2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-5388-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISSRE62328.2024.00043

Fix the Tests: Augmenting LLMs to Repair Test Cases with Static Collector and Neural Reranker

Jun Liu[1,3,†], Jiwei Yan[2,§,‡], Yuanyuan Xie[1,4,†], Jun Yan[1,2,3,†] and Jian Zhang[1,3,4§,†]
[1]Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[2]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences (UCAS)
[4]School of Intelligent Science and Technology, Hangzhou Institute for Advanced Study, UCAS
Email: [†]{liuj2022, xieyy, yanjun, zj}@ios.ac.cn, [‡]yanjiwei@otcaix.iscas.ac.cn

***Abstract*—During software evolution, it is advocated that test code should co-evolve with production code. In real development scenarios, test updating may lag behind production code changing, which may cause compilation failure or bring other troubles. Existing techniques based on pre-trained language models can be directly adopted to repair obsolete tests caused by such unsynchronized code changes, especially syntactic-related ones. However, the lack of task-oriented contextual information affects the repair accuracy on large-scale projects. Starting from an obsolete test, the key challenging task is precisely identifying and constructing Test-Repair-Oriented Contexts (TROCtxs) from the whole repository within a limited token size.**

In this paper, we propose SYNTER (SYNtactic-breaking-changes-induced TEst Repair), a novel approach based on LLMs to automatically repair obsolete test cases via precise and concise TROCtxs construction. Inspired by developers' programming practices, we design three types of TROCtx: class context, usage context, and environment context. Given an obsolete test case to repair, SYNTER firstly collects the related code information for each type of TROCtx through static analysis techniques automatically. Then, it generates reranking queries to identify the most relevant TROCtxs, which will be taken as the repair-required key contexts and be input to the large language model for the final test repair.

To evaluate the effectiveness of SYNTER, we construct a benchmark dataset that contains a set of obsolete tests caused by syntactic breaking changes. The experimental results show that SYNTER outperforms baseline approaches both on textual- and intent-matching metrics. With the augmentation of constructed TROCtxs, hallucinations are reduced by 57.1%.

***Index Terms*—Software Evolution, Obsolete Test Repair, LLM, Static Analysis**

I. INTRODUCTION

Software evolution is a fundamental and significant aspect of software development [1]. For large-scale software such as *Kafka* [2], the project frequently evolves, where a new version is released in one to two weeks on average and new commits are submitted nearly every day. As production code usually changes during software evolution, it is crucial to maintain and co-evolve associated test code to ensure that they remain effective in validating the software's functionality [3], [4]. Specifically, for software invoked as libraries, it is important to co-evolve the test code to follow the production code changes, which can help developers quickly notice the backward incompatible changes that may affect its clients [5].

To automatically co-evolve production code and test code, previous studies analyze and mine the software code to extract production-test co-evolution rules and patterns [6], [7]. However, as real-world code changes come in a great many forms, they are hard to summarize into a small number of general patterns. In recent years, with the rapid advancement of machine learning and Large Language Models (LLMs) [8], [9], [10], many studies have utilized learning-based techniques to assist the production and test code co-evolution, including obsolete test case identification [11], production-test co-evolution pair extraction [12], etc., which yielded favorable results. To further reduce the developer's burden, researchers are also concerned with repairing obsolete test cases automatically. For example, Hu et al. [13] identified and updated obsolete test methods by fine-tuning a pre-trained model initialized from CodeT5 [14], which is currently the SOTA approach for repairing obsolete test cases.

For this task, though directly using learning-based techniques resulted in some positive outcomes, it still faces difficulties in complex repositories. For example, API signature changing is the most common and straightforward code-changing type during software evolution. However, focusing on this type of production change, the SOTA work CEPROT [13] fails to repair about three-quarters of test cases on real-world projects (refer to Table II in Section V-A), which means there is still some gap before existing learning-based approaches can be applied in practice.

To fill this gap, we target repairing signature-related code changes using the power of LLMs. Here, the signature-based code changes are also called **Syntactic Breaking Changes (SynBCs)** as they may lead to compilation errors if associated tests are not co-evolved. According to our investigation, accurately repairing synBC-related obsolete test cases based on LLMs faces the following key challenges.

C1: **The repair-oriented code contexts are unclear.** When developers manually repair a test case, not only the signature changes of the focal method but also many other related code contexts are considered for better understanding. When fixing tests with learning-based

[§]Corresponding authors

approaches, the repair-oriented code context must be explicitly extracted. As existing works repair test cases solely with the original and updated focal methods, they are unaware of related contexts. It is essential to determine which types of contexts are essential for accurate repair and how to extract them.

C2: **The token size of code contexts is limited.** Compared to small-scale models, LLMs have demonstrated their extraordinary capabilities. However, even though they are designed with increasingly larger context windows, it is impractical to simply include the entire repository contents as input. Moreover, extra irrelevant information may bring negative effects as well, which means that the extracted code context is not the more the better. As the number of context tokens for LLM input is restricted, for all the code contexts that may have relations with the changed signature, it is required to sort and pick out the most relevant ones as input.

To this end, we propose SYNTER (SYNtactic-breaking-changes-induced TEst Repair, originally called SYNBCIATR), a novel approach for repairing obsolete test cases caused by SynBCs at the method level. To address challenge ***C1***, SYNTER designs three types of contexts, including Class Contexts (*ClassCtxs*), Usage Contexts (*UsageCtxs*), and Environment Contexts (*EnvCtxs*) as **Test-Repair-Oriented Contexts (TROCtxs)**. These contexts focus on different aspects of changed codes to provide adequate contextual information for test repair. For challenge ***C2***, SYNTER identifies and constructs qualified TROCtx from the repository in two stages. ❶ First, SYNTER collects all types of TROCtx by static code analysis, specifically, using the *Language Server* [15]. ❷ Then, inspired by the idea of Retrieval-Augmented Generation (RAG) [16], SYNTER generates reranking queries to identify the most relevant TROCtxs in each type according to the repair requirement extracted from the original test case, which is based on *Neural Rerankers* [17]. After constructing TROCtxs, SYNTER aggregates the test-repair-required information to a final prompt and generates the repaired test case.

To evaluate the effectiveness of SYNTER, we construct a benchmark dataset based on existing work which consists of 136 samples with diverse SynBCs. The evaluation is based on both the textual match and intent match metrics. In terms of the textual match, SYNTER achieves the best performance against baselines specifically on CodeBLEU (83.3), DiffBLEU (46.7), and Accuracy (32.4%). In terms of the intent match, we conduct a human evaluation on verifying whether test cases are correctly repaired without changing their original intents, in which SYNTER correctly repairs 90.4% test cases, achieving improvements of 248.6% and 9.8% when compared to CEPROT and NAIVELLM respectively. Moreover, SYNTER is capable of reducing 57.1% hallucinations caused by NAIVELLM.

We make the following contributions in this paper:

- We design three types of TROCtx to provide adequate contextual information for repairing obsolete test cases caused by SynBCs.
- We propose SYNTER to construct TROCtx by combining the static collector and neural reranker, which is utilized to enhance the repairing ability of naive LLM.
- Experimental results on the benchmark dataset demonstrate that SYNTER can repair obsolete test cases caused by SynBCs more effectively compared to both CEPROT and NAIVELLM.

The data and code are both publicly available at: *https://github.com/nonsense-j/SynTeR*.

II. BACKGROUND AND MOTIVATION

A. Task Definition

Referring to previous studies [5], [18], [19], Syntactic Breaking (SynB) issues represent signature-based compilation errors in API evolution, such as *ClassNotFoundException* and *NoSuchMethodError*. In this paper, we name method signature changes that may cause SynB issues as Syntactic Breaking Changes (SynBCs).

