2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00165

Identify and Update Test Cases when Production Code Changes: A Transformer-based Approach

Xing Hu*, Zhuang Liu*, Xin Xia†§, Zhongxin Liu†, Tongtong Xu‡, Xiaohu Yang†
*School of Software Technology, Zhejiang University, Ningbo, China
†College of Computer Science and Technology, Zhejiang University, Hangzhou, China
‡Software Engineering Application Technology Lab, Huawei, Hangzhou, China
{xinghu,liuzhuang,liu_zx,yangxh}@zju.edu.cn, xin.xia@acm.org,xutongtong9@huawei.com

***Abstract*—Software testing is one of the most essential parts of the software lifecycle and requires a substantial amount of time and effort. During the software evolution, test cases should co-evolve with the production code. However, the co-evolution of test cases often fails due to tight project schedules and other reasons. Obsolete test cases improve the cost of software maintenance and may fail to reveal faults and even lead to future bugs. Therefore, it is essential to detect and update these obsolete test cases in time. In this paper, we propose a novel approach CEPROT (Co-Evolution of Production-Test Code) to identify outdated test cases and update them automatically according to changes in the production code. CEPROT consists of two stages, i.e., obsolete test identification and updating. Specifically, given a production code change and a corresponding test case, CEPROT first identifies whether the test case should be updated. If the test is identified as obsolete, CEPROT will update it to a new version of test case. To evaluate the effectiveness of the two stages, we construct two datasets. Our dataset focuses on method-level production code changes and updates on their obsolete test cases. The experimental results show that CEPROT can effectively identify obsolete test cases with precision and recall of 98.3% and 90.0%, respectively. In addition, test cases generated by CEPROT are identical to the ground truth for 12.3% of samples that are identified as obsolete by CEPROT. We also conduct dynamic evaluation and human evaluation to measure the effectiveness of the updated test cases by CEPROT. 48.0% of updated test cases can be compiled and the average coverage of updated cases is 34.2% which achieves 89% coverage improvement over the obsolete tests. We believe that this study can motivate the co-evolution of production and test code.**

***Index Terms*—Test code maintenance, Mining Software Repositories, Software Evolution**

I. INTRODUCTION

Software testing is generally considered as one of the most crucial parts of the software development lifecycle [1], [2]. It helps to identify potential faults and ensure software system quality [3]. Generally, the source code continuously changes to satisfy new requirements or cope with possible issues during the software evolution [4]. To ensure the software quality, the corresponding test cases should co-evolve alongside the changed production code. Unfortunately, the co-evolution of the production code and test code is often missing during the software evolution due to the lack of time to maintain tests or enough knowledge to identify whether tests need to be updated [5]. To illustrate the influence of the obsolete test cases, we show an example of project Conekta [6] in Figure 1. We can find that the API version in the production code was updated to 2.0.0 on 16 Feb 2017. However, it was only on 18 Oct 2019, the API version used in the corresponding test cases was set to 2.0.0. Although the production code passed the test and did not report failures, the new version APIs were never tested during this period and might fail to reveal faults in the production code. Just as the description in the test updating commit, *"The java library Junit test needed to be fixed. Many of the tests where failing due to an encapsulating error with the method* `SetApiVersion`*. Also, other minor bugs where fixed regarding outdated references in the test"*, and many bugs may be introduced from obsolete test cases.

Production Code committed on 16 Feb 2017 4553dd1 src/io/conekta/Conekta.java

```
13
14          public static String apiKey;
15          public static String apiBase = "https://api.conekta.io";
16  -       public static String apiVersion = "1.1.0";
17  -       public static final String VERSION = "2.0.4";
16  +       public static String apiVersion = "2.0.0";
17  +       public static final String VERSION = "2.1.0";
18          public static String locale = "es";
```

Test Code committed on 18 Oct 2019 60f0b9f test/io/conekta/ConektaListTest.java

```
31      public void testSuccsessfulPrevious() throws JSONException, Error, ErrorList{
32  +       setApiVersion("2.0.0");
33          Order last = (Order) list.get(0);
34          JSONObject paginateParams = new JSONObject("{ 'limit': 10 }");
35          ConektaList lastWindow = Order.where(paginateParams);
```

Fig. 1. An example with obsolete test code.

Therefore, many techniques are proposed to mine and analyze the production and test code co-evolution rules and patterns [7], [8], [5], [4]. For example, Zaidman et al. [7] mined the data from version control systems to study the co-evolution of production and test code. Some studies use association rule mining techniques to generate co-evolution patterns [4], [9]. Recently, Wang et al. [8] conducted an empirical study to understand the practice of production-test co-evolution. Then, they proposed an approach named SITAR to facilitate the co-evolution of production and test code by extracting features and leveraging machine-learning techniques. According to their findings, updating existing test code with the production code changes is significantly more frequent than other co-evolution types, e.g., adding/deleting test code. However, their approach SITAR performed worst on predicting whether existing test code should be updated with

§Corresponding author

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00165

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

the production code changes than other types. Besides, the test co-evolution is a difficult task since even minor changes in the production code can significantly affect test code [10].

Inspired by their study, if existing test cases that need to be updated can be automatically identified and co-evolved, it is possible to reduce and even avoid the introduction of obsolete test cases. Different from Wang et al. [8] that only propose to identify obsolete test cases, in this paper, we further propose updating the identified obsolete test cases.

We propose a novel approach named CEPROT to automate the task, i.e., co-evolution of the production and test code (shorted as production-test co-evolution task in the rest of this paper). It includes two stages: ❶ CEPROT identifies whether a test case should be updated while the production code changes; ❷ it updates the test case according to the production code changes simultaneously. However, the co-evolution task of identifying and updating a test case is non-trivial. Existing techniques usually define heuristic rules and extract features manually to associate the production and test code changes [4]. It is time-consuming and labor-intensive to construct rules and features. Moreover, there is a lack of effort to combine the detection and updating of obsolete test cases as a whole and perform automatic end-to-end production-test co-evolution.

To build a more effective tool that helps developers co-evolve production code and test code, we propose to learn from code changes of production code and update existing test code. However, making such a tool is difficult with respect to the following challenges:

- **Representing production code changes**. Identifying and updating obsolete test cases should understand production code changes. Code changes include two versions of production code that consist of a sequence of code tokens. Each position of the code token has its corresponding edit action, such as, "add", "delete", and "replace". Therefore, we should learn fine-grained code changes from edit actions to determine whether the corresponding test should be updated and how to update.
- **Identifying and updating obsolete test based on code changes**. In this study, we should learn from two versions of production code, code change edits, and the existing test to resolve the co-evolution of the production-test code task. This task includes two stages, the obsolete test cases identification in stage 1 and updating them in stage 2. We need to capture semantic correlations among code changes and test cases to identify whether they are obsolete or not. Considering stage 2, we need to update obsolete tests for production code changes. Thus, we should learn where and how to update them during the obsolete test updating stage.

