Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction

Sungmin Kang*
School of Computing
KAIST
Daejeon, Republic of Korea
sungmin.kang@kaist.ac.kr

Juyeon Yoon*
School of Computing
KAIST
Daejeon, Republic of Korea
juyeon.yoon@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00194

***Abstract*—Many automated test generation techniques have been developed to aid developers with writing tests. To facilitate full automation, most existing techniques aim to either increase coverage, or generate exploratory inputs. However, existing test generation techniques largely fall short of achieving more semantic objectives, such as generating tests to reproduce a given bug report. Reproducing bugs is nonetheless important, as our empirical study shows that the number of tests added in open source repositories due to issues was about 28% of the corresponding project test suite size. Meanwhile, due to the difficulties of transforming the expected program semantics in bug reports into test oracles, existing failure reproduction techniques tend to deal exclusively with program crashes, a small subset of all bug reports. To automate test generation from general bug reports, we propose LIBRO, a framework that uses Large Language Models (LLMs), which have been shown to be capable of performing code-related tasks. Since LLMs themselves cannot execute the target buggy code, we focus on post-processing steps that help us discern when LLMs are effective, and rank the produced tests according to their validity. Our evaluation of LIBRO shows that, on the widely studied Defects4J benchmark, LIBRO can generate failure reproducing test cases for 33% of all studied cases (251 out of 750), while suggesting a bug reproducing test in first place for 149 bugs. To mitigate data contamination (i.e., the possibility of the LLM simply remembering the test code either partially or in whole), we also evaluate LIBRO against 31 bug reports submitted after the collection of the LLM training data terminated: LIBRO produces bug reproducing tests for 32% of the studied bug reports. Overall, our results show LIBRO has the potential to significantly enhance developer efficiency by automatically generating tests from bug reports.**

***Index Terms*—test generation, natural language processing, software engineering**

I. INTRODUCTION

Software testing is the practice of confirming that software meets specification criteria by executing tests on the software under test (SUT). Due to the importance and safety-critical nature of many software projects, software testing is one of the most important practices in the software development process. Despite this, it is widely acknowledged that software testing is tedious due to the significant human effort required [1]. To fill this gap, automated test generation techniques have been studied for almost half a century [2], resulting in a number of tools [3], [4] that use implicit oracles (regressions or crash detection) to guide the automated process. They are useful when new features are being added, as they can generate novel tests with high coverage for a focal class.

However, not all tests are added immediately along with their focal class. In fact, we find that a significant number of tests originate from *bug reports*, i.e., are created in order to prevent future regressions for the bug reported. This suggests that *the generation of bug reproducing tests from bug reports* is an under-appreciated yet impactful way of automatically writing tests for developers. Our claim is based on the analysis of a sample of 300 open source projects using JUnit: the number of tests added as a result of bug reports was on median 28% of the size of the overall test suite. Thus, the bug report-to-test problem is regularly dealt with by developers, and a problem in which an automated technique could provide significant help. Previous work in bug reproduction mostly deals with crashes [5], [6]; as many bug reports deal with semantic issues, their scope is limited in practice.

The general report-to-test problem is of significant importance to the software engineering community, as solving this problem would allow developers use a greater number of automated debugging techniques, equipped with test cases that reproduce the reported bug. Koyuncu et al. [7] note that in the widely used Defects4J [8] bug benchmark, bug-revealing tests *did not exist* prior to the bug report being filed in 96% of the cases. Consequently, it may be difficult to utilize the state-of-the-art automated debugging techniques, which are often evaluated on Defects4J, when a bug is first reported because they rely on bug reproducing tests [9], [10]. Conversely, alongside a technique that automatically generates bug-revealing tests, a wide range of automated debugging techniques would become usable.

As an initial attempt to solve this problem, we propose prompting Large Language Models (LLMs) to generate tests. Our use of LLMs is based on their impressive performance on a wide range of natural language processing tasks [11] and programming tasks [12]. In this work, we explore whether their capabilities can be extended to generating test cases from bug reports. More importantly, we argue that the performance of LLMs when applied to this problem has to be studied along with the issue of *when* we can rely on the tests that LLMs produce. Such questions are crucial for actual

*: these authors contributed equally.

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00194

developer use: Sarkar et al. [13] provide relevant examples, showing that developers struggle to understand when LLMs will do their bidding when used for code generation. To fill this gap of knowledge, we propose LIBRO (LLM Induced Bug ReprOduction), a framework that prompts the OpenAI LLM, Codex [14], to generate tests, processes the results, and suggests solutions only when we can be reasonably confident that bug reproduction has succeeded.

We perform extensive empirical experiments on both the Defects4J benchmark and a new report-test dataset that we have constructed, aiming to identify the features that can indicate successful bug reproduction by LIBRO. We find that, for the Defects4J benchmark, LIBRO can generate at least one bug reproducing test for 251 bugs, or 33.5% of all studied bugs from their bug reports. LIBRO also successfully deduced which of its bug reproducing attempts were successful with 71.4% accuracy, and produced an actual bug reproducing test as its first suggestion for 149 bugs. For further validation, we evaluate LIBRO on a recent bug report dataset that we built, finding that we could reproduce 32.2% of bugs in this distinct dataset as well, and verifying that our test suggesting heuristics work in this different dataset as well.

In summary, our contributions are as follows:

- We perform an analysis of open source repositories to verify the importance of generating bug reproducing test cases from bug reports;
- We propose a framework to harness an LLM to reproduce bugs, and suggest generated tests to the developer only when the results are reliable;
- We perform extensive empirical analyses on two datasets, suggesting that the patterns we find, and thus the performance of LIBRO, are robust.

The remainder of the paper is organized as follows. We motivate our research in Section II. Based on this, we describe our approach in Section III. Evaluation settings and research questions are in Section IV and Section V, respectively. Results are presented in Section VI, while threats to validity are discussed in Section VIII. Section IX gives an overview of the relevant literature, and Section X concludes.

II. MOTIVATION

As described in the previous section, the importance of the report-to-test problem rests on two observations. The first is that bug-revealing tests are rarely available when a bug report is filed, unlike what automated debugging techniques often assume [9], [10]. Koyuncu et al. [7] report that Spectrum-Based Fault Localization (SBFL) techniques cannot locate the bug at the time of being reported in 95% of the cases they analyzed, and thus propose a completely static automated debugging technique. However, as Le et al. [15] demonstrate, using dynamic information often leads to more precise localization results. As such, a report-to-test technique could *enhance the practicality and/or performance of a large portion of the automated debugging literature*.

The other observation is that the report-to-test problem is a perhaps underappreciated yet nonetheless important and recurring part of testing. Existing surveys of developers reveal that developers consider generating tests from bug reports to be a way to improve automated testing. Daka and Fraser [16] survey 225 software developers and point out ways in which automated test generation could help developers, three of which (deciding what to test, realism, deciding what to check) can be resolved using bug reports, as bug reproduction is a relatively well-defined activity. Kochhar et al. [17] explicitly ask hundreds of developers on whether they agree to the statement "during maintenance, when a bug is fixed, it is good to add a test case that covers it", and find a strong average agreement of 4.4 on a Likert scale of 5.

