Framing Program Repair as Code Completion

Francisco Ribeiro
francisco.j.ribeiro@inesctec.pt
HASLab/INESC TEC
Universidade do Minho
Braga, Portugal

Rui Abreu
rui@computer.org
INESC-ID & FEUP
University of Porto
Porto, Portugal

João Saraiva
saraiva@di.uminho.pt
HASLab/INESC TEC
University of Minho
Braga, Portugal

ABSTRACT

Many techniques have contributed to the advancement of automated program repair, such as: generate and validate approaches, constraint-based solvers and even neural machine translation. Simultaneously, artificial intelligence has allowed the creation of general-purpose pre-trained models that support several downstream tasks. In this paper, we describe a technique that takes advantage of a generative model — CodeGPT — to automatically repair buggy programs by making use of its code completion capabilities. We also elaborate on where to perform code completion in a buggy line and how we circumvent the open-ended nature of code generation to appropriately fit the new code in the original program. Furthermore, we validate our approach on the *ManySStuBs4J* dataset containing real-world open-source projects and show that our tool is able to fix 1739 programs out of 6415 — a 27% repair rate. The repaired programs range from single-line changes to multiple line modifications. In fact, our technique is able to fix programs which were missing relatively complex expressions prior to being analyzed. In the end, we present case studies that showcase different scenarios our technique was able to handle.

CCS CONCEPTS

• **Software and its engineering → Software evolution**; *Software post-development issues*; Software creation and management.

KEYWORDS

program repair, code generation, code completion

ACM Reference Format:
Francisco Ribeiro, Rui Abreu, and João Saraiva. 2022. Framing Program Repair as Code Completion. In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524459.3527347

1 INTRODUCTION

Automated Program Repair (APR) is a prominent field of software engineering. The continuing increase in complexity and size of software systems urges the community to invest its efforts on developing techniques that automatically identify patches that are able to fix faults arising from the implementation of new functionalities and code maintenance [11]. These patches are generated based on the original buggy program and take advantage of the fact that the developers' efforts result in almost accurate programs or, as DeMillo et al. [6] put it: "they create programs that are close to being correct!". Many approaches have been developed that successfully achieve this repair task. Early works [1, 16] utilize genetic programming by considering a buggy program as seed which is then continuously evolved at each generation by producing different programs through mutation and crossover. Other approaches [9, 21, 30] are constraint-based and analyze information from test executions to create constraints which are then fed to a solver to generate a patch.

More recently, APR techniques have taken advantage of machine learning advancements to build deep learning models. Some of these techniques [5, 8, 17, 19] employ Neural Machine Translation (NMT) to translate buggy code into fixed code. Likewise, general-purpose tools and models [10, 18, 27] supporting code understanding and code generation tasks have been developed.

In this paper, we argue that the code generation capabilities of pre-trained models like CodeGPT can be leveraged to specifically target program repair, effectively treating it as a code completion task. Let us consider an example from a real-world open-source software project.

```
171c171
< ... keyValueSequence = new ArrayList<Data>(|);
---
> ... keyValueSequence = new ArrayList<Data>(|entries.size());
271c271
< ... int mapLoadChunkSize = |nodeEngine.getGroupProperties().
      MAP_LOAD_CHUNK_SIZE.getInteger();
---
> ... int mapLoadChunkSize = |getLoadBatchSize();
```

The previous code shows two buggy lines and their corresponding fixes underlined. The expressions that repair this program — `entries.size()` and `getLoadBatchSize()` — are not trivial to figure out, even if we know that lines 171 and 271 are responsible for this bug. More precisely, repairing this bug implies the developer not only determines the incorrect expressions but also how to expand them. However, the complexity of this task can be reduced to simply performing code completion on the spot highlighted by the vertical bar to replace the leading code. We assume faulty line numbers are identified beforehand by well-established and accurate fault localization techniques [4, 14, 22, 23, 29].

This paper presents a repair technique that, given a file and buggy line numbers, seeks to fix a program by computing the most appropriate columns to perform code completion and incorporating the generated code in the original program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APR'22, May 19, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527347

The contributions of this paper are: (1) a technique that repairs buggy programs based on code completion; (2) a publicly available implementation of such technique; (3) a validation on a dataset of real-world projects with results showing our technique is able to fix 1739 programs out of 6415, representing a 27% repair rate; (4) a case study investigation highlighting some capabilities of our work.

The original aim of code completion is to assist the developer while writing code. Throughout the paper, we use annotations like the vertical bar representing the cursor position in a text editor and color highlights showing intended or generated completions. However, these serve to better visualize our approach's behavior. The primary goal of this work is to develop a technique and tool that uses code completion to produce patches without developer intervention.

