Multilingual Code Co-Evolution Using Large Language Models

Jiyang Zhang
UT Austin
USA
jiyang.zhang@utexas.edu

Pengyu Nie
UT Austin
USA
pynie@utexas.edu

Junyi Jessy Li
UT Austin
USA
jessy@austin.utexas.edu

Milos Gligoric
UT Austin
USA
gligoric@utexas.edu

ABSTRACT

Many software projects implement APIs and algorithms in multiple programming languages. Maintaining such projects is tiresome, as developers have to ensure that any change (e.g., a bug fix or a new feature) is being propagated, timely and without errors, to implementations in other programming languages. In the world of ever-changing software, using rule-based translation tools (i.e., transpilers) or machine learning models for translating code from one language to another provides limited value. Translating each time the entire codebase from one language to another is not the way developers work. In this paper, we target a novel task: translating code changes from one programming language to another using large language models (LLMs). We design and implement the first LLM, dubbed CODEDITOR, to tackle this task. CODEDITOR explicitly models code changes as edit sequences and learns to correlate changes across programming languages. To evaluate CODEDITOR, we collect a corpus of 6,613 aligned code changes from 8 pairs of open-source software projects implementing similar functionalities in two programming languages (Java and C#). Results show that CODEDITOR outperforms the state-of-the-art approaches by a large margin on all commonly used automatic metrics. Our work also reveals that CODEDITOR is complementary to the existing generation-based models, and their combination ensures even greater performance.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Software and its engineering** → **Software evolution**.

KEYWORDS

Language models, code translation, software evolution

ACM Reference Format:
Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-Evolution Using Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616350

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616350

1 INTRODUCTION

To ensure flexibility and a wide adoption of their software, companies provide application programming interfaces (APIs) for their services in several programming languages. Services, such as Google Cloud [20] and MongoDB [23], offer APIs written in most popular programming languages, including C++, C#, Java, and Python. Furthermore, popular software packages, like Antlr [43] and Lucene [47], have options to target different programming languages for the purpose of being used across various platforms easily.

Maintaining software that offers the same functionality in multiple programming languages is challenging. Any code change, due to a feature request or a bug fix, has to be propagated timely to all programming languages. At present, developers have to manually *co-evolve* code. This requires developers to manually find the correspondence between code snippets and apply necessary *edits*.

There has been work that could, in theory, help with translation. Rule-based migration tools [3, 18, 50] have been designed to translate between high-level programming languages (e.g., Java and C#). However, rule-based systems require developers who have expertise with both programming languages to manually write rules to specify the translation mappings. And the rules need to be updated with the evolution of programming languages themselves; they quickly become outdated [3, 9]. Recent work on automatic code translation [27, 33, 46, 49, 60] aim to directly translate between a source and a target programming language with the help of LLMs, which are pretrained on multiple programming languages. While these techniques could be used to produce code snippets that look correct, they make irrelevant changes that deviate substantially from the newly introduced features in the source programming language, or they fail to precisely infer the project-specific data types and class names.

Figure 1 illustrates the limitation of existing models. Developers changed `PdfException` to `LayoutExceptionMessageConstant` in method `docWithInvalidMapping02` in the Java project `itext/itext7`. In a later commit in the corresponding C# project `itext/itext7-dotnet`, developers revised method `DocWithInvalidMapping02` with exactly the same edits while keeping other parts of the method unchanged. We provide the Java code change, the prediction of an existing large language model, CodeT5 [55], fine-tuned for code translation, and the correct C# code change in Figure 1. The added lines of code are highlighted in green and the removed ones are highlighted in red. Although the existing model is able to correctly translate the updated exception type from Java to C#, it misses the class name for the field `HtmlRoles` and incorrectly infers the function call `Assert.Catch` as it does *not* use the prior version of C# code for reference.

To build more robust and accurate techniques that help software developers co-evolve projects implemented in different languages,

Figure 1: Example of using LLMs to help developers co-evolve code in two programming languages. The top box shows developer-made changes in a Java project `itext/itext7`, which needs to be propagated to the corresponding C# project `itext/itext7-dotnet`. The middle box shows the prediction by an existing generation-based large language model, which incorrectly changes irrelevant parts of the code. The bottom box shows the correct prediction by our model, CODEDITOR.

we explicitly model the *changes* that need to be made. We formulate a novel task: automatically *updating* code snippets in a target programming language, based on the *changes* made in the source programming language.

Most of the existing models implicitly tackle the code evolution tasks by generating tokens one by one in accordance with the underlying learned probability instead of focusing on how the code should be *modified* or retained. Prior work [10, 11, 15, 41, 51, 57, 59] have shown that standard generation-based models underperform models that explicitly model the edits on software-editing tasks.

To model code evolution across programming languages, we design an LLM, dubbed CODEDITOR, which learns to align the edits across programming languages and explicitly performs edits on the old version of the code in a target programming language. Following prior work [15, 41, 48, 59], we enable the model to reason about necessary edits and learn to apply them by directly generating an edit sequence.

For training and evaluation, we collect the first dataset with aligned Java and C# code changes on the methods with similar functionality and implementations. Specifically, we extract 6,613 pairs of code changes from 8 open-source Java projects and the corresponding C# projects on GitHub by mining the commit histories. This is the first dataset containing parallel code changes of two programming languages. We conduct the evaluation in two directions, updating C# method based on the Java changes (source language is Java and target language is C#) and updating Java method based on the C# changes (source language is C# and target language is Java).

Our results show that CODEDITOR outperforms all existing models across all the chosen automatic metrics, including the large pretrained generative models Codex [12] under few-shot setting and ChatGPT [40] under zero-shot setting. CODEDITOR achieves 96 (out of 100) CodeBLEU score on the task of updating C# code based on Java changes, which is more than 25% higher than the large pretrained generation-based model fine-tuned on this task.

Further, we find that CODEDITOR and generation-based models are complementary to each other as CODEDITOR is better at updating longer code snippets while generation model is better at handling the shorter ones. Thus, we combine the two models by choosing either model's prediction based on the size of the input code. Our results show that the combination can further improve our CODEDITOR model's exact-match accuracy by 6%.

The main contributions of this paper include:

- **Task**. We formulate a novel task of automatically updating code written in one programming language based on the changes in the corresponding code in another programming language.
- **Model**. We design and implement CODEDITOR, the first LLM for this task which learns to align the edits across programming languages and explicitly performs edits on the old version of the code in target programming language.
- **Dataset**. We create the first dataset with aligned code changes for two programming languages (Java and C#) from 8 open-source project pairs.
- **Results**. We show that CODEDITOR significantly outperforms the existing LLMs fine-tuned for code translation on exact-match accuracy by 77%. We also show that CODEDITOR is complementary to generation-based LLMs and the combination can further improve CODEDITOR's exact-match accuracy by 6%.