To address the above challenges, we first construct edit sequences from two versions of the production code. The edit sequence consists of tokens of the original version, new version production code, and the edit action in a specific token position. Then, we leverage the pre-trained encoder-decoder Transformer model CodeT5 [11] that has shown remarkable performance on learning from source code and generating code to learn the semantic correlations to identify and update the obsolete test cases. Transformer leverages the self-attention techniques [12] that can learn fine-grained correlations among inputs. We fine-tune the model on the identification and updating stages, respectively. In the online application, CEPROT combines the two stages. To build and evaluate our model, we build two datasets, one for method-level production-test co-evolution identification, and the other one used to update the obsolete test cases. We compare CEPROT with different baselines for the two tasks. Evaluation results show that: 1) CEPROT outperforms its three baselines in the production-test co-evolution identification task in terms of Precision, Recall, and F1-score. 2) CEPROT performs better than its two baselines in terms of Accuracy and CodeBLEU [13] by substantial margins in the updating stage. The experimental results show that CEPROT can help developers better understand where and how to perform production-test co-evolution.

Except for static evaluation, we also conduct dynamic evaluation on the CEPROT updated test cases. We build five popular Java projects and execute updated test cases. The experimental results show that 48.0% generated test cases by our approach CEPROT can be compiled and and the average coverage of updated cases is 34.2% which achieves 89% coverage improvement over the obsolete tests. To explore the quality of generated test cases from the developers' perspective, we conduct a human evaluation. Each practitioner is asked to evaluate the updated test cases from two aspects, the quality of the updated test cases and whether the updating is co-evolved with production code changes. Experiments show that our approach can generate high-quality test cases that co-evolve with production code.

To summarize, our work makes the following contributions:

- We propose a novel two-stage approach, i.e., CEPROT, to automatically identify obsolete test cases and update them for production code changes. It can effectively maintain the co-evolution of production-test code on method level.
- We construct two datasets to identify and update obsolete test cases for the production code changes. We conduct extensive experiments on the dataset. CEPROT is shown to outperform baselines in both two tasks. In addition, it outperforms baselines by combining two stages by a substantial margin.
- We provide our replication package [14] to help researchers and practitioners to repeat our work and verify their studies.

II. BACKGROUND

In this section, we briefly introduce the task definitions of the two stages of CEPROT and its usage scenarios. Then, we illustrate the details of code change representation and CodeT5 that we exploit in this paper.

A. Task Definition

This work aims to identify and update obsolete test cases given code changes at method level. A code change contains two versions of production code, namely, x and x'. Similarly, their associated test cases have two versions, namely, t and t'. The obsolete test identification task can be formulated as:

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

$$\text{Identify}(x, x', t) = \begin{cases} 1 & \text{if } t \neq t' \\ 0 & otherwise \end{cases} \quad (1)$$

If $t \neq t'$, CEPROT will update the obsolete test case t:

$$\text{Update}(x, x', t) = t' \quad (2)$$

We refer x, x', t and t' as original method, updated method, original test, and updated test in the following parts of our paper. Note that the updated test t' is unknown before identifying and updating in the practical application.

B. Usage Scenarios

In this paper, our tool aims to co-evolve the production-test code at method-level. It consists of two stages, i.e., an obsolete test identification stage and an obsolete test updating stage. The typical usage scenario of our tool is to provide test updating suggestions when a developer makes production code changes. Consider Alice is a developer in a large project team. Daily, she performs the development and makes some changes. Without our tool, she ignores to check whether the test cases for the changed production code need to be updated. Further, the obsolete test would not be updated due to her neglect. The obsolete test cases may fail to test the corresponding code changes and might even introduce bugs.

However, with our tool, there are several usage scenarios for Alice while committing the production code change: (1) The original test fails. In this scenario, Alice can realize from the failure that the test case is out of date and should be updated without our tool. However, our tool can accurately remind her whether the original test case needs to be updated before compiling and running. It can reduce her time waiting for compilation and test results and avoid Alice from switching the development context. Besides, our tool can generate a new test case and reduce her edits on updating. (2) The original test passes but becomes inadequate for quality assurance on the new production code. With our tool, it can automatically check whether the test needs to be updated. If the answer is yes, CEPROT will update it subsequently. In summary, with the help of our tool, Alice can successfully identify and update obsolete test cases with fewer efforts, which can increase the maintenance of the system and decrease the likelihood of introducing bugs from production code changes.

C. Code Change Representation

To better represent fine-grained production code changes, we follow existing studies [15], [16] to construct code edits. The code edit is represented as $e = \{x_i \overset{a_i}{\rightarrow} x'_i\}_{i=1}^{N}$ where x_i and x'_i are the token of original version and new version production methods in the position of i, respectively. a_i indicates the edit action that converts x_i into x'_i. There are four types of edit actions, i.e., insert, delete, equal, and replace. To get the code change representation, we first tokenize the original method and the modified method into sequences of code tokens, and use them to construct the edit sequence. We compute word-level alignment to get an edit triple in position i, i.e., $\langle x_i, x'_i, a_i \rangle$. Finally, these edit triples form the edit sequence e for a production code change. For example, the edit sequence e for x: `l.push(1)` $\rightarrow$ x': `l.pop()` is $e = \{$*<l, l, equal>, <., ., equal>, <push, pop, replace>, <(, (, equal>, <1, ∅, delete>, <),), equal>*$\}$.

D. CodeT5

In this paper, the identification task and updating task can be formulated as binary classification and sequence-to-sequence learning problems, respectively. In recent years, CodeT5 has shown outstanding ability in code learning and generation tasks, such as code clone detection and code completion. In this paper, we leverage the Transformer-based model CodeT5 [11] for modeling the production code and test cases, and identifying and generating the updated test cases. CodeT5 is a unified pre-trained encoder-decoder Transformer model. Different from CodeBERT [17] which relies on encoder pre-training, or GPT [18] which relies on decoder pre-training, CodeT5 uses an encoder-decoder architecture. Existing studies show that the CodeT5 has achieved effective results in code-related tasks, such as code generation and defect detection. Thus, we exploit the CodeT5 to initialize our model and finetune it on the two tasks.

III. APPROACH

In this section, we first introduce the overall framework of CEPROT, then we describe the details of the main modules in CEPROT including the outdated test identifier and updater.