2 BACKGROUND

Research in Natural Language Processing (NLP) focuses on how natural language can be processed, analyzed and manipulated by computers. Although its roots are based on symbolic rules and statistical modeling, the more recent adoption of machine learning models has allowed this field to flourish as one of the most prevalent areas of study in computer science. The continued work by the community has led to the specialization of certain subtasks within NLP into well-defined processes. Natural Language Understanding (NLU) analyzes natural text and appropriately encodes it into more low-level representations, while Natural Language Generation (NLG) transforms these machine representations into natural language text. NLP has benefited immensely from the application of neural networks, which allowed for the development of complex but highly effective models like BERT [7] and GPT [24] that have achieved tremendous success in language understanding and language generation tasks, respectively. These state-of-the-art models are based on the Transformer [28] neural architecture, which has shown to be more advantageous than previously used RNN-based architectures for analyzing longer and deeply-rooted dependencies. This is, in most part, thanks to a self-attention mechanism which allows the model to create connections between every token in a sequence no matter the distance between them. As a consequence, the representation of each token will be affected by its relationship with other ones, creating a more meaningful aggregate representation.

More recently, inspired by the significant advances in this area, the community has also directed its focus to the application of NLP principles regarding programing languages. In fact, software developers have been incorporating these tools into their workflow as they find them to have a positive effect in their productivity. One of the most sought-after capabilities in these systems is code completion [3] and every IDE or code editor supports this key feature. However, many of them provide this at a basic level, such as API call and parameter completion, limiting their usage to scenarios in which a developer needs to have a specific idea already typed in. Because of the success of pre-trained models like BERT and GPT, the architectures behind them have been used to create corresponding adaptations directly suited for programming languages. Thus, code understanding and code generation have allowed for advancements regarding the previous limitations through models such as IntelliCode Compose [27], CodeBERT [10] and CodeGPT [18].

CodeGPT is able to generate long and complex code sequences that are computed based on the context provided to the model. This input context consists of code preceding the point from which we wish the model to start generating more code. Essentially, the produced code sequence acts as a continuation of the original code piece. This way, CodeGPT can be used to perform code completion. In this work, we do not use this capability to help developers fill in the most suitable names for method calls or variable identifiers. Instead, we leverage it to inject new segments of code into existing buggy programs and modify their behavior. As a result, we show that we can take advantage of a code completion mechanism to conduct an entirely different task — automated program repair.

3 REPAIR TECHNIQUE

Our approach can be divided in four components, as shown in Figure 1:

(1) **Cutting:** the two inputs are the buggy file and the buggy line number. After the buggy file is parsed into its abstract syntax tree, we extract the nodes located at the buggy line number. As we mentioned, we assume the faulty line is already provided by some fault localization technique. Then, based on the criteria implemented in Algorithm 1 (described in Section 4), we compute the column numbers representing the places for which code completion is to be performed. Lastly, we truncate the buggy file at those columns, creating a file for each alternative;
(2) **Code Generation:** we perform code completion for each truncated file by providing an array of tokens as context to *CodeGPT*. Through random sampling, the model generates several token sequences, thus producing alternative ways of continuing the input code. These sequences are decoded and output as strings.
(3) **Bounding:** the code completion step is open-ended. That is, the model generates code without necessarily stopping at some suitable character regarding the language's syntax. As such, it is very likely that the last generated token does not terminate a well-formed expression or statement, as the output will finish once the maximum context size is reached. Likewise, the essence of the generated code may be sound except for the initial tokens. For this reason, we limit the generated code sequences based on relevant characters regarding the language's syntax to extract valid completions.
(4) **Character Synchronization:** the final step is to attach the generated completions to the original buggy code. The way we do this is by using characters that allow each completion to fall into place in the original buggy line, combining both pieces of code to produce a potential patch.

4 TRUNCATION ALGORITHM

As we have discussed before, in this work we leverage code generation by using the *CodeGPT* model to produce potential patches. We can interpret this code generation step as code completion being performed at a specific column in a line of code — similar to what is normally seen in text editors and IDEs.

Let us consider the the introductory example again. Listing 1 represents the desired fixed lines.

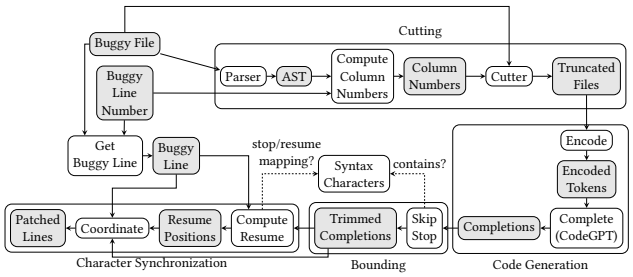

Figure 1: Architecture

```
> ... keyValueSequence = new ArrayList<Data>(entries.size());
> ... int mapLoadChunkSize = getLoadBatchSize();
```

Listing 1: Introductory example — desired completion

For this case, code completion would happen at the illustrated cursor position and the code to be generated is highlighted in grey. Therefore, we need to compute the places in that line for which we want to perform code completion. As such, we devised an algorithm that aims to compute suitable column numbers for the purpose of generating code sequences. We implemented it as a tool and make it available [1].