CODEDITOR and our corpus are publicly available on GitHub: https://github.com/EngineeringSoftware/codeditor.

2 TASK

At a high level, we work on a system that is triggered when a software developer, who maintains projects written in multiple programming languages, makes changes to one method in one of the languages, i.e., the "source" language. The system would automatically suggest updates to the methods with identical functionality in other language(s), i.e., the "target" language(s). To scope our work in this paper, we focus on Java as the source language, and C# as the target language. We leave evaluation that targets other programming languages as future work.

Table 1: The mappings between concise edit sequence and unambiguous edit sequence.

Edit	Concise	Unambiguous
Insertion	<Insert>	<ReplaceKeepBefore> <ReplaceKeepAfter>
Deletion	<Delete>	<Delete> <ReplaceKeepBefore> <ReplaceKeepAfter>
Replacement	<Replace>	<Replace> <ReplaceKeepBefore> <ReplaceKeepAfter>

In Figure 1, consider a method $M_{S;old}$ (`docWithInvalidMapping02`) written in the source language S and a method $M_{T;old}$ (`DocWithInvalidMapping02`) written in the target language T with identical functionality (hence similar implementation). Given the updated method $M_{S;new}$ in S, we define the task to generate the new method $M_{T;new}$ in T leveraging context provided by the code changes E_S, such that its functionality is consistent with $M_{S;new}$. Namely, we model the conditional probability distribution

$$P(M_{T;new}|M_{T;old},\ M_{S;new}, E_S)$$

and generate $M_{T;new}$ by sampling from the distribution.

3 MODEL

We present the overview of the proposed CODEDITOR model in Figure 2. CODEDITOR is built upon the encoder-decoder framework which consists of a transformer-based encoder and a transformer-based decoder [53]. Many conditional generation tasks, including code summarization and translation, are being addressed with encoder-decoder models [1, 21, 37, 54, 55].

We initialize CODEDITOR's parameters with the pretrained language model CoditT5 [59]. CoditT5 has shown promising results on various software-related editing tasks *in a single programming language*, but nonetheless would provide us with a "warm-start" that carries the necessary inductive biases towards modeling edits. To adapt to the multilingual co-editing task, we then fine-tune the CODEDITOR model exploring two key components: (i) the context fed into the model; (ii) the output format of the model.

To encourage our CODEDITOR model to leverage the (synchronous) code change histories of multiple programming languages in its training data, we provide the model with context from three sources as shown in Figure 2: (i) code changes on source programming language (E_S); (ii) old version of the code written in target programming language ($M_{T;old}$); (iii) new version of the code written in source programming language ($M_{S;new}$).

We explore two formats to represent the generated code changes: (i) the code edits in the target programming language (E_T); (ii) a meta edit sequence that translates the code edits from the source programming language to the target programming language, followed by the code edits in the target programming language (this is similar to the output format of CoditT5). In both cases, we then apply the generated code edits in the target programming language (E_T) to the old version of the code ($M_{T;old}$) to obtain the new version of the code ($M_{T;new}$).

3.1 Edit Representations

3.1.1 Concise Edit Sequence. We first represent edits using a sequence of edits identical to that used in CoditT5 [59], which we call concise edit sequence. Each edit is represented as:

```
<Operation> [token span] <OperationEnd>
```

Here, `<Operation>` is either `Insert`, `Delete` or `Replace`. Note that the `Replace` is represented in a slightly different structure since we must specify both the old contents to be replaced and the new contents to replace with:

```
<ReplaceOld> [old contents] <ReplaceNew>
[new contents] <ReplaceEnd>
```

For example, in Figure 1, the code change on the old Java method can be represented by "`<ReplaceOld> PdfException <ReplaceNew> LayoutExceptionMessageConstant <ReplaceEnd>`".

We use `difflib` [17] to compute the set of minimal edit sequence from the old and new versions of code.

3.1.2 Unambiguous Edit Sequence. One drawback of CoditT5 [59]'s representation specified above is that the concise edit sequence can be ambiguous due to the absence of positional information. For example, the Java code change in Figure 1 can be represented using `Replace` as: "`<ReplaceOld> PdfException <ReplaceNew> LayoutExceptionMessageConstant <ReplaceEnd>`". Without further specification, the edit does not contain any clues regarding which `PdfException` should be replaced as there are two occurrences of `PdfException` in the old code sequence. For similar reasons, `Insert` is always ambiguous because of not indicating where to add the new contents and `Delete` is ambiguous in cases where multiple occurrences of token spans can be removed.

To eliminate the potential ambiguity in the concise edit sequence, we design the format of unambiguous edit sequence by adjusting the condensed edit sequence proposed by Panthaplackel et al. [41], which uses anchor tokens to specify the location to perform edits.

Insertion. We do not use `Insert` since it will always introduce ambiguity without location information. To represent insertion, we first find unique anchor tokens that are the shortest span of tokens that is either before or after the edit location and is unique in the input sequence. Then we use `ReplaceKeepBefore` or `ReplaceKeepAfter`, which represents replacing the anchor tokens with the inserted contents and the anchor tokens. For example, in Figure 1, suppose the Java code change entails adding a blank return statement after the `assertEquals` statement on line 7. The token span "`getMessage());`" will serve as the minimal span of anchor tokens because it is unique among the old Java code sequence, and it occurs right before the edit to be performed. We disambiguate the edit sequence:

```
<Insert> return; <InsertEnd>
```

with the unambiguous edit sequence:

```
<ReplaceOldKeepBefore> getMessage());
<ReplaceNewKeepBefore> getMessage()); return;
<ReplaceEnd>
```

This edit sequence indicates that "`getMessage());`" should be replaced with "`getMessage()); return;`". We introduce `ReplaceKeepBefore` where the tokens that follows the `<ReplaceOldKeepBefore>`

