Advancing Code Coverage: Incorporating Program Analysis with Large Language Models

CHEN YANG, Tianjin University, China
JUNJIE CHEN*, Tianjin University, China
BIN LIN, Hangzhou Dianzi University, China
ZIQI WANG, Tianjin University, China
JIANYI ZHOU, Huawei Cloud Computing Technologies Co., Ltd., China

Automatic test generation plays a critical role in software quality assurance. While the recent advances in Search-Based Software Testing (SBST) and Large Language Models (LLMs) have shown promise in generating useful tests, these techniques still struggle to cover certain branches. Reaching these hard-to-cover branches usually requires constructing complex objects and resolving intricate inter-procedural dependencies in branch conditions, which poses significant challenges for existing techniques. In this work, we propose TELPA, a novel technique aimed at addressing these challenges. Its key insight lies in extracting real usage scenarios of the target method under test to learn how to construct complex objects and extracting methods entailing inter-procedural dependencies with hard-to-cover branches to learn the semantics of branch constraints. To enhance efficiency and effectiveness, TELPA identifies a set of ineffective tests as counter-examples for LLMs and employs a feedback-based process to iteratively refine these counter-examples. Then, TELPA integrates program analysis results and counter-examples into the prompt, guiding LLMs to gain deeper understandings of the semantics of the target method and generate diverse tests that can reach the hard-to-cover branches. Our experimental results on 27 open-source Python projects demonstrate that TELPA significantly outperforms the state-of-the-art SBST and LLM-enhanced techniques, achieving an average improvement of 34.10% and 25.93% in terms of branch coverage.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Test Generation, Program Analysis, Large Language Models

ACM Reference Format:
Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2018. Advancing Code Coverage: Incorporating Program Analysis with Large Language Models. 1, 1 (November 2018), 31 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Automatic test generation holds significant importance in software quality assurance, which aims to efficiently detect software bugs by automatically covering various behaviors of the software

*Corresponding author (junjiechen@tju.edu.cn)

Authors' Contact Information: Chen Yang, College of Intelligence and Computing, Tianjin University, Tianjin, China, yangchenyc@tju.edu.cn; Junjie Chen, College of Intelligence and Computing, Tianjin University, Tianjin, China, junjiechen@tju.edu.cn; Bin Lin, Hangzhou Dianzi University, Hangzhou, China, b.lin@hdu.edu.cn; Ziqi Wang, College of Intelligence and Computing, Tianjin University, Tianjin, China, wangziqi123@tju.edu.cn; Jianyi Zhou, Huawei Cloud Computing Technologies Co., Ltd., Beijing, China, zhoujianyi2@huawei.com.

under test. Recently, many test generation techniques have been proposed, among which Search-Based Software Testing (SBST) [29, 39, 40] is one of the most widely-studied categories due to its effectiveness. Typically, SBST techniques leverage heuristic search algorithms to explore the vast test space by mutating already-generated tests with the guidance of maximizing code coverage. While receiving extensive attention, they are still unable to reach specific parts of code. For instance, they often fail to generate tests to cover the branches that can be triggered only by some scenario-specific values that may require deep understanding of the code semantics.

The recent advancement of Large Language Models (LLMs) provides new opportunities for tackling these problems, and several LLM-based techniques have been proposed to generate effective test cases [35, 61]. These techniques utilize the code comprehension ability of LLMs by incorporating the source code of the target method to be tested and some contextual information (such as other methods around the target method) into the prompt. While the limitation of SBST can be relieved to some degree with the help of LLMs, the automatically generated tests still tend to be generic and only able to cover the branches without complicated constraints [44]. That is, many branches are still left for manual test design based on experts' domain knowledge, leading to significant costs. An automated approach that can further improve the coverage of these branches can largely reduce the testing effort.

