Test Intention Guided LLM-based Unit Test Generation

Zifan Nan, Zhaoqiang Guo, Kui Liu* , Xin Xia
Software Engineering Application Technology Lab, Huawei Technologies Co., Ltd., Hangzhou, China
{nanzifan, guozhaoqiang, kui.liu, xin.xia}@huawei.com

***Abstract*—The emergence of Large Language Models (LLMs) has accelerated the progress of intelligent software engineering technologies, which brings promising possibilities for unit test generation. However, existing approaches for unit tests directly generated from Large Language Models (LLMs) often prove impractical due to their low coverage and insufficient mocking capabilities. This paper proposes IntUT, a novel approach that utilizes explicit test intentions (e.g., test inputs, mock behaviors, and expected results) to effectively guide the LLM to generate high-quality test cases. Our experimental results on three industry Java projects and live study demonstrate that prompting LLM with test intention can generate high-quality test cases for developers. Specifically, it achieves the improvements on branch coverage by 94% and line coverage by 49%. Finally, we obtain developers' feedback on using IntUT to generate cases for three new Java projects, achieving over 80% line coverage and 30% efficiency improvement on writing unit test cases.**

***Index Terms*—unit test generation, mocking, LLM, program analysis, test intention**

I. Introduction

Unit tests play a significant role in software development [1], aiming to verify the functional correctness of each software unit (i.e., function or method) by covering code lines and branches of each function when executing the well-designed and developed test code of test cases. However, designing and developing high-quality unit test cases is a heavy and time-consuming task [2], which requires a comprehensive understanding of the requirements, design, and functional behavior of the code under test. To relieve developers from such a heavy task, researchers have explored various ways of automatic unit test generation (search-based [3], constraint-based [4], and random-based strategies [5]). However, these approaches still have the weaknesses of high randomness and cannot meet the high requirements of readability, maintainability, meaningfulness, and F.I.R.S.T. principles (Fast, Isolated, Repeatable, Self-validating, Thorough), making developers less willing to adopt these techniques in practice [6]. Recently, the emergence of Large Language Models (LLMs) has created a new milestone in the exploration of intelligent software engineering technologies and achieved promising results on solving concrete software engineering tasks [7], including unit test generation [8], [9].

Unit test generation (UTGen) with LLMs is mainly explored in two directions: ① **pre-training or fine-tuning LLMs**, which focuses on collecting high-quality test data [10], [11] and selecting efficient fine-tuning algorithms [12], [13]; and ② **designing effective prompts**, which aims at extracting accurate focal code context [14]–[17], constructing similar few-shot unit test examples [9] and designing strict output instructions [18]–[20]. Despite the promising results of generating unit tests with LLMs, two significant challenges are still not well addressed by LLM-based UTGen, which impedes its practical adoption for generating unit tests for developers in the industry: ❶ **Explicit branch coverage**: the generated test case can explicitly cover each branch of the focal method that has multiple or complicated branches. It is crucial to achieve high branch coverage that ensures a substantial portion of the unequivocal code within the focal method is thoroughly tested, thereby maintaining the quality of the source code. and ❷ **Accurate mock code**: the generated test case enables mocking of static methods, constructors, final classes and methods, private methods, invocations of dependent libraries, and more, allowing for the isolation of the unit being tested and increasing the accuracy of test results. For instance, Fig. 1 illustrates a real-world Java function that includes several branches and objectives that should be covered and mocked. Developers typically strive to write test cases that exhaustively cover all conditions with nine scenarios (i.e., seven `if` branches, one `try` block, and one `catch` exception block). The focal method also includes several objectives that should be mocked when writing the corresponding test cases. In our preliminary study, we fine-tuned the CodeLLaMA-34B model with 200K focal-test pairs with the context of the focal method. The fine-tuned LLM failed to generate accurate test cases for this focal method, but just generated the simple and useless test cases

```java
// Get the related license info with the related id.
public String getLicenseMessage(String id) {
   LicenseInfo licenseInfo = LicenseInfo.builder().build();
   if (StringUtils.isEmpty(id)) {return "Null License Info";}
   if (!isExistedID(id)) {return ...;}
   if (!SingletonUtil.getInstance().isMasterNode()) {...}
   if (LicenseUtil.isActivate(id) || LicenseUtil.isValid(id)) { return ...;}
   if (!SingletonUtil.getInstance().isInitSuccess(id)) {
      return "Init failed";}
   String result = "";
   try {
      result = ...;
      if (result == null) { result = ...;}
   }
   catch (Exception e) {...}
   return result;
}
```

Figure 1. Excerpted example of a focal method.

We also tried GPT-4o by providing the focal method with contexts and instructing it to generate high coverage unit tests. The results do not satisfy with the exact requirements of the

*Corresponding author.

```java
// Generated by CodeLLaMA-34B-SFT
@Test
public void testGetLicenseMessage_1() {
   String result = licenseController.getLicenseMessage("1");
   Assert.assertNotNull(result);
}

@Test
public void testGetLicenseMessage_3() {
   String appId = "0";
   LicenseController licenseController = new LicenseSubController();
   licenseController.getLicenseMessage(appId);
}

// Generated by GPT-4
@Test
void testGetLicenseMessage_InitializationFailed() {
   String id = "failedInitid";
   // error: should mock external class first
   when(LicenseController.isExistProduct(id)).thenReturn(true);

   // error: singletonMasterNodeUtil is not defined
   when(singletonUtil.isMasterNode()).thenReturn(true);
   when(singletonUtil.isInitSuccess(id)).thenReturn(false);

   // contradictory return value to reach the target branch
   when(LicenseUtil.isActivate(id)).thenReturn(true);
   when(LicenseUtil.isValid(id)).thenReturn(true);

   String expectedResponse = JSON.toJSONString(...);
   String result = LicenseController.getLicenseMessage(id);
   assertEquals(expectedResponse, result);
}
```

Figure 2. UTs generated by CodeLLaMA-SFT and GPT-4.

concrete branch coverage and mocking behavior, but ignore some branches and contain incorrect mock statements, such as the examples generated by CodeLLaMA-SFT and GPT-4o presented in Fig. 2 (more details cf. Section II-A).

In practice, when developers are writing test cases, they usually have the naïve intention of writing test code in their mind based on their understanding of the behavior of the focal methods, the concrete branches that will be covered by the test case, the objectives that should be mocked, the necessary test inputs and the oracle. Such intentions can guide developers to complete the task of writing test cases. For LLMs, well-designed prompts are used to shepherd LLMs to generate the expected results. The naïve intention and prompt inspire us to design effective prompts with accurate test intention to guide LLMs to generate high-quality test cases. To this end, we define the **test intention** as an explicit specification for covering a concrete branch of a focal method, encompassing constraints for input arguments, mock behaviors, and expected values, as exemplified in Fig. 3 (lines 3-16).