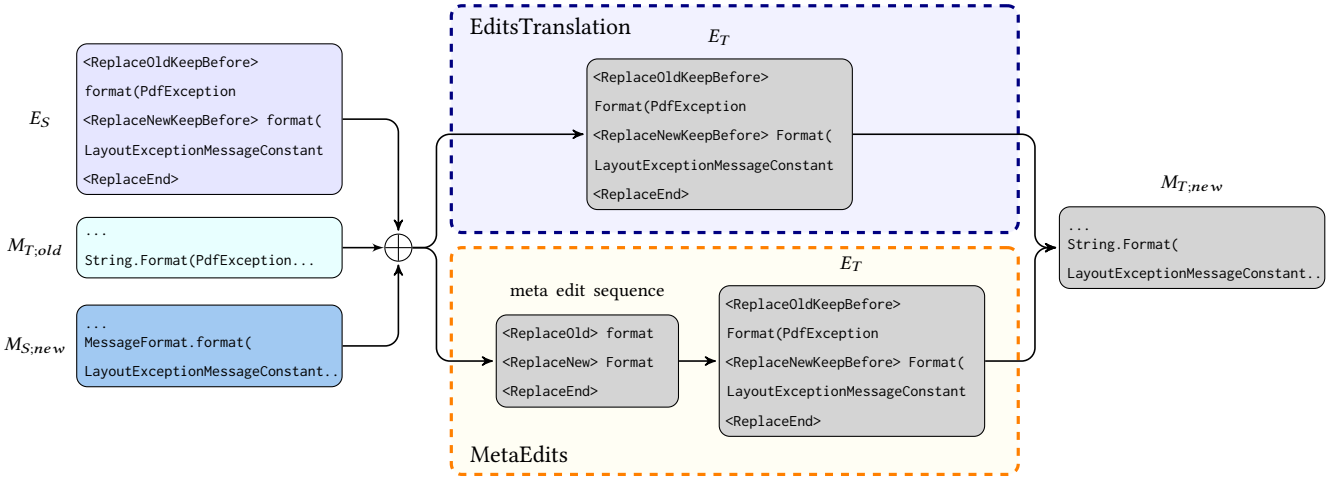

Figure 2: Workflow of CODEDITOR for multilingual co-editing. CODEDITOR leverages the context of code change histories of multiple programming languages from three sources: code changes on the source programming language (E_S), the old version of code in the target programming language ($M_{T;old}$), and the new version of code in the source programming language ($M_{S;new}$). CODEDITOR has two variants that both generate the code changes in the target programming language (E_T) but in different formats: EditsTranslation directly generates the code changes; MetaEdits generates the meta edit plan which edits E_S to E_T, followed by the code changes. Finally, we apply the code changes (E_T) on the old version of code ($M_{T;old}$) to obtain the new version of code ($M_{T;new}$) in the target programming language.

should be removed and the tokens following `<ReplaceNewKeepBefore>` should be inserted. Different from `Replace`, there is some overlap between the tokens to be removed and tokens to be inserted. If anchor tokens do not exist before the edit location, we use `ReplaceKeepAfter` with the tokens after the edit location instead.

Replacement. If the span of tokens to be replaced is unique in the old sequence, regular `Replace` sequence is sufficient and deterministic; in that case we will keep using it. Otherwise, it is unclear which occurrence of token span should be replaced. As an example, in Figure 1, the Java code change is changing from `PdfException` to `LayoutExceptionMessageConstant` in the `assertEquals` statement on line 6. The replacement in the concise edit sequence is ambiguous because there are two usages of `PdfException` (on lines 5 and 6) in the old Java code sequence after tokenization. To address this, similar to the insertion case, we search for the minimal anchor tokens before or after the edit location that can form a unique span in the old sequence. For example, the concise edit sequence:

```
<ReplaceOld> PdfException <ReplaceNew>
LayoutExceptionMessageConstant <ReplaceEnd>
```

can be disambiguate into the following unambiguous edit sequence:

```
<ReplaceOldKeepBefore> format(PdfException
<ReplaceNewKeepBefore> format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

Deletion. Similar to replacement, if the span of tokens to be deleted is unique across the old sequence, we will keep using `Delete` because it is unambiguous. Otherwise, it will be transformed to `ReplaceKeepBefore` or `ReplaceKeepAfter`. For example, suppose the token "`PdfException.`" should be removed from the old Java method on line 6 in Figure 1. The concise edit sequence:

```
<Delete> PdfException. <DeleteEnd>
```

will be transformed to:

```
<ReplaceOldKeepBefore> format(PdfException.
<ReplaceNewKeepBefore> format( <ReplaceEnd>
```

This edit sequence indicates that "`format(PdfException.`" should be replaced with "`format(`", unambiguously implying the deletion of "`PdfException.`".

To summarize, the unambiguous edit sequence contains 4 types of edits: `<Replace>`, `<Delete>`, `<ReplaceKeepBefore>` and `<ReplaceKeepAfter>`. The mappings between concise edit sequence and unambiguous edit sequence are summarized in Table 1. Given the unambiguous edit sequence, we can apply it to the old input sequence to derive the new edited sequence deterministically.

3.2 Model Input

We aim to build performant machine learning models for the multilingual co-editing task by providing the model with code evolution information, namely the revisions of code of both source and target programming languages. Instead of directly translating the entire code snippet between programming languages, CODEDITOR translates the code *changes* between programming languages.

3.2.1 Source Code Edits. To encourage the model to learn the alignment between developer-made changes across programming languages, we provide CODEDITOR with code changes in the source programming language. To maintain both precision and conciseness of the edits, we adopt the unambiguous edit sequence (Section 3.1.2) to represent the code changes. As shown in Figure 2, the Java code changes (E_S) of replacing the `PdfException` with `LayoutExceptionMessageConstant` is structured in the form of

```
<ReplaceOldKeepBefore> format(PdfException
<ReplaceNewKeepBefore> format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

3.2.2 History-Related Context. In addition to the learned representation of code changes in source programming language (E_S), we provide CODEDITOR with the old code in target programming language ($M_{T;old}$) to better help the model to infer the correlated code changes in the target programming language. The intuition is that the model will reason about how to transfer and tune the edits in source programming language grounding the specific implementation of the method in target programming language.

Furthermore, we append the new code in source programming language ($M_{S;new}$) as one of the contexts. We believe this will give the model more context to understand the edits in source programming language and promote the consistency of the updated methods in two programming languages.

To sum up, we combine history-related context from three sources: code changes in the source programming language (E_S), old code in the target programming language ($M_{T;old}$), and new code in the source programming language ($M_{S;new}$). We concatenate them into a sequence separated by a special *SEP* token as the model input.

3.3 Model Output

We propose two formats as the model's target output which lead to two modes of CODEDITOR: *EditsTranslation* and *MetaEdits*. Both modes use the same input and both modes' target outputs entail a sequence of edits on the target programming language.

