An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-based Automated Program Repair

1st Dániel Horváth
Department of Software Engineering
University of Szeged
Szeged, Hungary
hoda@inf.u-szeged.hu

2nd Viktor Csuvik
Department of Software Engineering
University of Szeged
Szeged, Hungary
csuvikv@inf.u-szeged.hu

3rd Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

4th László Vidács
Department of Software Engineering
University of Szeged
Szeged, Hungary
lac@inf.u-szeged.hu

***Abstract*—Bug fixing is one of the most time-consuming and resource-intensive tasks in the software development life cycle. Automated Program Repair (APR) might be able to help in this process, but it still has to overcome many obstacles. Deep learning models have shown promise for automated program repair in recent years, but their effectiveness can depend on the representation of the source code used as input. In this paper, we conduct an experimental study to compare the performance of deep learning models on two popular programming languages, Java and JavaScript, using three different code representations: raw text, command sequences, and abstract syntax trees (ASTs). We also experiment with varying models, including T5, CodeT5, (for solving sequence-to-sequence tasks) RoBERTa, and GPTNeo (to encode/decode AST graph information). We evaluate the models on a set of real-world defects from open-source projects and compare the performance, and the repair patches generated by the models. Our results show that training on command sequence representation outperforms most other configurations. We achieve a best of 19.88% accuracy on the java-small dataset, and 11.87% on java-medium, using text representation. Using command sequence representation, we achieve 30.64% on java-small and 18.53% on the medium dataset. However, when representing the source with ast+text information, our models significantly underperform compared to other representations, achieving results below one percent. Our findings contribute to a better understanding of the strengths and limitations of deep learning models for automated program repair and provide practical guidance for their use in practice.**

***Index Terms*—Automated Program Repair, Transformers, T5, BERT, Deep Learning, Machine learning**

I. Introduction

Automated Program Repair (APR) is a field that aims to automatically fix software bugs without human intervention. While the idea of automatically repairing software bugs is attractive, this problem has proven to be a significant challenge for researchers. In the past, APR approaches have relied on an "oracle function", usually a test suite, to generate patches and validate them. One example of this is GenProg, which uses a genetic algorithm to generate patches and test them against the test suite [1]. However, there are no guarantees that the patches generated by these approaches are actually correct, and there has been a lot of research focused on addressing this issue [2], [3], [4], [5]. Despite these efforts, patch correctness remains an open question in the APR community [6]. Instead of solely focusing on generating high numbers of plausible patches, the APR community should also prioritize human studies and ways to improve the explanation and presentation of patches.

Data-driven repair approaches form a separate area of research within the field of Automated Program Repair [7]. These techniques typically involve creating a large database of train, test, and validation data and evaluating the APR tool on this dataset. The criteria for accepting a patch as correct are more stringent in these approaches, as the patch must be exactly the same as the one created by the developer who fixed the bug [5]. This approach has the advantage of eliminating two issues: the question of dataset bias, as it is a rigorous task to create patchable programs with a test suite, and the patch correctness issue, as it is assumed that the developer's patch is correct. Recently, this research direction has made significant progress with promising results reported in various studies [8], [9].

One recent development in APR is the use of transformer models, which are a type of neural network architecture that has been successful in various natural language processing tasks [10]. There have been several studies that have demonstrated the effectiveness of these models in the domain of Automated Program Repair. For example, a study by Phan *et al.* showed that a transformer-based APR system was able to

Ministry for Culture and Innovation and Ministry of Innovation and Technology

979-8-3503-0214-1/23/$31.00 ©2023 IEEE
DOI 10.1109/APR59189.2023.00013

fix a large number of real-world errors in Java programs with high accuracy [11]. Another study by Svyatkovskiy *et al.* used transformer models to address the problem of synthesizing missing code in programs and found that their approach was able to generate high-quality repair patches for a variety of code faults [12]. Overall, the use of transformer models for automated program repair has shown promise as an approach for addressing errors and bugs in software.

The motivation of this paper is to find out which program representation fits better for the APR task and whether these representations behave similarly with different Transformer architectures. To do so, we created different model configurations and fed 3 source code representations to them. The models are trained on 2 datasets of different languages: Java and JavaScript. This work's contribution is to provide a broader vision of the importance of how we choose to represent the data, the model will be training on. Our setup, data, and methods used are also available in a GitHub repository [1].