In this paper, we propose IntUT, a novel test intention guided LLM-based unit test generation approach. The explicit test intention aims to enable LLMs to generate precise unit test code that effectively covers target branches with high branch coverage. To obtain the explicit test intention for covering all branches of the focal method, we propose PAINT, a program analysis-driven test intention generation approach. Leveraging the static analysis techniques, PAINT can thoroughly traverse all branches and precisely identify the conditions required to reach each branch. Additionally, it can identify all method calls, accurately determine the mocking objectives based on the features of callees, and set the mock behavior accordingly. Note that the generation of test intention is shepherded by the test intention template that is manually mined through extensive prompt design experiments for unit test generation with LLMs. Specifically, PAINT generates test intentions using Control Flow Graph (CFG)-based branch analysis, traversing the entire graph to extract paths with conditions and form a set

```java
 1 @Test
 2 public void testGetLicenseMessage() throws Exception {
 3    /**
 4     * Input parameter(s):
 5     * (1) Initialize a String object id, and set id based on below conditions:
 6     * - set appId so that StringUtils.isEmpty(id) return false
 7     * Mock condition(s):
 8     * (1) Mock class LicenseUtil, LicenseUtil.isExistID return true
 9     * (2) Mock class SingletonUtil, SingletonUtil.getInstance return
10     * SingletonUtil object called singletonUtil
11     * (3) Mock singletonUtil.isMasterNode return false
12     * (4) Mock LicenseUtil.isActivate return false
13     * (5) Mock LicenseUtil.isValid return false
14     * (5) Mock singletonUtil.isInitSuccess return false
15     * Expected result: "Init failed"
16     */
17    // Initialize the focal object
18    LicenseController lic = new LicenseController();
19
20    // Initialize the input parameters
21    String appId = "0";
22
23    // Mock condition(s)
24    mockStatic(LicenseUtil.class);
25    when(LicenseUtil.class, "isExistProduct", anyString()).thenReturn(true);
26
27    mockStatic(SingletonUtil.class);
28    SingletonUtil singletonUtil = mock(SingletonUtil.class);
29    when(SingletonUtil.class, "getInstance").thenReturn(singletonUtil);
30    when(singletonNodeUtil, "isMasterNode").thenReturn(false);
31    when(LicenseUtil.class, "isActivate", anyString()).thenReturn(false);
32    when(LicenseUtil.class, "isValid", anyString()).thenReturn(false);
33    when(singletonNodeUtil, "isInitSuccess", anyString()).thenReturn(false);
34
35    // Invoke the focal method
36    String result = lic.getLicenseMessage(id);
37
38    // Verify result
39    Assert.assertEquals("Init failed", result);
40 }
```

Figure 3. Example of a unit test generated with test intention.

of **condition chains**. These chains are then used to populate the placeholders within the test intention template. During the traversal process, information about all function calls is also extracted. A mock classifier trained with existing mock code data determines whether a function call requires mocking, and the mocking action is determined by the condition chains. Finally, the complete test intention is subsequently integrated into the final prompt, which is fed into the LLM to generate test cases for the focal method.

To validate the practical feasibility of IntUT, we implement it as an IDE plugin and evaluate its performance on three real-world industrial projects and the live study. Our experimental results show that the unit tests generated by IntUT achieve a 97% compile rate and a 78% execution rate. Furthermore, with test intentions, IntUT significantly improves line coverage and branch coverage by 49% and 94%, respectively. The IntUT plugin is also made available to developers who can leverage its capabilities in their projects. User feedback is overwhelmingly positive, with an 85.5% instance acceptance rate. We further obtain developers' feedback on using IntUT to generate test cases for three newly developed projects, IntUT achieves an impressive line coverage of over 80%, resulting in a 30% improvement in development efficiency.

Overall, this paper makes the following major contributions:

- To the best of our knowledge, IntUT is the first LLM-based unit test generator guided by test intentions extracted through ❶ code branch analysis and ❷ mocking information, which achieves the generation of high-accuracy and high-coverage unit tests for Java projects in the industry.
- We propose PAINT to facilitate IntUT's automation of test intention generation by utilizing program analysis on focal methods, thereby ensuring ❶ the coverage of all branches in the focal method and ❷ identifying the mocking callees.
- We conduct practical experiments and a live study to eval-

Table I
RESULTS OF UNIT TESTS GENERATED BY CODELLAMA-34B-SFT.

	Compilable		Executable		Coverage	
	UT	FM	UT	FM	Branch	LOC
apollo [21]	24%	48%	23%	48%	11%	29%
audit-impl [22]	**20%**	**41%**	**13%**	**41%**	14%	23%
milo [23]	30%	52%	30%	50%	18%	20%
collections [24]	31%	40%	19%	**38%**	13%	22%
pulsar [25]	**43%**	**62%**	**43%**	**62%**	5%	12%

∗UT and FM represent unit test and focal method.

uate the actual feasibility of IntUT in generating test cases for developers. The experimental results demonstrate the practical feasibility of generating test cases by applying IntUT to industry projects.

II. MOTIVATION AND PRELIMINARIES

A. Motivation

We first conduct a preliminary study on generating test cases for 5 open-source Java programs with CodeLLaMA-34B-SFT (CodeLLaMA-34B model is supervised fine-tuned with 200K pairs of focal methods and their unit tests collected from GitHub open-source Java projects, with the same collecting methodology presented in the last paragraph of Section VI-D) by feeding with a prompt consisting of the full code context (i.e., the complete focal method, the focal class name, the fields in the focal class, and signatures of all methods defined in the focal class [8]) of the focal method to generate unit test cases for it. The experimental details are illustrated in Table I.

As presented in Table I, in the best case (i.e., pulsar), 43% unit tests generated by LLM are compilable and executable, which can test 62% focal methods. In the worst case, it only achieves 20% compilable unit tests, 13% executable unit tests, and test 38% focal methods, respectively. The coverage details of executing generated unit tests are far from the actual requirements expected by developers. We use the CodeLLaMA-34B-SFT to generate test cases for one cloud service related project in industry, and we get the similar (a bit lower) results as the five open-source projects. Although compilable and executable, test cases generated by LLM are unacceptable to developers since they address naïve simple tests (e.g., results presented in Fig. 2 for the focal method in Fig. 1).

We further analyze the test cases generated by LLM. We observed that most generated test cases are very simple (e.g., most test cases are generated with no more than five lines). They cannot really reflect developers' intention/requirement for testing the given focal methods as they expected. For example, Fig. 1 presents an excerpted example of a focal method (the concrete code is replaced with abstract comments and method calls as required by the company) implementing a simple functionality to obtain the license message and handling all possible exceptions which may invoke method calls defined in other classes, packages, or third-part libraries. We feed the LLM with a prompt with the aforementioned full code contexts to generate test cases. Lines 2-13 in Fig. 2 are the test cases generated by LLM for the focal method, which are useless for developers.

First of all, those test cases are simply generated with a method call of the focal method and directly verify its returned value without considering its concrete implementation, and half of them do not have the assert statement. After executing the generated test cases, all of them can only cover the first `if` statement of the focal method, but ignore its other code, since the test cases are generated with the random and simple test inputs. The generated test case does not completely achieve the test intention of mocking the input and output of invoking `nullIdCaseImpl()` since such method invocation relies on the third-party lib and it is lack of the concrete input and output. In addition, with the generated test cases, developers cannot easily figure out which branch is covered by the LLM-generated test case since the information of covering branch for each test case is not provided by LLM. This may be attributed to that LLM cannot understand which branch should be covered by each test case. Actually, branch coverage is an important test intention for developers to understand which code branch is (not) covered by test cases.

We further try GPT-4o to generate test cases by providing focal methods with their contexts and instructions of reaching all branch coverage. However, the test cases generated by GPT-4o do not reach developers' expectation. For example, for the focal method in Fig. 1, GPT-4o analyzes the code and generate 7 unit tests which aims to cover 7 different branches, but 2 branches are ignored. Even so, the generated test cases cannot be used directly by devleopers which includes some critical errors: such as ❶ the mock statement at line 20 in Fig. 2 mocks the method of the wrong `LicenseController` class, the mock statements at lines 23-24 uses the objects that are not initialized; ❷ the returning values of mock statements at lines 27-28 is conflicted with the required conditions (they should be false) to of reaching the target branch (*if statement* at line 9 in Fig. 1). Moreover, an experienced developer spent around 10 minutes on modifying the generated unit tests to be executable, which only achieves 53.8% line coverage.

The focal method has nine branches (i.e., seven `if` branches, one `try` body branch, and one `try-catch` branch). Each branch implements the different and concrete behavior for the related requirement, which will have different input and output that are independent on each other. It means that there are at least six unit test cases for this focal method to make all code lines in the focal method covered by the test cases, and they should be independent on each other by following the F.I.R.S.T principle of writing unit tests. The detailed implementation of each branch may invoke method calls that are defined in third-party libraries or other packages, some of them need to be mocked by simulating the behavior of their real objects to isolate the behaviour of the being tested objects from the simulated ones since the real objects are impractical to incorporate into the unit test. Such information is not comprehensively understood by LLMs when generating test cases for the focal method. Leveraging LLMs to generate accurate test cases only with the aforementioned code context for such scenario faces three challenges: ❶ LLMs are short of accurately distinguishing the different branches from one another. ❷ LLMs cannot accurately identify the mocking objects and generate appropriate mocking code. ❸ LLMs is short of generating accurate test case for each branch which can follow the F.I.R.S.T. principle of writing unit tests.