EditsTranslation. The output of EditsTranslation mode is the unambiguous edit sequence in target programming language which suggests how the code in target programming language should be changed. Note that the model-generated unambiguous edit sequence can be parsed and applied to old version of code deterministically. EditsTranslation essentially learns to translate the code edits from the source programming language (E_S) to the target programming language (E_T) grounding the code history context. EditsTranslation mode's target output for the C# example in Figure 1 is:

```
<ReplaceOldKeepBefore> Format(PdfException
<ReplaceNewKeepBefore> Format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

MetaEdits. In this mode, we adopt the output format of CoditT5 [59] for multilingual co-editing since our model is built upon CoditT5, and it had showed promising performance on software editing tasks. CoditT5 is pretrained to generate the following output format: "[Edit Plan] <SEP> [Target Sequence]". The edit plan is a concise edit sequence that represents the steps to edit the input sequence; the target sequence is the edited sequence after applying the proceeding edit plan. We tailored this format to the multilingual co-editing task; the edit plan represents the edits between the code edits on source programming language (E_S) and target programming language (E_T) which we call the *meta edit sequence*. And the final target sequence should be the unambiguous edit sequence on the target programming language (E_T). For the example in Figure 1, the expected meta edit sequence that converts Java edit to C# edit is the following:

Table 2: Open-source projects used in our dataset and number of examples from each project.

Java Project	C# Project	Count
antlr/antlr4	tunnelvisionlabs/antlr4cs	12
apache/lucene	apache/lucenenet	40
apache/poi	nissl-lab/npoi	5
eclipse/jgit	mono/ngit	808
formicary/fpml-toolkit-java	formicary/fpml-toolkit-csharp	20
itext/itext7	itext/itext7-dotnet	5,121
quartz-scheduler/quartz	quartznet/quartznet	17
terabyte/jgit	mono/ngit	590
SUM		6,613

Table 3: Statistics of our dataset. Number of examples of training, validation and test data; average number of tokens in the old version of method and new version of method; average number of edits for the code change; average number of added and deleted tokens.

		Train	Val	Test
Count		4,391	623	1,599
Java	Avg. len(M_{old})	193.05	192.88	159.06
	Avg. len(M_{new})	195.99	192.36	159.37
	Avg. # edits	2.71	2.68	2.43
	Avg. # add. tks	19.57	16.64	10.90
	Avg. # del. tks	16.62	17.16	10.59
C#	Avg. len(M_{old})	200.37	199.71	168.60
	Avg. len(M_{new})	203.49	199.47	169.22
	Avg. # edits	2.73	2.75	2.47
	Avg. # add. tks	20.30	17.69	11.86
	Avg. # del. tks	17.18	17.92	11.25

```
<ReplaceOld> format <ReplaceNew> Format <ReplaceEnd>
```

The target sequence after applying the meta edit sequence is:

