ChatGPT-Based Test Generation for Refactoring Engines Enhanced by Feature Analysis on Examples

Chunhao Dong† [1], Yanjie Jiang† [2], Yuxia Zhang [1], Yang Zhang [3], and Hui Liu* [1]

[1]Beijing Institute of Technology, Beijing, China, {dongchunhao22, yuxiazh, liuhui08}@bit.edu.cn
[2]Peking University, Beijing, China, {jiangyanjiese}@gmail.com
[3]Hebei University of Science and Technology, Shijiazhuang, Hebei, China, {uzhangyang}@foxmail.com

***Abstract*—Software refactoring is widely employed to improve software quality. However, conducting refactorings manually is tedious, time-consuming, and error-prone. Consequently, automated and semi-automated tool support is highly desirable for software refactoring in the industry, and most of the main-stream IDEs provide powerful tool support for refactoring. However, complex refactoring engines are prone to errors, which in turn may result in imperfect and incorrect refactorings. To this end, in this paper, we propose a ChatGPT-based approach to testing refactoring engines. We first manually analyze bug reports and test cases associated with refactoring engines, and construct a feature library containing fine-grained features that may trigger defects in refactoring engines. The approach automatically generates prompts according to both predefined prompt templates and features randomly selected from the feature library, requesting ChatGPT to generate test programs with the requested features. Test programs generated by ChatGPT are then forwarded to multiple refactoring engines for differential testing. To the best of our knowledge, it is the first approach in testing refactoring engines that guides test program generation with features derived from existing bugs. It is also the first approach in this line that exploits LLMs in the generation of test programs. Our initial evaluation of four main-stream refactoring engines suggests that the proposed approach is effective. It identified a total of 115 previously unknown bugs besides 28 inconsistent refactoring behaviors among different engines. Among the 115 bugs, 78 have been manually confirmed by the original developers of the tested engines, i.e., IntelliJ IDEA, Eclipse, VScode-Java, and NetBeans.**

***Index Terms*—refactoring, ChatGPT, refactoring engines, differential testing**

I. Introduction

The practice of software refactoring plays a pivotal role in modern software development, serving as a crucial process for improving software quality [16]. It can enhance source code's maintainability, readability, and scalability, mitigating technical debts [16]. However, it is often tedious, time-consuming, and error-prone to conduct refactorings manually [27]. Even the most experienced programmers can inadvertently introduce bugs or overlook complexities during manual refactoring, which could compromise the applications' integrity. To alleviate the shortcomings of manual refactoring, automated and semi-automated tool support is highly desirable for software refactoring in the industry, and most of the main-stream IDEs provide powerful tool support for refactoring. Such tool support thereby significantly reduces the likelihood of human errors and enhances productivity.

Testing of refactoring engines is critical because buggy engines may result in buggy programs when developers conduct refactorings with the engines. Although this issue has been studied for decades and refactoring engines have been widely and frequently used for decades by millions of developers, state-of-the-practice refactoring engines still suffer from defects. Furthermore, testing these refactoring engines presents its own set of unique challenges. On one side, the input of the refactoring engines is complete programs that are often lengthy and complex. As a result, the search space for test input generation (i.e., generation of test programs) is large, making it challenging to find suitable input that may trigger the defects in refactoring engines. On the other side, it is often difficult, if not impossible, to distinguish failed test cases from past ones because we cannot automatically generate or infer the expected output for the test cases.

Test case generation approaches have been proposed to improve the reliability of refactoring engines. For example, Daniel et al. [21] proposed a novel approach to generating test programs for given refactoring types, e.g., *encapsulate field refactoring*. To the best of our knowledge, it is the first testing approach specially designed for refactoring engines. They manually analyzed the execution of given types of refactoring, designed heuristics-based rules to generate desired source code elements and structures, and then merged the generated elements/structures into a program that was finally fed into the refactoring engine. Such a novel approach successfully identified some unknown bugs in multiple refactoring engines. Gligoric et al. [25] proposed an end-to-end testing approach for refactoring engines. They randomly applied refactorings in open-source applications, compiled the refactored applications, and retrieved the failed cases where the refactored version could not be compiled or the refactoring engines reported errors. They manually analyzed such failed cases, removed duplicate bugs, and reported the remaining bugs. Although such specially designed approaches have significantly improved the state of the art in the testing of refactoring engines, refactoring engine bugs are being reported from time to time.

To this end, in this paper, we propose a ChatGPT-based approach to generating test programs for refactoring engines. It differs from existing approaches in the following two aspects.

† Chunhao Dong and Yanjie Jiang made equal contributions to this work.
* Corresponding author: Hui Liu

First, it exploits existing bugs of refactoring engines. We first collect bug reports associated with refactoring engines, manually analyze each of them, and specify fine-grained features of the test programs that could trigger the bugs. Such features are then employed to guide the test generation by requesting the generated programs to contain features randomly selected from the feature library. Such a feature-guided test program generation may improve the efficiency of the proposed approach by reducing the number of unfruitful test programs, i.e., those that cannot trigger bugs of refactoring engines. To the best of our knowledge, we are the first to analyze and exploit existing bugs for the automated testing of refactoring engines. Second, we leverage ChatGPT [17], one of the most advanced large language models, to generate test programs according to the requested features. While traditional program synthesis methods are struggling to create diverse test programs that effectively represent these features (described in natural language), LLMs show promise in automating this process due to their strengths in both natural language understanding and program generation. Existing studies suggest that LLMs have great potential in understanding natural languages and generating source code according to given specifications [26], [40]. From the manually identified critical features associated with existing bugs, we randomly select some fine-grained features and request ChatGPT to generate complete programs that contain the requested features. To the best of our knowledge, we are the first to employ LLMs to generate test programs for refactoring engines. Our initial evaluation of four mainstream refactoring engines suggests that the proposed approach is effective. It identified a total of 115 previously unknown bugs besides 28 inconsistent refactoring behaviors. Among the 115 bugs, 78 have been manually confirmed by the original developers of the tested refactoring engines.

The contribution of this paper is as follows:

- **A novel approach** to testing refactoring engines. To the best of our knowledge, it is the first approach in this line that exploits existing bugs. It is also the first in this field to exploit large language models.
- **A list of manually identified features** that are associated with defects in refactoring engines. Such fine-grained features could be employed to guide the testing of refactoring engines. The features (as well as the replication package of the paper) are publicly available at [13].
- **A list of previously unknown defects** associated with widely-used refactoring engines. We reported 115 previously unknown bugs associated with IntelliJ IDEA, Eclipse, VScode-Java, and NetBeans, and **78 of them have been manually confirmed by the original developers** of the tested refactoring engines. The bug reports as well as their triggering tests are publicly available at GitHub [3].

II. RELATED WORK

A few approaches have been proposed to test refactoring engines. Daniel et al. [21] are the pioneers in this field. They built a generator to generate test programs for refactoring engines. At the heart of the technique is a framework that iteratively generates structurally complex test inputs that are instantiated to generate abstract syntax trees representing Java programs. Besides the test program generation, they also constructed six oracles to check the correctness of conducted refactoring. With the program generator and test oracles, they found previously unknown bugs in Eclipse and NetBeans.