A. Overall Framework

The goal of CEPROT is to automate the production-test co-evolution task. Figure 2 shows the overall framework of CEPROT, which consists of three phases, i.e., *input construction*, *model training*, and *online production-test co-evolution application*. In the *input construction* phase, we convert the input tokens into embeddings before feeding them into the neural network. The *model training* phase consists of two stages, i.e., *outdated test identification* and *outdated test updating*. In the *outdated test identification* stage, we train a neural network classifier to identify outdated tests that are needed to be updated. In the *outdated test updating* stage, we train the updater model to generate new test cases. Different from the identification stage, the updating stage only uses the positive samples for training and considers the new test cases as ground truth to be generated. In the *application* phase, given a code change (with two versions of production code and edit sequence) and its associated old test, CEPROT first leverages the trained identifier to predict whether the old test case needs to be updated. If the answer is yes, the trained updater will be used to generate a new test to replace the old one. We elaborate on the details of each phase as follows:

B. Input Construction

The input of the model includes an original production method x, an updated production method x', and an original test t. As shown in Figure 2, for each code segment, we first tokenize it into a sequence of tokens, i.e., $x = \{x_1, ..., x_m\}$,

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

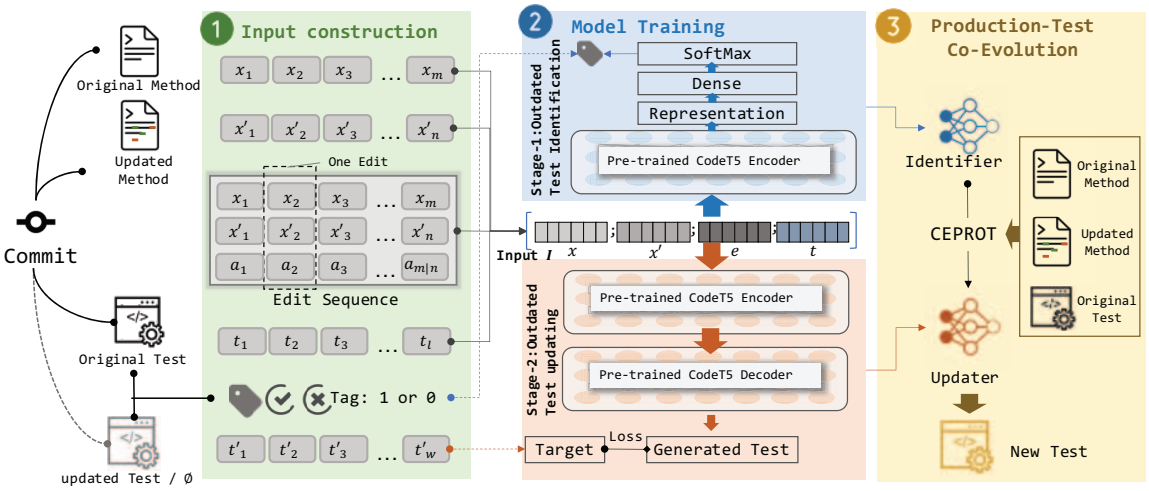

Fig. 2. The overall framework of our proposed approach CEPROT.

$x' = \{x'_1, ..., x'_n\}$, and $t = \{t_1, ..., t_l\}$, in which m, n and l represent the lengths of the original method, the updated method, and the original test, respectively. We exploit the Roberta [19] tokenizer to get these tokens. Different from building tokens by separating source code according to the whitespace, it leverages the Byte-Pair Encoding (BPE) [20] to alleviate the Out-of-Vocabulary (OoV) issues. In addition, we also build edit sequences e for code changes. For an edit $e_i = \langle x_i, x'_i, a_i \rangle$, we concatenate these three parts to represent the edit e_i. We get a_i by comparing the token x_i and x'_i at position i. The inputs of x, x', e, t are then concatenated into one input I and are used to predict the probability distributions in target. For the identification task, the target $l = \{0, 1\}$ represents that a test does not need to be updated or needs to be updated, respectively. For the updating task, the target is the updated test cases for positive samples. The model aims to generate the new test case t' as accurately as possible. In this work, we concatenate different inputs into one input instead of using multi-encoders to encode each input separately. Compared to multi-encoders, one input is beneficial to the self-attention mechanism used in CodeT5 to capture the relationship across different inputs.

C. Outdated Test Identification

At this stage, we propose an obsolete test identifier to identify if a test case needs to be updated. It includes an encoder and a classifier.

1) Encoder: The encoder is responsible for acquiring the contextual representative embedding of the input sequence. In this paper, we exploit the pre-trained CodeT5 encoder to initialize the input I. The CodeT5 has shown promising results in code understanding and generation due to its strong ability to capture code semantics conveyed from the developer-assigned identifiers. CodeT5 is a pre-trained model based on the Transformer [12]. The encoder of the Transformer converts the input I into the contextual vector representation $\mathbb{R}$.

For each input token I_i, the Transformer generates three embeddings for it, i.e., a query vector q_i, a key vector k_i, and a value vector v_i. The Transformer encoder exploits the dot products to calculate the attention scores for I_i by using the query vector q_i and the key vector k_j of each token in the input, as follows: $\alpha_{i,j} = \frac{q_i \cdot k_j}{\sqrt{d}}$ in which d is the dimension of q_i and k_j. The attention score represents how much focus to place on the jth input as we encode the ith input. Then, CEPROT gets the normalized scores by a softmax function:

$$\hat{\alpha}_{i,j} = \text{softmax}(\alpha_i) = \frac{\exp(\alpha_{i,j})}{\sum_t \exp(\alpha_{i,t})} \tag{3}$$

To learn relevant/irrelevant tokens, softmax is used to multiply each value vector and these vectors are summed up:

$$z_i = \sum_j \hat{\alpha}_{i,j} v_j \tag{4}$$

We use the last hidden state $z_{|I|}$ as the contextual vector representation $\mathbb{R}$ of the input I.

2) Classifier: This step aims to make binary classification according to the learned representation $\mathbb{R}$. To better capture the relationships of these four input information ($I_i = \{x, x', e, t\}$) from $\mathbb{R}$, we use a dense layer equipped with a non-linearity function to learn latent interactions between them. Then, the output of the dense layer is used to predict the likelihood of the final label $l = \{0, 1\}$. More precisely, the classifier is defined as follows:

$$\begin{aligned} f &= \tanh(W\mathbb{R} + b) \\ P(l|\langle x, x', e, t\rangle) &= \text{Softmax}(f) \end{aligned} \tag{5}$$

W and b denote the weight matrix and bias vector, respectively. tanh is an activation function for the dense layer's perceptron. The Softmax function will output the final probability of the label l between 0 and 1. For the probability score, we want this score to be high if the original test needs to be updated and to be low if the test does not need to be updated.

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

D. Obsolete Test Updating

At this stage, we propose an obsolete test updater to update obsolete test cases that need to be updated. It also includes two parts, an encoder and a decoder.

1) Encoder: We exploit the same encoder as the outdated test identifier. The input of the updater also includes four parts, i.e., $I = \{x, x', e, t\}$. As different parts of the input contribute to different parts of the test to be generated, we should learn the correlation between each token of the input I and the target updated test t'. Therefore, different from the identification task, we use the attention scores $z = \{z_1, ..., z_{|I|}\}$ of I instead of the last hidden state as the input to the decoder.

2) Decoder: Different from the identification task, the outdated test updating should generate new test cases given the input. The decoder learns to generate the corresponding new test t' one token at a time based on the input and all preceding tokens that have been generated so far. Mathematically, the test updating task is defined as finding $\bar{t}'$, such that: $\bar{t}' = \text{argmax}_{t'} P_\theta(t'|I)$ where $P_\theta(t'|I_u)$ is:

$$P_\theta(t'|I) = \prod_{i=1}^{w} P_\theta(t'_i|t'_1, ..., t'_{i-1}; x, x', e, t) \tag{6}$$

$P_\theta(t'|I)$ can be seen as the conditional log-likelihood of the predicted new test case t' given the input I. This model can be trained by minimizing the negative log-likelihood of the predicted test cases and the ground truth.