B. Preliminaries

Here we present the preliminaries for two concepts related to IntUT in details: control-flow graph and mocking technique.

A **Control-flow graph (CFG)** is a directed graph, $G = (B, E)$, which models the flow of execution between basic blocks in a program [26]. Each basic block $b \in B$ is a maximal length sequence of branch-free code that always execute together. Each edge $e = (b_i, b_j) \in E$ corresponds to a transfer of control from block b_i to b_j. The control-flow naturally corresponding to the branches of the source code, providing a clear map to identify the control-flow paths that can reach different branches. The conditions along these paths offer a rich source of information for generating test intentions.

Mocking is a crucial technique used in unit testing to isolate dependencies (e.g., databases, file systems, or network connections) of the focal method [27]. In unit testing, dependencies can cause several issues, including slow connection, flaky external systems, time-consuming setup, and tight coupling that makes implementation changes difficult. To address these issues, the mocking technique creates a fake implementation of a dependency, enabling tests to focus on the unit under test without worrying about dependencies [28]. In the industry, Mockito, EasyMock and PowerMock are the most popular mocking frameworks [29]. Specifically, PowerMock [30] provide a more comprehensive mock solution by extending the first two frameworks. It is used by our proposed UTGen tool.

III. Overall Framework

The primary challenge of generating unit tests using LLMs lies in providing the precise test intention that can distinguish and cover different branches in focal methods, as well as accurately identify the mocking callees, as discussed in Section II-A. To address this challenge, we propose a novel methodology IntUT, test intention guided unit test generation with LLMs. Instead of generating unit tests with the code context of the focal methods based on LLMs fine-tuned with unit test data, we propose extracting test intention from the context of focal method to guide LLMs to produce accurate and high-coverage unit tests. To this end, we propose PAINT, a CFG-guided branch analysis technique that aims to automatically generate test intentions by analyzing static analysis results of focal methods and mock decisions made by mock classifiers. The overall solution is illustrated in Fig. 4, which consists of two steps: ❶ generating test intentions, and ❷ assembling the full prompt to request the model to generate unit tests.

The first step contains the major steps of PAINT (cf. Section V). It first performs static analysis on the focal method to generate the CFG. An in-depth analysis of the blocks and edges within the CFG is then conducted. Specifically, block

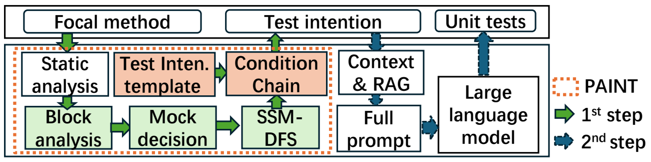


Figure 4. Test intention guided UT generation.

analysis is applied to various types of blocks (e.g., `if` and `try` branches) to reconstruct and label the CFG. The Subgraph-Split-Merge Depth-First Search (SSM-DFS) algorithm is used to traverse the CFG edges to generate condition chains after the mocking function callees are identified. Finally, the conditions inside each chain are filled into the test intention templates. These templates (cf. Section IV-A) are manually mined from over 1,000 empirical study cases (lasted for 3 months) of prompting LLMs to generate unit test cases for the focal method with hand-written test intention discussed with over 20 senior UT developers, to generate explicit test intentions to cover all branches of each focal method. The second step assembles the full prompt (cf. Section IV-C) by integrating test intentions with the code context of a focal method and RAG contents. The resulting final prompt is thus fed to the LLM to guide LLM generating the corresponding unit test cases.

IV. Prompt and Test Intention

With the prompt templates for test case generation mined from an abundant empirical study, we carefully craft a general-purpose (useful for general test case generation, but not for specific project) prompt in a code-like format to guide a code LLM (e.g., CodeLLaMA) to precisely complete the unit test code . As shown in Fig. 6, the prompt consists of four components in order of full code context of the focal method, example of test cases (retrieved by RAG), prefix of the test case, and **test intention**. Note that in Fig. 6, the segmentation words (e.g., "— 1. Program context —") in front of each section are only used to distinguish different content and are not included in the actual prompt.

A. Test Intention

Test intentions serve as guidance and constraints for LLM to generate unit tests. A test intention typically consists of four essential components as presented in Fig. 5: input parameters, mock behavior, hint specification, and expected results. These are fundamental elements within a unit test.

Input Parameters: Input parameters within test intentions describe the values or conditions for all parameters of the focal method. They aim to guide LLMs to set up or inference the inputs for a unit test, as shown in lines 4-5 in Fig. 5. For example, the parameter `id` of the focal method `getLicenseMessage` is used by `StringUtils.isEmpty(id)`. The condition *"set appId so that StringUtils.isEmpty(id) return false"* is set to guide LLM to infer suitable `id` values. Since LLM cannot always provide correct values, the prompt is editable by developers to ensure accurate inputs.

```
/**
* Case name: {XXX}
* Input parameter(s):
* (1) Initialize para1 = {XXX}
* (2) Initialize para2 = {XXX}
* ......
* Mock condition(s):
* (1) Mock {method call} return/throw/do nothing {XXX}
* (2) Mock class {XXX}, {static method call} return/throw/do nothing {XXX}
* ......
* Hint specification:
* (1) Use Whitebox.invoke to call focal method
* ......
* Expected result: {XXX}
* ......
*/
```

Figure 5. Template definition of test intention.

Mock Behavior: Mock behaviors direct LLMs to generate the necessary mock statements. A mock behavior specification comprises a mocking object and its corresponding actions with respect to key elements within a mock statement that are populated by PAINT with conditions to enter target branches. Different mock behavior specifications are designed to adapt to various mocking actions, e.g., the template *"Mock ... throw {XXX}"* is used for the case of throwing an exception, while the template *"Mock ... return {XXX}"* is used for returning certain values. For instance, to ensure a test that can achieve the second `if` branch of the focal method `getLicenseMessage` (Fig. 1), in Fig 3, the specification *"Mock class LicenseUtil, LicenseUtil.isExistID return true"* is generated for the mock condition.

Hint Specification: To cope with general testing scenarios, we reserve an optional hint that can be customized by users in the test intention. For example, it is necessary to specify the use of the `Whitebox.invoke` API in Fig. 5 as the hint specification when IntUT is used to generate test for private focal methods. Note that additional specifications can be added when encountering other test scenarios.

Expected Result: It specifies the test oracle within the unit test to verify the correctness of the focal method. For example, to verify that the unit test enters the third `if` branch (line 9 in Fig. 1) of the focal method `getLicenseMessage` during execution, the *"Non-master node Info"* should be written in the expected results.

B. Example of Test Cases Retrieved by RAG

RAG (Retrieval-Augmented Generation) technology has been widely used to improve the performance of LLMs by providing retrieved relevant similar example cases as a part of prompt, which has been proven in different domains [31], [32]. IntUT leverages RAG to retrieve high-quality unit test examples and mock examples, of which focal methods are similar to the focal method fed to LLM. The retrieved examples are further used to guide LLMs to generate unit tests that can be suitable for the same test and mock framework, and adhere to the same coding convention. To ensure the quality of the retrieved examples, RAG database is dynamically built on the developers' working projects.

RAG for Unit Test Examples. The unit test example refers to a complete test case used to provide the structure of a test case for LLM. Specifically, we first extract all {*historical focal method, list of historical test methods*} data pairs from the target project, and calculate the Jaccard similarity (with

```
--- 1. Program context ---
// Focal class file path
public class $$focal_class_name$$ {
   $$fields$$ (There may be multiple instances)
   $$dependency_methods$$ (There may be multiple instances)
   $$focal_method_body$$
}

// The definition of the dep methods invoked by the focal method:
$$dependency_methods$$ (There may be multiple instances)

--- 2. Example code ---
// possible test code: (There may be multiple instances)
$$test_cases$$

 (There may be multiple instances)
// mock statements for $$mocked_method$$ is:
// mock example 1: (There may be multiple mock examples)
// $$mock_code$$

--- 3. Test code prefix ---
public class $$focal_class_name$$Test {

   $$focal_object_declaration$$

   // Existing other test cases of the focal method
   $$test_cases$$

   @Test
   public void test_$$focal_method_name$$ {

--- 4. Test intention ---
   /** $$test_intention$$ */
```

Figure 6. Final prompt format of IntUT.

a threshold of 0.8) between the given focal method and each historical focal method to retrieve the corresponding historical test methods as UT examples.