To further verify that developers regularly deal with the report-to-test problem, we analyze the number of test additions that can be attributed to a bug report, by mining hundreds of open-source Java repositories. We start with the `Java-med` dataset from Alon et al. [18], which consists of 1000 top-starred Java projects from GitHub. From the list of commits in each repository, we check (i) whether the commit adds a test, and (ii) whether the commit is linked to an issue. To determine whether a commit adds a test, we check that its diff adds the `@Test` decorator along with a test body. In addition, we link a commit to a bug report (or an *issue* in GitHub) if (i) the commit message mentions "(fixes/resolves/closes) #NUM", or (ii) the commit message mentions a pull request, which in turn mentions an issue. We compare the number of tests added by such report-related commits to the size of the current (August 2022) test suite to estimate the prevalence of such tests. As different repositories have different issue-handling practices, we filter out repositories that have no issue-related commits that add tests, as this indicates a different bug handling practice (e.g. `google/guava`). Accordingly, we analyze 300 repositories, as shown in Table I.

TABLE I: Analyzed repository characteristics

Repository Characteristic	# Repositories
Could be cloned	970
Had a JUnit test (`@Test` is found in repository)	550
Had issue-referencing commit that added test	300

We find that the median ratio of tests added by issue-referencing commits in those 300 repositories, relative to the current test suite size, is 28.4%, suggesting that a significant number of tests are added due to bug reports. We note that this does not mean 28.4% of tests in a test suite originate from bug reports, as we do not track what happens to tests after they are added. Nonetheless, it indicates the report-to-test activity plays a significant role in the evolution of test suites. Based on this result, we conclude that the report-to-test generation problem is regularly handled by open source developers. By extension, an automated report-to-test technique that suggests and/or automatically commits confirmed tests would help developers in their natural workflow.

Despite the importance of the problem, its general form remains a difficult problem to solve. Existing work attempts to solve special cases of the problem by focusing on different

aspects: some classify the sentences of a report into categories like observed or expected behavior [19], while others only reproduce crashes (crash reproduction) [6], [20]. We observe that solving this problem requires good understanding of both natural and programming language, not to mention capabilities to perform deduction. For example, the bug report in Table II does not explicitly specify any code, but a fluent user in English and Java would be capable of deducing that when both arguments are NaN, the 'equals' methods in 'MathUtils' should return `false`.

One promising solution is to harness the capabilities of pre-trained Large Language Models (LLMs). LLMs are generally Transformer-based neural networks [13] trained with the language modeling objective, i.e. predicting the next token based on preceding context. One of their main novelties is that they can perform tasks without training: simply by 'asking' the LLM to perform a task via a textual prompt, the LLM is often capable of actually performing the task [11]. Thus, one point of curiosity is how many bugs LLMs can reproduce given a report. On the other hand, of practical importance is to be able to know *when* we should believe and use the LLM results, as noted in the introduction. To this end, we focus on finding heuristics indicative of high precision, and minimize the hassle that a developer would have to deal with when using LIBRO.

III. APPROACH

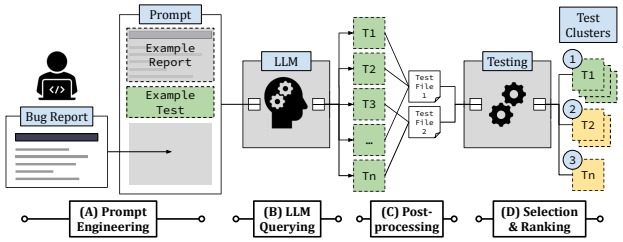

Fig. 1: Overview of LIBRO

An overview diagram of our approach is presented in Figure 1. Given a bug report, LIBRO first constructs a prompt to query an LLM (Figure 1:(A)). Using this prompt, an initial set of test candidates are generated by querying the LLM multiple times (Figure 1:(B)). Then, LIBRO processes the tests to make them executable in the target program (Figure 1:(C)). LIBRO subsequently identifies and curates tests that are likely to be bug reproducing, and if so, ranks them to minimize developer inspection effort (Figure 1:(D)). The rest of this section explains each stage in more detail using the running example provided in Table II.

A. Prompt Engineering

LLMs are, at the core, large autocomplete neural networks: prior work have found that different ways of 'asking' the LLM to solve a problem will lead to significantly varying levels of performance [21]. Finding the best query to accomplish the given task is known as *prompt engineering* [22].

To make an LLM to generate a test method from a given bug report, we construct a Markdown document, which is to be used in the prompt, from the bug report: consider the example in Listing 1, which is a Markdown document constructed from the bug report shown in Table II. LIBRO adds a few distinctive parts to the Markdown document: the command "Provide a self-contained example that reproduces this issue", the start of a block of code in Markdown, (i.e., ```), and finally the partial code snippet `public void test` whose role is to induce the LLM to write a test method.

TABLE II: Example bug report (Defects4J Math-63).

Issue No.	MATH-370[1]
Title	NaN in "equals" methods
Description	In "MathUtils", some "equals" methods will return true if both argument are NaN. Unless I'm mistaken, this contradicts the IEEE standard. If nobody objects, I'm going to make the changes.

Listing 1: Example prompt without examples.

```
# NaN in "equals" methods
## Description
In "MathUtils", some "equals" methods will return true if both argument
      are NaN.
Unless I'm mistaken, this contradicts the IEEE standard.
If nobody objects, I'm going to make the changes.

## Reproduction
>Provide a self-contained example that reproduces this issue.
```
public void test
```

We evaluate a range of variations of this basic prompt. Brown et al. [11] report that LLMs benefit from question-answer examples provided in the prompt. In our case, this means providing examples of bug reports (questions) and the corresponding bug reproducing tests (answers). With this in mind, we experiment with a varying number of examples, to see whether adding more examples, and whether having examples from within the same project or from other projects, significantly influences performance.

As there is no real restriction to the prompt format, we also experiment with providing stack traces for crash bugs (to simulate situations where a stack trace was provided), or providing constructors of the class where the fault is located (to simulate situations where the location of the bug is reported).

Our specific template format makes it highly unlikely that prompts we generate exist verbatim within the LLM training data. Further, most reports in practice are only connected to the bug-revealing test via a chain of references. As such, our format partly mitigates data leakage concerns, among other steps taken to limit this threat described later in the manuscript.

### B. Querying an LLM

Using the generated prompt, LIBRO queries the LLM to predict the tokens that would follow the prompt. Due to the

[1]https://issues.apache.org/jira/browse/MATH-370
```

nature of the prompt, it is likely to generate a test method, especially as our prompt ends with the sequence `public void test`. We ensure that the result only spans the test method by accepting tokens until the first occurrence of the string ```, which indicates the end of the code block in Markdown.

It is known that LLMs yield inferior results when performing completely greedy decoding (i.e., decoding strictly based on the most likely next token) [11]: they perform better when they are doing weighted random sampling, a behavior modulated by the *temperature* parameter. Following prior work, we set our temperature to 0.7 [11], which allows the LLM to generate multiple distinct tests based on the exact same prompt. We take the approach of generating multiple candidate reproducing tests, then using their characteristics to identify how likely it is that the bug is actually reproduced.