```
<ReplaceOldKeepBefore> Format(PdfException
<ReplaceNewKeepBefore> Format(
LayoutExceptionMessageConstant <ReplaceEnd>
```

Note that during inference, we only use the target unambiguous edit sequence to get the updated code in target programming language as MetaEdits mode's prediction.

4 DATASET

This is the first work to consider the history of software projects in a multilingual task; hence, we also created a new dataset that includes aligned code changes between programming languages. As the first step, we build the dataset by mining histories of the open-source Java and C# projects. We first collect the changed methods from the commits of the Java and C# projects. We then design heuristics to pair (i.e., align) those changes on methods with similar implementations and functionalities. We consider two directions on our dataset: J2CS (updating C# method based on Java changes) and CS2J (updating Java method based on C# changes). In this section, we describe the approach we use to collect the data (Section 4.1), split and preprocess data (Section 4.2), and finally present the statistics of our dataset (Section 4.3).

4.1 Data Collection

To build the dataset, we extract aligned Java and C# code changes at the method level as tuples (Java old method; Java new method, C# old method; C# new method). The code changes are mined from the git commits. We consider 8 open-source projects as listed in Table 2 which have both Java and C# implementations and are used in prior work [13, 33, 36]. All the projects were first developed in Java and then ported to C#.

To collect the paired changes, we first assign a unique identifier to each method in the projects (for both Java and C# projects) based on the method signature, class name and path to the file where the method is defined. Similar to the strategy used by Lu et al. [33], we then pair the Java methods and C# methods according to the similarity of their unique identifiers. This strategy is effective because the ported C# project has very similar structure and naming rules for classes and methods to the corresponding Java project.

We use the following rules to extract the aligned code changes:

(1) For each Java method change, we extract the code changes in the paired C# method that happen no later than 90 days of the Java change as the *possible matched code change*. We use the commit date as the time of the change.

(2) To filter unrelated code changes, we compute the Jaccard similarity [24] between C# and Java added and deleted lines. We further refine the filtering by sub-tokenizing these lines based on camelCase conventions (e.g., `lastModified` to `last modified`) and compute Jaccard similarity only for the added and deleted tokens. We only keep possible matched code changes that have the token-level Jaccard similarity higher than 0.4 and the line-level Jaccard similarity higher than 0.5.

(3) For each Java code change and C# code change, we only select the most similar corresponding code change if there are multiple possible matched code changes.

4.2 Data Preprocessing and Splitting

For both Java and C# methods, we remove the inline natural language comments and tokenize the method into tokens using the language-specific lexers generated by Antlr [43].

We envision the following use case for the machine learning model: whenever a developer makes a change in the project written in the source programming language, the developer will use the model trained on the existing historical aligned code changes to migrate that change to projects written in other target programming languages. To evaluate the models under this use case, following the recommendations from prior work [38], we split the dataset into training, validation and test sets using the *time-segmented* approach. Namely, the changes in the training set took place before the changes in the validation set, which in turn took place before the changes in the test set. More specific, for each Java and C# code change pair, we first collect the time of the C# commit and then sort the code change pairs in chronological order. We then select the oldest 70% of the code change pairs from each project as training data, next oldest 10% as validation data, the remaining as test data.

To more rigorously assess the generalization capabilities of the models, we also evaluated them when splitting the dataset using the *cross-project* approach [38], which is frequently used in prior work on machine learning models for code. Specifically, the aligned code changes in the training set are from different projects compared to those in the validation and test sets.

4.3 Statistics

The statistics of the collected dataset are shown in Table 3. We present the number of examples in the training, validation, and test dataset using time-segmented split approach. We show the average number of tokens in the old methods (Avg. len(M_{old})) and new methods (Avg. len(M_{new})) after tokenization by the lexers. To measure the size of the code changes, we calculate the average number of added tokens (Avg. # add. tks) and deleted tokens (Avg. # del. tks) in the changed Java and C# methods as well as the average number of edits (Avg. # edits) needed for those changes. For computing these edit-related statistics, we represent the code changes using concise edit sequences (Section 3.1.1).

For both Java and C# code changes, the difference between average number of added tokens and deleted tokens is usually small, fewer than 4 tokens. Similarly, we find that the average number of edits needed is fewer than 3 and the edits happened in the newer commits are generally smaller than prior ones. This is expected as the software projects are becoming more stable as they evolve, and thus there will be smaller code changes to be made. For evaluation, we run all the models and baselines on this dataset in two directions: (1) updating C# method based on Java changes, and (2) updating Java method based on C# changes. We denote the former one as J2CS and the latter one as CS2J.

5 EXPERIMENTS

In this section, we describe the baselines we compare to with our CodeEditor model (Section 5.1), the evaluation metrics (Section 5.2) and the detailed experiment setup (Section 5.3).

5.1 Baselines

We evaluate our approach against rule-based models, pretrained encoder-decoder models, the state-of-the-art code-editing model (which targets a single programming language), and large generative models pretrained on billions of lines of code.

Copy. This is a rule-based model which copies the old code in target programming language ($M_{T;old}$) as the prediction. This is not a trivial baseline since there are quite a few examples in the dataset that entail small edits between two versions. We include this to benchmark the models that actually update the code.

CopyEdits. Based on our observations, there are cases where the code change in source programming language (E_S) is exactly the same as the change in target programming language (E_T) , such as changing the variable name or updating the log message. This rule-based model copies the E_S and directly applies it to the old code in target programming language ($M_{T;old}$).

CodeT5-Translation. We consider a state-of-the-art model that does not have access to the code change history. Namely, a code translation model that translates code between the programming languages (from $M_{S;new}$ to $M_{T;new}$). We use CodeT5 [55], an LLM pretrained on large amount of developer-written code from GitHub, which we fine-tune on our constructed dataset.

CodeT5-Update. This model has the same architecture as CodeT5-Translation except that we supply it with code change history. The model input is the same as for our CODEDITOR models, i.e., with extra context of the old code in target programming language ($M_{T;old}$) and the code change in source programming language (E_S). Different from CODEDITOR model, it is trained to directly generate the new code in target programming language ($M_{T;new}$).

CoditT5. This is the state-of-the-art model for software editing tasks [59]. It has the same model architecture and input as CODEDITOR, while the output consists of the edit plan to represent the edits on the target programming language and the target sequence which represents the updated code ($M_{T;new}$) after applying the edit plan.

Codex-few-shot [12]. Large pretrained generative models such as GPT-3 [8] have shown impressive results under the context of *few-shot learning* or even *zero-shot learning* on various generation tasks. They are able to generalize to new tasks they have not seen during pretraining with only a few or even no labeled examples. To compare the fine-tuned CODEDITOR model with generative models, we include Codex, a large generative model built on GPT-3 and is further pretrained on billions of GitHub data. Following prior work [2, 26], for each example in test data, we randomly select several labeled examples in the training data as the context. Note that the labeled examples are selected from the same project as the test data. For J2CS dataset, each labeled example is formed as: "Java: $M_{S;old}$ => $M_{S;new}$ C#: $M_{T;old}$ => $M_{T;new}$" to inform the model the aligned updates between two programming languages. The designed prompt for inference is "Java: $M_{S;old}$ => $M_{S;new}$ C#: $M_{T;old}$ => ". The model output is the prediction for the new code in target programming language ($M_{T;new}$). To conform to the required input length limit, we include 2 labeled examples in the prompt.