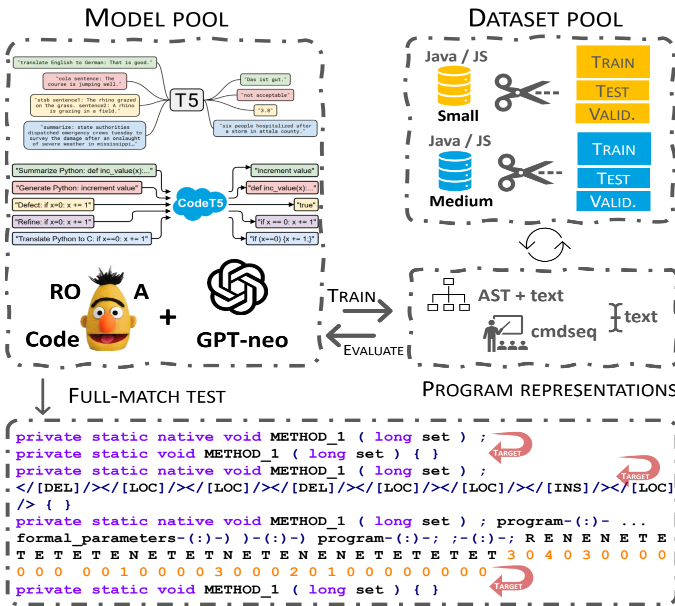

Fig. 1. High-level approach overview of this paper

II. METHOD

In Figure 1 we depicted the high-level approach of this paper. First, we created a "model pool" consisting of different Transformer model architectures including T5, CodeT5, RoBERTa, and GPTNeo. Once a model is selected, it is trained with a matching code representation on each of the observed datasets (Java and JavaScript). After training the models are evaluated using the standard evaluation procedure in learning-based APR approaches: the generated patch should be exactly the same as the one in the dataset (which is essentially the developer fix). In the following sections, we are going to introduce the used models and program representations in detail.

[1]https://github.com/AAI-USZ/APR23-representations

A. Deep-learning models

In this paper we experiment with different models, including T5 [10], CodeT5 [13], RoBERTa [14], CodeBERT [15], and GPTNeo [16]. T5-base and CodeT5-base are used mostly during our experimentation as pre-trained models for fine-tuning. When not using pre-trained models, we simply use the same configurations as T5-base or CodeT5-base, without initializing the model weights. However, as these models are designed to solve sequence-to-sequence tasks, when experimenting with ASTs, to handle graph and source code information, we use RoBERTa for encoding the graph part, and CodeBERT for encoding the source code part. These parts are concatenated, and then we use this embedding as the encoder state in GPTNeo, which acts as the decoder. When using these models, they are also initialized with pre-trained weights. Namely, for the program-encoder, we use codebert-base, and for GPTNeo we use gpt-neo-125M. Graph-encoder is initialized with empty weights since it has a custom vocabulary. In the following sections, we are going to present these models briefly.

1) T5 and CodeT5: T5, or Text-to-Text Transfer Transformer, is a Transformer based architecture that uses a text-to-text approach. It was first introduced by Raffel et. al. It was trained on a variety of natural language text scraped from the web, introducing the dataset "Colossal Clean Crawled Corpus" (C4) [10], a data set consisting of hundreds of gigabytes of clean English text. Every task – including translation, question answering, and classification – is cast as feeding the model text as input and training it to generate some target text. This allows for the use of the same model, loss function, hyperparameters, etc. across a diverse set of tasks. Compared to BERT, T5 adds a causal decoder to the bidirectional architecture and replaces the fill-in-the-blank cloze task with a mix of alternative pretraining tasks. CodeT5 has the same architecture, as the previously mentioned T5, and has been trained on CodeSearchNet [17]. It was further trained on a large number of public GitHub repositories written in C/C#. Transformer models have proven to excel in many Natural Language Processing (NLP) tasks, and while processing source code is different, they share many similarities.

2) CodeBERT, RoBERTa and GPTNeo: RoBERTa (Robustly Optimized BERT Approach), and CodeBERT has very similar architecture as BERT, as it is derived from the same base model. Basically, it is an encoder part of a Transformer model and has been proven to be a powerful tool for encoding. One key difference between RoBERTa and BERT is that RoBERTa was trained on a dataset of 160GB of text, which is more than 10 times larger than the dataset used to train BERT. Additionally, RoBERTa uses a dynamic masking technique during training that helps the model learn more robust and generalizable representations of words. To encode source code, as a pre-trained model, we chose codebert-base, which is an improved version of RoBERTa for encoding source code information [15]. To encode graph information we used a customized RoBERTa model, with a different vocabulary size that fits the number of tokens represented by graph nodes and

node-to-edge connections. Aside from this change, the model is the same as that used for text encoding. For the decoder, we use GPT-Neo [18], which is an open-source implementation of GPT-3 (Generative Pre-trained Transformer). It's architecture is based on the Transformer, which is an attention model - it learns to focus attention on the previous words that are the most relevant to the task at hand: predicting the next word in the sentence. GPT models proven to be highly efficient in generative tasks [18]. These models are combined to encode text and AST information from the source code and generate the fixed or improved version of it, using gpt-neo-125M as the decoder.