Specifically, the architecture of the decoder consists of two parts, namely, the self-attention layer and the encoder-decoder attention layer. The calculation of the self-attention layer is similar to the encoder except that it only deals with the words generated so far. The encoder-decoder attention layer learns the correlation between the output sequence and the input sequence. When calculating the attention scores between the encoder and the decoder, the key vector $\boldsymbol{K}$ is from the outputs of the encoder $\boldsymbol{z} = (\boldsymbol{z_1}, ..., \boldsymbol{z_{|I|}})$. The attention distribution $\boldsymbol{\alpha^j}$ between the target words y_j and the source code tokens $w_1, ..., w_m$ is:

$$\boldsymbol{\alpha^j} = \text{softmax}(\frac{\boldsymbol{Q_j K}^T}{\sqrt{d_k}}) \tag{7}$$

Specifically, the identifier and updater are initialized by CodeT5 and fine-tuned on the two tasks, respectively.

E. Online Production-Test Co-Evolution

After the training process, the two models can be combined together to maintain the co-evolution of production-test code. When the developer makes changes to the production code, CEPROT can be applied to these changes and identify whether the test case should be updated. If the test is identified as obsolete, CEPROT will then generate a new test case that can test the new production code.

IV. EXPERIMENTAL SETUP

In this section, we introduce the details of the dataset construction process. Then, we describe the baselines and evaluation metrics. Finally, we show the experimental settings of different approaches.

A. Dataset Construction

In this study, we aim to identify and update obsolete test cases for method-level production code. We build two datasets for training and evaluating CEPROT on two stages. This dataset is constructed from Java projects that contains a large amount of high-quality unit tests. To build such a dataset, we need to collect co-evolution Java methods and their corresponding test cases annotated with the `@Test` annotation, i.e., *<original method, updated method, original test, updated test>* quadruples. We notice that Liu et al. [16] collected a large scale high-quality dataset that consists of 6,106K method-level changes from Top 1,500 popular Java projects provided by Wen et al. [21]. These method-level changes include non-test methods and unit tests. Thus, we ask them for the dataset and use it to construct our dataset.

1) Construct Co-Evolve production-test samples: Then, we extract co-evolution method-level production-test samples from the extracted modified methods above. As the quality of dataset is essential for deep learning models, we regard that a method and its corresponding test change within the same commit as a co-evolving production-test pair. For each non-test method (without `@Test` annotation), we check if there exists a co-changed unit test in the same commit for it. According to Wang et al. [8] and Tufano et al. [22], test names are often similar to the corresponding production methods and can be matched according to the naming convention. We follow them to define heuristic rules to extract unit tests for methods by name matching that is widely leveraged to associate production and test code [8], [22], [7], [23].

For the identification task, we need to build a dataset with positive samples and negative samples. If a test case should be updated along with the changes of its corresponding production code, we label it as a positive sample. Otherwise, we will label it as a negative sample. For test updating task, we only use the positive samples whose test cases need to be updated in the identification task to construct the dataset. The details of positive/negative samples construction and test updating dataset are shown as follows:

Positive Samples Construction. First, we extract changed test cases with `@Test` annotation (i.e., t and t') from the collected method-level code changes in Section 4.1.1. Second, we extract the test case names by using Tree-Sitter [24]. Third, we conduct name matching to find the production code. If the production code is in the same project and also changed in the same commit, we regard it as a positive sample. Finally, we get 9,985 positive samples and each sample includes four parts, i.e., *<original method, updated method, original test, updated test>*.

Negative Samples Construction. To construct the negative samples, we should extract changed methods with test cases that do not need to be updated. First, we exclude methods in the positive samples and extract test cases for the remaining changed methods. For each candidate changed method, we search its test case as follows:

- **Path Matching**: According to best practices for JUnit

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

TABLE I
STATISTIC INFORMATION OF OUR DATASET

Set	Identification task		Updating task
	# Positive	# Negative	
Training	4,676	16,496	4,676
Test	520	1,833	520
Total	5,196	18,329	5,196

testing, production code, and its corresponding test cases are in the mirrored folder. Thus, we heuristically find test classes by path matching. We get the filename where the method is located and identify the file path of the test cases. For example, the test file for `/src/main/java/Connect.java` is usually in `/src/test/java/ConnectTest.java` or in `/src/test/java/TestConnect.java`.

- **Name Matching**: After the path matching, we extract corresponding tests for the candidate method. Similar to the construction of positive samples, we match the production code and test by name matching.

Then, we exclude samples whose test cases have changes and get 52,565 negative samples and each sample includes three parts, i.e., *<original method, updated method, original test>*.

2) Data filtering, splitting, and processing: Generally, a commit may contain duplicate change samples since developers may perform systematic or recurring code changes in one commit [25]. To avoid the bad performance of duplicate samples, we remove duplicated production method samples from the collected dataset. Finally, we get 5,196 positive samples and 18,329 negative samples, respectively. Then, we construct training data and test set for the two tasks. For the obsolete test identification task, we randomly select 90% positive samples and negative samples to train the model and the rest samples are used to test the effectiveness of CEPROT. Each sample includes four parts, i.e., *<original method, updated method, original test, label>*. For obsolete test updating task, we only use the positive samples and each sample consists of *<original method, updated method, original test, updated test>*. The detailed information of our dataset is shown in Table I.

B. Baselines

1) Baselines for Identifier: To assess the effectiveness of CEPROT on identifying obsolete test cases, we compare it with SITAR, K-Nearest Neighbor (KNN), and LSTM:

KNN K-Nearest Neighbor (KNN) is an IR-based approach. In this paper, we refer to the idea of NNGen [26] and build a KNN classifier to identify obsolete test cases. For each case in the test set, we compare it with each sample in the training set using two versions of production code and edit sequence. We return the top-k samples that have most overlapped $\langle x_i, x_i', a_i \rangle$ triples. After that, we use the cosine similarity to measure the similarity between the original test of this case and that of the returned top-k samples. Then, the label corresponding to the most similar sample in the top-k data is regarded as the predicted label. By default, we set k as 5.

SITAR is the state-of-the-art approach that aims to automatically identify outdated unit tests. According to Wang et al. [8], Random Forest [27] outperforms other classifiers in test update prediction. Thus, we take Random Forest as the classifier and replicate SITAR on our collected dataset.

LSTM. Long short-term memory is a widely used recurrent neural network. We use it to encode the input that is similar to CEPROT. Then, the dot-product attention mechanism [28] is adopted to fuse the input information. Similar to CEPROT, we exploit the softmax to predict the probability of each label. Specifically, due to the length limitation of LSTM, we truncate the overlong input, the lengths of edit sequence and original test are set to 300 and 150, respectively.