We define the signature of a method (**Method Signature**) as a 5-tuple $ms = \langle n, P, r, M, E \rangle$, where:

- n is the name of the method;
- P is a list of parameter types;
- r is the type of the return value;
- M is a set of modifiers;
- E is a set of exception types that can be thrown.

Given a method that changes from m to m', we define a **Syntactic Breaking Change (SynBC)** as:

$$m \xrightarrow{\text{SynBC}} m' \quad \text{iff } m.ms \neq m'.ms,$$

where $m.ms$ and $m'.ms$ denote the method signatures before and after the change respectively. To determine whether SynBCs are common in production-test co-evolution, we conducted an empirical study to count the SynBCs in the breaking change dataset released by CEPROT [13]. Each sample in the dataset contains a change of focal method on a real-world commit from GitHub. It reveals that signature-based focal changes occur in over 40% samples of the dataset, in which parameter and return type-related changes account for a large proportion.

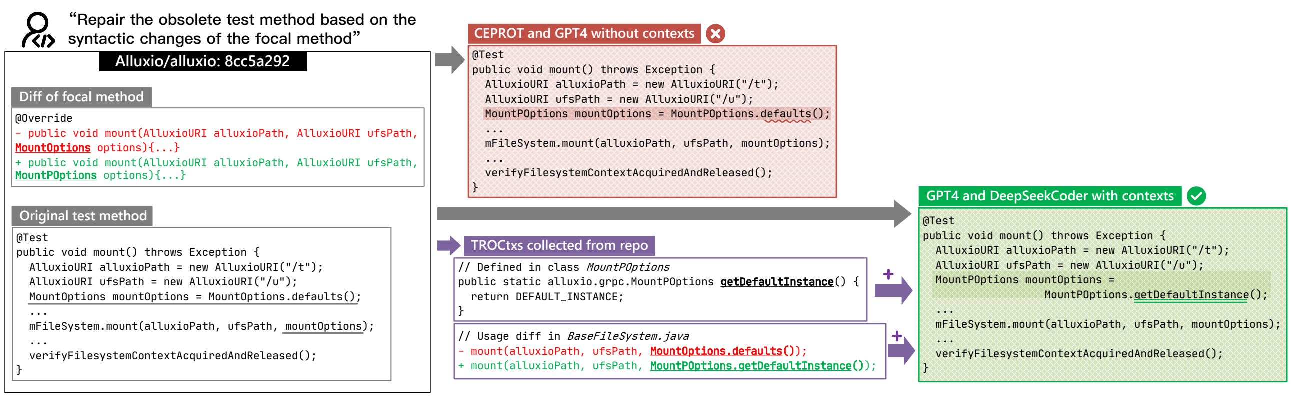
Specifically, we further categorize SynBCs into the following three types according to the changed elements in the method signature. Note that, a SynBC whose parameter and return types in the method signature both change is a ParamSynBC and RetSynBC at the same time.

- **Parameter-related Syntactic Change (ParamSynBC)**. For a SynBC from m to m', if the parameter type list in the method signature changes, it is a ParamSynBC.

$$m \xrightarrow{\text{ParamSynBC}} m' \quad \text{iff } m.ms.P \neq m'.ms.P$$

- **Return-related Syntactic Change (RetSynBC)**. For a SynBC from m to m', if the type of the return value in the method signature changes, it is a RetSynBC.

$$m \xrightarrow{\text{RetSynBC}} m' \quad \text{iff } m.ms.r \neq m'.ms.r$$


“Repair the obsolete test method based on the syntactic changes of the focal method”
Alluxio/alluxio: 8cc5a292
Diff of focal method
Original test method
CEPROT and GPT4 without contexts
TROCtxs collected from repo
GPT4 and DeepSeekCoder with contexts

```
@Override
- public void mount(AlluxioURI alluxioPath, AlluxioURI ufsPath,
MountOptions options){...}
+ public void mount(AlluxioURI alluxioPath, AlluxioURI ufsPath,
MountPOptions options){...}
```

```
@Test
public void mount() throws Exception {
  AlluxioURI alluxioPath = new AlluxioURI("/t");
  AlluxioURI ufsPath = new AlluxioURI("/u");
  MountOptions mountOptions = MountOptions.defaults();
  ...
  mFileSystem.mount(alluxioPath, ufsPath, mountOptions);
  ...
  verifyFilesystemContextAcquiredAndReleased();
}
```

```
@Test
public void mount() throws Exception {
  AlluxioURI alluxioPath = new AlluxioURI("/t");
  AlluxioURI ufsPath = new AlluxioURI("/u");
  MountPOptions mountOptions = MountPOptions.defaults();
  ...
  mFileSystem.mount(alluxioPath, ufsPath, mountOptions);
  ...
  verifyFilesystemContextAcquiredAndReleased();
}
```

```
// Defined in class MountPOptions
public static alluxio.grpc.MountPOptions getDefaultInstance() {
  return DEFAULT_INSTANCE;
}
```

```
// Usage diff in BaseFileSystem.java
- mount(alluxioPath, ufsPath, MountOptions.defaults());
+ mount(alluxioPath, ufsPath, MountPOptions.getDefaultInstance());
```

```
@Test
public void mount() throws Exception {
  AlluxioURI alluxioPath = new AlluxioURI("/t");
  AlluxioURI ufsPath = new AlluxioURI("/u");
  MountPOptions mountOptions =
                MountPOptions.getDefaultInstance();
  ...
  mFileSystem.mount(alluxioPath, ufsPath, mountOptions);
  ...
  verifyFilesystemContextAcquiredAndReleased();
}
```

Fig. 1: A motivating example with a ParamSynBC collected from a commit (8cc5a292) of Alluxio/alluxio. The original test method can be correctly repaired only when we provide precise Test-Repair-Oriented Contexts (shown as TROCtxs) to LLMs.

- **Normal Syntactic Change (NormSynBC)**. For a SynBC from m to m', if it does not belong to ParamSynBC or RetSynBC type, it is a NormSynBC.

$$m \xrightarrow{\text{NormSynBC}} m' \quad \text{iff } m.ms \neq m'.ms \wedge$$
$$m.ms.P = m'.ms.P \wedge m.ms.r = m'.ms.r$$

Here, for a SynBC that happens between the original focal method m and the updated focal method m', ro' denotes the repository code that the method m' belongs to, and t denotes the obsolete test case associated with m. Based on this knowledge, the test-repair task hopes to get the repaired test case t' as the final output. The whole task consists of two main steps. First, we extract the repair-oriented contexts for test t from repository ro' according to the SynBC from m to m'. That process can be denoted as:

$$Construct(m, m', ro', t) = C,$$

where C represents the constructed contexts. Then, we repair the obsolete test with the constructed contexts C, which can be expressed as:

$$Repair(m, m', C, t) = t',$$

where t' denotes the final repaired test case.

B. Language Server and Neural Reranker

A *Language Server* consists of programming language-specific tools like static analyzers and compilers. In modern Integrated Development Environments (IDEs), language servers provide language-specific features like ‘*autocomplete*’, ‘*goto definition*’, ‘*find usages*’, and others [20]. Recently, the Microsoft team has created a standard JSON-RPC-based protocol, called **Language Server Protocol (LSP)** [15], based on which multiple IDEs can communicate with the same language server to access intelligent programming features.

A *Neural Reranker* is a type of machine learning model [21] used to reorder a given set of documents based on their relevance to a given query, which is the initial request for information expressed as keywords or complex expressions [17]. It is widely used in research fields such as information retrieval, natural language processing, and recommendation systems. Instead of ranking based on simple heuristics like the frequency of query terms appearing in the query, neural rerankers are trained to take into account more complex features, like the semantic similarity between the query and the documents.