We targeted two scenarios when designing the algorithm. Code completion is frequently useful when developers want to predict:

- Code to continue specific language constructs (e.g. methods to invoke after ".");
- Candidate names for partially written identifiers (e.g. writing a variable's name halfway through).

As such, our algorithm computes column numbers based on:

- Textual boundaries of language constructs represented by AST nodes;
- Camel-case and underscore separation of words according to Java naming conventions.

Algorithm 1: Algorithm for computing column numbers

```
Data: An AST T1
Result: The list of computed column numbers ColNrs
ColNrs ← [];
foreach node ∈ T1 do
    start ← node.firstColNr; end ← node.lastColNr;
    add(start, ColNrs);
    add(end, ColNrs);
    if node is Identifier then
        name ← node.name;
        for i ← 1,size(name) do
            if (i ≠ size(name) and
                name_i is lowerCase and name_{i+1} is upperCase)
             then
                add(start+i+1, ColNrs);
            else if name_i == '_' then
                add(start+i, ColNrs); add(start+i+1, ColNrs);
            end
        end
    end
end
```

[1] https://github.com/FranciscoRibeiro/code-truncater

This means that our algorithm would truncate Listing 1 at the following columns:

```
>     ... key|Value|Sequence| = |new |Array|List|<|Data|>|()|;|
>     ... int| |map|Load|Chunk|Size = |node|Engine|.|get|Group|Properties()|.|MAP|_|
      LOAD|_|CHUNK|_|SIZE|.|get|Integer()|;|
```

Code generation would then be performed at each computed column. As executing the *CodeGPT* model is a time consuming task, the aim of our algorithm is to minimize the amount of requested predictions. A brute-force alternative would truncate the source code lines at every column (i.e. every character). However, such an approach would incur in a lot of computational effort as the number of columns to perform code completion on would increase considerably. As mentioned, the purpose of the truncation algorithm is to save time and effort on the code generation step by reducing the number of code sequences provided to the model, albeit with the drawback that some columns will be missed. In the provided example, the computed columns do not include the ideal one, as Listing 1 highlights. However, we shall see ahead that these occurrences are not necessarily a problem and that this program can still be fixed.

5 CODE GENERATION

We use the CodeGPT-adapted model to perform open-ended code generation. CodeGPT-adapted inherits the same model architecture of GPT-2, which is a natural language model conceived to perform multiple tasks such as text translation, question answering and text summarization. These tasks imply text generation, which makes GPT-2 a generative model. CodeGPT-adapted is based on GPT-2 as a starting point and is trained on code samples, making it a language model pre-trained for programming language (PL). Two separate versions of CodeGPT-adapted are provided for Python and Java, with the latter being the focus of this work. These models are made available through HuggingFace's Transformers library which provides a Python API.

One of the motivations of this work is to assess how program repair can be seen as a code generation task, more specifically code completion. As such, to perform code completion on buggy programs, we first need to provide a sequence of input tokens to the model. This will be the context to consider to continuously generate new sequences of tokens. After establishing the column to perform code completion on (as per the previous section), we retrieve the previous 1000 tokens and feed them to the model in order to generate the sequence to follow[2]. However, we do not want to limit ourselves to a single prediction and wish to explore several completion possibilities. *Greedy search* generates a sequence of tokens by following the path with the highest probability and *beam search* allows us to explore different hypothesis each time by keeping track of multiple high probability paths. Although *beam search* avoids restricting ourselves to only one completion, it is still based on the tokens with highest probability, making the different generations similar to each other [12]. To circumvent this, we use an indeterministic scenario to produce several completion possibilities. Instead of deciding the next token based on the highest probability,

[2] CodeGPT has a context size limit of 1024 tokens, so we have to leave some space for the output tokens. However, the number of input and output tokens is easily parameterizable in our tool and different values can be explored.

we ask the model to make this selection based on a conditional probability distribution through *sampling*. This way, we introduce randomness in the generation task and have more diversification.

For each column, we consider a sequence of 20 newly generated tokens and repeat this step 10 times in order to produce different code completions for the same input as shown in Algorithm 2.

Algorithm 2: Algorithm for generating code sequences

```
Data: A truncated program TP, a code generation model CodeGPT
Result: The list of code completions Completions
Completions ← [];
tokens ← lastTokens(TP, 1000);
foreach i ∈ 1..10 do
    completion ← complete(CodeGPT, tokens, length=20, sampling=true);
    add(Completions, completion);
end
```

6 BOUNDING CODE GENERATION

Code completion is done by using *CodeGPT* to perform code generation. As this process is open-ended, we need to focus on portions of the generated sequences before inserting them in the buggy code. To do this, we defined criteria that bound code sequences at various places, generating different sub-sequences of interest. The boundaries are specified by combinations of characters that are relevant regarding code syntax. Sometimes, the context provided to the code generation model may produce slightly off but almost correct results. As such, it is crucial that we discard elements in the beginning and in the end of token sequences to remove such noise from the nearly accurate predictions.