An example output from the LLM given the prompt in Listing 1 is shown in Listing 2: at this point, the outputs from the LLM typically cannot be compiled on their own, and need other constructs such as import statements. We next present how LIBRO integrates a generated test into the existing test suite to make it executable.

Listing 2: Example LLM result from the bug report described in Table II.

```
1 public void testEquals() {
2     assertFalse(MathUtils.equals(Double.NaN, Double.NaN));
3     assertFalse(MathUtils.equals(Float.NaN, Float.NaN));
4 }
```

C. Test Postprocessing

We first describe how LIBRO injects a test method into an existing suite then how LIBRO resolves the remaining unmet dependencies.

1) Injecting a test into a suitable test class: If a developer finds a test method in a bug report, they will likely insert it into a test class which will provide the required context for the test method (such as the required dependencies). For example, for the bug in our running example, the developers added a reproducing test to the `MathUtilsTest` class, where most of the required dependencies are already imported, including the focal class, `MathUtils`. Thus, it is natural to also inject LLM-generated tests into existing test classes, as this matches developer workflow, while resolving a significant number of initially unmet dependencies.

Listing 3: Target test class to which the test in Listing 2 is injected.

```
1 public final class MathUtilsTest extends TestCase {
2     ...
3     public void testArrayEquals() {
4         assertFalse(MathUtils.equals(new double[] { 1d }, null));
5         assertTrue(MathUtils.equals(new double[] {
6             Double.NaN, Double.POSITIVE_INFINITY,
7         ...
```

To find the best test class to inject our test methods into, we find the test class that is *lexically* most similar to the generated test (Algorithm 1, line 1). The intuition is that, if a test method belongs to a test class, the test method likely uses similar

Algorithm 1: Test Postprocessing

Input: A test method tm; Test suite $\mathcal{T}$ of SUT; source code files $\mathcal{S}$ of SUT;
Output: Updated test suite $\mathcal{T}'$

```
1  c_best ← findBestMatchingClass(tm, T);
2  deps ← getDependencies(tm);
3  needed_deps ← getUnresolved(deps, c_best);
4  new_imports ← set();
5  for dep in needed_deps do
6      target ← findClassDef(dep, S);
7      if target is null then
8          new_imports.add(findMostCommonImport(dep, S, T));
9      else
10         new_imports.add(target);
11 T' ← injectTest(tm, c_best, T);
12 T' ← injectDependencies(new_imports, c_best, T');
```

methods and classes, and is thus lexically related, to other tests from that test class. Formally, we assign a matching score for each test class based on Equation (1):

$$sim_{c_i} = |T_t \cap T_{c_i}|/|T_t| \quad (1)$$

where T_t and T_{c_i} are the set of tokens in the generated test method and the ith test class, respectively. As an example, Listing 3 shows the key statements of the `MathUtilsTest` class. Here, the test class contains similar method invocations and constants with those used by the LLM-generated test in Listing 2, particularly in lines 4 and 6.

As a sanity check, we inject ground-truth developer-added bug reproducing tests from the Math and Lang projects of the Defects4J benchmark, and check if they execute normally based on Algorithm 1. We find execution proceeds as usual for 89% of the time, suggesting that the algorithm reasonably finds environments in which tests can be executed.

2) Resolving remaining dependencies: Although many dependency issues are resolved by placing the test in the right class, the test may introduce new constructs that need to be imported. To handle these cases, LIBRO heuristically infers packages to import.

Line 2 to 10 in Algorithm 1 describe the dependency resolving process of LIBRO. First, LIBRO parses the generated test method and identifies variable types and referenced class names/constructors/exceptions. LIBRO then filters "already imported" class names by lexically matching names to existing import statements in the test class (Line 3).

As a result of this process, we find types that are not resolved within the test class. LIBRO first attempts to find public classes with the identified name of the type; if there is exactly one such file, the classpath to the identified class is derived (Line 7), and an import statement is added (Line 11). However, either no or multiple matching classes may exist. In both cases, LIBRO looks for `import` statements ending with the target class name within the project (e.g., when searching for `MathUtils`, LIBRO looks for `import .*MathUtils;`). LIBRO selects the most common import statement across all project source code files. Additionally, we add a few rules that allow

Algorithm 2: Test Selection and Ranking

Input: Pairs of modified test suites and injected test methods $\mathcal{S}_{\mathcal{T}'}$; target program with bug P_b; bug report BR; agreement threshold Thr;
Output: Ordered list of ranked tests $ranking$;

```
1  FIB ← set();
2  for (T', tm_i) ∈ S_T' do
3      r ← executeTest(T', P_b);
4      if hasNoCompileError(r) && isFailed(tm_i, r) then
5          FIB.add((tm_i, r));
6  clusters ←clusterByFailureOutputs(FIB);
7  output_clus_size ← clusters.map(size);
8  max_output_clus_size ←max(output_clus_size);
9  if max_output_clus_size ≤ Thr then
10     return list();
11 FIB_uniq ← removeSyntacticEquivalents(FIB);
12 br_output_match ← clusters.map(matchOutputWithReport(BR));
13 br_test_match ← FIB_uniq.map(matchTestWithReport(BR));
14 tok_cnts ← FIB_uniq.map(countTokens);
15 ranking ←list();
16 clusters ← clusters.sortBy(
     br_output_match, output_clus_size, tok_cnts);
17 for clus ∈ clusters do
18     clus ← clus.sortBy(br_test_match, tok_cnts);
19 for i = 0; i <max(output_clus_size); i ← i + 1 do
20     for clus ∈ clusters do
21         if i < clus.length() then
22             ranking.push(clus[i])
23 return ranking;
```

assertion statements to be properly imported, even when there are no appropriate imports within the project itself.

Our postprocessing pipeline does not guarantee compilation in all cases, but the heuristics used by LIBRO are capable of resolving most of the unhandled dependencies of a raw test method. After going through the postprocessing steps, LIBRO executes the tests to identify candidate bug reproducing tests.

D. Selection and Ranking

A test is a Bug Reproducing Test (BRT) if and only if the test fails due to the bug specified in the report. A *necessary* condition for a test generated by LIBRO to be a BRT is that the test compiles and fails in the buggy program: we call such tests FIB (Fail In the Buggy program) tests. However, not all FIB tests are BRTs, making it difficult to tell whether bug reproduction has succeeded or not. This is one factor that separates us from crash reproduction work [20], as crash reproduction techniques can confirm whether the bug has been reproduced by comparing the stack traces at the time of crash. On the other hand, it is imprudent to present all generated FIB tests to developers, as asking developers to iterate over multiple solutions is generally undesirable [23], [24]. As such, LIBRO attempts to decide when to suggest a test and, if so, which test to suggest, using several patterns we observe to be correlated to successful bug reproductions.

Algorithm 2 outlines how LIBRO decides whether to present results and, if so, how to rank the generated tests. In Line 1-10, LIBRO first decides whether to show the developer any results at all (selection). We group the FIB tests that exhibit the same failure output (the same error type and error message) and look at the number of the tests in the same group (the $max\_output\_clus\_size$ in Line 8). This is based on the intuition that, if multiple tests show similar failure behavior, then it is likely that the LLM is 'confident' in its predictions as its independent predictions 'agree' with each other, and there is a good chance that bug reproduction has succeeded. LIBRO can be configured to only show results when there is significant agreement in the output (setting the agreement threshold Thr high) or show more exploratory results (setting Thr low).