RAG for Mock Examples. The mock example refers to concrete mock statements for external dependent methods invoked in the focal method. We extract {*historical method call, list of mock example code*} data pairs from the target project to build mock indices, and retrieve the mock example code for method calls that appear in the focal method and add it to the final prompt.

C. Final Prompt fed to LLM

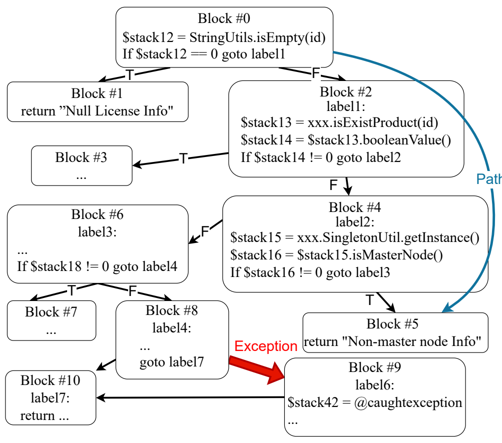
Fig. 6 shows the final prompt format of IntUT, which consists of the full code context and the test code prefix, in addition to the test intention presented previously. The full code context includes the complete focal method, the focal class name, the fields in the focal class, and signatures of all methods defined in the focal class [8]. All contents are formatted in original code style. The test code prefix is used to guide LLM to directly complete the test code in the given formatted test prefix without generating irrelevant tokens.

V. PAINT: TEST INTENTION GENERATOR

This section presents PAINT, a program analysis-driven test intention generating approach that aims to automatically generate test intentions for the focal method. As shown in Fig. 4, PAINT has three major steps: ❶ block analysis (cf. Section V-A), pre-processing blocks within the CFG, transforming the graph for subsequent traversal, and extracting conditions and function call information from blocks, ❷ mock classifier (cf. Section V-C), identifying the mocking callees, ❸ SSM-DFS algorithm (cf. Section V-D), traversing the transformed CFG to generate condition chains (cf. Section V-B), of which contents are filled into test intention templates to precisely guide LLMs to generate unit tests.

A. Block Analysis

This step pre-processes the blocks within the original CFG for subsequent traversing steps, involving adding, removing,



Block #0
$stack12 = StringUtils.isEmpty(id)
If $stack12 == 0 goto label1
T
F
Block #1
return "Null License Info"
Block #2
label1:
$stack13 = xxx.isExistProduct(id)
$stack14 = $stack13.booleanValue()
If $stack14 != 0 goto label2
Block #3
...
Path
Block #6
label3:
...
If $stack18 != 0 goto label4
Block #4
label2:
$stack15 = xxx.SingletonUtil.getInstance()
$stack16 = $stack15.isMasterNode()
If $stack16 != 0 goto label3
Block #7
...
Block #8
label4:
...
goto label7
Block #5
return "Non-master node Info"
Exception
Block #10
label7:
return ...
Block #9
label6:
$stack42 = @caughtexception
...

Figure 7. PAINT: CFG-based branch analysis.

or labeling edges, merging blocks, and extracting condition expressions, which can provide information for searching condition chains and making mock decisions in test intentions. The blocks are categorized into four types with different analysis methods: *If blocks*, *Switch blocks*, *Exception blocks*, and *Hybrid blocks*. *If* and *Switch* blocks are grouped as *conditional blocks* that govern the program's execution flow. *Exception blocks* handle exceptions, while *Hybrid blocks* encompass both conditional and exception handling functionalities.

Fig. 7 shows an CFG example corresponding to the focal method in Fig. 1 (some details are omitted due to space limitation). This CFG includes several *if blocks* (e.g., Block #0, 2) and one *exception block* (e.g., Block #9). *If blocks* are the most common type of blocks within a CFG. The last statement inside an *If block* typically contains a boolean expression, whose `True/False` value determines the subsequent blocks. Thus, this step involves extracting the full boolean expressions of *if blocks* and labeling the out-edges with True/False values. For example, the boolean expression of Block #0 in Fig. 7 is `StringUtils.isEmpty(id)`, while the out-edge $e = (0, 1)$ is labeled *True*, and edge $e = (0, 2)$ is labeld *False*.

Exception blocks represent the branches responsible for exception handling within the focal method. In the original brief block CFG, exception blocks lack in-edges, making them inaccessible from the starting block (i.e., block #0). To resolve this limitation, IntUT locates the corresponding try block(s) from static analysis results and establishes connections to the exception block by adding in-edges. For example, edge $e = (8, 9)$ are added as exception edges of Block #9 in Fig. 7.

A switch block typically involves several blocks. We merge and transform these blocks into one block with several out-edges labeled with corresponding *cases*. For Hybrid blocks, which are both conditional and exception blocks, we apply both analysis methods. These methods do not conflict, as the exception block handling step adds in-edges, while the condition handling step labels out-edges.

B. Condition Chains for Test Intention

After labeling the CFG, PAINT utilizes the SSM-DFS algorithm to search for paths. Each path represents a series

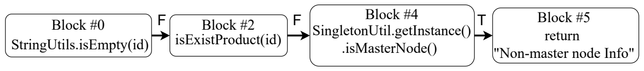


Figure 8. Condition chain.

of conditions required to reach a leaf block (i.e., a branch that needs covering). These paths are saved as condition chains and then merged with the test intention template to form test intentions. This section focuses on condition chains, while the SSM-DFS algorithm is explained in Section V-D.

A condition chain $ch = (B, E)$, searched from the CFG, is a single directed path. Each block $b = (stmt, cond\_stmt) \in B$ has a list of branch-free statement and end with a conditional statement $cond\_stmt$, whose value determines execution directions of the focal method. Each edge $e = (b_s, b_t, cond\_value) \in E$ is a direct edge points from the source block b_s to the target block b_t, with $cond\_value \in \{true, false, exception\}$ indicating the condition value for b_s to reach b_t.

Fig. 8 shows an example of a condition chain, which is the path labeled in Fig. 7, and corresponding to the test intention shown in Fig. 3 (lines 4-11, the last condition at line 11 should be set true for this chain). This condition chain indicate that, to reach Block #5 (i.e., the 3^{rd} if branch in the focal method), The value for `StringUtils.isEmpty(id)`, `isExistProduct(id)`, and `SingletonUtil. getInstance().isMasterNode()` should be `False`, `True`, and `True`, respectively.

We then fill these conditions into the test intention template. Based on whether a condition needs mocking (cf. Section V-C) and whether it is related to inputs, we place the conditions and their values into the mock behavior or input area in the template, forming the final test intentions. For instance, in Fig. 8, the `StringUtils.isEmpty(id)` with the `False` condition does not need mocking, hence is filled into the input parameters part (Fig. 3, line 5-6) within the template. Meanwhile, the `isExistProduct(id)` with the `True` condition needs mocking, thus is filled into mock behavior part (Fig. 3, line 8). These intentions guide the LLMs to generate unit tests precisely.

C. Mock Decision with Mock Classifier

A focal method could invoke other functions, leading to a crucial decision during unit test development: whether to mock or execute the callee. The challenge, after consulting with several experienced Java developers, is that *there are no explicit precise rules to determine whether a function should be mocked*. Developers typically make decisions based on their understanding and knowledge of the called functions. Therefore, we build an experience-based classifier using historical mocking data.

The features for the mocking classifier are the function full signature and the function body. The function signature includes the full package paths, the return type, the function name, and the input types. Note that the function body may not always be accessible, as it could be calls to library APIs. The labels for the data are binary, indicating whether the function is mocked (`true`) or not (`false`).