While previous work focused on the generation of Java and C programming languages, Drienyovszky et al. [24] designed an automated testing framework to randomly generate Erlang programs for testing of refactoring engines. They also proposed a novel approach to validating the equivalence of code before and after refactoring. Different from such random generation, JDOLLY [9] exhaustively generates programs within a given scope, e.g., *declare a class without fields*. Soares et al. [36] proposed a fully automated framework for testing refactoring engines. It is composed of two parts, i.e., a program generator, called *JDOLLY* [9], and a safety validation tool, called *SAFEREFACTOR* [34]. It first generates numerous test programs with *JDOLLY*, invokes refactoring engines to conduct refactorings on the generated test programs, and then validates the correctness of refactoring by *SAFEREFACTOR*.

Different from approaches introduced in the preceding paragraphs, the approach proposed by Gligoric et al. [25] does not generate test programs. Instead, it takes open-source applications as input, and randomly applies refactoring to them. It distinguishes failure cases where the refactored versions fail to compile or the refactoring engines throw exceptions. It presents such failure cases to human experts for validation.

Refactoring engines employ pre-conditions to prevent illegal refactorings. However, such pre-conditions could be overly strong, preventing legal refactorings. Mongiovi et al. [28] proposed a novel approach to detecting overly strong preconditions. If the precondition for a warning message is removed and the final version of the refactoring engine can safely perform the refactoring without altering the external behavior of the test program, the precondition is considered overly strong. Soares et al. [35] generate test cases, and run multiple refactoring engines with the test cases. If some of the engines reject a refactoring whereas others conduct it, they leverage Safe Refactor [34] to assess the correctness of the conducted refactoring. If the refactoring is conducted correctly, refactoring engines that refuse to conduct this refactoring must have overly strong conditions for the refactoring.

Silva et al. [33] explored the efficiency of automatic test case generation tools, i.e., Randoop [12] and EvoSuite [5], in detecting faulty refactoring. For each refactoring, they generated test cases with Randoop and EvoSuite, ran the test cases before and after the refactoring, and compared the results of the test cases. However, their evaluation results suggest that such a test case generation-based approach missed more than half of the faulty refactoring.

Some studies [20], [22] have explored the capabilities of LLMs in software refactoring. Pomian et al. [30], [31] introduced EM-Assist that leverages LLMs to generate refactoring suggestions. Dilhara et al. [23] proposed a PyCraft that

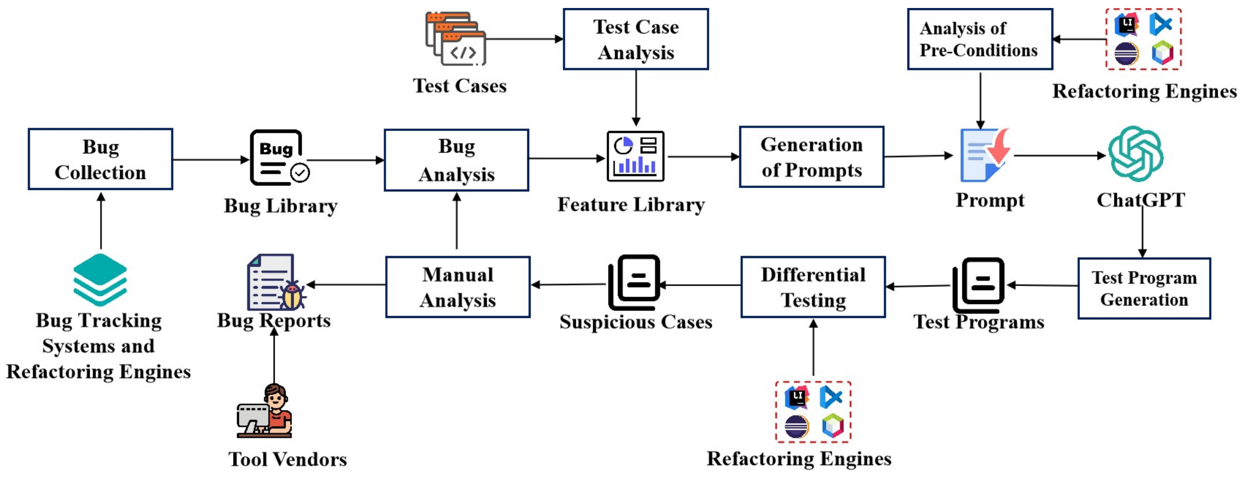


Fig. 1. Overview

combines static and dynamic analysis with LLMs capabilities to generate semantically correct and useful code variations. AlOmar et al. [20] explored conversations between developers and ChatGPT to gain deeper insights into how developers identify areas for code improvement. Shirafuji et al. [32] proposed an approach to selecting the most suitable refactoring examples for LLMs' few-shot prompts.

Our approach proposed in this paper differs from the approaches introduced in the preceding paragraphs in two aspects. First, our approach is the first in this line that guides the generation of test programs for refactoring engines with features derived from known defects of refactoring engines. Another difference is that our approach is the first approach in this line that generates test programs for refactoring engines with large language models.

III. Approach

In this section, we present the proposed approach. We first present an overview of the approach in Section III-A and then present the details in the rest part of this section.

A. Overview

Fig.1 presents an overview of the proposed approach. Notably, the approach is semi-automated, requesting human interventions. In the early stage, it requests experts to analyze history bugs and to manually identify fine-grained features of test programs that may trigger the given bugs. In the last stage, it requests software engineers to manually confirm the reported bugs. Besides these two states, however, the proposed approach is fully automated. Overall, it works as follows:

Step 1: We collect bug reports associated with refactoring engines from bug tracking systems.

Step 2: We manually analyze the collected bug reports. For each bug, we should identify a set of fine-grained features that can trigger the bug. A sample feature is "*the original class of the method to be moved contains a field of the target class type*" that is identified from an example bug associated with *move method refactoring*. The resulting features are added to the feature library.

Step 3: We collect public test suites specially designed for the refactoring engines. Such test cases were manually designed by the tool vendors to validate reported bugs or to prevent potential bugs.

Step 4: We manually analyze the resulting test suites. For each test case, we identify a set of features that a testing program breaking the given test case should have. Such features are also added to the feature library.

Step 5: From the resulting feature library, we randomly select some features that are described in natural languages, and request ChatGPT to generate ten programs that contain all such features.

Step 6: Each of the generated test cases is fed into refactoring engines to conduct differential testing. If any of the refactoring engines results in syntax errors, the test program and the errors are reported for manual analysis. If the refactoring engines generate inconsistent results, we also report it as a potential bug.

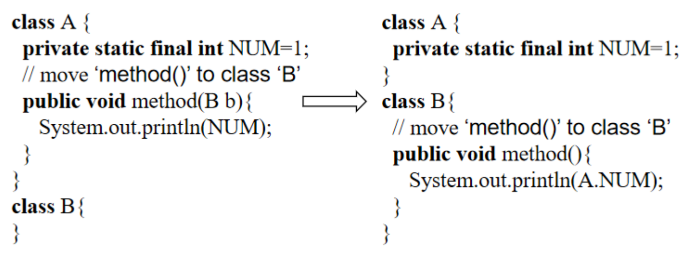
Step 7: Manually confirmed bugs are analyzed to identify additional features that are associated with the bug. Such features are added to the feature library.