ChatGPT-zero-shot [40]. ChatGPT is an upgraded version of GPT-3 model and is further fine-tuned for conversation generation following human instructions with the help of supervised and reinforcement learning methods. It has showed strong performance on code completion benchmarks like HumanEval and MBPP [4, 12, 39]. For each example in test data, we provide instructions including both the previous and the updated versions of the code written in the source programming language, subsequently prompting ChatGPT to update the old code in the target programming language accordingly. For J2CS dataset, the prompt is formed as: "The developer updates the Java method from: $M_{S;old}$ to: $M_{S;new}$. Please update the C# method accordingly. This is the old C# method: $M_{T;old}$."

5.2 Evaluation Metrics

Following prior work [1, 41, 55, 59], we use metrics for evaluating the quality of code generation: BLEU [42], CodeBLEU [45], xMatch, and metrics for evaluating the quality of software editing: SARI [56] and GLEU [35]. Note that for all the metrics we report in this paper, they range from 0 to 100 and higher scores are better.

xMatch. When the generated code matches exactly with the expected code in target programming language, this metric is 100; otherwise, this metric is 0. This metric reflects the percentage of exact matches among the models' predictions on test data.

BLEU. It is a widely used metric originally proposed for evaluating the quality of machine translation. It measures the n-gram overlap between the generated sequence and the expected one. Concretely, we report the 1~4-grams overlap between the tokens in the predictions and tokens in the ground truth.

CodeBLEU. The metric is proposed for evaluating the quality of code generation. In addition to measuring the n-gram overlap, it considers the overlap of the Abstract Syntax Tree (AST) and data-flow graph between generated code and the expected code.

SARI. It measures quality of the systems that are designed to make edits. Specifically, it is computed as the average of the F1 score for kept and inserted spans of tokens, and the precision of deleted spans of tokens.

GLEU. It is a variant of BLEU. It was originally proposed for grammatical error correction and designed for rewarding the correct edits while penalizing the incorrect ones.

5.3 Experimental Setup

We run all experiments on machines with 4 NVidia 1080-TI GPUs, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz for training. We implement our models using PyTorch 1.9.0. All the hyper-parameters of the CodeT5 and CoditT5 baselines are set to the same values as in prior work [55, 59]. For CODEDITOR, CodeT5-Translation, CodeT5-Update, and CoditT5, we early stop the training when the BLEU score on the validation set does not improve for 5 epochs, and use beam search with beam size 20 during inference. For Codex and ChatGPT, we set temperature to 0.2 during inference.

Note that Codex and ChatGPT are closed-source and may be updated/deprecated over time. We used the code-davinci-002 version of Codex when performing experiments in the time-segmented split; however, OpenAI deprecated Codex in March 2023 before we could complete our experiments in the cross-project split, as such we did not include Codex in this part of results.

6 RESULTS

We organize our evaluation around three main research questions:

RQ1: What is the benefit of using code change history in multilingual co-editing?

RQ2: How does our edit-based model, CODEDITOR, compare to generation-based models for the multilingual co-editing?

RQ3: How can a generation-based model complement CODEDITOR model to further improve the performance?

6.1 Quantitative Analysis

In tables 4-7, we present results for baselines and our proposed CODEDITOR models on J2CS, CS2J for both time-segmented and cross-project splits. We conducted statistical significance testing through bootstrap tests [6] under confidence level 95%.

RQ1: Conntribution of code change histories. We divide models into two categories with respect to whether a model has access to the information on code change histories: Copy and CodeT5-Translation are history-agnostic models, and the remaining are history-aware models. Overall, the history-aware models outperform the history-agnostic ones. The rule-based model CopyEdits,

Table 4: Results on the J2CS dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	CodeBLEU	SARI	GLEU
Copy	0.00	83.11	90.42	30.68	74.58
CopyEdits	38.21^{β}	$90.29^{\alpha\chi}$	91.34	76.92	87.93
CodeT5-Translation	38.02^{β}	87.45	77.15	83.77	85.59
CodeT5-Update	60.41^{ϵ}	$90.00^{\chi\eta}$	76.63	80.11	88.72
CoditT5	60.29^{ϵ}	$89.84^{\alpha\eta}$	75.20	80.99	89.29
Codex-few-shot	48.84	80.71	59.63	72.80	79.74
ChatGPT-zero-shot	29.52	85.60	73.00	68.44	84.74
CodeEditor (MetaEdits)	63.48	94.55	94.78	85.63	93.20
CodeEditor (EditsTranslation)	67.23	95.44	96.02	$\mathbf{87.23}^{\delta}$	94.21
Hybrid	**71.79**	**96.12**	**96.09**	87.08^{δ}	**95.07**

Table 5: Results on the CS2J dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	CodeBLEU	SARI	GLEU
Copy	0.00	83.06	89.82	30.66	74.55
CopyEdits	38.15	89.36^{α}	90.31^{β}	75.86	87.02^{χ}
CodeT5-Translation	40.21	89.10^{α}	77.99^{β}	83.99	87.21^{χ}
CodeT5-Update	55.97	90.62	76.38^{γ}	79.65	89.72
CoditT5	60.98	90.88	75.87^{γ}	81.41	90.15
Codex-few-shot	55.53	82.54	60.35	76.23	82.13
ChatGPT-zero-shot	32.52	86.95	76.01	69.05	86.33
CodeEditor (MetaEdits)	$\mathbf{68.61}^{\epsilon\eta}$	93.98	94.43	85.74	92.61
CodeEditor (EditsTranslation)	$67.92^{\delta\epsilon}$	95.29	94.83	**86.24**	94.23
Hybrid	$67.67^{\delta\eta}$	**96.44**	**95.36**	84.46	**95.75**

Table 6: Results on the cross-project split using J2CS dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	CodeBLEU	SARI	GLEU
Copy	0.00	79.96^{α}	89.08	30.53	71.89
CopyEdits	14.19	87.95	89.73	67.58	85.55
CodeT5-Translation	10.64	77.34	64.35	71.73	74.16
CodeT5-Update	29.38	80.56^{α}	66.15	64.70	79.65
CoditT5	34.59	81.59	65.17	83.29	80.91
ChatGPT-zero-shot	39.58	86.97	74.90	70.15	86.14
CodeEditor (MetaEdits)	38.36	90.79^{β}	91.60	73.45	88.94^{χ}
CodeEditor (EditsTranslation)	41.91	90.86^{β}	91.35	74.59	88.94^{χ}
Hybrid	**43.35**	**92.51**	**91.70**	**89.13**	**91.18**

which directly applies the code change in source programming language (E_S) to the old code in target programming language without any adaptation, has comparable performance to the machine learning history-agnostic model CodeT5-Translation. This emphasizes the importance of contextual information provided by code change histories in multilingual co-editing. Interestingly, we find that Codex-few-shot, which is used under the few-shot learning setting without fine-tuning, performs better than fine-tuned CodeT5-Translation on xMatch, while worse than other history-aware fine-tuned machine learning models. This again underlines the value of code change histories and suggests that fine-tuning will give better performance by leveraging more code history contexts in the training data.

RQ2: CodeEditor vs. generation-based models. Among all the history-aware models, machine learning models, such as CodeT5-Update and CoditT5, achieve much higher performance than the

Table 7: Results on the cross-project split using CS2J dataset. The results with the same suffixes (e.g., β) are NOT statistically significantly different.

Models	xMatch	BLEU-4	CodeBLEU	SARI	GLEU
Copy	0.00	80.02	88.60	30.51	71.94
CopyEdits	13.86	86.99	88.70	66.50	84.14
CodeT5-Translation	6.21	76.84	62.27	67.37	69.81
CodeT5-Update	31.60	81.98	65.82	65.49	81.07
CoditT5	35.81	82.89	65.20	83.27	81.67
ChatGPT-zero-shot	39.80	89.35	76.69	72.36	88.62
CODEDITOR (MetaEdits)	41.91	91.54^{α}	91.21	73.46	89.36^{β}
CODEDITOR (EditsTranslation)	40.35	91.63^{α}	90.99	74.15	89.58^{β}
Hybrid	**46.34**	**93.17**	**91.32**	**89.62**	**91.24**

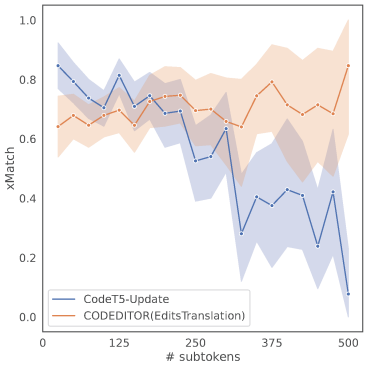

Figure 3: Average percentage of model's predictions that exactly match the ground truth on examples that have different number of subtokens. The bands represent the 95% confidence interval.

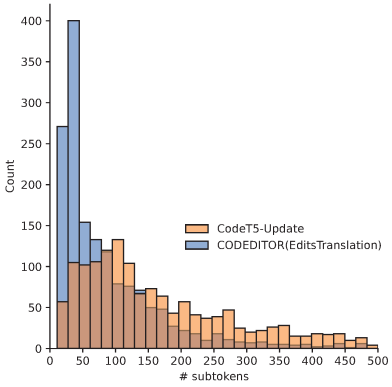

Figure 4: Distribution of number of sub tokens in models' target outputs.

rule-based CopyEdits, which demonstrates that the machine learning models effectively learns to reason about the correlated code changes and adjust them to the target programming language. We observe that CODEDITOR (in both EditsTranslation and MetaEdits modes), which is trained to first translate code changes on source programming language to target programming language and then apply the edits to the old code in target programming language, achieve even higher performance across all the metrics than the large pretrained generation-based model (CodeT5-Update) which directly generates the new code in target programming language from scratch. This highlights that the models that are trained to explicitly perform edits by predicting the edit sequence are better suited for editing tasks in the software domain than generation-based models.