The features are extracted from focal methods, while the labels are obtained from corresponding unit tests. We apply two constraints to select the data: ❶ the function call statement must be covered by the unit test, and ❷ the unit test must contain a mock statement. The first constraint prevents the *false* label from being miss assigned, and the second constraint make sure that developer will mock the function if necessary.

We formulate the mock classification task as a text classification problem, as the semantic embedding of the function signature and body could convey information for mocking decision. We train the classifier based on the CodeBERT model, which is well-suited for this task due to its superior code embeddings and reasonable inference speed, requiring fewer computational resources. The trained classifier shows more than 90% accuracy on mocking decisions test cases. For example, in Fig. 8, there are three function calls within the condition chain. The mock classifier determines that the function call in the later two blocks needs mocking, resulting in the mock conditions shown in Fig. 3, line 8-11, while the return values are determined by conditions on chain edges.

D. SSM-DFS Algorithm for Extracting Condition Chains

To achieve a high branch coverage, we identify condition chains that lead to each branch of the focal method through a comprehensive traversal of all edges, ultimately enabling the generation of high-coverage unit tests. Starting with the head block, we designed a subgraph-split-merge depth-first search (SSM-DFS) algorithm to traverse the entire CFG. This traversal yields a comprehensive collection of all paths from the head block to the leaf nodes, which are then transformed into a list of condition chains.

We design the SSM-DFS algorithm instead of directly using DFS due to two issues encountered during CFG traversal: exception branches and function calls. Both of them require recursive handling for different reasons. Exception branches serve as handlers for exceptions raised during the focal method execution. It is not necessary to combine an exception block with all condition combinations, as long as the exception is caught. Furthermore, a focal method often invokes other functions. If the function call, such as a private method in the same class, requires execution rather than mocking, we must consider the CFG of the called function.

The SSM-DFS algorithm is designed to address the above issues. In detail, during the traversal of the CFG, when exception blocks are reached, they are **split** from the current traversal along with their successors, which forms **exception-subgraphs**. After finishing traversing the parent graph, we apply the SSM-DFS algorithm to all exception-subgraphs. The traversal starts from the exception block and ends at leaf nodes or nodes in parent graph. Since conditional blocks and exception blocks can also appear in the exception-subgraphs, we handle them recursively.

After completing the traversal of subgraphs, we **merge** the paths back into the parent graph paths. Since all paths start

```
public int sum(int a, int b, int c) {
   if (a < 0) { // 2 conditions | a < 0, a >= 0
      if (a > -10) {...} // 2 conditions | a > -10, a <= -10
   } // 3 conditions
   if (b < 0) {...} // 2 conditions | b < 0, b >= 0
   if (c < 0) {...} // 2 conditions | c < 0, c >= 0
   return ...
}
```

Figure 9. Example of nested and parallel conditions.

from the same exception block, we only need to find the shortest path from the parent graph that includes the trigger block (and the tail block if it ends on the parent node), and merge the paths from the subgraph into this path. We choose the shortest path because it has the fewest conditions, making it easier for LLMs to generate correct unit tests.

Function calls are handled similarly. A function call is typically a call statement within a CFG block. Thus, the SSM-DFS algorithm **splits** this block at the call statement into two blocks. After traversing the current graph, the CFG of the callee is treated as a **subgraph**, and then SSM-DFS is recursively applied. The returned paths of the subgraph are then **merged** into the paths of the parent graph that include this function call.

One more case that needs consideration is loops. Within the CFG, loops are formed into a loop condition statement and a backwards edge that originates from the end of the loop and points back to the loop condition block. We observe that once the loop body has been traversed, the corresponding branch is considered covered. Therefore, to prevent infinite loops during traversal, we explicitly remove the return edges of loops.

E. Mitigating the Path Explosion in SSM-DFS Algorithm

Ideally, developers aim to test all condition combinations to ensure maintainability in all possible situations. The SSM-DFS algorithm searches for all combinations of conditions within the focal methods. However, the number of paths grows exponentially with the number of conditions and execution function calls in the focal method. Consider a real case where a focal method with 10 *if statements* can lead to over 400 paths, which will be transformed into unit test cases—an overwhelming number for developers. Therefore, based on the actual needs of developers, we offer two options to reduce the number of paths while maintaining the same coverage rate.

The first option focuses on nested conditions. Condition combinations are one of the major causes of the path explosion. As the SSM-DFS algorithm collects all paths from the head block to the leaf nodes, many edges appear in the path list multiple times. For instance, in Fig. 9, the edges for conditions $a < 0$ and $a > -10$ are visited four times by combining with the later conditions (2×2 for b and c). Since an edge is is considered covered as soon as it is visited once, condition combinations involving $a > -10$ && $b < 0$ and $a > -10$ && $b >= 0$ can be deemed less relevant, as pruning one of them do not significantly affect the overall coverage.

We employed a heuristic path pruning based on the structure of *if statements*, focusing on the distinction between nested and parallel conditions. A nested condition occurs when one *if statement* is within the body of another, while parallel conditions are separate *if statements*. In Fig. 9, the *if statements* at line 2-3 are nested conditions, while line 2, 5 and 6 are parallel conditions. We found that nested conditions are more essential because the execution of inner conditions depends on the boolean value of outer conditions, making them highly related. In parallel conditions, the latter condition is executed regardless of the boolean value of the former condition. Therefore, we pruned the combinations of parallel conditions based on this heuristic. Consequently, the number of paths in the aforementioned *10 if statement case* reduced from 400+ to 38 paths, while maintaining the same branch coverage.

Note that, such heuristic could lead to the possibility of excluding some paths (i.e., condition combinations) since there is an unavoidable trade-off between missing some potential critical paths and the explosion of listing **ALL** condition branches. To address this trade-off, in the implementation of IntUT, it leaves the weakness to users by allowing them to update the generated intentions when the chosen path is not covered by the intention.

In addition to conditions, function calls are another cause of path explosion, as their paths are merged into the paths of focal methods, resulting in a multiplication of path numbers. Therefore, if a focal method calls many functions or has a deep call hierarchy, we limit the number of functions executed to mitigate path explosion by configuring two parameters: ① the maximum number of execution functions and ② the maximum level of function call subgraph recursion. When these limits are reached, the remaining functions are mocked instead of being executed. Additionally, developers can provide ③ a list of function names that require execution, which entails more effort but results in more precise test intentions.

VI. Experiment Setup

A. Research Questions

To evaluate the feasibility of IntUT in generating unit tests, we conduct comprehensive experiments with the data collected from real-world development scenarios to answer the following research questions (RQs):

RQ-1: To what extent quality of test cases can be generated by IntUT?

RQ-2: To what extent test intentions contribute to unit test generation?

RQ-3: Are developers willing to use IntUT to assist their daily test writing tasks?

RQ-4: Can IntUT generate unit tests efficiently when test intentions are equipped?

B. Evaluation Data

To answer the aforementioned research questions from the practical aspect, we collect the ground-truth dataset from 3 Java projects in industry. We collect test cases in those 3 Java projects written by their developers, and asked 4 experienced senior UT developers to manually check whether each collected test case satisfies the F.I.R.S.T. principles and whether each case covers all code that should be covered. Finally, 237 test cases for 57 industrial focal methods are collected and curated as the evaluation data. Detailed statistics of the collected data are presented in Table II.

Table II
Statistics of curated evaluation data.

Project	# Unit Tests	# Focal Methods	# Branches
***Mgr	65	12	96
****Lic	101	25	196
****Res	71	20	134

*The project name is anonymized as required by the industry company.

C. Evaluation Metrics

This section presents the metrics used to assess the performance of generating unit tests with IntUT.