Step 8: We turn to Step 5 and repeat the test generation (Step 5) and differential testing (Step 6 and Step 7) until the allocated resource runs out.

Details of the key steps are presented in the following subsections.

B. Bug Collection

The proposed approach is driven by existing bugs, and thus we should first collect a set of bugs associated with refactoring engines. To do this, we first identify the bug tracing systems for well-known refactoring engines. Note that Eclipse and IntelliJ IDEA have their own bug-tracking system at https://bugs.eclipse.org/bugs and https://youtrack.jetbrains.com/issues, respectively. Bug reports for Netbeans and VScode can be found at *https://github.com/apache/netbeans* and



```
class A {
  private static final int NUM=1;
  // move 'method()' to class 'B'
  public void method(B b){
    System.out.println(NUM);
  }
}
class B{
}
```

```
class A {
  private static final int NUM=1;
}
class B{
  // move 'method()' to class 'B'
  public void method(){
    System.out.println(A.NUM);
  }
}
```

Fig. 2. Example Bug of *Move Method Refactoring*

https://github.com/microsoft/vscode, respectively. Notably, bug reports for refactoring engines (integrated into IDEs) are often mixed with other bug reports associated with the IDEs. To distinguish them, we conducted a pilot analysis, which revealed that most refactoring-related bug reports explicitly mention the refactoring names. Based on this finding, we search for refactoring engine-related bug reports on the bug tracking systems with refactoring names (e.g., "*move method*" and "*rename*"). Note that some of the bug reports have not yet been confirmed by the developers, and thus we only retrieved bug reports whose labels are *"bug"* or *"fixed"*.

Note that we focus on the correctness of the output (source code) of the refactoring engines, and thus bugs irrelevant to it should be excluded. A typical example [2] comes from Eclipse where the buggy behaviors concern the GUI interface: After a developer clicks the "*OK*" button to conduct a *move method refactoring*, the "*OK*" button should have been disabled because the refactoring has already been done. It is excluded because the buggy behavior does not influence the correctness of the output (source code) of the engine. In one word, a bug is excluded if it does not influence the resulting source code. Notably, the resulting bug library contains 4 bugs reported by Gligoric et al. [25], and all of them have been fixed.

C. Bug Analysis

We manually analyze each of the collected bugs by following the widely-used analysis pattern [37], [39]. Following this pattern, we should answer the following questions for each of the collected bug reports:

- ***How did it happen?*** The participants should first understand how to repeat the buggy behaviors, what is the input (including the testing program and executed refactoring command as well as its associated parameters), what is the output (buggy program), and why the output is buggy. Only if the participants fully understand the bug, they have the chance to figure out the features that could trigger the buggy behaviors of the refactoring engines.
- ***Which features of the test program trigger the bug?*** Based on the understanding of the buggy behaviors and the test program, participants should generalize the given test program, specifying what kind of programs may trigger the buggy behaviors as the given program did. More specifically, participants should explicitly specify the indispensable fine-grained features of the test programs.

We take the example bug report [6] from IntelliJ IDEA as an example to illustrate the process. The buggy behavior appears when the user wants to apply the refactoring in Fig.2, i.e., transferring the source code on the left of the figure to the one on the right. It moves method *method* from class *A* to class *B*. The compilation message (*error message*) suggests that the refactoring is buggy because the statement "*System.out.println(A.NUM)*" is incorrect: *A.NUM* is illegal since the field *NUM* of class *A* is private. So far, we have answered the question: "*how did it happen?*"

To generalize the triggering tests, we ignore irrelevant code that may not change the test results. For the example in Fig. 2, the keyword "*final*" could be removed without changing the testing result. The same is true for the data type of the field (i.e., "*int*") and the literal "*1*". The statement "*System.out.println*" is not indispensable, either: we can safely replace "*System.out.println(NUM)*" with "*int x=NUM*" or simply "*NUM*". After the exclusion, we found that the indispensable features include:

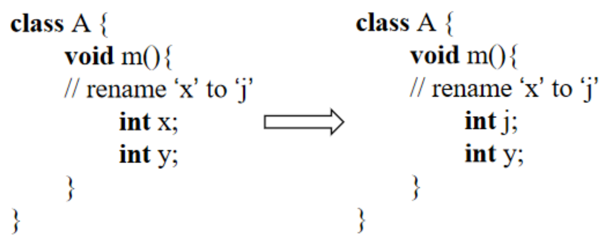
- The program should have at least two classes;
- The source class should have a private field;
- The to-be-removed method within the source class should access the private field;
- The to-be-removed method should have a parameter that is an instance of the target class;
- The source class and the target class should not have an inheritance relationship or inclusion relation.

We notice that disabling any of the features would destroy the triggering test case, which suggests that the identified features are concise.

D. Test Case Analysis

As specified in Section III-B, we collect not only bug reports but also test cases associated with refactoring engines. We have identified program features for each bug as specified in Section III-C. In this section, we explain how the collected test cases are analyzed and how fine-grained features are identified from them.

For a given test case, we should first figure out the tested refactoring (i.e. *which refactoring is invoked by the test case*) and which test program is used as the test input. All such information could be identified by static analysis and dynamic execution of the test case. Second, we should figure out whether it specifies a case where the refactoring should be rejected or a case where the refactoring should be conducted successfully. Note that some test cases are created to validate the pre-condition checking of the refactoring by presenting cases where the refactoring should not be applied. A good example is the test case *test3()* [14] from Eclipse that illustrates when *rename local variable refactoring* should be rejected: If you rename a local variable from "*i*" to "*j*" that is identical to another local variable of the same method, the refactoring should be rejected. For such a triggering test case, we should run the test case, check the warnings of the refactoring engine, and figure out why the refactoring should be rejected. Based on the analysis, we can specify the required features of the



```
class A {                          class A {
    void m(){                          void m(){
    // rename 'x' to 'j'               // rename 'x' to 'j'
        int x;          ==>                int j;
        int y;                             int y;
    }                                  }
}                                  }
```

Fig. 3. Test Case for Normal Scenarios

Description:
Please generate ten java test programs that test the move method refactoring and meet the following constraints
Conditions:
1. The original class of the method to be moved contains a field of the target class type.
2. The name of the method to be moved is 'methodToBeMoved'
3. The target class name for the method to be moved is 'TargetClass'
Features:
1. The method to be moved calls a private field within the original class
2. The modifier of the method to be moved is 'protected'
3. The target class of the method to be moved is an anonymous class

Fig. 4. An Example Prompt for Testing 'Move Method' Refactorings

triggering test program as well as the refactoring parameters. For example, for the preceding test case, the feature is that *"a local variable is renamed with a new name that is identical to one of its sibling local variables"*.

Some test cases are in contrast created to validate some normal refactoring scenarios. A typical example for *renaming local variable refactoring* is shown in Fig. 3. Although we do not identify features directly from such normal cases, they are useful for extracting features from triggering test cases (*as described in the preceding paragraph*). Only if the triggering tests are compared against normal cases, we can identify accurately the indispensable features of the triggering tests.