Once it decides to show its results, LIBRO relies on three heuristics to rank generated tests, in the order of increasing discriminative strength. First, tests are likely to be bug reproducing if the fail message and/or the test code shows the behavior (exceptions or output values) observed and mentioned in the bug report. While this heuristic is precise, its decisions are not very discriminative, as tests can only be divided into groups of 'contained' versus 'not contained'. Next, we look at the 'agreement' between generated tests by looking at output cluster size ($output\_clus\_size$), which represents the 'consensus' of the LLM generations. Finally, LIBRO prioritizes based on test length (as shorter tests are easier to understand), which is the finest-grained signal. We first leave only syntactically unique tests (Line 11), then sort output clusters and tests within those clusters using the heuristics above (Lines 16 and 18).

As tests with the same failure output are similar to each other, we expect that, if one test from a cluster is not BRT, the rest from the same cluster are likely not BRT as well. Hence, LIBRO shows tests from a diverse array of clusters. For each ith iteration in Line 19-22, the ith ranked test from each cluster is selected and added to the list.

IV. EVALUATION

This section provides evaluation details for our experiments.

A. Dataset

As a comprehensive evaluation benchmark, we use *Defects4J* version 2.0, which is a manually curated dataset of real-world bugs gathered from 17 Java projects. Each Defects4J bug is paired to a corresponding bug report[2], which makes the dataset ideal for evaluating the performance of LIBRO. Among 814 bugs that have a paired bug report, we find that 58 bugs have incorrect pairings, while six bugs have different directory structures between the buggy and fixed versions: this leaves **750** bugs for us to evaluate LIBRO on. 60 bugs in the Defects4J benchmark are included in the JCrashPack [25] dataset used in the crash reproduction literature; we use this subset when comparing against crash reproduction techniques.

As Codex, the LLM we use, was trained with data collected until July 2021, the Defects4J dataset is not free from data leakage concerns, even if the prompt format we use is unlikely to have appeared verbatim in the data. To mitigate such

[2] Except for the Chart project, for which only 8 bugs have reports

concerns, from 17 GitHub repositories[3] that use JUnit, we gather 581 Pull Requests (PR) created after the Codex training data cutoff point, ensuring that this dataset could not have been used to train Codex. We further check if a PR adds any test to the project (435 left after discarding non-test-introducing ones), and filter out the PRs that are not merged to the main branch or associated with multiple issues (84 left).

For these 84 PRs, we verify that the bug can be reproduced by checking that a developer-added test added by the PR fails on the pre-merge commit without compilation errors, and passes on the post-merge commit. We add the pair to our final list only when all of them have been reproduced. After the final check, we end up with **31** reproducible bugs and their bug reports. This dataset is henceforth referred to as the GHRB (GitHub Recent Bugs) dataset. We use this dataset to verify that trends observed in Defects4J are not due to data leakage.

B. Metrics

A test is treated as a Bug Reproducing Test (BRT) in our evaluation if it fails on the version that contains the bug specified in the report, and passes on the version that fixes the bug. We say that a bug is *reproduced* if LIBRO generates at least one BRT for that bug. The number of bugs that are reproduced is counted for each evaluation technique.

We use the `PRE_FIX_REVISION` and `POST_FIX_REVISION` versions in the Defects4J benchmarks as the buggy/fixed versions, respectively. The two versions reflect the *actual* state of the project when the bug was discovered/fixed. For the GHRB dataset, as we gathered the data based on code changes from merged pull requests, we use pre-merge and post-merge versions as the buggy/fixed versions.

EvoCrash [20], the crash reproduction technique we compare with, originally checks whether the *crash stack* is reproduced in the *buggy version*. For fair comparison, we evaluate EvoCrash under our reproduction criterion: EvoCrash-generated tests are executed on the buggy and fixed versions, and when execution results change, we treat the test as a BRT.

To evaluate the rankings produced by LIBRO, we focus on two aspects: the capability of LIBRO to rank the actual BRTs higher, and the degree of effort required from developers to inspect the ranked tests. For the former, we use the $acc@n$ metric, which counts the number of bugs whose BRTs are found within the top n places in the ranking. Additionally, we report the precision of LIBRO by dividing $acc@n$ with the number of all selected bugs, representing how often a developer would accept a suggestion by LIBRO. To estimate developer effort, we use the wef metric: the number of non-reproducing tests ranked higher than any bug reproducing test. If there are no BRTs, we report wef as the total number of the target FIB tests in ranking. We also use $wef@n$, which shows the wasted effort when using the top n candidates.

[3]These repositories have been manually chosen from either Defects4J projects that are on GitHub and open to new issues, or Java projects that have been modified since 10th July 2022 with at least 100 or more stars, as of 1st of August 2022. A list of 17 repositories is available in our artifact.

C. Environment

All experiments are performed on a machine running Ubuntu 18.04.6 LTS, with 32GB of ram and Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz CPU. We access OpenAI Codex via its closed beta API, using the `code-davinci-002` model. For Codex, we set the temperature to 0.7, and the maximum number of tokens to 256. We script our experiments using Python 3.9, and parse Java files with the `javalang` library [26]. Our replication package is online[4].

V. RESEARCH QUESTIONS

We aim to answer the following research questions.

A. RQ1: Efficacy

With RQ1, we seek to quantitatively evaluate the performance of LIBRO using the Defects4J benchmark.

- **RQ1-1: How many bug reproducing tests can LIBRO generate?** We evaluate how many bugs in total are reproduced by LIBRO using various prompt settings.
- **RQ1-2: How does LIBRO compare to other techniques?** In the absence of generic report-to-test techniques, we compare against EvoCrash, a crash reproduction technique. We also compare against a 'Copy&Paste' baseline that directly uses code snippets (identified with the HTML `<pre>` tag or via infoZilla [27]) within the bug report as tests. For code that could be parsed as a Java compilation unit, we add the code as a test class and add JUnit imports if necessary to run it as a test. Otherwise, we wrap the code snippet in a test method and evaluate it under the same conditions as LIBRO.

B. RQ2: Efficiency

With RQ2, we examine the efficiency of LIBRO in terms of the amount of resources it uses, to provide an estimate of the costs of deploying LIBRO in a real-world context.

- **RQ2-1: How many Codex queries are required?** We estimate how many queries are needed to achieve a certain bug-reproduction rate on the Defects4J dataset based on a pool of generated tests.
- **RQ2-2: How much time does LIBRO need?** Our technique consists of querying an LLM, making it executable, and ranking: we measure the time taken at each stage.
- **RQ2-3: How many tests should the developer inspect?** We evaluate how many bugs could be reproduced within 1, 3, and 5 suggestions, along with the amount of 'wasted effort' required from the developer.

C. RQ3: Practicality

Finally, with RQ3, we aim to investigate how well LIBRO generalizes by applying it to the GHRB dataset.

- **RQ3-1: How often can LIBRO reproduce bugs in the wild?** To mitigate data leakage issues, we evaluate LIBRO on the GHRB dataset, checking how many bugs can be reproduced on it.