2) Baselines for Updater: To the best of our knowledge, there is no prior work on automated outdated test case updating. We also use two baselines belonging to different types for evaluating CEPROT updater, i.e., KNN and NMT. The details of experimental settings are shown as follows:

KNN. We obtain predictions for samples in the test set the same as the identification task. In the updating task, the KNN method will return the updated test of the most similar training sample as the final generation of KNN. In this task, k is also set to 5 by default.

NMT. Neural Machine Translation model is a typical sequence-to-sequence model. Similar to the Identification task, we build the model based on the LSTM. Except for the encoder, we use another LSTM to generate the new test case. To better learn correlations from the decoder and the encoders, we also use the attention mechanism. Similar to the identifier, we truncate the overlong input and limit the generation length. Specifically, the length of edit sequence is set to 300, and the length of original test and updated test is set to 150.

3) Baselines for two-stage CEPROT*:* As CEPROT is a two-stage approach, we compare it with above baselines by combining identification stage and updating stage mentioned above, i.e., KNN and NMT. The KNN includes a KNN identifier and a KNN updater. The NMT consists of an LSTM identifier and a NMT updater. For each baseline, we first identify whether a test case should be updated and then we update it with corresponding updating models.

C. Evaluation Metrics

1) Metrics for Identification: In this paper, the identification of obsolete test cases can be formulated as a binary classification task, i.e., it is a binary classifier. Therefore, we adopt **Precision**, **Recall**, and **F1**, which are well-known metrics for binary classification, to measure the performance of CEPROT on outdated test detection.

2) Metrics for Updating: We use **CodeBLEU** [13] and **Accuracy** to evaluate CEPROT and the baselines.

CodeBLEU. CodeBLEU is proposed to evaluate the code synthesis task. It is implemented based on BLEU that is originally proposed to evaluate the generated sentences for neural machine translation task [29] and widely used in software engineering tasks [30], [31], [32], [33]. However, existing studies [13] show that it does not take into account

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

TABLE II
OBSOLETE TEST IDENTIFICATION AND UPDATING EVALUATION

	Obsolete Test Identification			Obsolete Test Updating	
Approaches	**Precision**	**Recall**	**F1**	**CodeBLEU**	**Accuracy**
KNN	83.4%	72.3%	77.5%	37.6%	3.9%
SITAR	78.3%	38.9%	52.0%	-	-
NMT	96.3%	89.2%	92.6%	32.3%	5.0%
CEPROT	**98.3%**	**90.0%**	**94.0%**	**63.1%**	**12.3%**

TABLE III
EVALUATION RESULTS ON THE TWO-STAGE APPLICATION

Approach	**#TPC/#TP**	**#FPC/#FP**
KNN	20/376 (5.3%)	2/75 (2.7%)
SITAR	-	-
NMT	11/464 (2.4%)	4/18 (22.2%)
CEPROT	**62/468 (13.2%)**	**6/8 (75.0%)**

the grammatical and logical correctness, resulting in favoring candidates with high n-gram accuracy and serious logical errors. They propose CodeBLEU to better evaluate the code generation tasks. Thus, we exploit CodeBLEU to evaluate the effectiveness of CEPROT and more details can be found in [13].

Accuracy. Accuracy aims to measure the ability of CEPROT to generate correct test cases that are identical to the references.

3) Metrics for the two-stage framework: Our proposed approach CEPROT is the combination of identification and updating models. We evaluate the performance of CEPROT and baselines in terms of #TPC/#TP and #FPC/#FP. #TP and #FP represent the numbers of true positive (TP) samples and false positive samples (FP), respectively. #TPC refers to the number of TP samples that are correctly updated by an approach. #FPC denotes the number of FP samples that are not updated by an approach, i.e., the generated tests are the same as the corresponding original tests.

D. Experimental Settings

In this paper, we exploit the PyTorch framework [34] to implement CEPROT. The hidden size of the dense layer before classifier is 768. During the fine-tuning process, we use AdamW [35] with shuffled mini-batches to optimize the parameters of our model. We use the CodeT5-base model that includes 12 layers of the Transformer in our model. The learning rate of the AdamW is set to 1e-5 and the batch size is 2. The training process lasts 15 epochs. Both two tasks are trained separately and aim to minimize the cross entropy. Our experiments are conducted on Ubuntu 20.04.3 LTS 64bits, with an NVIDIA GeForce RTX 2080 Ti 12GB.

V. RESULTS

We investigate the following research questions:

RQ1: How effective is CEPROT on obsolete test identification?
RQ2: How effective is CEPROT on obsolete test updating?
RQ3: Can CEPROT effectively co-evolve production-test code with two-stage given code changes?
RQ4: How efficient is CEPROT?

A. RQ1: The effectiveness of CEPROT identifier

To answer this research question, we evaluate CEPROT on the collected dataset in terms of Precision, Recall, and F1. The left part of Table II illustrates the evaluation results on the obsolete test identification task. We can observe that CEPROT achieves a precision of 98.3%, a recall of 90.0%, and an F1 score of 94.0%. Specifically, CEPROT outperforms all baselines in terms of all metrics. Among all baselines, LSTM (i.e., the NMT in Table II) performs best and SITAR performs worst among all baselines. SITAR is designed for file-level code changes and may not perform well on method-level code changes. Although its precision is 78.3%, its recall is very low. Since the co-evolution of production-test code is a two-stage task, we should take into account both precision and recall. On one hand, we should ensure that obsolete tests are identified as many as possible. On the other hand, we should ensure that the identified tests are indeed obsolete to avoid the redundant update in the updating stage. If an approach achieves 100% Recall but to find one obsolete test cases developers need to inspect many candidates, developers may not be happy to use it in practice. Considering that CEPROT achieves improvements of 21.3%, 80.8%, and 1.5% in terms of F1 when compared to KNN, SITAR and LSTM, respectively, our approach can effectively identify obsolete test cases.

B. RQ2: The effectiveness of CEPROT updater

The right part of Table II shows the results on updating obsolete test cases. We compare CEPROT with two baselines (i.e., KNN and NMT) in terms of Accuracy and CodeBLEU. We can observe that CEPROT outperforms all baselines in terms of all metrics with large improvements. The high CodeBLEU of CEPROT demonstrates that many generated test cases are syntactically correct with higher AST match and data-flow match. When compare it with baselines in terms of Accuracy, the KNN and the NMT can update 3.9% and 5.0% test cases correctly. The experimental results show that KNN and NMT can hardly generated correct test cases. As for CEPROT, it can correctly update 12.3% test cases and has improvements of 215.4% and 146.0% compared to KNN and NMT, respectively.

C. RQ3: Two-stage Evaluation

This research question aims to investigate how effective CEPROT can be by combining the identification and the updating of obsolete test cases. We evaluate and compare the CEPROT with baselines on a two-stage setting, i.e., identifying the obsolete test case and then updating them. We compare them in terms of #TPC/#TP and #FPC/#FP. Table III illustrates the evaluation results.