E. Analysis of Refactorings' Pre-conditions

Some refactorings have strong pre-conditions and the refactorings would be rejected unless such pre-conditions are satisfied. For example, if you want to move a method M from class A to class B whereas the method does not access any instance of class B and class A does not have any field whose type is B, the refactoring engines would reject the refactoring because invocations to M (in class A) cannot be delegated to it after moving to class B. However, if we request large language models to randomly generate testing programs, most refactoring attempts to move methods within the resulting programs would be rejected by refactoring engines for the same reason we discussed above. As a result, most (more than 80% in our initial study) of the generated programs are useless for the testing of refactoring engines, wasting expensive testing resources.

To reduce the chance of generating such useless programs, we analyze the pre-conditions explicitly specified by the refactoring engines. Notably, the mainstream refactoring engines, like Eclipse, explicitly specify their preconditions for each category of refactoring, which significantly facilitates the collection of pre-conditions. Since the pre-conditions specified by the refactoring engines could be imperfect (even incorrect), we only collect the most common pre-conditions that are accepted by all refactoring engines. Such common pre-conditions would be employed to guide LLM-based program generation.

F. Generation of Prompts

An example prompt is presented in Fig. 4. The prompt is composed of three parts: description, conditions, and features.

Description: The description part specifies the task, i.w., what we expect the LLM to do. Notably, the description should explicitly specify the type of to-be-tested refactoring. In the example case, the refactoring type is *move method refactoring*.

Conditions: This part presents the pre-conditions for the given type of refactoring. All such conditions have been identified and documented in Section III-E. Notably, besides the pre-conditions, we also specify some parameters for the targeted refactoring. For example, the prompt in Fig. 4 specifies the name of the to-be-moved method as well as the class name where the method should be moved. With such parameters, we can automatically identify where the refactoring should be applied as well as the parameters requested by the refactoring. As a result, the following differential testing could be automated.

Features: This is the key of the prompt, specifying what kind of features the generated program should have. The example in Fig. 4 presents three features.

The features in the prompt are randomly selected from the feature library constructed in the preceding steps. The feature library is composed of records where each record presents the features collected from a bug report (as specified in Section III-C) or a test case (as specified in Section III-D). Each record is composed of the description of the bug (or test case), the associated refactoring engine (e.g., Eclipse or IDEA), the type of the involved refactoring (noted as *refactoring type*), and a list of features. For a given type of refactoring rt (e.g., *move method refactoring*), we automatically generate the "*features*" section as follows:

- From the feature library, retrieve all records whose refactoring types equals rt. The resulting records are noted as RCs.
- Compute the maximal size of the feature lists in RCs, noted as Max.
- Randomly generate a positive integer n where $1 \leq n \leq Max$.
- From all feature lists in RCs, randomly select n unique features, and insert them into "*features*" section of the prompt.

As shown in Fig. 4, a prompt consists of three components: DESCRIPTION, CONDITION, and FEATURES. The first two components remain constant for a given refactoring type, while FEATURES are generated automatically as specified in the preceding paragraphs. Consequently, we manually define the DESCRIPTION and CONDITION for each refactoring type, after which the prompts are generated automatically. To support reproducibility, we have made the templates (i.e., DESCRIPTION and CONDITION) for each refactoring type publicly available online [11].

G. Differential Testing

The automatically generated prompts are fed automatically into LLMs (e.g., ChatGPT), and the latter generates the requested test program for each prompt. From the generated program as well as the prompt (especially its '*conditions*' part), we can automatically generate a script to invoke the intended refactoring on the generated program because all parameters requested for the refactoring have been explicitly specified in the prompt and the test program. What the script should do is deliver such information to the targeted refactoring engines via public APIs.

The most challenging part of the differential testing is the test oracle problem, i.e., how we can know whether the conducted refactoring is correct or incorrect. To identify buggy refactoring, we automatically conduct the same refactoring (with exactly the same parameters and an identical test program) with multiple refactoring engines. A refactoring r conducted by refactoring engine E is reported as potentially buggy if and only if:

- The refactoring r conducted by E results in an error message or runtime exception;
- r is refused by E whereas r is conducted by any other refactoring engines;
- r is conducted by E whereas r is rejected by any other refactoring engines;
- The resulting program (after refactoring) has some syntax errors that do not exist in the initial test program;
- Or the resulting program generated by E is different from that generated by other refactorings.

The comparison of the resulting programs (generated by different refactoring engines) is conducted in two steps. First, we compare them to normal texts with a token-by-token comparison. If two programs are identical concerning the token-by-token comparison, they are deemed identical. Otherwise, we conduct an AST-based comparison to validate whether they are essentially the same concerning the semantics of programs. Notably, refactoring may change the formats of the program, or insert comments to explain the refactoring. Consequently, it is likely that two outputs are essentially equivalent even if they are not literally identical. To this end, we compare the programs as AST and compare their AST-based similarity. If the trees are different, we report the refactoring as a suspicious refactoring that is potentially buggy.

H. Manual Confirmation and Feature Identification

The potentially buggy refactoring should be manually confirmed by test engineers before they are reported to tool vendors of the refactoring engines. The manual confirmation is exactly the same as test engineers to confirm bugs in traditional software systems. However, after the buggy refactorings are confirmed, we should take them as additional bugs and conduct the feature identification on them as we have done in Section III-C. The resulting features are added to the feature library, and this updated library is employed for future testing.

You are an expert in testing refactoring engines, and your task is to summarize the kind of features a test Java program should have so that it could trigger the same bug in **[refactoring type]** as the following test program triggers.
Notably, "features" refer to code entities with specific characters as well as relationships among such entities.
The given test program that triggers a bug in **[refactoring type]**: **[source code]**
The error messages of **[refactoring type]**: **[error message]**
To make it clear, we present the following three examples to illustrate how to extract different features that result in a bug during the **[refactoring type]** process.
Example 1
The given test program that triggers a bug in **[refactoring type]**: **[source code]**
The error messages of **[refactoring type]**: **[error message]**
The summarized features: **[features]**
Example 2
The given test program that triggers a bug in **[refactoring type]**: **[source code]**
The error messages of **[refactoring type]**: **[error message]**
The summarized features: **[features]**
Example 3
The given test program that triggers a bug in **[refactoring type]**: **[source code]**
The error messages of **[refactoring type]**: **[error message]**
The summarized features: **[features]**

Fig. 5. Prompt Template for Automated Feature Extraction

I. ChatGPT-Based Automatic Feature Extraction

As introduced in Section III-C, it is often time-consuming to manually extract features from real-world bugs. To this end, this section proposes a ChatGPT-based approach to automate the extraction. The prompt template is presented in Fig.5. It explicitly specifies the task and responsibility of ChatGPT. It also presents the triggering test program and the associated error message. To facilitate in-context learning, the prompt template also presents three examples of the same refactoring type. Notably, such examples are constructed manually in advance: for each kind of software refactoring, we select three earliest (according to their submission time) bugs of the given refactoring type, and manually summarize features from them. Prompts for each type of refactoring are available online [1].

IV. Case Study

In this section, we tested four well-known and widely used refactoring engines with the proposed approach, which resulted in 78 confirmed (by tool vendors) previously unknown bugs and 37 to-be-confirmed bug reports.