C. Motivating Example

The example in Fig. 1 is used to demonstrate our motivation. It was collected from a real commit in project *Alluxio* [22]. As shown in the given commit, there is a ParamSynBC for the focal method (named `mount`), where the third parameter changes from `MountOptions` to `MountPOptions`. If the associated original test case (also named `mount`) does not co-evolve with the change of the focal method, the test will fail for compilation errors (*cannot resolve type*).

To automatically repair the obsolete test method, we first apply existing learning-based techniques directly. However, both CEPROT and GPT-4 fail due to using an undefined method (`MountPOptions.defaults()`) to construct the third parameter, where the correct method invocation should be `MountPOptions.getDefaultInstance()`. This hallucination occurs for lacking Test-Repair-Oriented Contexts (TROCtxs) in the input. After providing the required contexts shown in Fig. 1, by including either the definition of `getDefaultInstance` in class `MountPOptions` or the usage change of focal method in other production code, LLMs can generate the correct repaired test.

When developers are asked to repair test for this case in IDEs, it is convenient for them to refer to the related contexts (TROCtxs in Fig. 1) with the programming features (such as ‘*goto definition*’ and ‘*find usages*’) provided by language servers. Inspired by these practices, SYNTER constructs TROCtxs by simulating developers’ behaviors in IDEs. Specifically, SYNTER collects related contexts by interacting with the language server and filters out unnecessary ones based on neural rerankers. Finally, SYNTER uses LLMs to repair obsolete test cases with constructed TROCtxs.

III. Methodology

In this section, we first show the overall framework of SYNTER. Then we demonstrate all the types of TROCtxs identified by SYNTER. Finally, we introduce the technical details of the main modules in SYNTER.

A. Overview

The overall pipeline of SYNTER is depicted in Fig. 2. Given the change of the focal method and the obsolete test method as inputs, SYNTER consists of three major steps. **(1) Collecting TROCtxs**: SYNTER analyzes all the related contexts from inputs and requests the language server to collect and process them into candidate chunks; **(2) Reranking TROCtxs**: SYNTER reranks candidate chunks with queries constructed from the inputs; **(3) Generating full prompt**: SYNTER aggregates the inputs and final TROCtxs to generate the full prompt, which is used to repair the test with LLM. We also provide a detailed overview in Fig. 3.

B. Types of TROCtx

Based on the characteristics of the test-repair task, we categorize TROCtxs into three types: Class Contexts (ClassCtxs), Usage Contexts (UsageCtxs), and Environment Contexts (EnvCtxs). In alignment with real-world developers' practices, these categories of contexts can provide comprehensive and sufficient contextual information for test repair.

★ **ClassCtxs** include the member accesses (method and field accesses) of a specific class and its parent classes. These contexts indicate the accurate operations supported by a given class type, serving to alleviate the hallucination of LLMs. Since new class types can be introduced in parameter types and the return type, ClassCtxs will be collected specially for ParamSynBCs and RetSynBCs. For example, the ClassCtxs for the case caused by a ParamSynBC in Fig. 1 are partly demonstrated in Listing 1.

```
// defined in MountPOptions and its parent classes
// methods are simplified as signatures
...
public static final int READONLY_FIELD_NUMBER = 1;
public boolean hasReadOnly();
public static MountPOptions getDefaultInstance();
...
```

Listing 1: ClassCtxs of the new class `MountPOptions`.

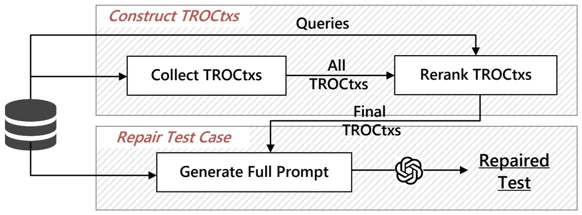


Fig. 2: Overall pipeline of SYNTER.

★ **UsageCtxs** include the changes of usages for the focal method in the diff format. Usages of the updated focal method in other parts of the repository can illustrate how to properly call the method in the associated test. UsageCtxs will be collected for all the types of SynBCs. For example, the UsageCtxs for case introduced in Fig. 1 are partly demonstrated in Listing 2.

```
...
- mount(alluxioPath, ufsPath, MountOptions.defaults()
+ mount(alluxioPath, ufsPath,
↪  MountPOptions.getDefaultInstance()
...
```

Listing 2: UsageCtxs of the focal method `mount()`.

★ **EnvCtxs** include the environmental changes of the focal and test method in the diff format. Given a method m, we define the class containing it and its parent classes as the environment of m. Only code changes external to m in the environment will be collected to construct its EnvCtxs. These contexts can indicate updates of related identifiers and similar change patterns. EnvCtxs will be collected for all the types of SynBCs. For example, the EnvCtxs of the focal method for the case introduced in Fig. 1 are partly demonstrated in Listing 3.

```
// code diffs of class containing mount and its parents
...
- return openFile(path, OpenFileOptions.defaults());
+ return openFile(path,
↪  OpenFileOptions.getDefaultInstance());
...
```

Listing 3: EnvCtxs of of the focal method `mount()`.

SYNTER aims to collect and rerank different types of TROCtxs respectively. In the following subsections, we will describe the details of how each type of TROCtx is collected, reranked, and ultimately aggregated as displayed in Fig. 3.

C. Collecting TROCtxs

To collect repo-level contexts, SYNTER interacts with language servers via *LSP* [15]. Every request message conforms to the JSON-RPC-based protocol, which consists of the request type (such as '*goto definition*') and the cursor position of the request identifier, including the file path and the indices of the line and column for the identifier within the file.

For every type of TROCtx, SYNTER first automatically analyzes the inputs and locate **key identifiers** (identifiers that need to find definitions and references by LSP) with their positions using the static parser *tree-sitter*[1]. The required contexts are further collected through the language server using the python library *multilspy*[2].

[1]https://tree-sitter.github.io/tree-sitter/
[2]https://github.com/microsoft/monitors4codegen

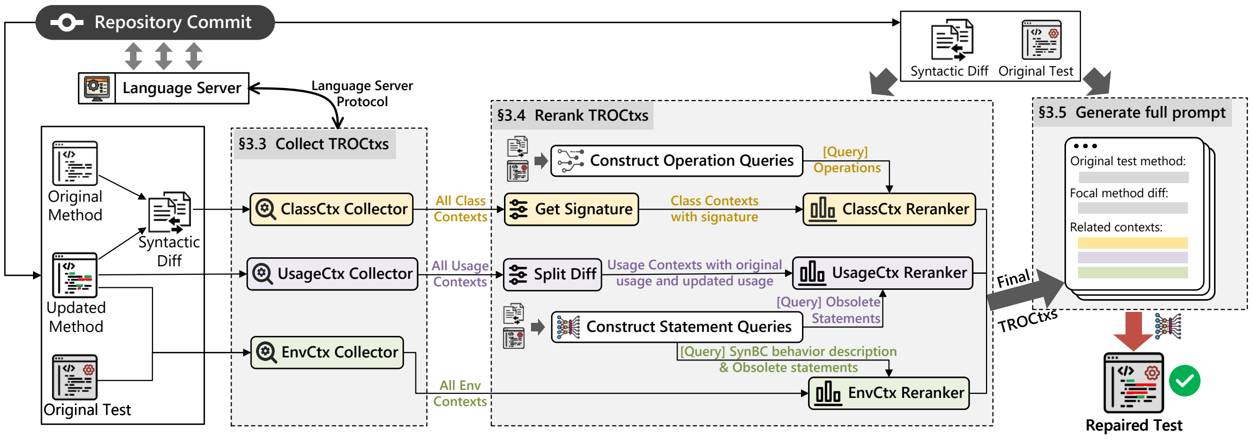


Fig. 3: Overview of SYNTER.