We can observe that CEPROT outperforms these two baselines in terms of all metrics by substantial margins. At the obsolete test identification phase, first, CEPROT identifies more obsolete tests with much higher precision. Then, at the updating stage, CEPROT updates true positive samples and false positive samples with higher Accuracy. We can find that the #FP of KNN and NMT is much higher than CEPROT. In

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
TRAINING AND INFERENCE TIME CEPROT

Approach	Identification		Updating	
	Training	Inference	Training	Inference
KNN	36m	0.269s	2m	0.221s
SITAR	1.62s	0.002ms	-	-
NMT	40m	0.025s	55m	0.175s
CEPROT	1h21m	0.062s	2h44m	0.975s

TABLE V
PROJECT INFORMATION IN DYNAMIC HUMAN EVALUATION

Projects	Stars	#Versions	#Cases
springside/springside4 [36]	5.8k	2/2	3
openmrs/openmrs-core [37]	3.1k	21/34	22
apache/commons-lang [38]	1.4k	5/5	8
datumbox/datumbox-framework [39]	1.1k	7/7	9
dayatang/dddlib [40]	0.5k	2/4	8
Total	-	37/52	50

addition, the #TPC of CEPROT is 3.1 and 5.6 times of KNN and NMT, respectively.

In detail, CEPROT achieves an accuracy of 13.2% (62/468) on its predicted true positive samples, which is much better than those of KNN (5.3%: 20/376) and NMT (2.4%: 11/464) with improvements of 149% and 450%. We can observe that although the NMT model can identify obsolete test cases effectively, it fails to update them. On the other hand, although there are eight negative samples predicted as positive, six of the generated new test cases are the same as original test cases. In addition, CEPROT has less FP and more FPC when compared to KNN and NMT. We admit that the performance of CEPROT is not perfect, but we argue that CEPROT is still useful for developers, because: First, CEPROT has high precision and recall that can help developers to find as many as obsolete test cases and reduce the false positives. Second, it tries to update obsolete test cases correctly and does not update false positive tests to reduce redundant update.

D. RQ4: Time efficiency

To measure the time complexity of our approach and other baselines, we record the start time and the end time of their training process and the test process. We train and evaluate all models on the same machine containing an NVIDIA GeForce RTX 2080 Ti GPU with 12 GB memory for a fair comparison. Table IV shows the time costs of training and average inference per sample. The training time varies as it depends on the size of dataset. Once models have been trained, it only takes a few microseconds to identify and update obsolete test cases. CEPROT can identify and update obsolete tests within 0.062s and 0.975s, respectively. Considering the performance in obsolete tests identification and updating, the experimental results demonstrate that our approach is efficient for practical uses.

TABLE VI
DYNAMIC EVALUATION AND HUMAN EVALUATION RESULTS

Approach	Compilability	Cov.	Quality	Co-Evolvability
Obsolete Test	40%	18.1%	4.53	6%
KNN	44%	6.5%	3.11	4%
NMT	30%	22.2%	2.22	22%
CEPROT	**48%**	**34.2%**	**3.81**	**38%**
Human-Updated	100%	62.3%	4.74	66%

VI. DYNAMIC AND HUMAN EVALUATION

Although static metrics, e.g., Accuracy and CodeBLEU, can evaluate the gap between the updated test cases by CEPROT and tests written by humans, it cannot reflect the effectiveness of updated test cases dynamically and human perceptions on them. Thus, we further conduct a dynamic evaluation and a human evaluation on it. Table V shows detailed information on these projects. We randomly select five Java projects (all have more than 500 stars) and manually build different versions of them. *#Versions* means the number of versions we successfully build and versions we need to build. These projects have 52 versions in total in which each version corresponds to a code change. Except for openmrs/openmrs-core and dayatang/dddlib, we successfully build all versions for the other projects. In total, we successfully build 37 versions and there are 50 test cases that should be updated. Then, we conduct dynamic evaluation and human evaluation on these 50 cases.

A. Dynamic Evaluation

The dynamic evaluation measures CEPROT from two aspects, including the *compilability* and *coverage*. The *compilability* measures how many updated test cases can be compiled after building projects. The *converage* evaluates how much of the updated production code is tested by running generated test cases. As shown in Table VI, 48% (i.e., 24) updated test cases by CEPROT can be compiled successfully. We manually inspect test cases updated by CEPROT and find that all updated test cases of dayatang/dddlib can be successfully compiled and most compilation failed test cases (in projects openmrs/openmrs-core and datumbox/datumbox-framework) are caused by incomplete long test cases that are truncated while generating.

We then measure the coverage of updated cases by using JaCoCo [41] which is a free code coverage library for Java. Table VI reports the average statement coverage of these cases. The average coverage of CEPROT is 34.2%, which outperforms baselines by a large margin and improves the coverage of the obsolete tests a lot. Although the coverage of test cases updated by our approach is lower than human-updated, our approach has an 89% improvement over the obsolete tests.

B. Human Evaluation

We then conduct a human evaluation to evaluate the updated test cases that passed compiling from two aspects, *quality* and *co-evolvability*. We invite three senior engineers with more than five years of Java development experience to conduct the

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

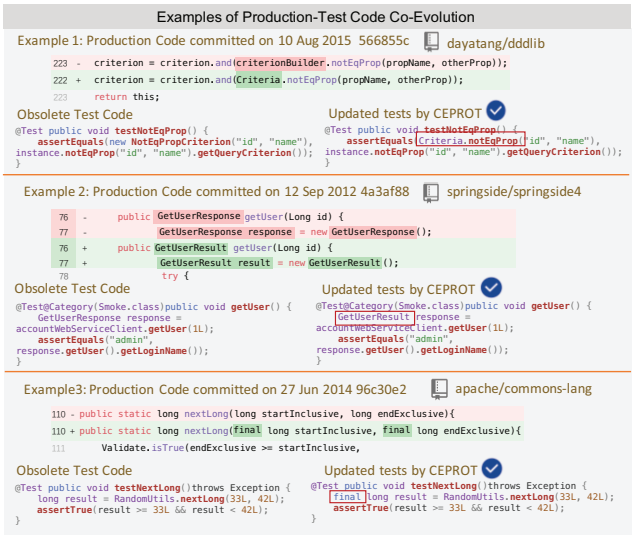

Fig. 3. Updating Examples

human evaluation. The *quality* measures a test case whether it follows testing practices [42], e.g., including necessary asserts for production code. The *quality* score ranges from 1 to 5 (the higher the better). Each engineer gives a score for the updated test case by reading the updated test and the corresponding new version of production code. We report the average *quality* scores among these three annotators. The *co-evolvability* measures whether updated test cases are co-evolve with production code. The answer *co-evolvability* of each case is *Co-Evolved* or *Not Co-Evolved*. The annotator reads the production code changes, old test cases, and updated test cases and determines whether the updated test cases are co-evolved with production code changes. The human evaluation results are shown in Table VI. The Fleiss' kappa among the three annotators is 0.97. The average quality score is 3.81 and 66% test cases are co-evolved with production code changes. The obsolete tests are written by humans, thus the *quality* of obsolete tests is higher than that of CEPROT. However, the *co-evolvability* of tests updated by CEPROT is much higher than obsolete ones or those generated by other approaches.