B. Program representation

Source code representation plays a crucial role in the effectiveness and efficiency of repair algorithms. One common approach is to represent source code as a variant of abstract syntax trees (ASTs), which capture the syntactic structure of the code and can be manipulated and transformed to generate repair patches. Other approaches have used lexical representations, such as token sequences or n-grams, or have employed more advanced techniques such as graph-based representations or neural network models [19]. Researchers have also explored the use of program slicing, which involves creating a reduced version of the program that focuses on the relevant portion of the code for the repair task [20]. These different representations can be used alone or in combination, and their choice can depend on the specific repair task, the programming language, and the available resources and constraints [21]. There are three program representations used in this paper, we summarized these in Table I. In the following sections, we describe the used representations in detail.

1) Text: One is used as the baseline representation, and it is also the most common one. Simply representing the source code as **text** is a straightforward, yet powerful enough representation. In text representation, the source and target code are simple series of tokens, where the model needs to learn how to transform the entire input, to its fixed version. However it is usually the case, that fixed versions of source code are pretty similar to the buggy version [22]. For us this means, that the decoder part of the model has to generate much of the redundant parts of the code.

2) Command sequence: It is also possible, to have the model generate only the code patches, where changes are required in the given source code [23]. This representation will be referred to as **command sequence** (cmdseq). As input it still contains the same information as in text representation, so the task for the encoder part of the models remains the same. However, when decoding, the model needs only to predict the necessary changes to be made to the source, to make it identical to the target. For this purpose, there are 3 special tokens added to the dictionary, [DELETE], [INSERT], and [LOCATION] tokens. Aside from this, the dictionary is the same as in text representation. For example, if a code segment needs to be cut from the source to generate the target, then upon learning, the model needs to predict something like: [DELETE] [LOCATION] [LOCATION]. The code patch to be deleted from the source. Location tokens need to be processed and filled with true positions, by another mechanism, e.g. pointer networks [23]. However, in this paper we only use the raw location tokens to see, if the decoder has an easier job figuring out targets represented by cmdseq only.

3) Abstract Syntax Tree + text: The third and last code representation we were experimenting with is to encode more information in the source. Namely, we try to encode AST information, along with the same text information as in text representation, referred to as **AST+text**. To represent the AST we use a representation of flattened nodes and edges, with added type information like whether the examined token is the root of the program, a node, an edge, or a terminal (leaf) node. We also include the child count of every node, as part of the AST information, which can be thought of as a special kind of positional embedding. The models used in this paper can handle such an encoding task with a minimal configuration, however, there exist other kinds of techniques for working with tree- or graph-like structures [24] [25]. The task of the decoder remains the same as in text representation. We need to predict the whole target sequence from the source. However the encoder, in theory, can have more representative power, since it has information about the AST representation of the program, too. This way the encoder consists of two main parts, one for encoding the AST information, which in our case a flattened graph, with some additional information, and two for encoding textual information given by the source code.

III. Datasets

A. Java

For Java, we used the dataset released by Tufano *et al.* in [26]. Later it became part of the popular CodeXGLUE benchmark [27], and can be found under the code-refinement task. The dataset contains Java source codes mined from GitHub. The programs have been normalized in a way, that they do not contain variable, method, or function names, but they have been abstracted away. For example, if a code contains a variable named *myVar*, then it will be referred to as *VARIABLE_1*. This preprocessing step is done file by file, resetting the variable index to 0, thus reducing the dictionary size. However, the dataset defines some **idioms** - common identifiers in the observed projects - which they see as "part of the language". These idioms occur frequently in all of the source codes, so they are part of the vocabulary, along with the language-specific keywords, or reserved words.