*1) **Collecting ClassCtxs**:* Based on the syntactic diff of the focal method, SYNTER first identifies the new class types introduced in the method signature of the updated version (both parameter types and the return type will be checked). These new class types are considered as key identifiers for collecting ClassCtxs. Through requesting the language server, SYNTER collects the definitions of these new class types as well as their parent classes. To split the collected contexts into identical chunks, SYNTER cleans up all comments and the definition body is divided into member access operations. Only non-private declarations of fields and methods are collected. Specifically, for classes and private field accesses with Lombok annotations[3](`@Data`, `@GETTER`, `@SETTER`), SYNTER also retains their related private field accesses.

For the case in Fig. 1 as an example, the updated focal method uses a new parameter type (`MountPOptions`). SYNTER will collect all the declarations of field and method to construct ClassCtxs, as demonstrated in Listing 1.

*2) **Collecting UsageCtxs**:* For UsageCtxs, the name identifier of the updated focal method is considered as the key identifier. The collecting procedures are described in Algorithm 1. On line 3, SYNTER locates the name identifier of the updated focal in the repository. On line 4, all the usages of the updated focal method are fetched by requesting the language server. For each usage, SYNTER collects the usage-diff texts (diffs that contain the change of usage) in two steps, generating the diff (lines 6- 10) and gathering the required contexts (lines 11-18). SYNTER formats the files that contain usages before generating diff to avoid missing information. Also, since usage-diff of invocation is not sufficient for ParamSynBCs and RetSynBCs, SYNTER collects additional backward and forward surrounding contexts respectively (lines 13 and 16).

For the case in Fig. 1, SYNTER finds ten usages in the repository. After filtering out usages in comments and repeated ones, four usage-diff texts are collected, one of which is demonstrated in Listing 2.

[3]https://projectlombok.org/features/

Algorithm 1 Algorithm of Collecting UsageCtxs

```
Input: m_ori: original focal method, m_upd: updated focal method,
    p: focal relative path, lsp: object interacting with the language
    server, repo: object interacting with the repository commit
Output: UsageCtxs: a set of usage contexts in diff format
 1: Initialize UsageCtxs as an empty set
 2: synbc ← getSynBC(m_ori, m_upd)
                              ▷ get the syntactic changes of the focal method
 3: pos ← getMethodNamePos(repo.getSrcFile(p), m_upd)
                    ▷ get the cursor position of the name identifier in m_upd
 4: usages ← lsp.requestUsages(p, pos)
                          ▷ request the language server for usages of m_upd
 5: for all usage in usages do
 6:     uf_ori ← repo.getSrcFile(usage.relpath)
 7:     uf_upd ← repo.getTgtFile(usage.relpath)
                       ▷ get the original and updated files that contain usage
 8:     uff_ori ← format(uf_ori)
 9:     uff_upd, pos_fmt ← formatWithCursor(uf_upd, pos)
                 ▷ format the original and updated files (pos also updates)
10:     udiffs ← generateDiff(uff_ori, uff_upd)
11:     utext ← collectInvokeStmt(udiffs, pos_fmt)
                         ▷ initialize usage-diff text with invocation statement
12:     if synbc contains change in parameter types then
13:         utext ← collectBeforeCtx(udiffs, pos_fmt) + utext
14:     end if▷ enrich usage-diff text with contexts before invocation

15:     if synbc contains change in return type then
16:         utext ← utext + collectAfterCtx(udiff, pos_fmt)
17:     end if ▷ enrich usage-diff text with contexts after invocation
18:     UsageCtxs.add(utext)
19: end for
```

*3) **Collecting EnvCtxs**:* EnvCtxs represent additional contexts indicating environmental changes, namely env-diff texts. SYNTER collects EnvCtxs for the focal and test method respectively. Specifically, the class file to which the method belongs and its parent classes are treated as the method's environment. Here, the class name identifiers are the key identifiers, which are used to find out the related parent classes and collect their diffs as env-diff texts. Also, diffs are generated after cleaning comments and reformatting.

For the case in Fig. 1, the environment of the test method remains unchanged. Therefore, only environmental changes of the focal method are collected as env-diff texts to constructed EnvCtxs, which are partly shown in Listing 3.

D. Reranking TROCtxs

SYNTER utilizes neural rerankers to filter and retain the most relevant contexts. As the test cases should be functionally consistent before and after the evolution, they have high similarity. Thus, our primary idea of rerankers is to make use of the original test to construct queries for TROCtx reranking. For the case in Fig. 1, from the collected ClassCtxs of the new class `MountPOptions`, we hope to filter out unrelated member accesses and precisely retain the required method declaration "`MountPOptions.getDefaultInstance()`". Given the text '`MountOptions.defaults()`" as a query, we can rerank the ClassCtxs to get the most similar APIs with neural rerankers. As shown in Listing 4, the reranking result reveals that the required API "`MountPOptions.getDefaultInstance()`" is ranked with the highest score under the given query.

```
// methods are simplified as signatures
public static MountPOptions getDefaultInstance(); // top1
public MountPOptions getDefaultInstanceForType(); // top2
...
```

Listing 4: Reranked ClassCtxs of class `MountPOptions`.

*1) **Constructing Queries**:* Based on the above ideas, the quality of the query texts decides the reranking results. However, as we have three types of TROCtxs but their formats (or granularity) are different, SYNTER reranks each type of TROCtxs with distinct queries instead of using a general one. In terms of granularity for TROCtx, ClassCtx is a fine-grained one where each context is a specific member access operation within the corresponding class (Listing 1), while UsageCtx and EnvCtx are coarse-grained ones whose contexts directly refer to the diff texts of statements (Listings 2 and 3). Overall, SYNTER constructs two kinds of corresponding queries, operation queries and statement queries.

Constructing operation queries. We define operation as member access to a given class. Operation queries are used to rerank method and field declarations in ClassCtxs for new class types. Based on our insight, the operations used in the original test can be reused as queries, namely operation queries, to rerank ClassCtxs. For new class types in ParamSynBCs and RetSynBCs, SYNTER constructs fine-grained queries by extracting operations that could potentially be accessed in the repaired test as shown in Algorithm 2. Specifically, SYNTER constructs operation queries for new parameter and return class types respectively.

To construct operation queries for collected ClassCtxs in ParamSynBCs, starting from the invocation of the focal method in the original test, SYNTER extracts backward operations related to the obsolete parameters, which are modified

Algorithm 2 Algorithm of Constructing Operation Queries

```
Input: $\mathsf{m_{ori}}$: original focal method, $\mathsf{m_{upd}}$: updated focal method, $\mathsf{t}$:
    original test method
Output: $\mathsf{OpQueries}$: a tuple of queries consisting of operations for
    new parameter and return class types
1:  Initialize ParamOpQ, RetOpQ as empty sets
2:  $\mathsf{synbc} \leftarrow getSynBC(\mathsf{m_{ori}}, \mathsf{m_{upd}})$
3:  if synbc is a ParamSynBC then
4:      for all arg in $getObsArgs(\mathsf{m_{ori}}, \mathsf{m_{upd}})$ do
5:          $\mathsf{op} \leftarrow getSetOp(\mathsf{arg})$
                ▷ construct operation queries for obsolete parameters
6:          $\mathsf{ParamOpQ}.add(\mathsf{op})$
7:      end for
8:      $\mathsf{ops} \leftarrow backwardParamsOps(\mathsf{t}, \mathsf{synbc})$
            ▷ backward analysis to extract operations based on synbc
9:      $\mathsf{ParamOpQ}.update(\mathsf{ops})$
10: end if
11: if synbc is a RetSynBC then
12:     $\mathsf{ops} \leftarrow forwardReturnOps(\mathsf{t}, \mathsf{synbc})$
            ▷ forward analysis to extract operations based on synbc
13:     $\mathsf{RetOpQ}.update(\mathsf{ops_o})$
14: end if
15: $\mathsf{OpQueries} \leftarrow (\mathsf{ParamOpQ}, \mathsf{RetOpQ})$        ▷ aggregate queries
```

during the given SynBC. Since the obsolete parameters may be refactored to be set by new parameters, SYNTER first adds additional operation queries in the form as `set_xxx` on lines 4- 6. Then, on line 8, SYNTER traverses the def-use chains [23] of the obsolete parameters to extract directly used operations, including method accesses and field accesses. These operations are also collected as operation queries.