To better understand how test co-evolves with production code changes, we manually inspect the results. As Figure 3 shows, the updating mainly includes the following types:

API Updating: CEPROT can update the API invocations along with the production code change. For example, an API invocation in production code is change (i.e., `criterionBuilder.notEqProp→Criteria.notEqProp` in the project dayatang/dddlib), the test code is co-evolved as `new NotEqPropCriterion → Criteria.notEqProp`.

Identifier Updating: Identifiers are often changed during the software evolution. Updating the corresponding identifiers in the test is important for developers. Through the manual inspection, we find that CEPROT can effectively update these identifiers, e.g., `GetUserResponse → GetUserResult` in springside/springside4 project.

Modifier Updating: Modifiers are one of the most important parts in Java projects. CEPROT can add corresponding modifiers along with the production code. For example, developers add *final* keyword into the production code (in project apache/commons-lang), it also adds a corresponding modifier.

In summary, we find that CEPROT performs well on learning fine-grained code changes from edit actions. To the best of our knowledge, CEPROT is the first work focusing on production-test co-evolution in two stages. The task itself is not easy considering the difficulties of "understanding" code changes and "updating" tests. In addition, CEPROT has outperformed the baseline in all metrics by substantial margins and has good performance during dynamic evaluation and human evaluation. Based on these facts, we believe CEPROT can reduce developers' efforts on test maintenance. Also, it can promote the development of this research direction and inspire other researchers to tackle this important task.

TABLE VII
TIME SERIES EVALUATION

	Obsolete Test Identification			Obsolete Test Updating	
Approaches	Precision	Recall	F1-score	CodeBLEU	Accuracy
KNN	58.9%	61.2%	60.1%	27.2%	1.5%
SITAR	39.9%	17.0%	23.9%	-	-
NMT	89.1%	79.2%	83.9%	40.6%	2.0%
CEPROT	**94.4%**	**81.6%**	**87.6%**	**59.3%**	**8.8%**

VII. DISCUSSION

A. Time Series Evaluation

To evaluate the ability of CEPROT on the future co-evolution of production and test code, we also conduct an experiment on a time series setting. We reconstruct a dateset according to the commit time. For each project, we use the historical data to train the model and apply the trained model to make predictions for new production code changes. For each project, we sort its commits in the ascending order of commit creation time, put the first 90% commits into the training set, and take the remaining 10% commits as the test set. Then, we retrain the models on the time series dataset using the same parameters. Table VII shows the results of different approaches on the time series experimental setting. We can observe that the performance of baselines decreases dramatically, especially the KNN and the SITAR. In addition, both KNN and NMT can hardly update the test cases correctly.

B. Effectiveness of different variants of CEPROT

To identify and update obsolete test cases, we take before/after versions of production code and edit sequences as inputs. Each part of input has different impacts on the effectiveness of CEPROT. To investigate the effectiveness of each input, we compare CEPROT with its two variants, i.e., CEPROT-w/o code and CEPROT-w/o edit. CEPROT-w/o edit uses two versions of production code and the original test to identify and update obsolete tests without edit sequence.

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

TABLE VIII
THE EFFECTIVENESS OF EACH COMPONENT OF CEPROT

Variants	Identification			Updating		Combination of two phases	
	Precision	Recall	F1	CodeBLEU	Accuracy	#TPC/#TP	#FPC/#FP
CEPROT-w/o code	97.6%	86.2%	91.5%	61.9%	4.4%	23/448 (5.1%)	9/11 (81.8%)
CEPROT-w/o edit	96.4%	86.3%	91.1%	62.6%	10.8%	51/449 (11.4%)	9/17 (52.9%)
CEPROT	98.3%	90.0%	94.0%	63.1%	12.3%	62/468 (13.2%)	6/8 (75.0%)

CEPROT-w/o code only exploits the edit sequence to realize this task. Table VIII shows the results of CEPROT and its variants. The performance of CEPROT-w/o edit is similar to CEPROT-w/o code and decreases when compared with CEPROT on the identification task. The F1 decreases about 3.3% of CEPROT when removing the production code or edit sequence. For the updating task and two-stage setting, the performance of both CEPROT-w/o code decreases dramatically when compared to CEPROT. Specifically, the #TPC/#TP of CEPROT decreases 158.8% when removing the production code. The production code provides the semantic information of the method to be tested, which helps CEPROT generate correct tests.

In addition, both CEPROT-w/o code and CEPROT-w/o edit outperform baselines by substantial margins. It demonstrates that both production code and edit sequence contribute to obsolete test updating. The production code provides the semantic information of the method to be tested and edit sequence helps CEPROT learn fine-grained code changes.

C. Why does CEPROT Fail?

We also investigate why CEPROT fails to co-evolve the test cases with changes in production code. We manually inspect samples that CEPROT fails to make correct predictions. The failure is mainly caused by truncated updated test cases. During the identification stage, there are some false positive cases. These cases are mainly caused by value updating in the production code, e.g., updating a string value. During the updating stage, CEPROT fails to update long test cases. Deep neural models are limited in generating too-long test cases. We find 270 incorrect updated test cases by CEPROT are incomplete in the test set. Updating long test cases effectively is an interesting and promising direction for future work.

D. Why updating test code instead of generating?

According to Wang et al. [43], developers update existing production code and its corresponding test code frequently. Besides, this behavior is much more frequent than adding new test code. Thus, compared to generating test code from scratch, updating existing test code with explicit production code changes is more helpful for developers.

E. Threats to Validity

Internal Vality. Threat to internal validity refers to the errors and bias in our experiments. Existing outdated test identification studies mainly rely on heuristic rules and feature extractions. We find that they do not provide replication packages. We try our best to replicate their approaches and they may not the same as theirs. Besides, there is no work focus on updating test cases and two-stage production-test co-evolution. Therefore, we construct baselines that are introduced in Section IV. As these baselines include both IR-based and DL-based approaches, we believe this threat is limited.

Data Validity. The quantity of our dataset (especially the obsolete test cases, i.e., positive data) is a threat in this study. Wang et al. [8] extract co-evolution production-test pairs from commits within 48 hours. To guarantee the quality of our dataset, we only extract the co-evolved pairs as positive ones. It may ignore some positive cases (co-evolved within 48h). We randomly check some code changes within 48h manually and find that some test changes within 48h are not co-evolved with production code. Thus, we use co-evolved pairs within the same commit instead of commits within 48h. Although it guarantees the high quality of collected data, some positive samples may be ignored in our study.

External Validity. Threats to external validity concern the generalization of CEPROT. Our dataset is built only from Java projects. It may not be representative of all programming languages. However, Java is one of the most popular programming languages. Instead of extracting code features and building heuristic links between co-evolve product-test cases, our approach is trained on the code tokens and is independent of code features. Thus, our approach is language-agnostic and we believe it can be easily adapted for other languages.