- **Executable rate:** the ratio of executable unit test cases generated by IntUT, is used to evaluate whether the generated unit tests can be executed successfully.
- **Line coverage** and **Branch coverage:** ratios of code lines and branches covered by executing unit test cases generated by IntUT. Such coverage is collected using Jacoco [33], a Java Code Coverage Library, to assess the quality of unit tests generated by IntUT.
- **Line acceptance rate (LAR)**, **token edit distance (TED)**, and **line edit distance (LED)** are used to assess the differences between generated unit tests and the ground truth. Line acceptance rate is calculated as $LAR = x/y$, where x represents the number of code lines in a unit test generated by IntUT that appear in the ground truth, and y represents the total number of lines in the unit test generated by IntUT. The TED and LED are used to assess the efforts for recovery after generation, which represent the minimum number of edits, on tokens and lines, required to modify the generated unit test to match the ground truth.
- **Response time** aims to evaluate the efficiency of generating test cases with IntUT. The response time is the time duration of triggering the action in the IDE and receiving the corresponding results.
- **Acceptance rate** evaluates to what extent developers are willing to accept the generated unit test cases. It is the ratio between the number of unit tests accepted by developers and the total number of requests. We use **instant acceptance rate** to evaluate the unit test accepted right after generation and use **traceable acceptance rate** to track the unit test accepted after a period of time.

D. Implementation of IntUT

We implement IntUT as an IntelliJ IDEA plugin that can interact with developers to generate high coverage unit tests. When a project is imported into the IDE, IntUT runs background analysis to build the RAG database. When a focal method is selected, the PAINT approach is then triggered to analyze the determined focal method. The identified condition chains are then transformed into test intentions, which are presented to developers for review and editing. After refinement, the context and RAG content are automatically assembled with the intention and fed into the LLM. The LLM generates multiple unit test candidates, allowing developers to

Table III
QUALITY OF TESTS GENERATED BY INTUT.

Project	#Pass Test	#Fail Test	#Fail Exec	#Comp. error
***Mgr	57 (88%)	1 (1.5%)	6 (9%)	1 (1.5%)
****Lic	67 (66%)	14 (14%)	17 (17%)	3 (3%)
****Res	31 (44%)	14 (20%)	22 (31%)	4 (6%)
Total	155 (65%)	29 (12%)	45 (19%)	8 (3%)

select the most suitable one. We present multiple candidates because the LLM's sampling process can produce varying results when generating long contexts, and some of these results may contain errors. By displaying multiple options, we increase the likelihood of correctness.

We employ the CodeLLaMA-34B-base [34] (CodeLLaMA), LLaMA-3.1-70B-base (LLaMA-3.1) and CodeLLaMA-34B-base-SFT (CodeLLaMA-SFT) models as LLMs in experiments. CodeLLaMA and LLaMA-3.1 are selected because they are the SOTA open source LLMs with high performance across various coding tasks. Additionally, we include models of varying sizes to validate the efficacy of test intentions and their adaptability to different model scales. We opted against using online service models (e.g., GPT-4) due to security restrictions that prohibit sending code to external servers.

The CodeLLaMA-SFT model is a fine-tuned version of the CodeLLaMA-34B-base model using LoRA [35]. We collect 10K data points from the aforementioned three Java projects in industry. Each data point consists of the following elements: focal context (class, fields, method signature), focal method, test context (test class, test fixture), and the corresponding unit test, all formatted in a code style similar to [8]. The fine-tuning process aims to adapt the model to the coding convention of the target projects. Subsequently, we evaluate the performance of the fine-tuned model in unit test generation, examining both the effectiveness of fine-tuning and the potential added value of explicit test intentions. Note that the test cases used were authored by project developers prior to this study, and the ground truth data were not included in the experiments.

VII. EXPERIMENTAL RESULTS

A. RQ-1: Quality of Tests Generated by IntUT

Table III reports the quality of tests generated by IntUT in terms of the number and rate of passed, failed (test and execution), and compile-error, respectively. Overall, 65% of generated tests can be compiled correctly and are free of execution errors. 31% (12% + 19%) of the test cases fail their execution and tests due to the test triggering a runtime exception and the incorrect assertion statements in the test code that do not match the program execution results, respectively. A small portion (3%) of tests encounter compilation errors and cannot be compiled successfully. Although IntUT shows varying effectiveness in different projects, these results show that IntUT can generate unit tests with high quality.

B. RQ-2: Contribution of Test Intentions

Branch and line coverage. Table IV compares the branch and line coverage of the tests generated by IntUT with and without test intention in its prompt. Overall, for CodeLLaMA-SFT model, IntUT with test intention generates test cases with the branch coverage of 64% and the line coverage of 75%, which achieve a significant improvement of 94% and 49% comparing against IntUT without test intention, respectively. IntUT achieves similar results on LLaMA-3.1 model, with 63% branch coverage (91% improvements) and 78% line coverage (59 % improvements), demonstrating the adaptability of IntUT on different models. Note that the improvement on the ***Res project is not significant compared to the other projects. It is because most of the test cases generated for this project cannot be executed correctly (see Table III) due to the lack of necessary preset runtime environment statements.

Table IV
COMPARING RESULTS WITH/WITHOUT TEST INTENTION.

Model	Project	Branch Coverage			Line Coverage		
		W/O	W	Impr.%	W/O	W/	Impr.%
Code-LLaMA-SFT	**Mgr	30%	79%	+162%	37%	80%	+118%
	**Lic	34%	70%	+108%	55%	84%	+54%
	**Res	34%	44%	+31%	59%	65%	+10%
	Total	33%	64%	+94%	51%	75%	+49%
LLaMA-3.1	**Mgr	32%	73%	+128%	42%	87%	+107%
	**Lic	31%	73%	+138%	57%	88%	+54%
	**Res	28%	42%	+49%	48%	62%	+28%
	Total	33%	63%	+91%	49%	78%	+59%

Table V
RESULTS WITH DIFFERENT PROMPT SETTINGS.

	CodeLLaMA			CodeLLaMA-SFT		
Prompt Setting	LAR	TED	LED	LAR	TED	LED
① FM	0.092	34.2	8.30	0.180	29.4	7.50
② FM + C + R	0.150	32.0	8.40	0.167	30.2	7.50
③ FM + C + R + TI	**0.529**	**15.0**	**5.20**	**0.565**	**13.4**	**4.80**
④ FM + C + TI	0.469	16.3	5.60	0.483	16.5	5.60

*FM: focal method; C: context; R: RAG; TI: test intention.

Ablation study on prompt settings. We further investigate the efficacy of generating test cases with test intentions by conducting an experiment involving three prompt content variations for each focal method: ① only with the focal method (FM), ② focal method with context and RAG (FM + C + R), ③ full prompt with text intententions (FM + C + R + TI), ④ full prompt but without RAG (FM + C + TI). These prompts are fed to the CodeLLaMA and CodeLLaMA-SFT models to generate 5 unit test candidates per request, respectively. For each generated candidate, we assess its LAR, TED and LED values against the ground truth.

As shown in Table V ① ② ③, only relying on supervised fine-tuning or prompting with the context and RAG of the focal method can achieve a bit improvement for unit test generation, but not always for the token and line edit distances. Notably, prompting with test intention achieves a significant improvement on the line acceptance rate (LAR is increased substantially from 0.092 and 0.18 to 0.529 and 0.565). At the token level, the edit distance decreases dramatically from 29.4 to 13.3. The line edit distance also decreases from 7.5 to 4.8. The decreases of TED and LED indicate that, with test intentions generated by IntUT, developers spend fewer efforts on modifying the generated test cases to satisfy with their expected results. These data reveals that prompting with test intention can significantly enhance the ability of LLMs to

Table VI
LAC OF UT GENERATED BY INTUT COMPARED WITH THE TEST CASES EVENTUALLY MERGED INTO PROGRAM REPO BY DEVELOPERS.

#UT	0%	⩽10%	⩽20%	⩽30%	⩽40%	⩽50%	>70%
632	40.5%	50.6%	63.0%	71%	75.2%	80.1%	13.8%

The first row presents the rates of code lines in IntUT-generated test cases modified by developers when they are merged into program repositories.

generating test cases.