B. JavaScript

As for the JavaScript experiments, we used FixJS [28], which contains bug-fixing information for GitHub commits. Similarly to the Java dataset, FixJS also uses abstraction for the variable names to reduce the vocabulary size, although it includes the raw commit information as well, thus the original names can be obtained. Worth noting that while duplicated samples can occur in the Java dataset, the authors of FixJS filtered their data, so there is a strong guarantee that samples

TABLE I
EXAMPLES OF PROGRAM REPRESENTATIONS

Source Rep.	Target Rep.	Description	Example
text	text	Code is represented as is. Simple sequential text input, and output.	private static native void METHOD_1 (long set) ; => private static void METHOD_1 (long set) { }
text	cmdseq-token	Input code is represented as text, target is represented as a sequence consisting of *DELETE* and *INSERT* commands	public void METHOD_1 (final boolean VAR_1) { VAR_2 . METHOD_1 (VAR_1) ; } => </[INS]/></[LOC]/> this .
ast+text	text	Input represented as chains of AST and textual info. The output is a text rep. of the fixed source code.	boolean METHOD_1 ( ) { } program-(:)-program program-(:)-local_variable_declaration local_variable_declaration-(:)-local_variable_declaration local_variable_declaration-(:)-boolean_type boolean_type-(:)-boolean_type local_variable_declaration-(:)-variable_declarator variable_declarator-(:)-variable_declarator variable_declarator-(:)-identifier identifier-(:)-identifier identifier-(:)-; ;-(:)-; formal_parameters-(:)-formal_parameters formal_parameters-(:)-((-(:)-(formal_parameters-(:)-))-(:)-) program-(:)-block block-(:)-block block-(:)-{ {-(:)-{ block-(:)-} }-(:)-} R E N E T E N E T E T N E T E T E N E T E T 3 0 3 0 0 0 1 0 0 0 0 2 0 0 0 0 0 2 0 0 0 0 => boolean METHOD_1 ( ) ;

with different input/same output or same input/out pair cannot occur.

IV. EXPERIMENTAL SETUP

When optimizing the models we chose a learning rate of $5e-5$ [27], using an Adam optimizer, leaving the epsilon as its default value $1e-8$. For the small datasets we used a maximum sequence length of 256 and a batch size of 16, and as for the medium dataset we used a sequence length of 384 and batch size of 8. This configuration is used throughout the conducted experiments, aside from ast+text representation, where the model was proven to be too large to use batches of this size on one GPU. To train the models we used an RTX 3090 GPU. The base models working with text, and command sequence representations are of size 222M parameters. The model used on ast+text representation needs more encoding power, and it has additional layers for encoding AST information. For this reason, this model is considerably bigger, with 346M parameters.

All of the models in this work come from *transformers* library [29]. For the environment, we used Python 3.8, and *PyTorch* library along with *PyTorch-Lightning*. We trained for a maximum of 50 epochs, and used early stopping with a minimum delta of 0.05 on validation loss, with patience set to 8. This means that if a model could not reduce its loss by 0.05 in 8 epochs, then we assume that the model converged, and stop the training, and load the best model so far for testing. The loss function used throughout the experiments is the torch implementation of cross-entropy loss. On Java dataset [26] training time per epoch mostly took about half an hour on small, and one hour on the medium dataset, on text and cmdseq-token representation. Training on text+ast representation took about 1 hour and 40 minutes on small, and 2 hours on medium dataset on average. Training time on FixJS dataset [28] was significantly lower, due to the size of the dataset. It only took about 5 and 13 minutes per epoch to train the model on small, and medium sequences on text and cmdseq-token representation. Also, it only took 20 minutes to train on text+ast representation in both cases. All in all, the models took from 1 hour to about 1 day to train, varying highly with the size of the model, the dataset, and the representation used.

V. RESULTS

In this section, we are examining the results presented in Table II. At first glance, it is obvious that there can be huge differences, based on which dataset we train our models on. Firstly, we can deduce that it is considerably harder to train on FixJS [28], compared to the Java dataset [26]. Models evaluated on the Java dataset outperform evaluation metrics of models on FixJS by a significant margin. This may be, due to the fact that FixJS consists of 9662 data samples on small and 11410 samples on medium dataset, compared to Java dataset, which contains 58350 and 65455 data samples on small and medium length program codes, respectively. Also, FixJS is processed in a way, that every non-unique abstracted sample is filtered out, keeping only unique data. Even samples having the same target with different source representations are filtered out, which can lead to a considerable decrease in data samples, while also reducing redundancy in the database, making it *harder* to train on. To the best of our knowledge, the Java dataset [26] does not apply such filters to the data.

We can see that representing the target as a sequence of command tokens [23] can give a performance boost in some cases. On the Java dataset, this representation outperforms all the other representations, by at least 6%. However, this is not the case with the FixJS dataset, where using text representation performs a little better on average. Comparison between representations and datasets can be seen in Table III.

Although we only conducted one experiment training only the language model (LM) head in a pre-trained model (T5-

TABLE II
RESULTS ON TRAINING DIFFERENT MODELS ON VARIOUS DATA REPRESENTATIONS.