A. Research Questions

The case study should answer the following research questions:

- **RQ1:** Can the proposed approach identify unknown defects in widely-used and extensively-tested refactoring engines?
- **RQ2:** To what extent can the proposed approach improve the state of the art?

- **RQ3:** Can the proposed approach be applied to test additional refactoring engines without additional human analysis/adaption?
- **RQ4:** Is ChatGPT-based automatic feature extraction comparable to human extraction?

RQ1 concerns the usefulness of the proposed approach, i.e., its ability to detect unknown defects in the state-of-the-practice refactoring engines. RQ2 concerns the comparison between the proposed approach and existing ones. Answers to this question would contextualize the proposed approach. RQ3 concerns the generality of the proposed approach. If we have to repeat the manual analysis requested by the proposed approach for every new refactoring engine, the testing could be expensive and less cost-effective. However, if the resulting approach calibrated on some engines could be applied without any adaption to other refactoring engines, the approach and its implementation could be taken as a ready-to-use tool, resulting in a highly cost-effective testing approach. RQ4 concerns the performance of ChatGPT-based automatic feature extraction.

B. Tested Refactoring Engines

According to existing studies [18], [19], the activities of refactoring and other programming tasks are often intertwined. Consequently, most of the mainstream refactoring engines are integrated into IDEs so that developers can switch between refactoring and other programming tasks. Consequently, in this study, we selected three state-of-the-practice refactoring engines integrated with mainstream IDEs, i.e., Eclipse, IntelliJ IDEA, and NetBeans. All these IDEs have built-in and powerful refactoring engines. To increase the diversity, we also selected a plugin for the case study. The plugin, called VScode-java [15], is a Java extension of VScode. Notably, VScode itself does not have any built-in refactoring engine, and thus it depends on third-party plugins like VScode-java.

We accessed the latest version of the selected refactoring engines for the case study. The versions of the engines (IDEs) are specified as follows:

- IntelliJ IDEA, version 2024.1 [8]
- Eclipse, version 2024.3 [4]
- NetBeans, version 21 [10]
- VScode-java (VScode for short), version 1.22.0 [15]

C. Implementation and Environment

Because some of the involved refactoring engines, like Eclipse, only support Java, our initial implementation is confined to Java as well. The full implementation of the proposed approach is publicly available at GitHub [13] to facilitate third-party replication or further extension/evaluation. The experiment uses the default settings (including the temperature) of ChatGPT.

D. Bug Collection and Feature Extraction

To make the proposed useful, we should construct a feature library as described in SectionsIII-C by collecting and analyzing bug reports as well as test suites. To this end, in this study, we collected bug reports from the bug-tracking systems of Eclipse [4], IntelliJ IDEA, and VScode-java [15]. We did not collect bug reports from NetBeans because we want to validate whether the feature library constructed for some refactoring engines (i.e., Eclipse, IntelliJ IDEA, and VScode-Java) could be applied without adaption to other engines (i.e., NetBeans). We also collected built-in test cases provided by the target refactoring engines. As a result, we collected 700 bug reports as well as 185 test cases. Notably, The case study was designed to test seven types of refactoring, i.e., *rename method*, *rename filed*, *rename variable*, *move method*, *extract variable*, *extract method*, and *inline method*. Such refactoring types were selected because all of them were supported by the selected refactoring engines and they are the most frequently applied refactoring accounting for 85% of refactoring activities [29].

Three authors of the paper manually analyzed the collected test cases and bug reports, and constructed a feature library according to the guidance specified in Section III-C. During the construction of the feature library, each bug report/test case was analyzed by three participants. The bug collection and feature extraction took the authors one man-months in total. They were all PhD students, with at least two years of experience in software refactoring and testing. For each bug report, we requested them to extract the features independently first, merge all features, and then check each feature together to remove duplicate or unnecessary features and polish the feature descriptions. The checking of suspicious cases was somewhat objective, and thus the kappa coefficient (0.91) among the participants is high. The resulting feature library is publicly available on GitHub [13] and a summary is presented in Table I.

TABLE I
RESULTING FEATURE LIBRARY

Metrics	Values
Number of bugs	700
Number of test cases	185
Number of refactoring types	7
Number of unique features	53
Average number of features per record	3.2

TABLE II
RESULTS ON THREE REFACTORING ENGINES

Metrics	Eclipse	IDEA	VScode	Total
# Test Programs	91,230	91,230	1,926	184,386
# Refactoring	80,368	82,742	1,697	163,110
# Suspicious Cases	861	1,028	1,164	3,053
# False alarms	72	72	72	216
# Buggy Cases	647	814	950	2,411
# Inconsistent Cases	142	142	142	426
# Confirmed Bugs	20	28	38	86
# Inconsistency	28	28	28	28
# Bugs Confirmed by Tool Vendors	15	18	26	59

E. RQ1: Effectiveness of the Proposed Approach

The evaluation results on three refactoring engines are presented in Table II. *#Test Programs* is the total number of test programs generated by the proposed approach. *#Refactoring* is the total number of refactoring the refactoring engine conducted. *#Suspicious Cases* is the total number of cases that the proposed approach suggested to be suspicious. *#False Alarms* is the total number of suspicious cases that were manually confirmed as safe, i.e., not buggy or inconsistent. *#Buggy Cases* is the total number of suspicious cases that were manually confirmed as buggy. *#Inconsistent Cases* is the total number of suspicious cases that were manually confirmed as inconsistent. *#Confirmed bugs* is the total number of bugs we reported to the tool vendors. *#Inconsistency* is the total number of inconsistencies between refactoring engines. Notably, *#Buggy Cases* is different from *#Confirmed bugs* in that some buggy cases may have been caused by the same bug, and thus such cases were merged and only a single bug was confirmed and reported. A similar relation exists between *#Inconsistent Cases* and *#Inconsistency*. All of the suspicious cases were analyzed and classified by three human experts in software refactoring, they have at least two years of research experience in software refactoring and more than five years of Java experience. They discussed together, and managed to reach a consensus on each of the suspicious cases. The human inspection took fifteen man-days. Notably, while deciding whether a refactoring was buggy, they followed the objective criterion: The refactoring is buggy if 1) the refactoring introduces new compilation errors or 2) we can find a test case (i.e., input to the generated test program) that makes the test program generate different outputs before and after the refactoring.

From Table II, we make the following observation:

- The approach successfully identified unknown bugs from the state-of-the-practice refactoring engines. It identified 20, 28, and 38 manually confirmed unknown bugs for Eclipse, IDEA, and VScode-Java, respectively. In total, 86 bugs were manually confirmed as previously unknown bugs. Notably, we have compared them against the existing bugs we collected, and manually confirmed that they were different from existing bugs. **59 of the bugs have been confirmed or fixed by the tool vendors of the tested engines.** In total, invoking ChatGPT during the evaluation incurred a cost of $260. That is, it costs $3 (charged by ChatGPT) on average to identify a unique bug in refactoring engines.
- The proposed approach successfully identified inconsistency among the evaluated refactoring engines. Although such inconsistency is not faulty, the consistency among different refactoring engines may confuse users. In total, the approach identified 28 inconsistencies among the three engines.
- The proposed approach was accurate, and most of the reported suspicious cases have been manually confirmed as questionable. Only 216 out of the 3,053 suspicious cases were denied, resulting in a high precision of 92.9%.