[4]https://anonymous.4open.science/r/llm-testgen-artifact-2753

- **RQ3-2: How reliable are the selection and ranking techniques of LIBRO?** We investigate whether the factors that were used during selecting bugs and ranking tests for the Defects4J dataset are still valid for the GHRB dataset, and thus can be used for other projects in general.
- **RQ3-3: What does reproduction success and failure look like?** To provide qualitative context to our results, we describe examples of bug reproduction success and failure from the GHRB dataset.

VI. EXPERIMENTAL RESULTS

A. RQ1. How effective is LIBRO?

1) RQ1-1: Table III shows which prompt/information settings work best, where $n = N$ means we queried the LLM N times for reproducing tests. When using examples from the source project, we use the oldest tests available within that project; otherwise, we use two handpicked report-test pairs (Time-24, Lang-1) throughout all projects. We find that providing constructors (*à la* AthenaTest [28]) does not help significantly, but adding stack traces does help reproduce crash bugs, indicating that LIBRO can benefit from using the stack information to replicate issues more accurately. Interestingly, adding within-project examples shows poorer performance: inspection of these cases has revealed that, in such cases, LIBRO simply copied the provided example even when it should not have, leading to lower performance. We also find that the number of examples makes a significant difference (two-example n=10 values are sampled from n=50 results from the default setting), confirming the existing finding that adding examples helps improve performance. In turn, the number of examples seems to matter less than the number of times the LLM is queried, as we further explore in RQ2-1. As the two-example n=50 setting shows the best performance, we use it as the default setting throughout the rest of the paper.

TABLE III: Reproduction performance for different prompts

Setting	reproduced	FIB
No Example (n=10)	124	440
One Example (n=10)	166	417
One Example from Source Project (n=10)	152	455
One Example with Constructor Info (n=10)	167	430
Two Examples (n=10, 5th percentile)	161	386
Two Examples (n=10, median)	173	409
Two Examples (n=10, 95th percentile)	184	429
Two Examples (n=50)	**251**	**570**
One Example, Crash Bugs (n=10)	69	153
One Example with Stack, Crash Bugs (n=10)	84	155

Under the two-example n=50 setting, we find that overall **251** bugs, or 33.5% of 750 studied Defects4J bugs, are reproduced by LIBRO. Table IV presents a breakdown of the performance per project. While there is at least one bug reproduced for every project, the proportion of bugs reproduced can vary significantly. For example, LIBRO reproduces a small number of bugs in the Closure project, which is known to have a unique test structure [29]. On the other hand, the performance is stronger for the Lang or Jsoup projects, whose tests are generally self-contained and simple. Additionally, we find that the average length of the generated test body is about 6.5 lines (excluding comments and whitespace), indicating LIBRO is capable of writing meaningfully long tests.

TABLE IV: Bug reproduction per project in Defects4J: x/y means x reproduced out of y bugs

Project	rep/total	Project	rep/total	Project	rep/total
Chart	5/7	Csv	6/16	JxPath	3/19
Cli	14/29	Gson	7/11	Lang	46/63
Closure	2/172	JacksonCore	8/24	Math	43/104
Codec	10/18	JacksonDatabind	30/107	Mockito	1/13
Collections	1/4	JacksonXml	2/6	Time	13/19
Compress	4/46	Jsoup	56/92	**Total**	**251/750**

Answer to RQ1-1: A large (251) number of bugs can be replicated automatically, with bugs replicated over a diverse group of projects. Further, the number of examples in the prompt and the number of generation attempts have a strong effect on performance.

2) RQ1-2: We further compare LIBRO against the state-of-the-art crash reproduction technique, EvoCrash, and the 'Copy&Paste baseline' that uses code snippets from the bug reports. We present the comparison results in Figure 2. We find LIBRO replicates a large and distinct group of bugs compared to other baselines. LIBRO reproduced 91 more unique bugs (19 being crash bugs) than EvoCrash, which demonstrates that LIBRO can reproduce non-crash bugs prior work could not handle (Fig. 2(b)). On the other hand, the Copy&Paste baseline shows that, while the BRT is sometimes included in the bug report, the report-to-test task is not at all trivial. Interestingly, eight bugs reproduced by the Copy&Paste baseline were not reproduced by LIBRO; we find that this is due to long tests that exceed the generation length of LIBRO, or due to dependency on complex helper functions.

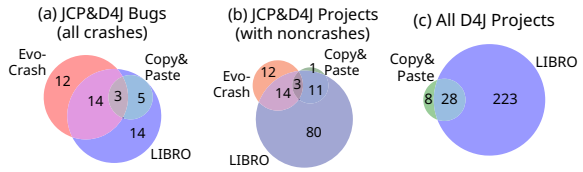

Fig. 2: Baseline comparison on bug reproduction capability

Answer to RQ1-2: LIBRO is capable of replicating a large and distinct group of bugs relative to prior work.

B. RQ2. How efficient is LIBRO?

1) RQ2-1: Here, we investigate how many tests must be generated to attain a certain bug reproduction performance. To do so, for each Defects4J bug, we randomly sample x tests from the 50 generated under the default setting, leaving a reduced number of tests per bug. We then check the number of bugs reproduced y when using only those sampled tests. We repeat this process 1,000 times to approximate the distribution.

The results are presented in Figure 3. Note that the x-axis is in log scale. Interestingly, we find a logarithmic relation holds

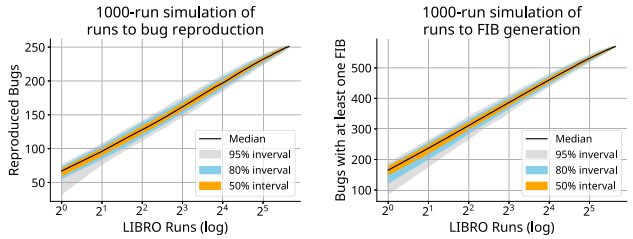

Fig. 3: Generation attempts to performance. Left depicts bugs reproduced as attempts increase, right for FIB

between the number of test generation attempts and the median bug reproduction performance. This suggests that it becomes increasingly difficult, yet stays possible, to replicate more bugs by simply generating more tests. As the graph shows no signs of plateauing, experimenting with an even greater sample of tests may result in better bug reproduction results.

Answer to RQ2-1: The number of bugs reproduced increases logarithmically to the number of tests generated, with no sign of performance plateauing.

TABLE V: The time required for the pipeline of LIBRO

	Prompt	API	Processing	Running	Ranking	Total
Single Run	<1 μs	5.85s	1.23s	4.00s	-	11.1s
50-test Run	<1 μs	292s	34.8s	117s	0.02s	444s

2) RQ2-2: We report the time it takes to perform each step of our pipeline in Table V. We find API querying takes the greatest amount of time, requiring about 5.85 seconds. Postprocessing and test executions take 1.23 and 4 seconds per test (when the test executes), respectively. Overall, LIBRO took an average of 444 seconds to generate 50 tests and process them, which is well within the 10-minute search budget often used by search-based techniques [20].

Answer to RQ2-2: Our time measurement suggests that LIBRO does not take a significantly longer time than other methods to use.