To further investigate the advantages of CODEDITOR over the best generation-based model (CodeT5-Update), we break down the performance of EditsTranslation and CodeT5-Update on each example in the test data of J2CS. In Figure 3, we show the average percentage of CODEDITOR (EditsTranslation) and CodeT5-Update's predictions that exactly match the ground truth with respect to the number of sub-tokens in the input old code ($M_{T;old}$). Note that the code are subtokenized using the Roberta tokenizer [31], which is used by all machine learning models. We exclude the examples that have more than 500 sub-tokens from being shown in this figure as those outliers only account for less than 5% of the test data. We can see that the performance of CodeT5-Update drastically drops with the increase of number of sub-tokens in the code to be edited ($M_{T;old}$), but EditsTranslation's performance is rather stable. This illustrates another benefit of CODEDITOR in accurately handling longer input, because of focusing on transforming the edits instead of generating the entire new code like CodeT5-Update.

Meanwhile, most of the existing transformer-based models have a length limit for the input sequence because the naive self-attention has quadratic complexity with regard to the input length. In Figure 4, we present the distribution of the number of sub-tokens in the models' target outputs for CODEDITOR (EditsTranslation) and CodeT5-Update on the test data of J2CS. We only show the distribution of target outputs with fewer than 500 sub-tokens for the same reason described in the previous paragraph. Most of CODEDITOR's target outputs (the sequence of edit operations) are shorter than CodeT5-Update's output (new code in target programming language). This might explain why CODEDITOR achieves better performance than generation-based models on longer code as generating longer sequence are generally more challenging to machine learning models. Recent studies [5, 7, 14] have focused on exploring approaches to address the limitation of the model's input context

Figure 5: Qualitative analysis of all the models on one example in the test data of J2CS dataset.

window size. Future research should examine the performance difference between translating edit sequences and generating entirely new code using models capable of handling longer context.

RQ3: Combining generation-based model with CODEEDITOR. To exploit the superiority of generation-based model on short code snippets, we combine our strongest generation model—CodeT5-Update—with the strongest CODEEDITOR mode—EditsTranslation—based on the size of the code snippet. Specifically, we use CodeT5-Update if the code to be updated has fewer sub-tokens than a threshold and use CODEEDITOR (EditsTranslation) otherwise. To pick the threshold for combining two models, we performed a grid-search on the validation set and selected the one that gives optimal xMatch score. We refer to the combined model as the *Hybrid* model and provide its results on the bottom row of Table 4 to Table 7. By combining generation-based model with CODEEDITOR, we can achieve improved performance on most of the reported automatic metrics.

6.2 Qualitative Analysis

Figure 5 shows an example in J2CS dataset and the models' predictions. We show the code changes from Java project `itext/itext7` in the method (`parseBodyFragment`). The newly added code is highlighted in green and removed code is highlighted in red. We also present the old version of the corresponding C# method (`ParseBodyFragment`) from `itext/itext7-dotnet`, and the predicted code changes from three models: CODEEDITOR (EditsTranslation), CodeT5-Update, CodeT5-Translation. Note that CodeT5-Translation only has access to the new version of Java method.

Although CodeT5-Translation is able to correctly translate the code change in Java, it fails to infer the full name of the type `Node` and makes an irrelevant edit, because it does not have the context of the old version of C# code. CodeT5-Update correctly captures the Java change while making an extra irrelevant edit on the C# code. Our proposed model, CODEEDITOR (EditsTranslation) accurately identifies the position in the C# method to make edits and correctly adjusts the Java edits.

7 LIMITATIONS

Studied programming languages.. We study the translation of code changes between two programming languages. In this paper, we focus on open-source Java and C# projects due to the ease of locating corresponding changes using heuristics. Nevertheless, it is important to note that our approach can be applied to other programming language pairs as well, and we leave the investigation of such pairs for future research.

Correspondence between programming languages. Our model, CODEEDITOR, is intended for developers to migrate code changes from a project written in a source programming language to projects

written in target programming languages, leveraging known correspondences (e.g., methods with similar functionalities) between the source and target programming languages. In this work, we adopt a similar strategy used in [33] to match Java and C# methods. In practice, a code retrieval system can be used as a first step to identify the locations where the code changes should be propagated. We leave the combination of code retrieval tool and CODEDITOR as future work.

Empirical evaluation. This paper presents the empirical study results for internal metrics that are of interest to researchers. However, the external measurements of the impact on software engineering effort are not included in this study. These measurements could be addressed by conducting user studies.

8 RELATED WORK

In this section, we describe related work on the rule-based code translation tools, existing machine learning models designed for code translation, and the machine learning models that are proposed for accelerating software evolution.

Rule-based code translation. Researchers and practitioners have designed rule-based tools for translating the source code between programming languages. Such tools, usually called transpilers, were built for pairs like Java and C# [3], C and Rust [18], C and Go [16]. Nguyen et al. [36] proposed PBSMT, a phrase-based statistical machine translation models for source code translation. Gyori et al. [22] proposed LAMBDAFICATOR to translate imperative Java code to using the functional Stream APIs. Radoi et al. [44] presented the rule-based model to translate sequential Java code to MapReduce framework. Prior work [34] has shown that existing rule-based code refactoring tools can only deal with stylized code snippets over common code patterns.

Learning-based code translation. Researchers have proposed various machine learning models for the code translation task. Chen et al. [13] proposed a tree-to-tree neural network with a tree-RNN encoder and a tree-RNN decoder. Motivated by the success of large pretrained LLMs for many Natural Language Processing tasks, domain-specific models that are pretrained on source code and technical text have emerged. Researchers have applied them to the code translation task. Lu et al. [33] proposed CodeXGLUE, a benchmark including the code translation dataset consisting of Java and C# methods with equivalent functionality. They fine-tuned and evaluated CodeBERT on the translation dataset. Results showed that it produced the best results among all the existing baselines. LLMs that are built on the encoder-decoder paradigm and pretrained with general unsupervised denoising auto-encoding objectives showed promising results on wide range of code generation tasks including code translation. Such models include CodeT5 [55], PLBART [1], and UniXcoder [21]. For the comparison of CODEDITOR with state-of-the-art code translation models, we include two variants of the CodeT5-based translation models (with history context and without) in our evaluation.

Researchers designed LLMs which are pretrained with the objective tailored for code translation. Tipirneni et al. [49] introduced tasks on predicting AST paths and data flows during pretraining. Lachaux et al. [27] proposed TransCoder which is pretrained to do code translation with back-translation objective. To improve the quality of pretraining data, Roziere et al. [46] leveraged an automated unit-testing system to filter out invalid generated programs during back-translation. Zhu et al. [60] proposed MuST, which is a multilingual code snippet translation pretraining objective. None of the above work leverages the code change history, which is the main contribution of our paper. We leave improving CODEDITOR with pretraining objectives tailored for code translation as future work.

Software evolution and machine learning. New research initiatives have emerged around building and evaluating models that aid the process of software evolution. Prior work [19, 29, 30, 32, 41] proposed to update the comment given the changes in the associated method, e.g., Panthaplackel et al. [41] built a model that takes the code change as context to make edits on the outdated comment. Nie et al. [38] present different approaches to split dataset into training, validation and test sets and studied how different approaches affect the evaluation of machine learning models. Kamezawa et al. [25] presented a dataset, RNSum, which consists of release notes and the associated commit messages collected from GitHub repositories and designed models to generate release notes based on the commit messages. Zhang et al. [59] proposed a novel pretraining objective designed for software editing tasks and built CoditT5. CoditT5 was fine-tuned on three downstream tasks related to the software evolution. Li et al. [28], Tufano et al. [52], Zhang et al. [58] proposed models that targeted various tasks through the code review process. The models are trained on the historical data and evaluated on the new pull requests submitted for code review. Our CODEDITOR model incorporates the context from the code changes in source programming language and the old version of method in target programming languages to improve its performance on the multilingual co-editing task, which helps developers co-evolve the projects implemented in different programming languages.