From Table III, we observe that *move method refactoring* is by far the most dangerous refactoring: We identified 24 bugs for it and 18 have been confirmed by tool vendors. The number of reported bugs (24) is by far greater than the second largest (14) in the table, and the number of confirmed bugs (18) is twice the second largest (9). It also has the largest number of inconsistencies. One possible reason for the great risk in *move method refactoring* is that it has complex pre-condition checking and involves complex cross-class modification. In contrast, the other refactoring types in Table III are often confined to a class or even a single method.

TABLE III
TYPES OF REFACTORING AND RAISED ISSUES

Refactoring Type	#Bugs	#Bugs Confirmed by Tool Vendors	#Inconsistency
Rename method	14	9	6
Rename field	5	3	3
Rename variable	10	7	1
Move method	24	18	8
Extract variable	9	7	4
Extract method	11	6	4
Inline method	13	9	2
Total	86	59	28

```
1 class ParentClass {
2     void method(){System.out. println ("1");}
3 }
4 class SubClass extends ParentClass{
5     void methodToBeRenamed(){System.out.println("2");}
6     void testMethod(){
7         method();
8     }
9 }
```

Listing 1. Example Bug with *Rename Method Refactoring*

A typical example of the identified bugs is presented in Listing 1. In this example, the test program is generated for testing *rename method refactoring*. When the method "*methodToBeRenamed*" (Line 5) is renamed as "*method*", the refactoring engines only update the name on Line 5, i.e., the declaration of the renamed method. However, the method invocation on Line 7 should have been updated. Before refactoring, the method invocation should invoke the method declared on Line 2. In contrast, the same statement after refactoring should invoke the method declared on Line 5: If the same method is declared in both superclass and subclass, the invocation in the subclass should access the one declared within the same class. As a result, the refactoring is incorrect. We notice that all of the tested refactoring engines make the same mistake in this example. We have reported this bug to the tool vendors, and IntelliJ IDEA has already acknowledged this bug (BugID will be presented upon acceptance of the paper.)

A typical inconsistent refactoring is presented in Listing 2. When we select the field *fieldToBeRenamed* on Line 4, and try to rename it as "*a*", IDEA would conduct the refactoring and yield the resulting code on Lines 10-16. However, Eclipse would refuse the renaming because the new name is

```
 1 //Source code before  refactoring :
 2 class SourceClass{
 3     // rename 'fieldToBeRenamed' to 'a'
 4     int fieldToBeRenamed;
 5     void method(int a){
 6         fieldToBeRenamed = 1;
 7     }
 8 }
 9 //Source code after  refactoring by IDEA:
10 class SourceClass{
11     // rename 'fieldToBeRenamed' to 'a'
12     int a;
13     void method(int a){
14         this.a = 1;
15     }
16 }
17 // Eclipse refuses the refactoring by warning 'Another
       name will shadow access to the renamed element.'
```

Listing 2. Example of Inconsistent Refactoring Policies

```
 1 //Source code before  refactoring :
 2 class SourceClass{
 3     void method(int i, int j, int k){
 4         // the refactoring is to extract "i*j" as a
              variable
 5         int v1=i*j;
 6         int v2=i*j*k;
 7     }
 8 }
 9 //Source code after  refactoring by IDEA:
10 class SourceClass{
11     void method(int i, int j, int k){
12         int i1 = i * j;
13         int v1=i1;
14         int v2=i*j*k;
15     }
16 }
17 //Source code after  refactoring by Eclipse:
18 class SourceClass{
19     void method(int i, int j, int k){
20         int i1 = i * j;
21         int v1=i1;
22         int v2=i1*k;
23     }
24 }
```

Listing 3. Example of False-Positives

identical to the parameter on Line 5, resulting in a warming "*Another name will shadow access to the renamed element*". In this case, neither of them produces any incorrect refactoring. IDEA conducts the refactoring without introducing any defects whereas Eclipse does not conduct any refactoring. However, such inconsistent policies for the same refactoring could confuse users who switch between IDEs. A possible reason for the inconsistency is that Eclipse would like to prevent any variable shadowing whereas IDEA prefers to follow developers' commands in case executing the commands would not introduce any syntax or semantic errors.

A typical example of a false positive is presented in Listing 3. In this example, we should extract the expression "*i*j*" on Line 5 as a local variable. IDEA extracts this expression (Line 13), and replaces it with the new variable (Line 14). However, Eclipse further replaces the identical expression "*i*j*" on Line 22 with the new variable as well. This case was reported by the proposed approach as suspicious because the Eclipse and IDEA made different modifications. However, with further analysis, we see that the two resulting programs (Lines 10–16 and Lines 18–24) are semantically equivalent although they have different texts and AST trees.

Notably, 6 bugs were not reported to the tool vendors. We found some bugs that are very challenging, if not impossible, to fix in the near future, and thus did not disturb the tool vendors. A typical example is *rename method*: When the renamed method is invoked via Java reflection, we should update the literal (i.e., the method name) accordingly. However, it is challenging to trace method invocation called with Java reflection by static source code analysis alone. Consequently, it is too difficult for refactoring engines to decide whether a given literal should be updated or not. Maybe a warning message is fine, and developers should make the final decision.

F. RQ2: Comparison against Existing Approach

To contextualize the performance of the proposed approach, we compared it against the approach proposed by Gligoric et al. [25]. For convenience, we called it *Gligoric's* in the rest of this paper. This approach was selected because it represented the state of the art in testing refactoring engines. Notably, SAFEREFACTOR [34] and Silva et al.'s work [33] could be employed to validate the correctness of conducted refactorings, but they could not generate test programs. What they generate are test cases for the to-be-refactored programs. Consequently, we can not compare our approach against them. *Gligoric's* should take some open-source applications as input, and thus we reused the applications Gligoric et al. [25] selected for its initial evaluation in their paper. To make the comparison fair, we allocated the same time slots for *Gligoric's* and our approach, i.e., five days with three human participants. Once the allocated time ran out, the evaluation terminated.

The evaluation results are presented in Table IV. From this table, we observe that it successfully identified previously unknown bugs from the state-of-the-practice refactoring engines. In total, 24 bugs were manually confirmed by the participants. It may suggest that *Gligoric's*, an approach proposed ten years ago, still works effectively on the latest refactoring engines. Notably, *Gligoric's* does not employ differential testing, and thus it does not report any inconsistent cases. It should be noted that the refactoring engines tested in our experiment are different (different products or different versions) from those tested by the original experiment of Gligoric's, and thus the results are not comparable between the two experiments.