In practice, two major challenges stand out when creating such approaches. First, some branch constraints involve objects with complex construction processes. Specifically, the construction of some objects relies on other (complex) types of objects, ultimately necessitating a specific sequence of constructor invocations. For example, the branch condition at Line 5 in Figure 1 requires a `field` object with a specific type (e.g., `Array`) and a specific attribute value (e.g., `items` should be a `(tuple, list)` with the type `typing.Union[Field, typing.Sequence[Field]]`). Constructing such a `field` object is challenging because it requires understanding the dependencies between multiple objects: (1) constructing valid `items` (which must be either a single `Field` object or a sequence of `Field` objects), and (2) creating a valid `field` object, ensuring that the `items` attribute has the correct type. This process involves multiple steps and dependencies that are difficult for existing techniques to handle without a deep understanding of the code semantics. As indicated by a previous study [61], the tests generated by LLMs exhibit a relatively low compilation success rate (about 39%), which directly impacts their effectiveness in achieving code coverage. This highlights a deficiency in the ability of LLMs to generate effective tests in such scenarios. Indeed, state-of-the-art LLM-based techniques (e.g., CODAMOSA [35]) fail to produce valid objects for such scenarios, let alone the objects with specific attribute values required by branch constraints. Second, some branch constraints entail intricate inter-procedural dependencies. That is, the outcome of certain conditions within the constraints are determined by the execution results of a series of method invocations. For instance, the branch condition at Line 3 in Figure 2 depends on the result of the method `is_magic`. This method is defined elsewhere in the project and may invoke other methods, creating a chain of inter-procedural dependencies. To cover this branch, the test must correctly simulate the action that `is_magic` performs and provide the test data that satisfy the condition. This requires a deep understanding of the code semantics to ensure that the generated test accurately reflects the dependencies and produces the correct result for the branch condition. In such scenarios, merely providing source code of the target method or some coarse-grained contextual information (the common practice of existing LLM-based techniques) is insufficient to enable LLMs to comprehend the semantics conveyed by a sequence of invoked methods, thereby struggling to generate effective tests. Therefore, we define "hard-to-cover branches" as those that meet either of the following criteria: (1) branches requiring values derived from complex object construction. For example, if a branch condition checks whether an object's attribute matches specific conditions (such as type, format, and other non-constant conditions), the test must first construct the object with the appropriate attribute to cover the branch; or (2) branches with complex inter-procedural dependencies. For example, if a branch condition depends on the result of a method invocation, which itself relies on other method invocations, the test must accurately simulate the sequence of method calls and their interactions to cover the branch.

To tackle these challenges in covering hard-to-cover branches, we propose a novel LLM-based test generation technique (called **TELPA** – **TE**st generation via **L**lm and **P**rogram **A**nalysis), which focuses on reaching hard-to-cover branches via program-analysis-enhanced prompting. Given that complex objects in branch constraints are often passed to the target method as parameters, TELPA addresses the first challenge by gathering invocations of the target method within the software module. These existing invocations might contain the whole process of constructing these objects. Specifically, TELPA conducts **object construction analysis**, which traces the sequence of method calls leading to the target method, focusing on understanding how objects are constructed and passed to the target method. This analysis starts from each invocation of the target method, to extract all sequences of method invocations with the target method as the endpoint. Then, TELPA prompts LLMs to generate tests by using different paths to enter the target method. By exploring different entry paths, TELPA not only facilitates the construction of valid objects but also captures various usage scenarios in practice (that may produce specific attribute values required by branch constraints), thus enhancing test diversity and improving coverage.

To address the second challenge arising from complicated inter-procedural dependencies in branch constraints, TELPA performs **branch dependency analysis**, which determines the methods invoked within the branch conditions of the target method, focusing on understanding the inter-procedural dependencies that influence branch outcomes. This analysis begins from each variable and invoked method in the constraints for a target branch, aiming to extract all associated methods. By incorporating the source code of these methods in their invocation order as the prompt, TELPA can elicit LLMs to gain a deeper understanding of the semantics pertaining to the target branch, thereby generating effective tests. This provides precise contextual information concerning the target branch and it is more effective than directly supplying the source code of all methods within the software module to LLMs, as the latter could introduce excessive noise irrelevant to the target branch and confuse LLMs during test generation.