3) RQ2-3: With this research question, we measure how effectively LIBRO prioritizes bug reproducing tests via its selection and ranking procedure. As LIBRO only shows results above a certain agreement threshold, Thr from Section III-D, we first present the trade-off between the number of total bugs reproduced and precision (i.e., the proportion of successfully reproduced bugs among all selected by LIBRO) in Figure 4. As we increase the threshold, more suggestions (including BRTs) are discarded, but the precision gets higher, suggesting one can smoothly increase precision by tuning the selection threshold.

We specifically set the agreement threshold to 1, a conservative value, in order to preserve as many reproduced bugs as possible. Among the 570 bugs with a FIB, 350 bugs are selected. Of those 350, 219 are reproduced (leading to a precision of $0.63(=\frac{219}{350})$ whereas recall (i.e., proportion

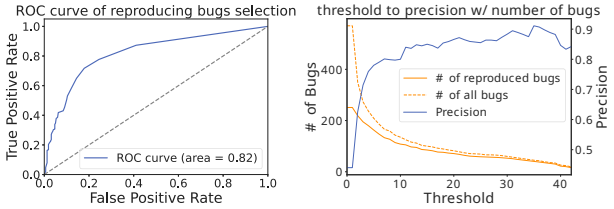

Fig. 4: ROC curve of bug selection (Left), Effect of thresholds to the number of bugs selected and precision (Right)

TABLE VI: Ranking Performance Comparison between LIBRO and Random Baseline

	Defects4J				GHRB			
	$acc@n$ (*precision*)		$wef@n_{agg}$		$acc@n$ (*precision*)		$wef@n_{agg}$	
n	LIBRO	random	LIBRO	random	LIBRO	random	LIBRO	random
1	**149 (0.43)**	116 (0.33)	**201 (0.57)**	234 (0.67)	**6 (0.29)**	4.8 (0.23)	**15 (0.71)**	16.2 (0.77)
3	**184 (0.53)**	172 (0.49)	**539 (1.54)**	599 (1.71)	**7 (0.33)**	6.6 (0.31)	**42 (2.0)**	44.6 (2.12)
5	**199 (0.57)**	192 (0.55)	**797 (2.28)**	874 (2.5)	**8 (0.38)**	7.3 (0.35)	**60 (2.86)**	64.3 (3.06)

of selected reproduced bugs among all reproduced bugs) is $0.87(=\frac{219}{251})$. From the opposite perspective, the selection process filters out 188 bugs that were not reproduced, while dropping only a few successfully reproduced bugs. Note that if we set the threshold to 10, a more aggressive value, we can achieve a higher precision of 0.84 for a recall of 0.42. In any case, as Figure 4 presents, our selection technique is significantly better than random, indicating it can save developer resources.

Among the selected bugs, we assess how effective the test rankings of LIBRO are over a random baseline. The random approach randomly ranks the syntactic clusters (groups of syntactically equivalent FIB tests) of the generated tests. We run the random baseline 100 times and average the results.

Table VI presents the ranking evaluation results. On the Defects4J benchmark, the ranking technique of LIBRO improves upon the random baseline across all of the $acc@n$ metrics, presenting 30, 14, and 7 more BRTs than the random baseline on $n = 1$, 3, and 5 respectively. Regarding $acc@1$, the first column shows that 43% of the top ranked tests produced by LIBRO successfully reproduce the original bug report on the first try. When n increases to 5, BRTs can be found in 57% of the selected bugs, or 80% of all reproduced bugs. The conservative threshold choice here, emphasizes recall over precision. However, if the threshold is raised, the maximum precision can rise to 0.8 (for $Thr = 10$, $n = 5$).

The $wef@n_{agg}$ values are additionally reported by both summing and averaging the $wef@n$ of all (350) selected bugs. The summed $wef@n$ value indicates the total number of non-BRTs that would be manually examined within the top n ranked tests. Smaller $wef@n$ values indicate that a technique delivers more bug reproducing tests. Overall, the ranking of LIBRO saves up to 14.5% of wasted effort when compared to the random baseline, even after bugs are selected. Based on these results, we conclude that LIBRO can reduce wasted inspection effort and thus be useful to assist developers.

Answer to RQ2-3: LIBRO can reduce both the number of bugs and tests that must be inspected: 33% of the bugs are safely discarded while preserving 87% of the successful bug reproduction. Among selected bug sets, 80% of all bug reproductions can be found within 5 inspections.

C. RQ3. How well would LIBRO work in practice?

TABLE VII: Bug Reproduction in GHRB: x/y means x reproduced out of y bugs

Project	rep/total	Project	rep/total	Project	rep/total
AssertJ	3/5	Jackson	0/2	Gson	4/7
checkstyle	0/13	Jsoup	2/2	sslcontext	1/2

1) RQ3-1: We explore the performance of LIBRO when operating on the GHRB dataset of recent bug reports. We find that of the 31 bug reports we study, LIBRO can automatically generate bug reproducing tests for 10 bugs based on 50 trials, for a success rate of **32.2%**. This success rate is similar to the results from Defects4J presented in RQ1-1, suggesting LIBRO generalizes to new bug reports. A breakdown of results by project is provided in Table VII. Bugs are successfully reproduced in AssertJ, Jsoup, Gson, and sslcontext, while they were not reproduced in the other two. We could not reproduce bugs from the Checkstyle project, despite it having a large number of bugs; upon inspection, we find that this is because the project's tests rely heavily on external files, which LIBRO has no access to, as shown in Section VI-C3. LIBRO also does not generate BRTs for the Jackson project, but the small number of bugs in the Jackson project make it difficult to draw conclusions from it.

Answer to RQ3-1: LIBRO is capable of generating bug reproducing tests even for recent data, suggesting it is not simply remembering what it trained with.

2) RQ3-2: LIBRO uses several predictive factors correlated with successful bug reproduction for selecting bugs and ranking tests. In this research question, we check whether the identified patterns based on the Defects4J dataset continue to hold in the recent GHRB dataset.

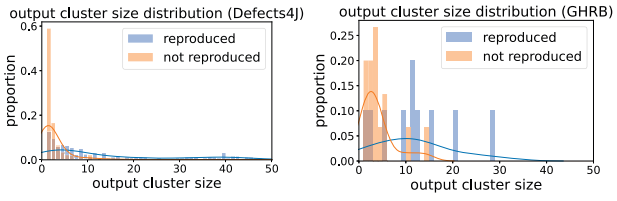

Fig. 5: Distribution of the $max\_output\_clus\_size$ values for reproduced and not-reproduced bugs

Recall that we use the maximum output cluster size as a measure of agreement among the FIBs, and thus as a selection criterion to identify whether a bug has been reproduced. To observe whether the criterion is a reliable indicator to predict the success of bug reproduction, we observe the trend of $max\_output\_clus\_size$ between the two datasets, with and without BRTs. In Figure 5, we see that the bugs with no BRT typically have small $max\_output\_clus\_size$, mostly under ten; this pattern is consistent in both datasets.

The ranking results of GHRB are also presented in Table VI. They are consistent to the results from Defects4J, indicating the features used for our ranking strategy continue to be good indicators of successful bug reproduction.