By comparing the performance of the proposed approach against *Gligoric's*, we make the following observation:

- Our approach is more effective. It identified 86 bugs and 28 inconsistencies whereas *Gligoric's* identified 24 bugs only. The improvement in number of bugs is up

TABLE IV
PERFORMANCE OF *Gligoric's*

Metrics	Eclipse	IDEA	VScode	Total
# Test Programs	141,869	141,869	689	284,427
# Refactoring	97,243	102,973	521	200,737
# Suspicious Cases	227	462	408	1,097
# False alarms	0	0	0	0
# Buggy Cases	227	462	408	1,097
# Confirmed Bugs	5	8	11	24
# Inconsistency	0	0	0	0
# Bugs Confirmed by Tool Vendors	2	4	4	10

TABLE V
TESTING NETBEANS

Refactoring Type	#Bugs	#Bugs Confirmed by Tool Vendors	#Inconsistency
Rename method	5	3	5
Rename field	1	1	0
Rename variable	1	1	0
Move method	11	6	4
Extract variable	5	3	0
Extract method	3	2	1
Inline method	3	3	1
Total	29	19	11

to 258%=(86-24)/24. Notably, all such bugs were found with the same time cost (time interval). Concerning the number of bugs confirmed by tool vendors, our approach is much more effective than *Gligoric's* by improving the number from 10 to 59, with a relative improvement of 490%(59-10)/10.

- Only 13 defects were identified by both of them, the two approaches are complementary to each other. *Gligoric's* generated more test programs than our approach, it resulted in substantially fewer suspicious cases (1,097 vs. 3,053). One possible reason for the high efficiency is that our approach guides the generation of test programs with features derived from history bugs, and thus the generated programs have a greater chance of triggering buggy or inconsistent behaviors. Applying refactorings randomly as *Gligoric's* may result in numerous useless cases, wasting computing resources.

The two evaluated approaches are complementary to each other. Our approach and Gligoric's identified 86 and 24 bugs, respectively. However, only 13 bugs were identified by both of them, suggesting that the proposed approach and *Gligoric's* successfully identified 73 and 11 unique bugs, respectively. Consequently, combining both of them would increase the number of identified bugs from 86 to 97.

A significant advantage of *Gligoric's* is that it did not report any false alarms. The reason is that it identified suspicious cases with compilation errors and error messages from refactoring engines instead of differential testing. A cost of this advantage, however, is that *Gligoric's* cannot identify buggy cases where refactoring is conducted without any warning or complication errors but resulting in semantics changes. In contrast, our approach has the potential to identify such bugs, and it identified 20 such bugs from refactoring engines.

G. RQ3: Testing Additional Refactoring Engine

In this section, we applied the proposed approach to detect buggy refactoring in another refactoring engine, *NetBeans*. Note that *NetBeans* is an IDE independent of the previously analyzed three refactoring engines, i.e., Eclipse, IDEA, and VSCode-Java. Consequently, if the feature library derived from Eclipse, IDEA, and VSCode-Java could be used directly to detect the refactoring engine in NetBeans, the approach could be readily applied to various refactoring engines without time-consuming feature extraction. Notably, the differential testing was conducted between NetBeans and IDEA.

We observe from Table V that the proposed approach succeeded in testing the new refactoring engines, i,e., NetBeans. It successfully identified previously unknown bugs associated with all targeted refactoring types. In total 29 bugs were manually confirmed among which **19 have been confirmed/fixed by tool vendors of the tested engine**. During the testing, it generated 1,926 test programs and reported 846 suspicious cases for *NetBeans*. 47 of the suspicious cases were manually excluded as false alarms, 708 were firmed as buggy cases, and 91 were confirmed as inconsistent cases. From the buggy/inconsistent cases, 29 bugs and 11 inconsistencies were reported. Finally, 19 bugs were confirmed by the tool vendor (developers of *NetBeans*). We also notice that the ratio of false alarms (5.6%=47/846) is comparable to 7.1%=216/3053 in Section IV-E. All these together may suggest that the feature library derived from some refactoring engines could be applied directly to new refactoring engines without substantial reduction in effectiveness and efficiency.

Furthermore, we classified the types of defects found in the four refactoring engines according to the taxonomy proposed by Wang et al [38]. The results suggest that the most common reasons for the defects are compile errors and behavior changes: Compile errors, behavior changes, failed refactorings, unnecessary changes, and other error causes account for 60%, 23%, 10%, 4%, and 3% of the defects, respectively.

H. RQ4: ChatGPT-Based Automated Feature Extraction

We applied the LLM-based feature extraction to the collected bug library, and it reported 79 unique features in total, which is substantially larger than the number (53) of unique features extracted by human experts. One possible reason is that ChatGPT often tended to extract more features than human experts. On average, it extracted 5.5 features from a single bug whereas human experts extracted 3.2 features only.

We replaced the manually extracted features with those automatically extracted by ChatGTP-4o, and repeated the evaluation as introduced in Section IV-E, which resulted in 24, 15, 32, and 27 manually confirmed bugs in IDEA, Eclipse,

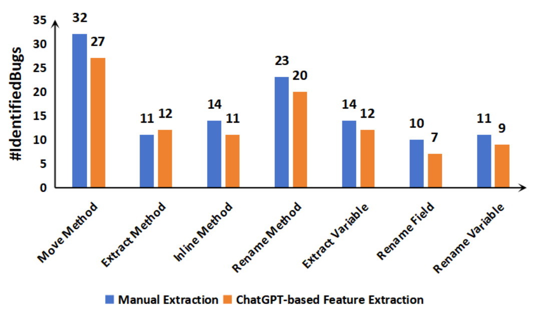


Fig. 6. Bugs Identified with Manual and Automated Feature Extraction

VScode, and NetBeans, respectively. The results are presented in Fig. 6 where we make the following observations:

- ChatGPT-based extraction was comparable to manual extraction. It only slightly reduced the number of identified bugs from 115 to 98=(27+12+11+20+12+7+9).
- For some refactoring types, e.g,. *extract method*, replacing the manually extracted features with those automatically extracted by ChatGPT can even increase the number of identified bugs.

ChatGPT-based extraction also resulted in 14 unique bugs that had not been found in the original evaluation (with manually extracted features), demonstrating the complementarity between the two approaches. The automatically extracted features as well as bugs identified with such features are available at [7].

I. Threats to Validity

A threat to external validity is that only four refactoring engines were involved in the case study. It remains unclear how effective the proposed approach would be once applied to other refactoring engines. Notably, it is expensive to test refactoring engines because it involves expensive human intervention, and thus it is difficult to involve a large number of refactoring engines in the case study. To minimize the threat, we selected four well-known and widely-used refactoring engines that represented the state of the practice in this field.

A threat to the internal validity is that the same participants analyzed the suspicious cases in Section IV-E (for the proposed approach) first and then again in Section IV-F (for the baseline approach). It may lead to the well-known testing effect, i.e., the participants could become more effective after the first testing. This testing effect may have increased the efficiency of the baseline approach because the participants became familiar with the inspection after inspecting suspicious cases for the proposed approach. However, despite such testing effects, the proposed approach outperformed the baseline approach.

V. Discussions

A. Limitations

The proposed approach is the first attempt to test refactoring engines with known bugs and LLMs. Although our case study in Section IV-E suggests that the proposed approach is efficient and effective, it suffers from the following limitations.

First, it is not yet fully automated. The test program generation and the conduction of refactoring are fully automated. However, the bug analysis, test case analysis, and inspection of suspicious cases have not yet been fully automated. Notably, the bug analysis should take bug reports as input and generate features that may trigger the associated bugs. Both input (bug reports) and outputs (features) are informally described in natural languages, making it challenging to fully automate the task. However, with the advances in natural language understanding and machine learning, it is likely that the task could be semi-automated or fully automated in the future.