Skip. In order to reject incorrect tokens from the start of the generated code sequences, we *skip* those characters. The code in Listing 2 denotes an example in which the sub-token `EXT` should be removed in order to fix the bug.

```
150c150
< GL.glGenTextures|EXT(n, textures, Memory.getPosition(textures));
---
> GL.glGenTextures(n, textures, Memory.getPosition(textures));
```

Listing 2: Change method call — sub-token `EXT` removal

Asking *CodeGPT* for 10 different code completions on the cursor position outputs the following predictions.

```
EXT(n, textures, Memory.getPosition(textures));}void glUniform3
EXT(n, textures);} public void glTexParameterf (int index, float fval
EXT(n, textures, Memory.getPosition(textures));} public void glSten
EXT(n, textures, memory.getPosition(textures));} public void glFramebuffer
EXT(n, textures, Memory.getPosition(textures));} @Override public void flush
EXT(n, textures, Memory.getPosition(textures));} public void glVertex
EXT(n, textures, Memory.getPosition(new Integer(n)));}public void
EXT(n, textures, Memory.getPosition(textures));} public int nGLObject
EXT(n, textures, Memory.getPosition(textures));}void glGetA0
EXT(n, textures);}} public void glVertexBegin (int x, int y
```

Listing 3: Generated completions for Listing 2

Every alternative begins with the undesired `EXT` word. However, the subsequent generated tokens of some completions are able to build the expected code until the character ending the statement: ';' (semi-colon). The character '(' (left parenthesis) has special significance in the language — in this case, establishing the beginning of the parameters. As such, we can use this knowledge and *skip* the starting tokens in the predictions until we find the left parenthesis.

Stop. As generated sequences can be potentially unlimited, there is still the problem of determining at what point we *stop* considering the output tokens. Similarly to the previous situation, code generation is performed without considering any syntactic aspects. As such, we again resort to specific characters in order to decide the locations after which we stop incorporating tokens for patch production.

The completions in Listing 3 have multiple characters that can be considered for the *stopping* criteria and that will lead to the creation of successful fixes. Taking the first completion line from Listing 3, considering the left parenthesis as a *skip* criteria and comma, space, left parenthesis and semi-colon as *stop* criteria would trim the sequence and produce the following possibilities:

```
EXT(n, textures, Memory.getPosition(textures));}void glUniform3
EXT(n,␣textures, Memory.getPosition(textures));}void glUniform3
EXT(n, textures, Memory.getPosition(textures));}void glUniform3
EXT(n, textures, Memory.getPosition(textures));}void glUniform3
```

Listing 4: Trimmed sequences from the previous completions

Although it may seem the first three alternatives stop earlier than intended, there is still a step to perform in order to fit the trimmed sequences in the buggy code. This last synchronization step will ensure these slices are applied correctly to the buggy code.

7 CHARACTER SYNCHRONIZATION

The last step to produce a candidate patch consists of fitting the trimmed completions in the buggy code. As explained, completions were altered regarding different criteria to produce adequate alternatives. The final step to correctly incorporate these pieces of code is to synchronize the sequences with the line of code from the original program. Similary to the previous section, this synchronization process is also achieved by coordinating the occurrence of relevant characters of the language's syntax.

Considering the example in the previous section, the first trimmed completion from Listing 4 (second line in Listing 5) can be incorporated in the buggy program (first line in Listing 5) by synchronizing both code sequences on the first occurrence of the comma character, thus producing the desired fix (third line in Listing 5).

```
bug:    GL.glGenTextures|EXT(n, textures, Memory.getPosition(textures));
completion:             |EXT(n, textures, Memory.getPosition(textures));}void...
patch: GL.glGenTextures(n, textures, Memory.getPosition(textures));
```

Listing 5: Produced fix for Listing 2

As we can see, the completion is successfully integrated in the original program and we are able to maintain the rest of the code accordingly.

8 EXPERIMENTS

To validate our approach we used the *ManySStuBs4J* dataset [15] to conduct a large scale experiment [3]. The dataset version mined from 100 open source Java projects, containing 11624 bugs, was filtered and bugs that did not fit the following criteria were removed:

- line numbers reported in the dataset match the ones obtained through our automated analysis;
- the bug can be repaired only through line changes, i.e. line additions and deletions are not necessary;

[3] https://gitlab.com/FranciscoRibeiro/manysstubs4j-experiments

- fixes are not produced by changing string literals;
- our truncation algorithm computes at least one column number.