Model	Repr.	Pretrained	Dataset	Accuracy	100% accuracy	Epochs
T5-base, LM	text	Yes	Java small	0.9104	0.0000	42
T5-base, LM	text	Yes	Java medium	0.8577	0.0000	25
T5-base, empty	text	No	Java small	0.9371	0.0524	27
T5-base, empty	text	No	Java middle	0.9795	0.0652	31
T5-base	text	Yes	Java small	0.9756	0.1491	9
T5-base	text	Yes	Java middle	0.9864	0.0588	9
CodeT5-base	text	Yes	Java small	0.9684	0.1988	9
CodeT5-base	text	Yes	Java middle	0.9817	0.1187	9
T5-base, empty	cmdseq-token	No	Java small	0.7884	0.2598	12
T5-base, empty	cmdseq-token	No	Java middle	0.7476	0.1488	15
T5-base	cmdseq-token	Yes	Java small	0.8201	0.2946	16
T5-base	cmdseq-token	Yes	Java middle	0.8107	0.1733	18
CodeT5-base	cmdseq-token	Yes	Java small	0.8371	0.3064	12
CodeT5-base	cmdseq-token	Yes	Java middle	0.8051	0.1853	10
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	Java small	0.3862	0.0077	18
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	Java middle	0.2783	0.0000	13
T5-base, empty	text	No	FixJS sm all	0.8264	0.0030	24
T5-base, emtpy	text	No	FixJS middle	0.8107	0.0000	18
T5-base	text	Yes	FixJS small	0.9245	0.0496	1
T5-base	text	Yes	FixJS middle	0.9369	0.0262	16
CodeT5-base	text	Yes	FixJS small	0.9078	0.0754	9
CodeT5-base	text	Yes	FixJS middle	0.9207	0.0464	9
T5-base, empty	cmdseq-token	No	FixJS small	0.6917	0.0714	21
T5-base, empty	cmdseq-token	No	FixJS middle	0.6924	0.0149	21
T5-base	cmdseq-token	Yes	FixJS small	0.6502	0.0000	17
T5-base	cmdseq-token	Yes	FixJS middle	0.7265	0.0184	30
CodeT5-base	cmdseq-token	Yes	FixJS small	0.6190	0.0000	13
CodeT5-base	cmdseq-token	Yes	FixJS middle	0.7045	0.0639	16
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	FixJS small	0.2073	0.0010	24
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	FixJS middle	0.1504	0.0000	19

Models trained on Java [26], and FixJS [28] database. **T5-base, LM** is a pre-trained model, initialized from t5-base, with a language model on top. During training, only the LM-head has been optimized. Similarly, **T5-base** and **CodeT5-base** means the same model, initialized from t5-base [10] and codet5-base [13] respectively, however, here the whole model is fine-tuned. Models with the **empty** tag, are not pre-trained. **RoBERTa-base, CodeBERT-base, gpt-neo-125M** is a composite model, trained on text+ast data representation, and all of them are initialized using the same pre-trained weights from roberta-base [14], codebert-base [15] and gpt-neo-125M [16] respectively.

base, LM), it is apparent that training only the LM head is not enough as it performed poorly even after training for 42 and 25 epochs on small and medium Java dataset. In the case of the *flattened graph* model consisting of roberta [14], codebert-base [13] and gpt-neo-125M [16], we did not achieve good results. However, to say more on the matter, further experiments would be needed. Representing the source code as AST and text, concatenating the results, may have more representative power in theory, however decoding it, may fail due to insufficient model size. Batch sizes also needed to be halved during training to fit inside the GPU.

Seeing Table III, only in a quarter of the examined cases benefit from learning on simple text representation, namely T5-base on small and medium, and CodeT5-base on small models trained on FixJS. In other cases, we can say that learning on command sequence representation is easier for most of the models, suggesting that choosing a good representation for the given dataset may be crucial. However, not all datasets benefit the same way from the same representation. It is also apparent, that given the same methods, results can highly vary based on the dataset used. Some examples on generated patches using the java dataset [26] can be seen in the following listings.

Empirical evaluation: in the following paragraphs, we are going to present some examples we generated throughout our experimentations (listings: 1, 3, 5, 7). In the listings below, the first line is the input (the prompt for the model), the second line is the expected output and the last line is the output of the model. Every second listing (2, 4, 6, 8) contains the non-abstracted version of the same bugs, and the developer fixes too, for readability. However, it is important to note that, the models here, have not been trained on such representations. They are only here to present the nature of the bug to be fixed, in a more readable form.

```
public void METHOD_1 ( ) { VAR_1 . METHOD_2 ( VAR_2 ) ; VAR_2 = null ; }
</[INS]/></[LOC]/> ) { if ( ( VAR_2 ) != null</[INS]/></[LOC]/> }
</[INS]/></[LOC]/> ) { if ( ( VAR_2 ) != null</[INS]/></[LOC]/> }
```

Listing 1. Correctly patched program, using cmdseq-token representation.

```
public void unregisterNSDService() {
  mNsdManager.unregisterService(networkRegistrationListener);
  networkRegistrationListener = null;
}
public void unregisterNSDService() {
if ((networkRegistrationListener) != null) {
  mNsdManager.unregisterService(networkRegistrationListener);
  networkRegistrationListener = null;
  }
}
```

Listing 2. Non-abstracted bug, and developer fix of the example above.