Answer to RQ3-2: We find that the factors used for the ranking and selection of LIBRO consistently predict bug reproduction in real-world data.

3) RQ3-3: We present case studies of attempts by LIBRO to reproduce bugs that either succeeded or failed.

TABLE VIII: Bug Report Successfully Reproduced: URLs are omitted for brevity (AssertJ-Core Issue #2666)

Title	**assertContainsIgnoringCase fails to compare i and I in tr_TR locale**
	See org.assertj.core.internal.Strings#assertContainsIgnoringCase [url] I would suggest adding [url] verification to just ban toLowerCase(), toUpperCase() and other unsafe methods: #2664

We first present a successfully reproduced bug report, issue #2685 for the AssertJ-Core project, in Table VIII. The bug is that, in `tr_TR` locale, `assertContainsIgnoringCase` method is mishandling the letter `I`. For this bug report, LIBRO generates the bug reproducing test in Listing 4. Despite the bug report not containing any executable code, LIBRO successfully generated a bug reproducing test that compares `I` and `i`, using the `containsIgnoringCase` method referenced within the report. A BRT for this bug is ranked at the second place, suggesting a developer would have been quickly provided with a reproducing test.

Listing 4: Generated FIB test for AssertJ-Core-2666.

```
public void testIssue952() {
  Locale locale = new Locale("tr", "TR");
  Locale.setDefault(locale);
  assertThat("I").as("Checking in tr_TR locale").containsIgnoringCase("i
      ");
}
```

Listing 5: Generated FIB test for Checkstyle-11365.

```
public void testFinalClass() throws Exception {
    final DefaultConfiguration checkConfig =
            createModuleConfig(FinalClassCheck.class);
    final String[] expected = CommonUtil.EMPTY_STRING_ARRAY;
    verify(checkConfig, getPath("InputFinalClassAnonymousClass.java"),
          expected);
}
```

We now turn to a bug report that is not successfully reproduced. Table IX contains issue #11365 in the Checkstyle project. The bug is that CheckStyle incorrectly decides that a class should be declared `final`, and mistakenly raises an error.

TABLE IX: Bug Report Reproduction Failure: Lightly edited for clarity (Checkstyle Issue #11365)

Title **FinalClassCheck: False positive with anonymous classes**

... I have executed the cli and showed it below, as cli describes the problem better than 1,000 words

```
→src cat Test.java
[...]
public class Test {
  class a { // expected no violation
    private a(){} } }
[...]
→java [...] -c config.xml Test.java
Starting audit...
[ERROR] Test.java:3:5: Class a should be declared as final.
```

A FIB test generated by LIBRO is presented in Listing 5, which fails as the Java file it references in Line 5 is nonexistent. This highlights a weakness of LIBRO, i.e., its inability to create working environments outside of source code for the generated tests. However, if we put the content of `Test.java` from the report into the referenced file, the test successfully reproduces the bug, indicating that the test itself is functional, and that even when a test is initially incorrect, it may reduce the amount of developer effort that goes into writing reproducing tests.

VII. DISCUSSION

A. Manual Analysis of LIBRO Failures

Despite successfully reproducing 33.5% of the Defects4J bugs, in many cases LIBRO could not reproduce the bugs from the bug reports. To investigate which factors may have caused LIBRO to struggle, we manually analyzed 40 bug reports and corresponding LIBRO outputs. The most common cause of failure, happening in 13 cases, was *a need of helper definitions*: while the developer-written tests made use of custom testing helper functions which at times spanned hundreds of lines, LIBRO-generated tests were generally agnostic to such functions, and as a result could not adequately use them. This points to a need to incorporate project-specific information for language models to further improve performance. Other failure reasons included low report quality in 11 cases (i.e., a human would have difficulty reproducing the issue as well), the LLM misidentifying the expected behavior in 8 cases, dependency on external resources in 6 cases (as was the case in Listing 5), and finally insufficient LLM synthesis length in 3 cases. This taxonomy of failures suggests future directions to improve LIBRO, which we hope to explore.

B. Code Overlap with Bug Report

As Just et al. point out [30], bug reports can already contain partially or fully executable test code, but developers rarely adopt the provided tests as is. To investigate whether LIBRO relies on efficient extraction of report content or effective synthesis of test code, we analyzed the 750 bug reports from Defects4J used in our experiment. We find that 19.3% of them had full code snippets (i.e., code parsable to a class or method), while 39.2% had partial code snippets (i.e., not a complete class or method but in the form of source code statements or expressions); finally 41.5% did not contain code snippets inside. Considering only the 251 bug reports that LIBRO successfully reproduced, the portion of containing the full snippets got slightly higher (25.1%), whereas the portion of bug reports with partial snippets was 37.9%, and 37.1% did not have code snippets. When LIBRO generated tests from bug reports containing any code snippets, we find that on average 81% of the tokens in the body of the LIBRO-generated test methods overlapped with the tokens in the code snippets.

We note that using full code snippets provided in reports does not always reproduce the bug successfully; recall that the Copy&Paste baseline in Figure 2 succeeded only on 36 bugs. Although whether a bug report contains full code snippets or not may affect the success or failure of LIBRO, LIBRO generated correct bug reproducing tests even from incomplete code, or without any code snippets. Thus, we argue that LIBRO is capable of both extracting relevant code elements in bug report and synthesizing code aligned with given a natural language description.

VIII. THREATS TO VALIDITY

Internal Validity concerns whether our experiments demonstrate causality. In our case, two sources of randomness threat internal validity: the flakiness of tests and the randomness of LLM querying. While we do observe a small number of flaky tests generated, the number of them is significantly smaller (<2%) than the overall number of tests generated, and as such we do not believe their existence significantly affects our conclusions. Meanwhile, we engage with the randomness of the LLM, performing an analysis in RQ2-1.

External Validity concerns whether the results presented would generalize. In this case, it is difficult to tell whether the results we presented here would generalize to other Java projects, or projects in other languages. While the uniqueness of our prompts and our use of GHRB cases provide some evidence that LIBRO is not simply relying on the memorization of the underlying LLM, it is true that LIBRO benefits from the fact that the underlying LLM, Codex, has likely seen the studied Defects4J projects during training. However, our aim is *not* to assess whether a specific instance of Codex has general intelligence about testing: our aim is to investigate the extent to which LLM architectures augmented with post-processing steps can be applied to the task of bug reproduction. For LIBRO to be used for an arbitrary project with a similar level of efficacy as in our study, we expect the LLM of LIBRO to have seen projects in a similar domain, or the target project itself. This can be achieved with fine-tuning the LLMs, as studied in other domains [31], [32] (note that Codex is GPT-3 fine tuned with source code data). As a due diligence, we checked how many tests generated from the Defects4J benchmark verbatim matched developer-committed bug reproducing tests. There were only such three cases, and all had the test code written verbatim in the report as well, suggesting it is likely they got verbatim answers from the report rather than from memorization. We also report a few general conditions for which LIBRO does not perform well: it does not generalize to

tests that rely on external files or testing infrastructure whose syntactic structure is significantly different from the typical JUnit tests (such as the Closure project in Defects4J).