Table V further demonstrate the benefit of fine-tuning and RAG. Comparing CodeLLaMA against CodeLLaMA-SFT, fine-tuned model achieved higher statistics than original model, as 9% improvement for the focal method prompt ①, and 3.6% for full prompt ③. Comparing ③ and ④, RAG helps improve around 6% and 8.2% on LAR, 1.3 and 3.1 on TED, 0.4 and 0.8 on LED for CodeLLaMA and CodeLLaMA-SFT, respectively. These results suggest that fine-tuning and RAG can enhance the models' ability to generate better unit tests, although the improvements are relatively modest compared to the benefits derived from test intentions.

C. RQ-3: Live Study on Generating Test Cases with IntUT

To evaluate the feasibility of generating test cases for developers in their daily test writing tasks, we conduct a live study with 185 UT developers from 17 software product teams. They submitted 3,337 requests of unit test generation to IntUT and successfully obtained 2,820 responses. Over 500 failed responses were caused by the long prompt length exceeding the limitation of prompt (16k tokens), network problems, or accident closing of IntUT plugin in the IDE. Some requests were repeated by developers, we consider the latest ones in the reported results. Eventually, 1,675 unique responses generating 513K LoC of test code are collected, and 1,414 of them (84.4%) are accepted by developers. The average length of all submitted full prompts is 7,528 characters, and the test intention does not enlarge the prompt length sharply.

Note that the instant acceptance rate of 84.4% for generated unit tests does not completely imply that the accepted test cases generated by IntUT are 100% correct without any changes. The accepted tests are just looked actionable for developers and selected by them for further modification. To have a deep understanding of the quality of test cases generated by IntUT, we further collect 12-hour traceable accepted test cases that can be traced by the method signatures of the focal method and its corresponding test case generated by IntUT. A test case may not be traced due to its deletion, the project is not loaded later, or the network problem. Eventually, 632 test cases generated by IntUT are collected, of which detailed results are reported in Table VI. 40% of them are directly merged into program repositories by developers without any modification. 63% test cases are merged into program repos with less than 20% modifications at code line level, and 80% test cases are finally accepted by developers with less than 50% modifications (including the cases with 0% modification). Only 13.8% of test cases generated IntUT need the over 70% modification at the code line level. Such results from live study further show that IntUT can generate high-quality test cases for developers in practice.

Table VII
FEEDBACK DATA FROM DEVELOPERS.

Proj.	LoC of UT	Line Coverage	Branch Coverage
A	500+	81.4%	74.2%
B	200+	80.9%	62.5%
C	1000+	80.2%	61.4%

Branch and Line Coverage from Live Study: We also receive feedback from developers who use IntUT to generate test cases for three Java projects that do not have any unit test cases before using IntUT, the results are shown in Table VII. They reported that IntUT help them generate over 500, 200, and 1000 code lines of unit tests, respectively, with the line coverage of 80+% and branch coverage of 60+%. Notably, developers assess that IntUT improved the efficiency of developing test cases by 30%.

D. RQ-4: Time Costs of UT Generation with Test Intention

In this work, we also assess the efficiency of generating test cases with test intention in terms of the time costs from the aforementioned live study, considering that generating test intention takes some time and adding test intention increases the prompt length. The first interaction of generating test intention, developers generally wait for an average of 4.58 seconds, and no additional time is needed for generating other test intention for the same focal method. After submitting the full prompt to LLM, it takes an average of 10.86 seconds for IntUT to generate test cases. Such a waiting time is acceptable for developers according to their responses.

E. Discussion

Practical Feasibility: General LLM-based test generation solutions rely on fine-tuning with high-quality data and prompting with the code context of the focal method [8], or conduct multiple-time interactions to generate unit tests. These methods fully rely on the ability of LLMs and the limited context at the code token level, but cannot help developers understand the test intention for each case as they expected. It would be challenging to correctly generate test cases, especially for the focal methods with complicated implementation (e.g., multiple branches and mocking callees), to achieve the high coverage. Our proposed test intention aims to address this challenge by analyzing the code naturalness in terms of branch and mocking calless to precisely express explicit information of generating unit tests for the given focal method. Both our experimental results, live study and users' feedback demonstrate that LLM-based unit test generation guided by test intention can achieve significant improvements on generating acceptable and high-quality unit tests for developers.

Extensibility: We extend our proposed test intention guided methodology of unit test generation to the test case generation tasks for C, C++, and Go programs, with PAINT algorithms implemented using LLVM and Go SSA. All implementations received positive feedbacks from users. Recently, more than 10,000 lines of unit test in C projects have been generated for developers, achieving over 80% line coverage and helping developers uncover four vital bugs related memory leaks in C programs. Such results reveal that our methodology could be extended to other programming languages.

Post-process: Some generated test cases did not cover target code/lines due to errors in generated codes or lack of environment setting, even the test intentions generated by IntUT are correct. As LLMs show the ability to fix coding errors, adding a recovery mechanism could increase the quality of generated unit tests, which is left to future work.

VIII. Threats to Validity

Construct Threat: The construct threat is that our defined prompt format with test intention might not be perfect. To mitigate this threat, we have communicated with several senior UT developers and conducted empirical study on manual experiments with over 1,000 cases of exploring the appropriate prompt format for generating test cases with LLMs.

Internal Threat: The internal threat may lie in the generation of the test input and oracle for each test case, since they are not explicitly specified in final prompt of IntUT. To reduce this threat, the generation of test input and oracle relies on the inference capability of our fine-tuned LLM, input parameters, and condition chains analyzed by PAINT. We plan to further reduce this threat by improving the quality of test intention with explicit specification about test input and oracle by exploring advanced learning or code analysis techniques. The second threat lies in the existence of conflict conditions in our analyzed condition chains that results in an inconsistent situation. It may be resolved by symbolic execution, which is considered in future improvement of IntUT.

External Threat: The external threat is that only CodeLLaMA is used by IntUT, GPT-4 is not considered due to the potential security restriction since code data must be sent to the GPT-4 server. Other open-source LLMs are not studied with two reasons: ① CodeLLaMA presents the SOTA performance on code generating task, and ② this work focuses on designing effective prompt but not LLMs.

IX. Related Work

UT generation by prompt engineering. Prompt engineering is a low-cost application technology , hence it is highly valued by researchers. Originally, researchers designed only one individual prompt to guide LLMs to generate tests at once. Researchers have proved that the focal code context [15] [14] [16] [17], few-shot examples [9] and output instructions [18] [19] [20] are necessary to create test generation prompts. Li et al. [36] revealed that the generated tests demonstrate a degree of insensitivity to prompt perturbations, suggesting that minor variations in the prompt content do not significantly impact test quality. Zhang et al.'s study [37] showed that vulnerability information is helpful for ChatGPT to generate security tests.

Afterwards, iterative prompts were designed to obtain higher quality test cases by invoking LLMs multiple times. To reduce the compilation error, Yuan et al. [8] proposed CHATTESTER that can ask ChatGPT three times to obtain the intent of focal method, generated test, and repair solutions, respectively. To improve the test coverage, Bhatia et al. [38] proposed to add and to update repeatedly the indexes of missed statements by the existing tests to the prompts. Similarly, Pizzorno et al. [39] also integrated uncovered lines and branches in their prompts. Meanwhile, Zhang et al. [40] added low coverage ineffective tests as counter-examples into their prompts and refined them iteratively. Ryan et al. [41] proposed to leverage execution path information to optimize their prompts. To improve the bug detection ability, Dakhel et al. [42] designed iterative prompts augmented by surviving mutant codes. Liu et al. [43] presented to create code specification based two-stage prompts to generate fault-revealing test inputs and oracles.

UT generation by model training. Model training is an alternative technique when prompt engineering is ineffective. Supervised fine-tuning is a popular training strategy. To fine-tune an efficient model, researchers have collected high-quality test data and selected efficient fine-tuning algorithms. Specifically, Tufano et al. [10] introduced a large Java dataset containing 780K test cases (i.e., METHODS2TEST [11]), and fine-tune BART transformers on the test generation task. Afterwards, Shin et al. [12] leveraged the METHODS2TEST data [11] to fine-tune the CodeT5 model and the target project data to fine-tune it adain for the purpose of learning the project-specific domain knowledge. Rao et al. [44] pointed out that it is vital to incorporate software-specifc insights when training a code LLM. They hence proposed CAT-LM model, which was trained by the novel pretraining signal that explicitly considers the code and test pairs. Additionally, reinforcement learning (RL) technique also were applied to train LLM. For instance, Steenhoek et al. [45] proposed RLSQM that collects static quality metrics from generated tests and uses the metrics as rewards of RL to optimize models. Takerngsaksiri et al. [13] mined test case characteristics (e.g., syntactically correct) as the feedback for their RL algorithm.