TABLE III
COMPARING RESULTS ON JAVA AND FIXJS DATASET ON TEXT AND COMMAND SEQUENCE TOKEN REPRESENTATION.

Model	**Dataset Size**	**Text**		**Command Sequence Tokens**		**Text vs. Cmd.**		**Text**	**Cmd.**
		Java 100%	**FixJS 100%**	**Java 100%**	**FixJS 100%**	**Java**	**FixJS**	**Java vs. FixJS**	
T5-base, empty	Small	0.0524	0.0030	0.2598	0.0714	0.2074	0.0684	0.0494	0.1884
T5-base, empty	Medium	0.0652	0.0000	0.1488	0.0149	0.0836	0.0149	0.0652	0.1339
T5-base	Small	0.1491	0.0496	0.2946	0.0000	0.1455	-0.0496	0.0995	0.2946
T5-base	Medium	0.0588	0.0262	0.1733	0.0184	0.1145	-0.0078	0.0326	0.1549
CodeT5-base	Small	0.1988	0.0754	0.3064	0.0000	0.1076	-0.0754	0.1234	0.3064
CodeT5-base	Medium	0.1187	0.0464	0.1853	0.0639	0.0666	0.0175	0.0723	0.1214
Average		0.1072	0.0334	0.2280	0.0281	0.1209	-0.0053	0.0737	0.1999
Min.		0.0524	0.0000	0.1488	0.0000	0.0666	0.0078	0.0326	0.1214
Max.		0.1988	0.0754	0.3064	0.0714	0.2074	0.0754	0.1234	0.3064

Columns under **Text vs. Cmd.** and **Java vs. FixJS** show the differences between model performance on different data representations, and datasets, respectively. In the **Text vs. Cmd.** column results are positive (+) if cmdseq representation performed better than text representation and negative (-) otherwise. Consequently, columns found under **Java vs. FixJS** are positive (+) if a model performed better on Java dataset, and negative (-) otherwise. **Java 100%** and **FixJS 100%** means that the whole predicted sequence matches the target.

In the example above, we can see that in the first insertion command, the network can correctly guess, that the program needs to be extended with a null check. The second insertion command with the curly bracket is also a part of the previous null-check, closing the if statement. We can see that using command sequence representation it is possible to learn multi-line modification of the program using a few instructions, that only concern those parts of the code, which need modification, leaving the other parts of the code unchanged.

```
private void METHOD_1 ( TYPE_1 VAR_1 ) { VAR_1. METHOD_2 ( new TYPE_2 ( ) {
      public void METHOD_3 ( TYPE_3 VAR_2 ) { METHOD_4 ( ) ; } } ) ; }
</[INS]/></[LOC]/> < java.lang.String >
</[DEL]/></[LOC]/></[LOC]/>
```

Listing 3. Incorrectly patched program, using cmdseq-token representation.

```
private void addNotifyListener(javax.swing.JComboBox combo) {
  combo.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent e) {
      notifyChange();
    }
  });
}
private void addNotifyListener(javax.swing.JComboBox<java.lang.String> combo)
    {
  combo.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent e) {
      notifyChange();
    }
  });
}
```

Listing 4. Non-abstracted bug, and developer fix of the example above.

This example shows that the program should be patched by inserting a generic type parameter. However, according to the model, a deletion is needed. Much of the programs found in java-dataset need only a single delete command to be fixed, so it is safe to assume, that the model is biased towards guessing single deletion commands if it feels uncertain about the changes to be made.

```
public TYPE_1 append ( int value ) { VAR_1. METHOD_1 ( VAR_2, value ) ; }
public TYPE_1 append ( int value ) { VAR_1. METHOD_1 ( VAR_2, value ) ; return
        this ; }
public TYPE_1 append ( int value ) { VAR_1. METHOD_1 ( VAR_2, value ) ; return
        this ; }
```

Listing 5. Correctly patched program, using text representation.

```
public cz.lidinsky.tools.ToStringBuilder append(int value) {
  style.appendValue(sb, value);
}
public cz.lidinsky.tools.ToStringBuilder append(int value) {
  style.appendValue(sb, value);
  return this;
}
```

Listing 6. Non-abstracted bug, and developer fix of the example above.