9 CONCLUSION

In this paper, we formulated a new task: translating code changes across programming languages with the goal to synchronize projects that provide the same APIs or implementations in multiple programming languages. We proposed CODEDITOR, a model which uses code change history as contextual information and learns to make edits on the existing version of code written in the target programming language. We showed that our model outperforms existing code translation models and is better than the generation-based models even if they use historical context. CODEDITOR is a significant advancement in supporting developers with the maintenance of their projects that incrementally provide identical functionalities in multiple programming languages.

ACKNOWLEDGMENTS

We thank Nader Al Awar, Yu Liu, Sheena Panthaplackel, Aditya Thimmaiah, Zhiqiang Zang, and the anonymous reviewers for their comments and feedback. We acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, IIS-2145479, CCF-2217696 and CCF-2313027.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[2] Toufique Ahmed and Premkumar Devanbu. 2022. Few-Shot Training LLMs for Project-Specific Code-Summarization. In *Automated Software Engineering*. 1–5.

[3] Christian Mauceri Alexandre FAU. 2013. Java2csharp. http://sourceforge.net/projects/j2cstranslator/

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[5] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[6] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An Empirical Investigation of Statistical Significance in NLP. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. 995–1005.

[7] Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. 2023. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625* (2023).

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Nghi DQ Bui and Lingxiao Jiang. 2018. Hierarchical Learning of Cross-Language Mappings Through Distributed Vector Representations for Code. In *International Conference on Software Engineering, NIER*. 33–36.

[10] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code Editing with Tree-based Neural Models. *Transactions on Software Engineering* 4 (2020), 1385–1399.

[11] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *Automated Software Engineering*. 443–455.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-Tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems*, Vol. 31.

[14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[15] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *Automated Software Engineering*. 275–286.

[16] Elliot Chance et al. 2021. A tool for transpiling C to Go. https://github.com/elliotchance/c2go

[17] Python Software Foundation. 2023. difflib — Helpers for computing deltas. Retrieved February 2, 2023 from https://docs.python.org/3/library/difflib.html

[18] Galois and Immunant. 2023. C2Rust. https://github.com/immunant/c2rust

[19] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[20] Google. 2023. Google Cloud. https://cloud.google.com/

[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Annual Meeting of the Association for Computational Linguistics*. 7212–7225.

[22] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the Gap from Imperative to Functional Programming Through Refactoring. In *International Symposium on the Foundations of Software Engineering*. 543–553.

[23] MongoDB Inc. 2023. MongoDB. https://www.mongodb.com/

[24] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New phytologist* (1912), 37–50.

[25] Hisashi Kamezawa, Noriki Nishida, Nobuyuki Shimizu, Takashi Miyazaki, and Hideki Nakayama. 2022. RNSum: A Large-Scale Dataset for Automatic Release Note Generation via Commit Logs Summarization. In *Annual Meeting of the Association for Computational Linguistics*. 8718–8735.

[26] Junaed Younus Khan and Gias Uddin. 2022. Automatic Code Documentation Generation Using GPT-3. In *Automated Software Engineering*. 1–6.

[27] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems*. 20601–20611.

[28] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. 2022. Automating Code Review Activities by Large-Scale Pre-Training. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1035–1047.

[29] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F Bissyandé. 2021. Automated Comment Update: How Far are We?. In *International Conference on Program Comprehension*. 36–46.

[30] Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao. 2022. Predictive comment updating with heuristics and ast-path-based neural learning: A two-phase approach. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1640–1660.

[31] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A Robustly Optimized Bert Pretraining Approach. *arXiv preprint arXiv:1907.11692* (2019).

[32] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering* (2021).

[33] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).

[34] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Isil Dillig. 2022. Automated Transpilation of Imperative to Functional Code using Neural-Guided Program Synthesis. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–27.

[35] Courtney Napoles, Keisuke Sakaguchi, Matt Post, and Joel Tetreault. 2015. Ground Truth for Grammatical Error Correction Metrics. In *Annual Meeting of the Association for Computational Linguistics and International Joint Conference on Natural Language Processing*. 588–593.

[36] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-Conquer Approach for Multi-Phase Statistical Migration for Source Code. In *Automated Software Engineering*. 585–596.

[37] Pengyu Nie. 2023. *Machine Learning for Executable Code in Software Testing and Verification*. Ph. D. Dissertation. The University of Texas at Austin.

[38] Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2022. Impact of Evaluation Methodologies on Code Summarization. In *Annual Meeting of the Association for Computational Linguistics*. 4936–4960.

[39] OpenAI. 2023. GPT-4 Technical Report. arXiv:arXiv:2303.08774

[40] OpenAI. 2023. Introducing ChatGPT. https://openai.com/blog/chatgpt

[41] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Annual Meeting of the Association for Computational Linguistics*. 311–318.

[43] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A Predicated-LL (k) Parser Generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[44] Cosmin Radoi, Stephen J Fink, Rodric Rabbah, and Manu Sridharan. 2014. Translating Imperative Code to MapReduce. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 909–927.

[45] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[46] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging Automated Unit Tests for Unsupervised Code Translation. *arXiv preprint arXiv:2110.06773* (2021).

[47] Apache Software. 2022. Apache Lucene. Retrieved March 2, 2022 from https://lucene.apache.org/

[48] Felix Stahlberg and Shankar Kumar. 2020. Seq2Edits: Sequence Transduction Using Span-level Edit Operations. In *Empirical Methods in Natural Language Processing*. 5147–5159.

[49] Sindhu Tipirneni, Ming Zhu, and Chandan K Reddy. 2022. StructCoder: Structure-Aware Transformer for Code Generation. *arXiv preprint arXiv:2206.05239* (2022).

[50] Marco Trudel, Manuel Oriol, Carlo A Furia, and Martin Nordio. 2011. Automated Translation of Java Source Code to Eiffel. In *International Conference on Objects, Models, Components, Patterns*. 20–35.

[51] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *International Conference on Software Engineering*. 25–36.

[52] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *International Conference on Software Engineering*. 2291–2302.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[54] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).

[55] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Empirical Methods in Natural Language Processing*. 8696–8708.

[56] Wei Xu, Courtney Napoles, Ellie Pavlick, Quanze Chen, and Chris Callison-Burch. 2016. Optimizing Statistical Machine Translation for Text Simplification. *Transactions of the Association for Computational Linguistics* 4 (2016), 401–415.

[57] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning Structural Edits via Incremental Tree Transformations. In *International Conference on Learning Representations*.

[58] Jiyang Zhang, Chandra Maddila, Ram Bairi, Christian Bird, Ujjwal Raizada, Apoorva Agrawal, Yamini Jhawar, Kim Herzig, and Arie van Deursen. 2023. Using Large-scale Heterogeneous Graph Representation Learning for Code Review Recommendations at Microsoft. *International Conference on Software Engineering, SEIP* (2023).

[59] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. CoditT5: Pretraining for Source Code and Natural Language Editing. In *Automated Software Engineering*. 1–12.

[60] Ming Zhu, Karthik Suresh, and Chandan K Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *AAAI Conference on Artificial Intelligence*. 11783–11790.

Received 2023-02-02; accepted 2023-07-27