To construct operation queries for collected ClassCtxs in RetSynBCs, starting from the invocation of the focal method in the original test, SYNTER applies forward propagatability analysis by traversing the references of the return object for the focal method. During the analysis, related operations of the return object are extracted as the operation queries.

For the focal method in Fig. 1, only the parameter types changed. Therefore, SYNTER only extracts operation queries for the newly added parameter class type `MountPOptions`, which is shown as follows.

▷ ParamOpQ: {'*MountOptions.defaults()*', '*setOptions()*'}

Constructing statement queries. SYNTER constructs coarse-grained queries for reranking diff texts of statements. In our settings, diff texts are collected after reformatting so that every line in the diff contains a separate statement. Since we only keep changed parts in diff texts, we should extract obsolete statements from the original test as the query, namely statement query, to rerank diff texts in UsageCtxs and EnvCtxs.

As recent studies have shown that LLM specializes in code summarizing and understanding tasks [24], [25], SYNTER uses an LLM to extract obsolete statements from the original test. Following existing works [26], [27], we adopt LLMs with *Few-Shots Learning* and *Chain-Of-Thought* prompting to identify obsolete statements with the syntactic change of the focal method and original test method as input. The LLM is first asked to summarize the syntactic differences of the focal method, and then find the obsolete test statements. Both the **SynBC behavior description** (in natural language) and **obsolete statements** (stmts) are collected as coarse-grained statement queries. Specifically, the SynBC behavior description is used as coarse queries for reranking the EnvCtxs of the focal method, while the EnvCtxs of the test method are reranked with the obsolete statements.

For the example in Fig. 1, the extracted statement queries are demonstrated as follows.

▷ SynBC behavior description: "*The method mount() has been updated to accept an object of type 'MountPOptions' instead of 'MountOptions' as its third parameter.*"

▷ obsolete stmts: "*MountOptions mountOptions = MountOptions.defaults(); mFileSystem.mount(alluxioPath, ufsPath, mountOptions;*"

*2) **Reranking TROCtxs with Queries**:* SYNTER reranks each type of TROCtxs with different queries as shown in Fig. 3. During a single reranking process, the top three most relevant contexts are retained by default.

To rerank ClassCtxs, first, all the constructor declarations in ClassCtxs are retained since they are necessary to construct instances for the class. Then, for other method and field declarations, SYNTER transforms method declarations in ClassCtxs into their signatures before reranking. According to the SynBC, ClassCtxs of new parameter and return class types are reranked with ParamOpQ and RetOpQ respectively.

To rerank UsageCtxs, every usage-diff text in the UsageCtxs is divided into original usage (by removing added lines in diffs) and updated usage (by removing deleted lines in diffs). SYNTER reranks both the original and updated usages with obsolete statements as query, in which the maximum reranking score determines the relevance of the given usage-diff text.

To rerank EnvCtxs, EnvCtxs of the focal method are directly reranked with the SynBC behavior description, while EnvCtxs of the test method are reranked with the obsolete statements.

E. Generating Full Prompt

The full prompt consists of the unified diff of the focal method, the original test, and the related contexts (TROCtxs).

Following existing works using LLM for code tasks [27], [28], SYNTER cleans comments in codes to avoid influencing inferring program intentions. Then, SYNTER generates the unified diff of the focal method after formatting.

The general structure of the final prompt is demonstrated in Fig. 3. Specifically, we start by setting up the system role of LLM as an expert in Java software evolution and briefly describe the test-repair task caused by syntactic changes of the focal method. Then, we introduce the original test that needs to be repaired followed by the unified diff of the focal method. At last, we categorize TROCtxs according to their types and ask LLM to treat these contexts as references. For each group of contexts, we also provide the basic description. For the case in Fig 1 as an example, we automatically generate a description ('Defined in class MountPOptions') for ClassCtxs of `MountPOptions`.

Finally, SYNTER requests the LLM with the generated full prompt to generate the repaired test.

IV. EXPERIMENTAL SETUP

Before the evaluation, in this section, we first introduce the construction of our benchmark dataset and then describe the baselines and metrics used in the evaluation. Finally, we will briefly show the implementation settings of our approach.

A. Benchmark Datasets

In this study, we focus on repairing obsolete test cases caused by syntactic changes in the focal method. To the best of our knowledge, CEPROT is the first and SOTA learning-based approach to co-evolve test cases at the method level [13]. Therefore, we reuse and refine the dataset provided by CEPROT, in which all the samples are collected from the top 1,500 Java projects on GitHub by Liu et al. [29].

The evaluation dataset, i.e., testing dataset, of CEPROT contains 520 samples extracted from 128 real-world Java projects. Based on that, we first filter out the samples without syntactic changes in the focal method. After this step, 211 samples remain. Then, to ensure the high quality of the dataset, we manually *filter noisy samples* and *augment new samples* referring to the corresponding repository commit.

Filtering noisy samples. Samples with the following characteristics are filtered out as noises in the dataset:

- the test method is incomplete or not yet implemented;
- the focal method is incomplete with only the signature;
- the focal method is not used in the given test.

We manually resolve the problems above by extracting correct co-evolution pairs with SynBCs from the repository.

Augmenting new samples. To improve the diversity and generality of the dataset, we mine new samples to enrich the dataset. Specifically, we augment at most two samples from the real-world repository with different changing patterns for each commit in the original dataset.

To sum up, the final benchmark dataset consists of 137 samples from 54 projects. As shown in Fig. 4, the samples are diverse in the types of associated SynBCs.

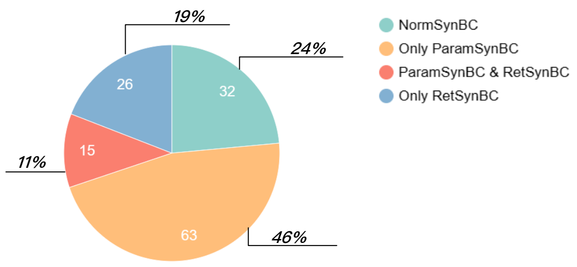


Fig. 4: Distribution of samples based on the type of SynBCs.

B. Baselines

To assess the effectiveness of SYNTER, we consider two baselines from existing studies and our investigation, namely CEPROT and NAIVELLM.

CEPROT is the SOTA learning-based approach in updating obsolete test cases. According to Hu et al. [13], CEPROT is built on a code language model fine-tuned from CodeT5 and outperforms previous techniques. Taking notes that CEPROT needs edit sequences of the focal method as input, we reproduce it with *clang-format*[4] and *difflib*[5]. With the replication package of CEPROT, we retrained the model and saved the best checkpoint with the highest F1. According to the result of our replication (accuracy: 11.9%), the performance is consistent with the statistics in the paper (accuracy: 12.3%) [13].