After this filtering step, 6415 bugs remained. For all these bugs, we applied a similar procedure to what was described in Section 3. However, there is a difference in the way column numbers were computed. Although our algorithm computes less column numbers than a brute-force approach, applying it to a dataset with such an amount of bugs would make the experiments impractical by taking considerable time to execute. Instead, for each line, we compared the buggy and the fixed version and used the first differing character (column) as the place to truncate the program. Considering Listing 2, the cursor position denotes the column used in that case, as that is where characters start differing.

Out of the 6415 bugs, our technique was able to fix 1739 programs, representing a 27% effectiveness rate. Even though *random sampling* introduces indeterminism during code completions done by *CodeGPT*, the implementation of our work allows for reproducibility as we fix the seeds for random number generation. This implies that if one of the column numbers computed by the truncation algorithm matches the first differing character, our approach will be able to fix it as the sequence of generated tokens will be the same and the rest of the pipeline is deterministic. The truncation algorithm is able to successfully compute the column numbers used in these experiments for 5674 bugs, which means the algorithm can infer the closest place to the bug 88% of times. Although this step fails to calculate the nearest column number for some cases, it does not mean that our technique is not able to repair them. In fact, performing code completion at different column numbers may still produce a fix. From the 1739 fixed programs, the truncation algorithm did not compute the nearest column for 97 of them. Nonetheless, our technique was still able to fix these programs.

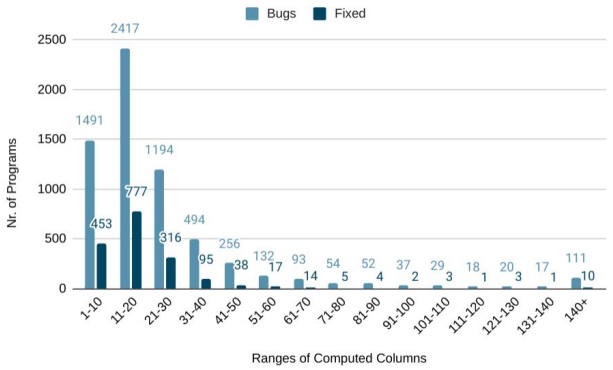

Figure 2: Bugs per range of computed column numbers

Even though code completion was not performed on every plausible column, we still applied the algorithm to compute such column numbers to all the bugs in the study. Figure 2 shows the number of bugs per ranges of computed columns — size 10 buckets. The algorithm computes between 10 and 20 column numbers for 2417 bugs, representing 38% of the analyzed programs, and we are able to fix 777 of them, which results in a 32% repair rate.

Figure 3 shows the amount of bugs per number of lines. The vast majority of the studied programs — 4610 — are single-line

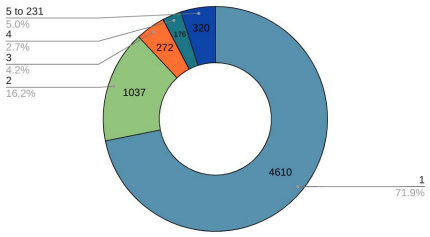

Figure 3: Bugs per lines to modify

bugs, which consists of 72% of the total amount. Multi-line bugs range from 2 to 231 lines with 1037 and 2 programs respectively. As single-line bugs are the most predominant, it is relevant to focus on this sizable segment of programs to understand how values are distributed.

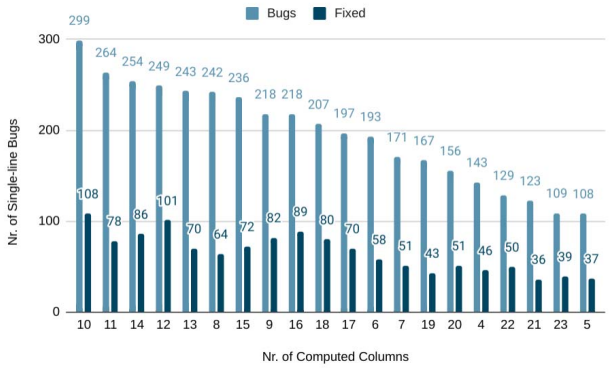

Figure 4: Single line bugs per computed column numbers — top 20

Figure 4 shows the top 20 computed column numbers for single-line bugs. This data does not present a discrepancy when compared to the distribution illustrated in Figure 2 for all the bugs, as the most frequent number of columns are contained within the top-3 buckets. Furthermore, our approach is able to fix 1502 of these programs, resulting in a repair rate of 33% for this set of bugs. The fact that this segment represents a significant portion of the dataset is also reflected in the number of repaired programs, with 86% of all 1739 fixed programs being part of this group of bugs.

The largest cluster of bugs in the dataset — single-line — has its top-20 most frequent numbers of computed columns in line with the global top-20 — Figure 5 — with only the last two places, 23 and 5, missing from it.