Second, the current implementation of our approach supports only seven types of refactoring. To support additional types of refactoring, we should collect bug reports associated with them, extract feature libraries for them, and implement the refactoring infrastructure that invokes refactoring engines according to given commands. The seven refactoring types obtained the first batch of support because they are popular, accounting for more than 85% of refactoring activities.

Third, the current implementation supports only Java refactoring engines. In this initial attempt, we only collect bug reports associated with Java refactoring engines. As a result, it is difficult to apply the resulting feature library to other languages because different languages often have unique language features that are not available for other languages.

B. Difficulty in Supporting New kinds of Refactorings

For a new kind of refactoring, the following tasks should be conducted manually or semi-automatically: 1) collect related bugs and test cases; 2) derive features from the collected bugs and test cases; 3) define a prompt template. In our case study (with manual feature extraction), it took us 41 man-days to accomplish these tasks for 7 kinds of refactorings. That is, it may take around 6 man-days on average to make the proposed approach work for a new kind of refactoring.

C. Testing of Other Kinds of Software Systems

The key idea of the paper is to derive features of useful test programs and guide LLMs with such features to generate additional test programs. It is likely that the idea could be applied to test other software systems, like compilers, interpreters, code analysis systems, code converters, code documentation generators, and code optimizers, that take programs as input. However, we have not yet exploited the potential so far. Ideally, the publication of this paper could serve as a catalyst, inspiring researchers to explore and harness the potential.

VI. Conclusions and Feature Work

In this paper, we propose a novel approach to generating test programs for refactoring engines. The key insight of the approach is that existing bugs associated with refactoring engines could be employed to guide the generation of test programs for the refactoring engines. Another key insight is that advanced large language models have the potential to generate diverse programs according to the given requirements on program features. To this end, we built a feature library by manually analyzing 700 bug reports associated with refactoring engines, and generated test programs with ChatGPT with the resulting feature library. As a result, we successfully identified 115 defects in four state-of-the-practice refactoring engines, and 78 of them have been confirmed/fixed by the original developers of the refactoring engines. We also successfully identified 28 inconsistencies among the four refactoring engines that may cause confusion among programmers.

VII. Acknowledgments

The authors would like to thank the anonymous reviewers from ICSE 2025 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037) and China Postdoctoral Science Foundation (2023M740078).

References

[1] Automatic feature extraction. https://assdfsdafasfa.github.io/AutoFeatureExtraction/AutoFeatureExtraction.html.

[2] Bug anaylsis case. https://bugs.eclipse.org/bugs/show_bug.cgi?id=433295.

[3] Bug reports. https://github.com/assdfsdafasfa/assdfsdafasfa.github.io.

[4] Eclipse idea bug repository. https://bugs.eclipse.org/bugs.

[5] Evosuite. https://www.evosuite.org/.

[6] A example of the bug report. https://youtrack.jetbrains.com/issue/IDEA-81323.

[7] Identified bugs. https://assdfsdafasfa.github.io/AutoFeatureExtraction/AutoFeatureExtraction.html.

[8] Intellij idea bug repository. https://youtrack.jetbrains.com/issues.

[9] Jdolly. http://www.dsc.ufcg.edu.br/ spg/jdolly.

[10] Netbeans bug repository. https://github.com/apache/netbeans.

[11] Prompt templates. https://github.com/assdfsdafasfa/OpenPaper/tree/main/AutomaticFeatureExtraction/PromptTemplates.

[12] Randoop. https://randoop.github.io/randoop/.

[13] Replication package. https://github.com/assdfsdafasfa/OpenPaper.

[14] test3(). https://github.com/eclipse-jdt/eclipse.jdt.ui/blob/master/org.eclipse.jdt.ui.tests.refactoring/resources/RenameTemp/cannotRename/A_testFail3.java.

[15] Vscode-java. https://github.com/redhat-developer/vscode-java.

[16] *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[17] Openai, chatgptblog. https://openai.com/blog/chatgpt, 2023.

[18] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. Refactoring practices in the context of modern code review: An industrial case study at xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 348–357, 2021.

[19] Eman Abdullah AlOmar, Moataz Chouchen, Mohamed Wiem Mkaouer, and Ali Ouni. Code review practices for refactoring changes: An empirical study on openstack. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 689–701, 2022.

[20] Eman Abdullah AlOmar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 202–206, New York, NY, USA, 2024. Association for Computing Machinery.

[21] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 185–194, New York, NY, USA, 2007. Association for Computing Machinery.

[22] Kayla DePalma, Izabel Miminoshvili, Chiara Henselder, Kate Moss, and Eman Abdullah AlOmar. Exploring chatgpt's code refactoring capabilities: An empirical study. *Expert Syst. Appl.*, 249(PB), July 2024.

[23] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. Unprecedented code change automation: The fusion of llms and transformation by example. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[24] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, page 75–80, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. Systematic testing of refactoring engines on real software projects. In Giuseppe Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, pages 629–653, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[26] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. Commitbart: A large pre-trained model for github commits, 2023.

[27] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving usability of software refactoring tools. In *2007 Australian Software Engineering Conference (ASWEC'07)*, pages 307–318, 2007.

[28] Melina Mongiovi. Scaling testing of refactoring engines. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 674–676, New York, NY, USA, 2016. Association for Computing Machinery.

[29] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, page 552–576, Berlin, Heidelberg, 2013. Springer-Verlag.

[30] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Timofey Bryksin, and Danny Dig. Together we go further: Llms and ide static analysis for extract method refactoring, 2024.

[31] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. Em-assist: Safe automated extractmethod refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE '24, page 582–586. ACM, July 2024.

[32] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, page 151–160. IEEE, December 2023.

[33] Indy P. S. C. Silva, Everton L. G. Alves, and Wilkerson L. Andrade. Analyzing automatic test generation tools for refactoring validation. In *Proceedings of the 12th International Workshop on Automation of Software Testing*, AST '17, page 38–44. IEEE Press, 2017.

[34] Gustavo Soares. Making program refactoring safer. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 521–522, 2010.

[35] Gustavo Soares, Melina Mongiovi, and Rohit Gheyi. Identifying overly strong conditions in refactoring implementations. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 173–182, 2011.

[36] Gustavo Soares Soares. Automated behavioral testing of refactoring engines. SPLASH '12, page 49–52, New York, NY, USA, 2012. Association for Computing Machinery.

[37] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2389–2401, New York, NY, USA, 2022. Association for Computing Machinery.

[38] Haibo Wang, Zhuolin Xu, Huaien Zhang, Nikolaos Tsantalis, and Shin Hwei Tan. An empirical study of refactoring engine bugs, 2024.

[39] Xiaowei Zhang, Zhifei Chen, Yulu Cao, Lin Chen, and Yuming Zhou. Multi-intent inline code comment generation via large language model. *International Journal of Software Engineering and Knowledge Engineering*, 0(0):1–24, 0.

[40] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, Junda He, and David Lo. Generation-based code review automation: How far are we*f*. *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 215–226, 2023.