X. Conclusion

This paper proposes IntUT, a new approach that using explicit test intention to guide LLMs to generate high-quality unit test cases. By specifying the inputs, mock conditions and expected results, a test intention can strictly guide the LLM to generate high-quality test cases. IntUT is armed with the automated test intention generation by using program analysis techniques. The experimental results on industry Java projects shows that, test cases generated by IntUT achieve 94% improvement on branch coverage and 49% improvement on line coverage. Surprisingly, we received developers' feedback of using IntUT to generate test cases in practical development of test cases, with over 80% line coverage and 30% efficiency improvement of writing test cases.

Acknowledgements

We genuinely acknowledge the insightful comments from Nan Liu, Wei Pu, Heng Sun, Lianyuan Jiao, Weimin Liu, Zheng Zhu, Dengjun Wu, Yichen Ding, and other Huawei developers with their practical development experiences.

References

[1] P. Runeson, "A survey of unit testing practices," IEEE Software, vol. 23, no. 4, pp. 22–29, 2006. [Online]. Available: https://doi.org/10.1109/MS.2006.91

[2] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014. IEEE Computer Society, 2014, pp. 201–211. [Online]. Available: https://doi.org/10.1109/ISSRE.2014.11

[3] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," IEEE Trans. Software Eng., vol. 36, no. 2, pp. 226–247, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.71

[4] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 246–256. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693084

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[6] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017. IEEE Computer Society, 2017, pp. 263–272. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.27

[7] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," CoRR, vol. abs/2308.10620, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.10620

[8] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," CoRR, vol. abs/2305.04207, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.04207

[9] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," IEEE Trans. Software Eng., vol. 50, no. 1, pp. 85–105, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2023.3334955

[10] 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

[11] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "METHODS2TEST: A dataset of focal methods mapped to test cases," in 19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022. ACM, 2022, pp. 299–303. [Online]. Available: https://doi.org/10.1145/3524842.3528009

[12] J. Shin, S. Hashtroudi, H. Hemmati, and S. Wang, "Domain adaptation for deep unit test case generation," 2024.

[13] W. Takerngsaksiri, R. Charakorn, C. Tantithamthavorn, and Y. Li, "TDD without tears: Towards test case generation from requirements through deep reinforcement learning," CoRR, vol. abs/2401.07576, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.07576

[14] S. K. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," CoRR, vol. abs/2208.05950, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2208.05950

[15] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," 2024.

[16] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. Lou, and W. Chen, "Codet: Code generation with generated tests," in The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net, 2023. [Online]. Available: https://openreview.net/pdf?id=ktrw68Cmu9c

[17] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," 2024.

[18] V. Guilherme and A. Vincenzi, "An initial investigation of chatgpt unit test generation capability," in 8th Brazilian Symposium on Systematic and Automated Software Testing, SAST 2023, Campo Grande, MS, Brazil, September 25-29, 2023, A. L. Fontão, D. M. B. Paiva, H. Borges, M. I. Cagnin, P. G. Fernandes, V. Borges, S. M. Melo, V. H. S. Durelli, and E. D. Canedo, Eds. ACM, 2023, pp. 15–24. [Online]. Available: https://doi.org/10.1145/3624032.3624035

[19] L. Plein, W. C. Ouédraogo, J. Klein, and T. F. Bissyandé, "Automatic generation of test cases based on bug reports: a feasibility study with large language models," in Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024. ACM, 2024, pp. 360–361. [Online]. Available: https://doi.org/10.1145/3639478.3643119

[20] M. Nabeel, D. D. Nimara, and T. Zanouda, "Test code generation for telecom software systems using two-stage generative model," CoRR, vol. abs/2404.09249, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.09249

[21] "apolloconfig apollo," https://github.com/ApolloAuto/apollo, accessed: 2024-03-15.

[22] "apollo-audit-impl," https://github.com/apolloconfig/apollo/tree/master/apollo-audit/apollo-audit-impl, accessed: 2024-03-15.

[23] "eclipse milo," https://github.com/eclipse/milo, accessed: 2024-03-15.

[24] "eclipse eclipse-collections," https://github.com/eclipse/eclipse-collections, accessed: 2024-03-15.

[25] "apache pulsar," https://github.com/apache/pulsar, accessed: 2024-03-15.

[26] K. Cooper and L. Torczon, Engineering a compiler. Elsevier, 2011.

[27] H. Zhu, L. Wei, M. Wen, Y. Liu, S. Cheung, Q. Sheng, and C. Zhou, "Mocksniffer: Characterizing and recommending mocking decisions for unit tests," in 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE, 2020, pp. 436–447. [Online]. Available: https://doi.org/10.1145/3324884.3416539

[28] H. Zhu, L. Wei, V. Terragni, Y. Liu, S. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," ACM Trans. Softw. Eng. Methodol., vol. 33, no. 1, pp. 16:1–16:31, 2024. [Online]. Available: https://doi.org/10.1145/3617171

[29] L. Xiao, G. Zhao, X. Wang, K. Li, E. Lim, C. Wei, T. Yu, and X. Wang, "An empirical study on the usage of mocking frameworks in apache software foundation," Empir. Softw. Eng., vol. 29, no. 2, p. 39, 2024. [Online]. Available: https://doi.org/10.1007/s10664-023-10410-y

[30] "Powermock," https://github.com/powermock/powermock, accessed: 2024-07-21.

[31] H. Li, Y. Su, D. Cai, Y. Wang, and L. Liu, "A survey on retrieval-augmented text generation," CoRR, vol. abs/2202.01110, 2022. [Online]. Available: https://arxiv.org/abs/2202.01110

[32] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, J. Jiang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," 2024.

[33] "jacoco," https://github.com/jacoco/jacoco, accessed: 2024-07-21.

[34] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024.

[35] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[36] V. Li and N. Doiron, "Prompting code interpreter to write better unit tests on quixbugs functions," CoRR, vol. abs/2310.00483, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.00483

[37] Y. Zhang, W. Song, Z. Ji, D. Yao, and N. Meng, "How well does LLM generate security tests?" CoRR, vol. abs/2310.00710, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.00710

[38] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative AI : A comparative performance analysis of autogeneration tools," CoRR, vol. abs/2312.10622, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.10622

[39] J. A. Pizzorno and E. D. Berger, "Coverup: Coverage-guided llm-based test generation," CoRR, vol. abs/2403.16218, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16218

[40] C. Yang, J. Chen, B. Lin, J. Zhou, and Z. Wang, “Enhancing llm-based test generation for hard-to-cover branches via program analysis,” CoRR, vol. abs/2404.04966, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.04966

[41] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, “Code-aware prompting: A study of coverage guided test generation in regression setting using LLM,” CoRR, vol. abs/2402.00097, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.00097

[42] A. M. Dakhel, A. Nikanjam, V. Majdinasab, F. Khomh, and M. C. Desmarais, “Effective test generation using pre-trained large language models and mutation testing,” CoRR, vol. abs/2308.16557, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.16557

[43] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, “Llm-powered test case generation for detecting tricky bugs,” CoRR, vol. abs/2404.10304, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.10304

[44] N. Rao, K. Jain, U. Alon, C. Le Goues, and V. J. Hellendoorn, “CAT-LM training language models on aligned code and tests,” in 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023. IEEE, 2023, pp. 409–420. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00193

[45] B. Steenhoek, M. Tufano, N. Sundaresan, and A. Svyatkovskiy, “Reinforcement learning from automatic feedback for high-quality unit test generation,” CoRR, vol. abs/2310.02368, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2310.02368