Due to the non-negligible cost of utilizing LLMs, we define the usage scenario of TELPA as its activation when the existing (lightweight) test generation tool (e.g., Pynguin [39]) fails to increase test coverage within a certain timeframe. In other words, TELPA is only employed for hard-to-cover branches to ensure cost-effectiveness. To enhance efficiency further, TELPA samples a diverse set of already-generated tests as counter-examples. By incorporating these counter-examples into the prompt, TELPA can instruct LLMs to generate tests that diverge from them, as these counter-examples have been recognized as ineffective for these hard-to-cover branches. In particular, the overall test generation process with TELPA is structured as a feedback-based process. This design allows for updating counter-examples based on the most recent coverage results, thereby enhancing test generation effectiveness.

We conducted an extensive evaluation on 27 open-source Python projects, which have been widely used in previous studies [35, 39]. Our results show that TELPA significantly outperforms both the state-of-the-art SBST tool (i.e., Pynguin [39]) and the state-of-the-art LLM-based/LLM-enhanced techniques (i.e., CODAMOSA [35] and CHATTESTER [61]). On average, TELPA achieves 34.10%, 25.93%, and 21.10% higher branch coverage than them respectively, given the same testing time budget. Our ablation study also confirms the contribution of each main component in TELPA, including object construction analysis for relieving the challenge arising from complex object construction, branch dependency analysis for relieving the challenge arising from intricate inter-procedural dependencies, counter-examples sampling and coverage-based feedback for improving the overall testing efficiency. Our investigation into various configurations of TELPA confirms that our designed usage scenario for TELPA strikes a balance between effectiveness and efficiency and TELPA consistently achieves higher coverage than other settings.

The main contributions of our study are as follows.

- We introduce TELPA, a novel LLM-based test generation technique designed to enhance the coverage of hard-to-cover branches through program-analysis-enhanced prompting.
- TELPA can address the challenges arising from complex object construction and intricate inter-procedural dependencies well by incorporating both object construction and branch dependency analysis within TELPA.
- The feedback-based test generation process in TELPA improves the efficiency and effectiveness of test generation by guiding LLMs with diverse counter-example tests.
- We present an extensive evaluation of TELPA across 27 open-source projects, demonstrating its effectiveness in comparison to the state-of-the-art SBST tool and two state-of-the-art LLM-based/LLM-enhanced techniques.

2 MOTIVATING EXAMPLE

In this section, we demonstrate the limitations of state-of-the-art automated test generation tools for Python, namely Pynguin [39], CODAMOSA [35], and CHATTESTER [61] with two real examples to motivate our study.

Figure 1 shows the simplified code snippet of the target method `set_definition` from the `typesystem` project[1]. With a time budget of 20 minutes, Pynguin, CODAMOSA, and CHATTESTER all fail to reach line 6, namely the condition for the `if` statement in line 5 never returns true. In fact, to make the condition true, the generated test needs to first construct an `Array` object (line 3), and then assign a proper value to its `items` attribute. More specifically, `items` should be either a tuple or a list of `Field` objects (line 4), according to the type hints from the constructor of `Array` (line 13). Constructing such a sophisticated `Array` object is certainly challenging for automatic test generation techniques, Pynguin, CODAMOSA, and CHATTESTER are no exception.

Another type of hard-to-cover branches are caused by complicated inter-procedural dependencies, as illustrated in Figure 2. In this case, our target method is `is_public_family` from the `apimd` project[2]. Pynguin, CODAMOSA, and CHATTESTER are unable to guide the condition in line 3 to return true. The challenge here is that the outcome of the `if` condition depends on the method `is_magic`, which is defined elsewhere in the project and might further invoke other methods. Those complicated inter-procedural dependencies are often hard to understand and analyze for test generation techniques.