This patch made on text representation correctly identifies a missing return statement. The method signature shows us that some type of value needs to be returned from the *append* function, however, the model likely encountered very similar examples during training, where a single *return this* statement was missing, as it can guess it correctly with seemingly no context suggesting this prediction.

```
public int METHOD_1 ( java.lang.Object VAR_1 ) { return list. METHOD_1 ( VAR_1
        ) ; }
public int METHOD_1 ( java.lang.Object VAR_1 ) { return 1 ; }
```

Listing 7. Incorrectly patched program, using text representation.

```
public int indexOf(java.lang.Object o) {
  return 0;
}
public int indexOf(java.lang.Object o) {
  return list.indexOf(o);
}
```

Listing 8. Non-abstracted bug, and developer fix of the example above.

Lastly, we can examine a wrongly predicted patch on text representation. While the model correctly guessed, that the return statement needs to be changed, and even the type is identified correctly, this case too shows symptoms of over-fitting. Simple patches like this can be found throughout the dataset, where a simple *return 1* statement is good enough. However, the generalization of more complex statements like this seems to fail.

Even though if we only consider the Java dataset it is clear that the CodeT5-base is superior on the command-sequence representation compared to others. On the other hand, the same setting does not seem to fit on the FixJS dataset. As we describe previously this might be the consequence that the learning problem is harder in the latter case (i.e. the nature of the programming language makes it hard to predict), or of the few training samples. Either way worth noting that we used

the same model size for each dataset, which is fair to compare, but not optimal from the data point of view. Optimizing the size of the model is out of the scope of this paper, but we consider it an important future research direction.

VI. Related Work

In this paper, we focused on the Transformer architecture, like most of today's modern tools that use deep-learning for specific code-related tasks. For example, in [6] authors fine-tune a T5 model for several tasks including fixing bugs, injecting code mutants, generating assert statements, and generating code comments. Their results show that the performance of the model is comparable to the state-of-the-art tools in each downstream task. In another recent work [30] Chen *et al.* address the problem of automatic repair of software vulnerabilities by training a Transformer on a large bug-fixing corpus and then tuning on a vulnerability fix dataset. They concluded that transfer learning works well for repairing security vulnerabilities in C compared to learning on a small dataset. Variants of the Transformer model are also used for code-related tasks, like in [31] where authors propose a grammar-based rule-to-rule model which leverages two encoders modeling both the original token sequence and the grammar rules, enhanced with a new tree-based self-attention. Their proposed approach outperformed the state-of-the-art baselines in terms of generated code accuracy. Another seminal work is DeepDebug [5], where the authors used pre-trained Transformers to fix bugs automatically. Here Drain *et al.* use the standard transformer architecture with copy-attention. They conducted several experiments including training from scratch, pretraining on either Java or English, and using different embeddings. They achieved their best results when the model was pre-trained on both English and Java with an additional syntax token prediction. This model is evaluated on the dataset by Tufano *et al.* [32], which was later included in the CodeXGLUE (General Language Understanding Evaluation benchmark for CODE) benchmark [33].

CodeXGLUE includes a collection of code intelligence tasks and a platform for model evaluation and comparison and refers to the data-driven repair approach as *Code Refinement (Code-Code)*. The team operates a leaderboard of the best-performing tools, where an approach called NSEdit [23] comes first at the time of writing this paper. NSEdit is a model that predicts the editing sequence given only the source code. They use both the encoder and decoder of the Transformer, where the encoder processes the buggy code, and the decoder predicts the editing sequence. As a grammar, the decoder uses a domain-specific regular language to write scripts that can transform source to target when executed. The grammar consists of two actions, delete and insert, which are added to the vocabulary of the language model as new tokens. They achieve a state-of-the-art result of 24.04% accuracy (100%) on small and 13.87% on medium dataset. Other approaches behind NSEdit on the leaderboard also usually use a transformer-based model for the code refinement task.

The question of how to present a buggy program to a DL model is of great importance. The use of Abstract Syntax Trees (ASTs) seems obvious, although this information does not help the learning process in some cases. In [34] Kim *et al.* examined the code prediction (next token prediction) task used in autocomplete systems. They showed that by making the Transformer architecture aware of the syntactic structure of code (the AST), it outperformed previous systems. In another work [19], authors introduce a tool named Hoppity, which predicts the changes to be made to the AST of JavaScript commits with a graph-based neural network. Hoppity was trained on a huge dataset consisting of 290,715 code change commits, however, neither their implementation nor the used dataset is available for further studies.