IX. Related Work

A. Test Generation

Automated test generation has been explored since almost 50 years ago [2]. The advent of the object-oriented programming paradigm caused a shift in test input generation techniques, moving from primitive value exploration to deriving method sequences to maximize coverage [3], [4]. However, a critical issue with these techniques is the oracle problem [33]: as it is difficult to determine what the correct behavior for a test should be, automated test generation techniques either rely on implicit oracles [3], or accept the current behavior as correct - effectively generating regression tests [4], [34]. Swami [35] generates edge-case tests by analyzing structured specifications using rule-based heuristics; its "rigid"ness causes it to fail when the structure deviates from its assumptions, whereas LIBRO makes no assumptions on bug report structure.

Similar to our work, some techniques focus on reproducing problems reported by users: a commonly used *implicit* oracle is that the program should not crash [33]. Most of existing work [5], [6], [36]–[38] aim to reproduce crashes given a stack trace, which is assumed to be provided by a user. Yakusu [39] and ReCDroid [40], on the other hand, require user reports written in specific formats to generate a crash-reproducing test for mobile applications. All the crash-reproducing techniques differ significantly from our work as they rely on the crash-based implicit oracle, and make extensive use of of SUT code (i.e., they are white-box techniques). BEE [19] automatically parses bug reports to classify sentences that describe observed or expected behavior but stops short of actually generating tests. To the best of our knowledge, we are the first to propose a technique to reproduce general bug reports in Java projects.

B. Code Synthesis

Code synthesis also has a long history of research. Traditionally, code synthesis has been approached via SMT solvers in the context of Syntax-Guided Synthesis (SyGuS) [41]. As machine learning techniques improved, they showed good performance on the code synthesis task; Codex demonstrated that an LLM could solve programming tasks based on natural language descriptions [14]. Following Codex, some found that synthesizing tests along with code was useful: AlphaCode used automatically generated tests to boost their code synthesis performance [42], while CodeT jointly generated tests and code from a natural language description [12]. The focus of these techniques is not on test generation; on the other hand, LIBRO processes LLM output to maximize the probability of execution, and focuses on selecting/ranking tests to reduce the developer's cognitive load.

X. Conclusion

In this paper, we first establish that the report-to-test problem is important, by inspecting relevant literature and performing an analysis on 300 open source repositories. To solve this problem, we introduce LIBRO, a technique that uses a pretrained LLM to analyze bug reports, generate prospective tests, and finally rank and suggest the generated solutions based on a number of simple statistics. Upon extensive analysis, we find that LIBRO is capable of reproducing a significant number of bugs in the Defects4J benchmark, and perform detailed analyses about the requirements of using the technique. We further experiment with a real-world report-to-bug dataset that we have collected: we find that LIBRO shows similar performance on this dataset when compared to the Defects4J benchmark, demonstrating its versatility. In both datasets, LIBRO successfully identifies when the bug is reproduced by which test, showing that LIBRO can minimize developer effort as well. We hope to expand upon these results and explore the synergy with existing test-generation techniques to further help practitioners.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and the Institute for Information & Communications Technology Promotion grant funded by the Korean government MSIT (No.2022-0-00995).

References

[1] R. Haas, D. Elsner, E. Juergens, A. Pretschner, and S. Apel, "How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. Association for Computing Machinery, 2021, pp. 1281–1291. [Online]. Available: https://doi.org/10.1145/3468264.3473922

[2] W. Miller and D. L. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. 2, no. 3, pp. 223–226, 1976.

[3] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*. ACM, 2007, pp. 815–816.

[4] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[5] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen, "Single-objective versus multi-objectivized optimization for evolutionary crash reproduction," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds. Springer International Publishing, 2018, pp. 325–340.

[6] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 101–110.

[7] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "Ifixr: Bug report driven program repair," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. Association for Computing Machinery, 2019, pp. 314–325. [Online]. Available: https://doi.org/10.1145/3338906.3338935

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. ACM, 2014, pp. 437–440.

[9] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 416–426.

[10] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. Association for Computing Machinery, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[12] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[13] A. Sarkar, A. Gordon, C. Negreanu, C. Poelitz, S. Srinivasa Ragavan, and B. Zorn, "What is it like to program with artificial intelligence?" *arXiv preprint arXiv:2208.06213*, 2022.

[14] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[15] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 579–590.

[16] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[17] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '19. IEEE Press, 2019, pp. 61–70. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00015

[18] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[19] Y. Song and O. Chaparro, "Bee: A tool for structuring and analyzing bug reports," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. Association for Computing Machinery, 2020, pp. 1551–1555. [Online]. Available: https://doi.org/10.1145/3368089.3417928

[20] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, Jan 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09762-1

[21] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2022. [Online]. Available: https://arxiv.org/abs/2205.11916

[22] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[23] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. Association for Computing Machinery, 2016, pp. 165–176. [Online]. Available: https://doi.org/10.1145/2931037.2931051

[24] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. Association for Computing Machinery, 2022, pp. 2228–2240. [Online]. Available: https://doi.org/10.1145/3510003.3510040

[25] M. Soltani, P. Derakhshanfar, X. Devroey, and A. Van Deursen, "A benchmark-based evaluation of search-based crash reproduction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 96–138, 2020.

[26] C. Thunes, "javalang: Pure Python Java parser and tools," https://github.com/c2nes/javalang, 2022.

[27] R. Premraj, T. Zimmermann, S. Kim, and N. Bettenburg, "Extracting structural information from bug reports," in *Proceedings of the 2008 international workshop on Mining software repositories - MSR '08*, ACM Press. ACM Press, 05/2008 2008, pp. 27–30.

[28] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2020.

[29] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Martin, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, pp. 1936–1964, 2016.

[30] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, "Comparing developer-provided to user-provided tests for fault localization and automated program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 287–297.

[31] R. Tinn, H. Cheng, Y. Gu, N. Usuyama, X. Liu, T. Naumann, J. Gao, and H. Poon, "Fine-tuning large neural language models for biomedical natural language processing," *CoRR*, vol. abs/2112.07869, 2021.

[32] L. Wang, H. Hu, L. Sha, C. Xu, K. Wong, and D. Jiang, "Finetuning large-scale pre-trained language models for conversational recommendation with knowledge graph," *CoRR*, vol. abs/2110.07477, 2021.

[33] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[34] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *CoRR*, vol. abs/2009.05617, 2020. [Online]. Available: https://arxiv.org/abs/2009.05617

[35] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 188–199.

[36] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2014.

[37] J. Xuan, X. Xie, and M. Monperrus, "Crash reproduction via test case mutation: Let existing test cases help," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 910–913.

[38] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Botsing, a search-based crash reproduction framework for java," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 1278–1282.

[39] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. ACM, 2018, pp. 141–152. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213869

[40] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G.J. Halfond, "Recdroid: Automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.

[41] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security, M. Irlbeck, D. A. Peled, and A. Pretschner, Eds. IOS Press, 2015, vol. 40, pp. 1–25. [Online]. Available: https://doi.org/10.3233/978-1-61499-495-4-1

[42] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago *et al.*, "Competition-level code generation with AalphaCode," *arXiv preprint arXiv:2203.07814*, 2022.