VIII. RELATED WORK

A. Co-Evolution of Production-Test Code

Co-Evolution of the source code and test code is an essential part for software maintenance. Wang et al. [8] investigate 975 open-source Java projects and mine factors (i.e., the complexity of the production code changes) that can determine whether the test code should be updated. They also propose an ML-based approach SITAR to identify whether a test should be updated while the source code is changed. In addition, various techniques are proposed to build links between the source code and test code [4], [44], [9]. They focus on mining co-evolution patterns, e.g., aggregating fine-grained changes [4] and introducing new class in commits [44]. Generally, these approaches define heuristic rules to build traceability links between production code and test code. Different from them, our approach cannot only identify obsolete test cases but update them automatically.

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

B. Test Generation

Considering the importance of software testing, several automated tools are proposed to generate tests automatically, such as EvoSuite [45] and Randoop [46]. EvoSuite [45] is a typical search-based unit test generation tool. Randoop [46] is a feedback-directed test generation technique that is based on random testing. Considering the effectiveness of deep learning techniques in learning the source code, researchers have proposed some techniques to generate test cases. Tufano et al. [22] propose an approach named ATHENATEST that leverages the pre-trained model BART [47] to generate test cases. To alleviate the uninformative identifiers issues in the generated test cases, Roy et al. [1] introduce the DeepTC-Enhancer that generates meaningful identifier names to enhance the readability of automatically generated test cases.

Recent studies also point out that the generated assert statements are often incomplete or lacking the necessary complexity to capture a designated fault [2], [48]. Watson et al. [2] propose a NMT based approach to generate meaningful assert statements for test cases. Similar to ATHENATEST, Tufano et al. [48] exploit the pre-trained model BART [47] and finetune it on the task of generating assert statements for unit test cases. Dinella et al. [49] propose a transformer-based approach TOGA to infer exceptional and assertion test oracles. These studies generate meaningful assert statements for test cases. They perform well on generating single assert statements.

Our work is different from them, we focus on updating preexisting tests instead of generating test cases from scratch, and it considers both the old test and the corresponding code change instead of only the new code. Also, the goal of this work is to combine obsolete test identification and updating, instead of generating tests.

IX. CONCLUSION AND FUTURE WORKS

In this paper, we propose CEPROT to motivate the co-evolution of production-test code. To the best of our knowledge, we are the first to use a two-stage framework to identify and update obsolete test cases. We build CEPROT on a pre-trained model, i.e., CodeT5, and fine tune it on the two tasks. To train and evaluate CEPROT, we construct two datasets. The evaluation results show that CEPROT outperforms baselines on the co-evolution production-test code by a substantial margin. We also conduct dynamic evaluation and human evaluation to investigate the quality of updated test cases. In the future, we will investigate other techniques to improve our approach by integrating contextual information or improving learning techniques for code changes. In addition, we will extend our approach to other programming languages, e.g., Python and JavaScript, to improve the generalizability of our approach.

ACKNOWLEDGMENT

This research was supported by the National Key Research and Development Program of China (No. 2021YFB2701102), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), the National Natural Science Foundation of China (No. 62141222, No. 62202420, and No. U20A20173).

REFERENCES

[1] D. Roy, Z. Zhang, M. Ma, V. Arnaoudova, A. Panichella, S. Panichella, D. Gonzalez, and M. Mirakhorli, "Deeptc-enhancer: Improving the readability of automatically generated tests," ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 287–298. [Online]. Available: https://doi.org/10.1145/3324884.3416622

[2] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1398–1409. [Online]. Available: https://doi.org/10.1145/3377811.3380429

[3] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[4] C. Marsavina, D. Romano, and A. Zaidman, "Studying fine-grained co-evolution patterns of production and test code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 195–204.

[5] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *2012 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 11–20.

[6] https://github.com/conekta/conekta-java/pull/92.

[7] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. v. Deursen, "Mining software repositories to study co-evolution of production & test code," ser. ICST '08. USA: IEEE Computer Society, 2008, p. 220–229. [Online]. Available: https://doi.org/10.1109/ICST.2008.47

[8] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 272–283.

[9] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production amp; test code," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 151–154.

[10] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. USA: IEEE Computer Society, 2001, p. 170. [Online]. Available: https://doi.org/10.1109/ICSM.2001.972727

[11] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[13] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[14] https://github.com/CEPROTest/CEPROT.git.

[15] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. Gaunt, "Learning to represent edits," in *ICLR 2019*, May 2019, arXiv:1810.13337 [cs.LG]. [Online]. Available: https://www.microsoft.com/en-us/research/publication/learning-to-represent-edits/

[16] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 585–597. [Online]. Available: https://doi.org/10.1145/3324884.3416581

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online:

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.

Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139
[18] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.
[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
[20] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725. [Online]. Available: https://aclanthology.org/P16-1162
[21] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 53–64.
[22] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2020.
[23] B. V. Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 209–218.
[24] https://tree-sitter.github.io/tree-sitter/.
[25] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. USA: IEEE Computer Society, 2009, p. 309–319. [Online]. Available: https://doi.org/10.1109/ICSE.2009.5070531
[26] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, *Neural-Machine-Translation-Based Commit Message Generation: How Far Are We?* New York, NY, USA: Association for Computing Machinery, 2018, p. 373–384. [Online]. Available: https://doi.org/10.1145/3238147.3238190
[27] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[28] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.0473
[29] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *Proceedings of ACL 2017, System Demonstrations*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 67–72. [Online]. Available: https://aclanthology.org/P17-4012
[30] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 631–642. [Online]. Available: https://doi.org/10.1145/2950290.2950334
[31] M. Allamanis, E. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys*, vol. 51, 09 2017.
[32] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 8984–8991, 04 2020.
[33] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 440–450. [Online]. Available: https://aclanthology.org/P17-1041
[34] https://pytorch.org/.
[35] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Bkg6RiCqY7
[36] https://github.com/springside/springside4.git.
[37] https://github.com/openmrs/openmrs-core.git.
[38] https://github.com/apache/commons-lang.git.
[39] https://github.com/datumbox/datumbox-framework.git.
[40] https://github.com/dayatang/dddlib.git.
[41] https://www.jacoco.org/jacoco/.
[42] https://testing.googleblog.com/.
[43] S. Wang, N. Shrestha, A. K. Subburaman, J. Wang, M. Wei, and N. Nagappan, "Automatic unit test generation for machine learning libraries: How far are we?" in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1548–1560.
[44] L. Vidács and M. Pinzger, "Co-evolution analysis of production and test code by learning association rules of changes," in *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 31–36.
[45] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179
[46] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902
[47] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 7871–7880. [Online]. Available: https://aclanthology.org/2020.acl-main.703
[48] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," 09 2020.
[49] E. Dinella, G. Ryan, T. Mytkowicz, and S. Lahiri, "Toga: A neural method for test oracle generation," in *ICSE 2022*. ACM, May 2022.

Authorized licensed use limited to: Zhejiang University. Downloaded on May 28,2025 at 06:07:02 UTC from IEEE Xplore. Restrictions apply.