A few studies have attempted to understand the effect of changing program representation or code embedding in the APR domain. Navamar *et al.* in [35] trained 21 different generative models that suggest fixes for name-based bugs, including 14 different homogeneous code representations, 4 mixed representations for the buggy and fixed code, and three different embeddings. However their work focuses on code representations, it is limited to the study of abstraction levels of token/AST representations. Our work also differs in the incorporated DL model, since they used an NMT model, compared to this we used a Transformer. Other SE fields also study different code representations [4].

VII. Conclusions

In this paper, we experimented with 3 different program representations that are being fed to 8 deep-learning model configurations trained and evaluated on 2 datasets built for Java and JavaScript Automated Program Repair. Our results show that choosing the right representation for a given dataset can give a huge performance boost in some cases. Using a sequence of command tokens can reduce redundancy between source and target, by focusing on edits needed in the source code, while in some cases this method under-performs compared to simple text-to-text representation. From our experiments we can conclude (1) that each deep-learning setting requires its own data representation, thus (2) no representation fits for every dataset and every model. Future studies may benefit from examining the effects of using different representations of source code, either when encoding, decoding, or both, as this may lead to better model performance, using the same data.

Acknowledgement

The research presented in this paper was supported in part by the ÚNKP-22-3-SZTE New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation fund and by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The national project TKP2021-NVA-09 also supported the work. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of

Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

References

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 364–374.

[2] A. Ghanbari, "Revisiting Object Similarity-based Patch Ranking in Automated Program Repair: An Extensive Study," *Proceedings - International Workshop on Automated Program Repair, APR 2022*, pp. 16–23, 2022.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[4] V. Csuvik, D. Horvath, F. Horvath, and L. Vidacs, "Utilizing Source Code Embeddings to Identify Correct Patches," in *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 2020, pp. 18–25.

[5] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021*, pp. 1–8, jun 2021.

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

[7] M. Monperrus, "The Living Review on Automated Program Repair," Tech. Rep., dec 2020.

[8] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," pp. 1161–1173, 2021.

[9] Y. Li, S. Wang, and T. N. Nguyen, "DEAR: A Novel Deep Learning-based Approach for Automated Program Repair," *Proceedings - International Conference on Software Engineering*, vol. 2022-May, pp. 511–523, 2022.

[10] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[11] L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian, and Y. Ye, "CoTexT: Multi-task Learning with Code-Text Transformer," pp. 40–47, may 2021.

[12] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1433–1443. [Online]. Available: https://doi.org/10.1145/3368089.3417058

[13] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021.

[14] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[16] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow," Mar. 2021, If you use this software, please cite it using these metadata. [Online]. Available: https://doi.org/10.5281/zenodo.5297715

[17] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[18] S. Black, L. Gao, P. Wang, C. Leahy, and S. Biderman, "GPT-Neo: Large scale autoregressive language modeling with meshtensorflow," Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5551208

[19] E. Dinella, H. Dai, G. Brain, Z. Li, M. Naik, L. Song, G. Tech, and K. Wang, "Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs," Tech. Rep., 2020.

[20] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 34–44.

[21] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 213–223.

[22] V. Csuvik, D. Horváth, M. Lajkó, and L. Vidács, "Exploring plausible patches using source code embeddings in javascript," *CoRR*, vol. abs/2103.16846, 2021. [Online]. Available: https://arxiv.org/abs/2103.16846

[23] Y. Hu, X. Shi, Q. Zhou, and L. Pike, "Fix bugs with transformer through a neural-symbolic edit grammar," 04 2022.

[24] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, apr 2022. [Online]. Available: https://doi.org/10.1109%2Ftse.2020.3020502

[25] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: Continual repair across programming languages," 2022. [Online]. Available: https://arxiv.org/abs/2205.10956

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

[27] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[28] V. Csuvik and L. Vidács, "Fixjs: A dataset of bug-fixing javascript commits," in *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 2022, pp. 712–716.

[29] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[30] Z. Chen, S. J. Kommrusch, and M. Monperrus, "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code," *IEEE Transactions on Software Engineering*, apr 2022.

[31] "Grammar-Based Patches Generation for Automated Program Repair," *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 1300–1305, 2021.

[32] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 25–36.

[33] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," *undefined*, 2021.

[34] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *Proceedings - International Conference on Software Engineering*, pp. 150–162, may 2021.

[35] M. Namavar, N. Nashid, and A. Mesbah, "A controlled experiment of different code representations for learning-based program repair," *Empirical Software Engineering*, vol. 27, no. 7, dec 2022.