NAIVELLM is developed based on LLM without related contexts. Compared with CodeT5, LLM is trained on huge sets of data with a much larger number of parameters. Therefore, LLM is more intelligent in general code tasks. In NAIVELLM, we directly use LLM to repair the test case with the focal diff (the unified diff of original and updated focal methods) and the original test method as input. Following the pre-processes in SYNTER, comments in codes are cleaned and all the filtered codes are reformatted to a standardized style.

C. Metrics

Based on previous studies [13] [30], we design metrics to measure the quality of repaired test cases from two aspects, and finally get five metrics.

Textual Match. We use three specific metrics to measure how the generated code is close to the ground truth. **(1) CodeBLEU.** Developed from the classical machine translation evaluation metric *BLEU* [31], *CodeBLEU* [32] is widely used in evaluating code generation tasks by measuring the similarity with code semantics. Thus, we use CodeBLEU to assess the similarity between the generated method and the ground truth. **(2) DiffBLEU.** As only a part of statements are modified, we design the metric *DiffBLEU* to concentrate on the modified ones specifically. To compute it, we calculate the BLEU score of the repaired test whose unchanged statements are removed. DiffBLEU serves as a complement to CodeBLEU. **(3) Accuracy (%).** This metric represents the percentage of samples where the generated method is identical to the ground truth (exactly match textually).

Intent Match. Besides textual match metrics, we also design two other specific metrics to measure whether the generated repaired test cases are correct. Ideally, test repair should not change the intent of the test case, which means that the repaired test should keep the same assertions and input values. Referring to the original test and the ground truth respectively, we use two specific metrics to evaluate the *repairability*. **(1) Repairability$_{ori}$ (%).** Repairability$_{ori}$ represents the repairability referring to the original test. Specifically, we calculate Repairability$_{ori}$ as the percentage of samples where the generated test can be successfully compiled and shares the same assertions and input values with *the original test*. **(2) Repairability$_{gt}$ (%).** Similar to Repairability$_{ori}$, Repairability$_{gt}$ is calculated referring to the ground truth. For each repaired test case, two developers with more than three years of Java programming experience are asked to perform manual evaluation separately. If they can't reach an agreement, they will have a discussion on uncertain samples until they agree on consistent conclusions.

In addition, we also record the **Syntax Pass Rate (SPR)** and **Compilation Pass Rate (CPR)** as the success rate of syntax checking and compilation for generated codes respectively.

[4]https://clang.llvm.org/docs/ClangFormat.html

[5]https://docs.python.org/zh-cn/3/library/difflib.html

D. Implementation Settings

SYNTER is built based on LangChain [33], which is a framework designed to simplify the procedures of developing applications powered by LLMs. With the APIs provided by LangChain, SYNTER utilizes the SOTA LLM (*GPT-4*) [34] developed by OpenAI and the widely-used open-source neural reranker (*bge-reranker-v2-m3*) [35] released by BAAI [36]. To reduce the impact of randomness, for each case in the dataset, we request LLM with the temperature as 0.1 three times and keep the best one for both NAIVELLM and SYNTER in evaluation.

V. EVALUATION

Based on the constructed benchmark dataset, we evaluate the effectiveness of our approach and conduct a comprehensive analysis of the results. We address these research questions:

- **RQ1:(*Effectiveness of* SYNTER)** Can SYNTER effectively repair obsolete tests caused by SynBCs?
- **RQ2:(*Effectiveness of TROCtx*)** To what extent do the TROCtxs contribute to correctly repairing the test?
- **RQ3:(*Failure Analysis*)** Under which cases does SYNTER fail to repair?
- **RQ4:(*Efficiency*)** What is the efficiency of SYNTER?

A. RQ1: Effectiveness of SYNTER

1) Basic Evaluation on Effectiveness: To evaluate the performance of SYNTER compared with baselines, we adopt different approaches to repair obsolete test cases in the benchmark dataset.

First, we conduct *syntax validation* on the generated test codes. As shown in Tab. I, the Syntax Pass Rate (SPR) of CEPROT is 47.8%, while LLM-based approaches all pass the validation (100%), which means that LLM demonstrates a higher proficiency in producing syntactically accurate code than CodeT5. Then we use the three textual metrics to measure the average similarity between the generated test code and the ground truth. The column ***Textual Match*** in Tab. I demonstrates that our approach outperforms baselines in terms of all three metrics. Specifically, LLM-based approaches have average improvements of 12.4% and 72.4% in terms of CodeBLEU and DiffBLEU compared with CEPROT, which indicates that LLM is more capable of understanding the semantics of code and generating repaired tests. With the highest scores on CodeBLEU and DiffBLEU, SYNTER is also able to accurately repair 32.4% of the test cases, which also achieves varying degrees of improvement over CEPROT (4.4%) and NAIVELLM (28.7%).

TABLE I: Effectiveness of repairing obsolete test cases caused by SynBCs based on textual match.

Approach	SPR(%)	*Textual Match*		
		CodeBLEU	**DiffBLEU**	**Accuracy(%)**
CEPROT	47.8%	73.5	26.3	4.4%
NAIVELLM	**100%**	81.9	44.0	28.7%
SYNTER	**100%**	**83.3**	**46.7**	**32.4%**

2) Human Evaluation on Effectiveness: Since textual metrics focus on measuring the similarity of tokens and AST structure for the given codes, they can not well represent the correctness of repair. To bridge this gap, in this part, we replace the obsolete test cases with the generated repaired ones in the repository and manually check whether the replaced test is correct or not, where a correct repair should pass the compilation and keep the intent of the test unchanged.

The column ***CPR*** in Tab. II represents the success rate of compilation after repairing the original test case in the repository. As a result, SYNTER fixes the most compilation errors caused by SynBCs. In terms of the two metrics of repairability, SYNTER both outperforms the baselines. As shown in the column ***Intent Match***, we can observe that SYNTER correctly repair 75.0% (102/136) and 90.4% (123/136) test cases in alignment with the intent of the ground truth and the original test respectively.

As for Repairability$_{ori}$ specifically, SYNTER achieves improvements of 248.6% and 9.8% when compared to CEPROT and NAIVELLM respectively.

Answering RQ1: SYNTER outperforms the baselines on all the metrics in the benchmark dataset. It indicates that our approach can effectively help developers to correctly repair obsolete test cases caused by SynBCs.

B. RQ2: Effectiveness of TROCtx

As introduced in Section I, SYNTER constructs TROCtxs to enhance LLMs, which serves as contextual information to reduce hallucinations during the generation of LLM (LLM below all refers to *GPT-4*). In this research question, we focus on illustrating the effectiveness of TROCtxs. Therefore, we compare our proposed SYNTER with NAIVELLM based on the remaining hallucinations.

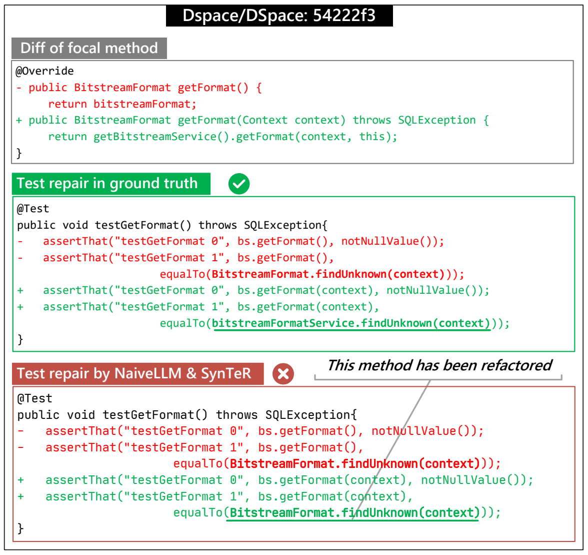
According to existing work [37] and our investigation, we define hallucinations as using undefined methods, variables, or classes in the LLM-generated codes. Based on the results of human evaluation, we manually identify hallucinations and summarize them into two types: common hallucinations and outdated hallucinations.