9 CASE STUDIES

In this Section we present specific examples of programs to explore relevant scenarios that showcase the advantages of our approach.

Case Study 1. Listing 6 shows a bug and its corresponding fix.

```
404c404
< } else if (itemActionLayout >= 0) {
---
> } else if (itemActionLayout > 0) {
```

Listing 6: Case study 1 — bug and fix

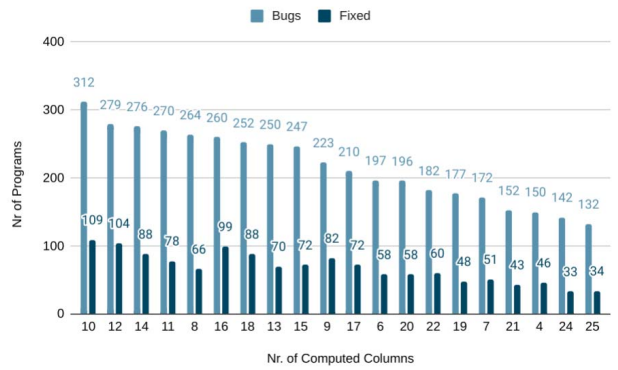

Figure 5: Bugs per computed column numbers — top 20

In the buggy line, we can see the column for which code completion should be performed. However, the algorithm defined in this work does not compute that column number because splitting a binary operator does not fit the defined criteria, as Listing 7 shows.

```
} else if (|item|Action|Layout| >= |0|) {
```

Listing 7: Case study 1 — computed columns

Nonetheless, we are still able to produce a fix in this situation by making use of a completion after `itemActionLayout`.

```
bug:  } else if (itemActionLayout| >= 0) {
completion:                      |!= null) {if (itemShowAsAction > 0) {item...
patch:} else if (itemActionLayout> 0) {
```

Listing 8: Case study 1 — produced fix

Even though *CodeGPT* does not complete the condition with the intended binary operator (>) and operand (0) straightaway, as a result of outlining the generated sequence (Section 6), we are able to *skip* (highlighted in grey) undesired tokens and make use of a subsequent comparison from the initial completion. By then *stopping* at the closing parenthesis and discarding tokens coming afterwards, we can synchronize (Section 7) the extracted portion of the completion (highlighted in green) with the buggy line and produce the patch shown in Listing 8.

Case Study 2. Listing 9 illustrates a scenario for which our technique was able to expand an existing condition.

```
78c78
<     return mModelClasses|.size() > 0;
---
>     return mModelClasses != null && mModelClasses.size() > 0;
```

Listing 9: Case study 2 — bug and fix

As we can see from Listing 10, the truncation algorithm computes the closest column to the bug. Therefore, our approach is able to successfully produce a repair for this program.

```
|return |m|Model|Classes|.|size|()| > |0|;|
```

Listing 10: Case study 2 — computed columns

As results are replicable, we can safely infer *CodeGPT* would generate the same code sequences for that column without incurring in the overhead of applying our approach to every computed column.

```
bug:    return mModelClasses|.size() > 0;
completion:                 |!=null && mModelClasses.size()>0;} public...
patch: return mModelClasses!=null && mModelClasses.size()>0;
```

Listing 11: Case study 2 — produced fix

For this example, code completion was able to generate useful tokens right from the beginning. As a consequence of using variable `mModelClasses` to produce the first comparison (`mModelClasses!=null`), the original expression, which constitutes the second part of the condition, is lost. Nonetheless, the continuation of the token sequence generates the desired code (`mModelClasses.size()>0`), eliminating the problem. After discarding tokens to the right of the semi-colon (grey color) and using the same character to synchronize the resulting sequence with the buggy code, we produce a patch that fixes this program as shown in Listing 11.

Case Study 3. The example in Listing 12 illustrates a multi-line bug that we are able to fix by performing the procedure on three separate lines.

```
176c176
<   if(request.get|TaskDefinitionKey() != null) {
---
>   if(request.getDueDate() != null) {
179c179
<   if(request.get|TaskDefinitionKey() != null) {
---
>   if(request.getDueBefore() != null) {
182c182
<   if(request.get|TaskDefinitionKey() != null) {
---
>   if(request.getDueAfter() != null) {
```

Listing 12: Case study 3 — bug and fix

For every buggy line, the column closest to the bug's location is computed as can be seen in Listing 13.

```
if(|request|.|get|Task|Definition|Key|()| != |null|) {
```

Listing 13: Case study 3 — computed columns

Similarly to the previous case study, as the experiments are reproducible, our pipeline will always produce the same results for this column, which assures us this bug is fixable.

```
bug:    if(request.get|TaskDefinitionKey() != null) {
completion:           |DueDate()!= null) {taskQuery.dueDate(request.get...
patch: if(request.getDueDate()!= null) {

bug:    if(request.get|TaskDefinitionKey() != null) {
completion:           |DueBefore()!= null) {taskQuery.dueBefore(request.get...
patch: if(request.getDueBefore()!= null) {

bug:    if(request.get|TaskDefinitionKey() != null) {
completion:           |DueAfter()!= null) {taskQuery.dueAfter(request.get...
patch: if(request.getDueAfter()!= null) {
```