Common hallucinations. Common hallucinations are caused by the lack of direct contextual information from the change of focal method signature. For the case in Fig. 1, a new class (`MountPOptions`) is set as the third parameter type. Therefore, the LLM falls into common hallucination without the required class contexts of `MountPOptions`.

TABLE II: Effectiveness of repairing obsolete test cases caused by SynBCs based on intent match.

Approach	CPR(%)	*Intent Match*	
		Repairability$_{gt}$(%)	**Repairability$_{ori}$(%)**
CEPROT	33.3%	19.9% *(27)*	25.7% *(35)*
NAIVELLM	88.5%	69.1% *(94)*	82.4% *(112)*
SYNTER	**96.2%**	**75.0%** ***(102)***	**90.4%** ***(123)***

Outdated hallucinations. Outdated hallucinations are caused by the lack of implicit contextual information beyond the focal method signature. For the example in Fig. 5, the method `findUnknown` used in the test is refactored to be accessed from an instance of class `BitstreamFormat`. Without this specific knowledge, the test generated by LLM still uses the method in an outdated way, which causes a hallucination.


Dspace/DSpace: 54222f3
Diff of focal method
Test repair in ground truth
Test repair by NaiveLLM & SynTeR
This method has been refactored

```
@Override
- public BitstreamFormat getFormat() {
      return bitstreamFormat;
+ public BitstreamFormat getFormat(Context context) throws SQLException {
      return getBitstreamService().getFormat(context, this);
}
```

```
@Test
public void testGetFormat() throws SQLException{
-   assertThat("testGetFormat 0", bs.getFormat(), notNullValue());
-   assertThat("testGetFormat 1", bs.getFormat(),
                     equalTo(BitstreamFormat.findUnknown(context)));
+   assertThat("testGetFormat 0", bs.getFormat(context), notNullValue());
+   assertThat("testGetFormat 1", bs.getFormat(context),
                     equalTo(bitstreamFormatService.findUnknown(context)));
}
```

```
@Test
public void testGetFormat() throws SQLException{
-   assertThat("testGetFormat 0", bs.getFormat(), notNullValue());
-   assertThat("testGetFormat 1", bs.getFormat(),
                     equalTo(BitstreamFormat.findUnknown(context)));
+   assertThat("testGetFormat 0", bs.getFormat(context), notNullValue());
+   assertThat("testGetFormat 1", bs.getFormat(context),
                     equalTo(BitstreamFormat.findUnknown(context)));
}
```

Fig. 5: An example of outdated hallucination in which LLM incorrectly generates the test with a refactored method.

To assess the effectiveness of TROCtxs constructed by SYNTER, we collect all the hallucinations for NAIVELLM and SYNTER. As shown in Tab. III, hallucinations occur in both approaches. For the 6 common hallucinations that NAIVELLM fails, SYNTER fixes all of them and correctly generates repaired tests with related contexts, indicating that SYNTER has the general capability to precisely and effectively collect required contexts from the syntactic change of the focal method. Besides, SYNTER also reduces outdated hallucinations by 25%, in which two cases are fixed with the constructed TROCtxs (UsageCtxs and EnvCtxs specifically).

TABLE III: The number of different types of Hallucinations.

Approach	Common Hall.	Outdated Hall.	Total
NAIVELLM	6	8	14
SYNTER	**0 (-100.0%)**	**6 (-25.0%)**	**6 (-57.1%)**

Answering RQ2: The TROCtxs constructed by SYNTER can effectively reduce the total hallucinations of LLM by 57.1%, in which all the common ones are fixed.

C. RQ3: Failure Analysis

SYNTER is designed to repair the original test case with its intent unchanged. Considering that the ground truth may contain semantic changes such as adding new assertions and altering input values, we further investigate the 13 cases that fail in terms of Repairability$_{ori}$ according to Tab. II. Finally, we summarize three reasons for the failure.

- **Uses unimported classes**. For four cases, LLM directly uses classes that are not imported in the file of the test method when generating codes. The key reason is that we focus on collecting contexts by analyzing the syntactic change of the focal method at the method level. Without analyzing the focal class and changes in the methods, the constructed contexts by SYNTER may be inadequate and result in failure. Fortunately, this type of failure is easy to fix.
- **Complex focal changes**. For eight cases, LLM fails to correctly generate the expected tests with limited contexts as the changes of focal methods in class-level or repo-level upgrades are complex. Specifically, the complex focal change leads to outdated hallucinations and incorrect invocations. This type is challenging to resolve because of the difficulty of collecting implicit contexts and the limited capability of current LLMs.
- **Fails to construct TROCtxs**. Specifically for one case, we find that the language server fails to provide intelligent features for repositories containing configuration errors, which results in the failure of SYNTER to construct TROCtxs. Overall, it is hard even for developers to manually repair tests in these repositories, in which they can not infer related contexts in IDEs either.

Answering RQ3: SYNTER fails to repair obsolete tests mainly for using unimported classes, being unaware of complex focal changes, and encountering errors when initializing the language server.

D. RQ4: Efficiency

As shown in RQ1, the performance of CEPROT lags behind other approaches because of the limited model backbone. Although CEPROT can repair tests fast, most of the generated codes are incomplete and contain syntax errors (52.2%). Therefore, we focus on comparing the efficiency of NAIVELLM and SYNTER in this research question.

Compared to NAIVELLM, SYNTER adds the steps to construct TROCtxs. When querying LLM to repair the test, the prompt of SYNTER is longer with the constructed TROCtxs. In this research question, we evaluate the time efficiency and the token count of SYNTER compared to NAIVELLM.

In particular, we divide the process of SYNTER into constructing TROCtxs and querying LLM for repair. As shown in Fig. 6, the time cost of SYNTER mainly depends on the process of constructing TROCtxs, while the response time of LLM API of the two approaches is close. We also notice that the language server may fail to synchronize old repositories due to connection timeout for missing dependencies, which results in the outliers with much longer time. In our design, SYNTER trades time for accuracy. Besides, the average time of SYNTER falls into 10s to 30s, which is acceptable for developers since test repair is not an interactive task.

In terms of the token cost of SYNTER, we observe that the tokens of the prompt roughly doubled in number after being augmented with TROCtxs from Fig. 7. This evidence also indicates that the TROCtxs constructed by SYNTER are precise considering the large token size of the whole repository.

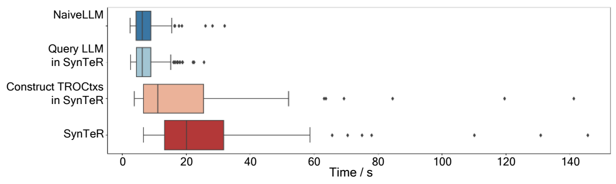


Fig. 6: The time cost of NAIVELLM and SYNTER

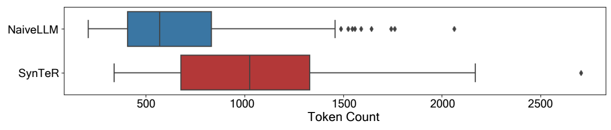


Fig. 7: The token cost of NAIVELLM and SYNTER

Answering RQ4: SYNTER trades time for accuracy, in which the average time of constructing TROCtxs is 10s to 30s. Besides, the final prompt of SYNTER contains roughly twice as many tokens as NAIVELLM.