Listing 14: Case study 3 — produced fix

All three lines are fixed in a similar way. However, the method names that need to be generated are different from each other. Code completion is able to produce the necessary tokens from the start and we only need to *stop* considering the sequence after the first opening parenthesis. Both the buggy line and the code sequence are

synchronized also using the opening parenthesis, thus creating the patches seen in Listing 14. On the other hand, some of the discarded tokens (`!= null){`) could also be utilized for patch production as they would correctly complete the buggy code. As such, this program could be fixed by using different characters for bounding the generated code sequence, like the closing parenthesis or the opening curly bracket.

Case Study 4. Listing 15 represents a bug for which a method call needs to be replaced with a constant.

```
272c272
<     buf.get(bulk, buf.position(), len);
---
>     buf.get(bulk, 0, len);
```

Listing 15: Case study 4 — bug and fix

Again, the truncation algorithm correctly computes the column nearest to the bug as Listing 16 illustrates.

```
|buf|.|get|(|bulk|, |buf|.|position|()|, |len|);|
```

Listing 16: Case study 4 — computed columns

As a result of reproducibility, the circumstances certify this bug is fixable under our approach.

```
bug:    buf.get(bulk, buf.position(), len);
completion:           |0, len); os.write(bulk); } dos.write(buf.array(),
patch: buf.get(bulk, 0, len);
```

Listing 17: Case study 4 — produced fix

Listing 17 shows the produced fix. In this case, the line is truncated at the beginning of the expression that needs to be replaced. However, the necessary expression (`0`) is much different from the original one (`buf.position()`). Nevertheless, the code completion step is able to infer the next tokens correctly from the provided context and the relevant part is extracted accordingly. There is no need to *skip* any unnecessary tokens. Additionally, we can simply *stop* considering the generated token sequence after the comma character and also use it to synchronize with the original code. As in the previous case study, this bug may be fixed in different ways. The remainder of the code sequence (`len);`) can be safely inserted in the original program as it corresponds to an already correct part. For this to happen, the closing parenthesis or the semi-colon need to be used for limiting token generation and synchronization.

10 THREATS TO VALIDITY

The main objective of our work is to assess whether program repair can be tackled as a code completion task. More precisely, can we use code generated by deep learning models such as CodeGPT to evolve faulty programs in order to correct their behavior? We believe our work shows the answer to this question to be yes. Nonetheless, we recognize that there are some elements to consider that may challenge our work's rationale.

Internal Validity: Some programs need larger and more complex changes in order to be repaired. That is, not all pairs of bugs and corresponding fixes are equivalent in kind. Essentially, this means that different programs need to meet different demands to be classified as correct. However, we consider that the results show our approach was successfully applied to programs of different kinds going from needing small adjustments to multiple intricate changes.

External Validity: The reported results are obtained by analyzing programs from a dataset aiming to provide a collection of single statement bugs. As such, these simple bugs may not be representative of the real-world complexity of software and its needed changes. However, the dataset used in our work was created by extracting actual occurrences from real-world open-source projects, showing that such instances typically arise. In addition, some multi-line changes may be seen as aggregates of multiple single-line modifications. Aside from that, our work targets *Java* programs and, thus, does not encompass a lot of other languages. Nonetheless, many of the language's features and constructs are common to other languages and *Java* is one of the most used by developers.

Construct Validity: Some typical NLP practices were put into place in our work. For text generation, maximizing the probability of the decoded segments leads to poor quality outputs, contrasting with the training objective used to build such models. Higher quality text can be obtained by employing a decoding strategy that uses sampling [12]. Even though these sampling techniques have their roots in NLP, we are convinced we successfully applied them to a PL setting as we were able to fix more programs by not only producing multiple alternatives but also making them more reliable.

11 RELATED WORK

Barr et al. [2] implement a technique for software transplantation that allows for transfering behavior from a *donor* to a *host* program. The part of the first program that is of interest is called the *organ* and the intention is to recreate the feature it represents in a potentially unrelated target program. This methodology has several applications in software development, with the authors using as an example the transfer of a video encoding implementation from the *x264* utility to the *VLC* media player to emphasize the transport of functionalities between different programs.

Similarly, Shariffdeen et al. [26] use an equivalent reasoning but focus on transfering patches from a *donor* to a *host*. The authors highlight the technique's usefulness for scenarios in which differing implementations may benefit from patch adaptation.

Transplantation considers a *host* benefits from having a *donor's* feature transferred to it. Likewise, the training process in code generation models like CodeGPT enhances output quality, showing the utility of learning from other programs. Thus, we consider other works' [2, 26] analogous methods as a validation of our solution.

APR techniques based on neural networks are a clear advance in software reliability. However, some of these [5, 8, 17, 19] focus on partial code snippets and do not acknowledge the entire source code, therefore missing the entire perspective and making learning the code syntax restrictive. Jiang et al. [13] point these limitations out and build a pre-trained model from a large code repository before perfoming any APR task. By using CodeGPT, our work shares the same logic as we leverage the capabilities of a pre-trained model that first understands the language it is trained on without influence from a specific task beforehand.

Ribeiro et al. [25] perform fault localization by identifying the semantics behind faults. This is done by translating the AST difference between two program versions — before and after the bug — into mutation operators. The authors are able to infer mutations 78% of times. They demonstrate how real-world programs can be

automatically repaired by applying mutation operators that revert the faulty modifications at the inferred places while leaving new but unrelated code unchanged. Likewise, we compute column numbers and consider them the most appropriate spots to generate new code and integrate it.

Pre-trained models like CodeBERT have been fine-tuned on the *ManySStuBs4J* dataset in order to automatically repair programs and shown to be able to produce patches of variable length and complexity while reporting accuracies between 19% and 72% [20].

12 CONCLUSION

This work presents an automated repair technique that, given a buggy file and line number, produces candidate patch lines in an attempt to fix the program. We devised a truncation algorithm that computes column numbers for which we use CodeGPT to perform code completion on. After that, we explained our implementation to limit the generated code sequences and how we fit the resulting string based on the language's syntax. Our approach was validated by analyzing the *ManySStuBs4J* dataset. The results show that 1739 programs were fixed out of 6415, which reflects a 27% repair rate and corroborates our work's soundness. As future work, we would like to minimize the number of patches created by taking into account structural aspects of the program to better restrict the generated code sequences. To do this, we want to examine the AST node types of the generated code to appropriately fit the expressions inside the original program, this way filtering out syntactically incorrect patches. Although out of this work's focus, we wish to explore how different sampling techniques impact code generation, like *top-k* and *nucleus* sampling. In some instances, the distribution of probabilities in decoded text has been shown to be very different from human-written text [12]. As such, we would also like to investigate if these findings generalize to programming languages.

Replication Package

All the necessary resources to replicate this study are publicly available:

- **Code Truncater:** *https://github.com/FranciscoRibeiro/code-truncater*
- **Experiments:** *https://gitlab.com/FranciscoRibeiro/manysstubs4j-experiments*

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. Francisco Ribeiro would like to acknowledge a PhD scholarship with reference SFRH/BD/144938/2019. Rui Abreu would like to acknowledge FCT through reference UIDB/50021/2020, the SecurityAware Project (ref. CMU/TIC/0064/2019) - also funded by the Carnegie Mellon Program, and the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017).

REFERENCES

[1] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *Applied Soft Computing* 11, 4 (2011), 3494–3514. https://doi.org/10.1016/j.asoc.2011.01.023
[2] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated Software Transplantation. In *ISSTA 2015*.
[3] Marcel Bruch, Monperrus Martin, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *ESEC/FSE '09*.
[4] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *ASE 2012*.
[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179
[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41.
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. https://doi.org/10.18653/v1/N19-1423
[8] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 275–286.
[9] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *AST '16*.
[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162
[12] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751 [cs.CL]
[13] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *ICSE '21*.
[14] J.A. Jones, M.J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE 2002*. https://doi.org/10.1145/581396.581397
[15] Rafael-Michael Karampatsis and Charles Sutton. 2020. *How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset*. Association for Computing Machinery, New York, NY, USA, 573–577. https://doi.org/10.1145/3379597.3387491
[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 38, 1 (jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104
[17] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *ICSE '20*. https://doi.org/10.1145/3377811.3380345
[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).
[19] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *ISSTA 2020*. https://doi.org/10.1145/3395363.3397369
[20] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. arXiv:2103.11626 [cs.SE]
[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *ICSE '13*. https://doi.org/10.1109/ICSE.2013.6606623
[22] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *ICSE '17*. https://doi.org/10.1109/ICSE.2017.62
[23] Alexandre Perez and Rui Abreu. 2018. Leveraging Qualitative Reasoning to Improve SFL. In *IJCAI '18* (Stockholm, Sweden).
[24] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.
[25] Francisco Ribeiro, Rui Abreu, and João Saraiva. 2021. On Understanding Contextual Changes of Failures. In *QRS '21*. https://doi.org/10.1109/QRS54544.2021.00112
[26] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 6 (dec 2021), 36 pages. https://doi.org/10.1145/3412376
[27] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation Using Transformer. In *ESEC/FSE '20*.
[28] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]
[29] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368
[30] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811