VI. DISCUSSION

A. Limitations

The key limitations of SYNTER exist in its two main modules, the static collector and the neural reranker. First, SYNTER cannot construct related contexts if the language server fails to initialize due to configuration errors in the repository. Second, the approach's effectiveness is influenced by its reranking strategy. Despite crafting tailored reranking queries for different contexts, we cannot always ensure the precision of reranking. Additionally, our approach focuses on method-level signature changes and does not improve the performance of test repair for class-level or complex implicit changes specifically.

B. Threats to Validity

External Validity. The main threats to external validity come from the evaluation dataset. We reuse the existing dataset, which may not be representative of all possible real-world syntactic changes, so we check and augment the dataset to collect diverse samples from related commits. Besides, it is inevitable to avoid data leakage from popular open-source projects in GitHub. These projects widely use standard design patterns [38], which can be learned by LLM during training. Although experimental results show that NAIVELLM correctly repairs some of the tests with data leakage, it does not affect the effectiveness evaluation of SYNTER. According to the analysis in RQ2, SYNTER outperforms NAIVELLM in cases where NAIVELLM fails for hallucinations.

Internal Validity. In our experiments, a major threat to internal validity is the possible bias in human evaluation. To mitigate it, we invite two senior developers to manually verify the generated tests and annotate explanations. The final result is collected after they reach an agreement after discussion.

VII. RELATED WORK

A. Production-test Co-evolution

Production-test co-evolution refers to co-evolving the test codes with the changes in production codes. Most of the previous studies focus on identifying production-test co-evolution pairs [39], [6], [7], [12]. Recently, more works adopt learning-based techniques to automatically identify obsolete tests [11], [12]. While these works focus on identification only, our approach targets automatically repairing the obsolete test cases directly to relieve the burden of developers.

Two types of obsolete test cases have attracted the attention of developers, one is the GUI-oriented event sequence test case, and the other is the code-oriented method test case. Nowadays, many researchers studied the automated repair of GUI test cases, especially on Android applications [40], [12]. However, for code-oriented method test cases, limited studies are focusing on repairing code-oriented method test cases automatically. One biggest difference between them is that the search space for the GUI-oriented test event is more obvious, which can be obtained by traversing the related GUI widget trees, while the code-oriented method test case has a larger search space in the whole repository.

For code-oriented method test case repairing, Hu et al. [13] proposed the first transformer-based approach to update obsolete tests with two stages, identifying and updating, which is the SOTA work in this area. However, it is based on pre-trained models with fewer parameters and lacks contextual information. Compared to this, our work uses larger language models with automatically constructed contexts from the repository.

B. LLM-based Code Generation

Automated code generation with LLMs brings huge improvements in production efficiency. Repo-level code generation represents the task of generating codes based on a broader context of the repository [41]. It is challenging due to the lack of domain-specific knowledge, which results in hallucination [42], [43]. Several studies leverage Retrieval-Augmented Generation (RAG) to improve the performance of LLM in specific code tasks by providing similar codes or results into the query prompt [41], [26]. However, the contexts are retrieved based on simple metrics such as textual similarity without code semantics. Recently, some works have focused on improving the capability of LLM by static analysis [44], [20]. Specifically, monitor-guided decoding (MGD) [20] is a novel approach to bring IDE-assistance from developers to LLMs to guide the decoding when generating codes, in which IDE-assistance is providing intelligent features by the language server. Compared to this work, we augment LLMs by combining static analysis with RAG to provide more contexts instead of guiding the decoding process.

VIII. CONCLUSIONS

We propose SYNTER, an LLM-powered approach to automatically repair obsolete test cases caused by syntactic changes of the focal methods. The key idea of SYNTER is to combine static analysis and neural rerankers to precisely construct test-repair-oriented contexts from the updated repository, which augments the capability of LLM. Experimental results show that SYNTER's effectiveness outperforms baseline approaches on both the textual- and intent-matching metrics. Besides, with the augmentation of TROCtx constructed by SYNTER, hallucinations are reduced by 57.1%. The overall results also demonstrate that adopting static analysis techniques to improve the capability of LLM yields excellent performance and could be extended to other code-related tasks.

ACKNOWLEDGEMENTS

Thanks to Dr. Xutong Ma for the initial discussions on this work and to all the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grant numbers 62132020 and 62102405, and Major Project of ISCAS (ISCAS-ZD-202302).

REFERENCES

[1] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[2] "apache/kafka," https://github.com/apache/kafka/tags.

[3] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 438–442.

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

[5] D. Jayasuriya, V. Terragni, J. Dietrich, S. Ou, and K. Blincoe, "Understanding breaking changes in the wild," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1433–1444.

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

[7] 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*. IEEE, 2014, pp. 195–204.

[8] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[9] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[10] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024.

[11] 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.

[12] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, "Revisiting the identification of the co-evolution of production and test code," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, 2023.

[13] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "Identify and Update Test Cases When Production Code Changes: A Transformer-Based Approach," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2023, pp. 1111–1122.

[14] Y. Wang, W. Wang, S. R. Joty, and S. C. H. 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, EMNLP 2021*. Association for Computational Linguistics, 2021, pp. 8696–8708.

[15] "Official page for Language Server Protocol," https://microsoft.github.io/language-server-protocol/.

[16] T. Zhang, S. G. Patil, N. Jain, S. Shen, M. Zaharia, I. Stoica, and J. E. Gonzalez, "Raft: Adapting language model to domain specific rag," *arXiv preprint arXiv:2403.10131*, 2024.

[17] T. Mei, Y. Rui, S. Li, and Q. Tian, "Multimedia search reranking: A literature survey," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, pp. 1–38, 2014.

[18] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[19] "Jour - Java API Signature verification," https://jour.sourceforge.net/signature.html, 2008.

[20] L. A. Agrawal, A. Kanade, N. Goyal, S. Lahiri, and S. Rajamani, "Monitor-guided decoding of code lms with static analysis of repository context," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[21] P. Yin and G. Neubig, "Reranking for neural semantic parsing," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[22] "Alluxio/alluxio@8cc5a29," https://github.com/Alluxio/alluxio/commit/8cc5a292f4c6e38ed0066ce5bd700cc946dc3803, 2019.

[23] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 29, no. 11, pp. 1005–1018, 2003.

[24] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.

[25] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[26] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.

[27] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[28] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, "Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 14–26.

[29] 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*, 2020, pp. 585–597.

[30] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "Codebertscore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*. Association for Computational Linguistics, 2023, pp. 13 921–13 937.

[31] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[32] 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.

[33] "Introduction — LangChain," https://python.langchain.com/.

[34] openai, "GPT-4," https://openai.com/index/gpt-4/.

[35] J. Chen, S. Xiao, P. Zhang, K. Luo, D. Lian, and Z. Liu, "Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation," 2023.

[36] BAAI, "Beijing academy of artificial intelligence — baai.ac.cn," https://www.baai.ac.cn/english.html.

[37] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[38] J. Bloch, *Effective java*. Addison-Wesley Professional, 2017.

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

[40] A. M. Memon and M. L. Soffa, "Regression testing of guis," *ACM SIGSOFT software engineering notes*, vol. 28, no. 5, pp. 118–127, 2003.

[41] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*. Association for Computational Linguistics, 2023, pp. 2471–2484.

[42] M. Liu, T. Yang, Y. Lou, X. Du, Y. Wang, and X. Peng, "Codegen4libs: A two-stage approach for library-oriented code generation," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 434–445.

[43] S. Lu, N. Duan, H. Han, D. Guo, S. Hwang, and A. Svyatkovskiy, "Reacc: A retrieval-augmented code completion framework," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022*. Association for Computational Linguistics, 2022, pp. 6227–6240.

[44] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, "Enhancing llm-based test generation for hard-to-cover branches via program analysis," *arXiv preprint arXiv:2404.04966*, 2024.