According to our analysis, our evaluation benchmark, which consists of 486 modules from 27 Python projects, contains 6,559 branches, of which 3,979 (60.7%) are classified as hard-to-cover. Similar challenges are also prevalent in Java, as evidenced by the Defects4J benchmark, which contains 9,391 branches, of which 5,494 (58.5%) are hard-to-cover. This highlights the prevalence of these challenges.

There are several reasons which might contribute to the inability of the state-of-the-art techniques to handle branches with complex objects or inter-procedural dependencies.

First, during the test generation process, existing techniques [35, 39] normally try to construct minimal or over-simplified objects, without taking into account the real usage scenarios involving

[1]https://github.com/encode/typesystem
[2]https://github.com/KmolYuan/apimd

```python
def set_definitions(field, definitions) -> None:
    <other code>
    elif isinstance(field, Array):
        if field.items is not None:
            if isinstance(field.items, (tuple, list)):
                <other code>
        else:
            <other code>
    <other code>


class Array(Field):
    def __init__(
        self,
        items: typing.Union[Field, typing.Sequence[Field]] = None,
        additional_items: typing.Union[Field, bool] = False,
        <other code>
    ) -> None:
```

Fig. 1. Target method *set_definitions*

```python
def is_public_family(name: str, <other paramters>) -> bool:
    for n in name.split('.'):
        if is_magic(n):
            <other code>
        if n.startswith('_'):
      <other code>
    return <return value>


def is_magic(name) -> bool:
    name = name.rsplit('.', maxsplit=1)[-1]
    return name[:2] == name[-2:] == '__'
```

Fig. 2. Target method *_is_public_family*

specific attributes. Second, for the inter-procedural dependencies, state-of-the-art techniques [35, 39] are not able to recursively analyze how dependent methods interact with variables from the target method, thus failing to interpret the concrete branch constraints. Third, existing LLM-based techniques [35, 61] typically provide either unnecessary or insufficient contexts for LLMs to generate tests. CODAMOSA provides the whole module as the context, where many parts of code are irrelevant to the target method and thus introduce noise to LLMs. CHATTESTER provides the class declaration, constructor signatures, relevant fields, and the target method as the context, which misses the broader context dependent to the target method.

Based on these potential reasons behind the unsatisfactory performance of existing techniques, we propose TELPA, an LLM-based test generation technique leveraging program-analysis-enhanced prompting. TELPA will 1) conduct object construction analysis to learn real usage scenarios of objects and overcome the difficulty of constructing complex objects, 2) conduct branch dependency analysis to understand the behavior of complicated inter-procedural dependencies, and 3) supply relevant information to LLMs for test generation, avoiding imprecise context.

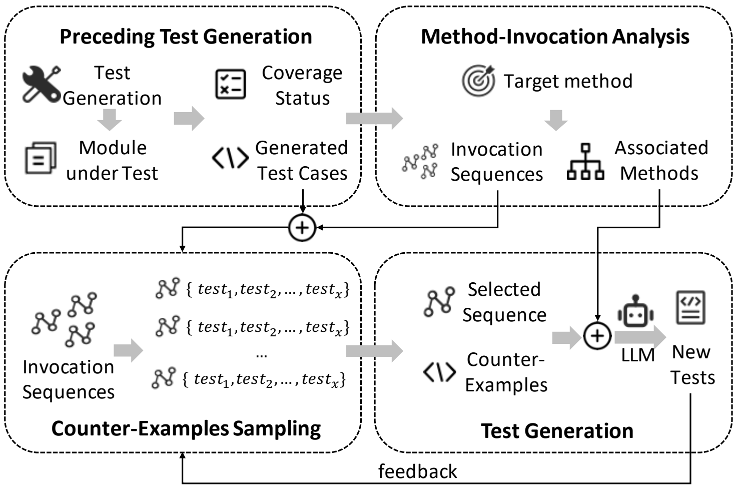


Fig. 3. Overview of TELPA

3 APPROACH

The overall workflow of TELPA is illustrated in Figure 3. First, TELPA leverages existing test generation tools to generate a set of tests, which can reach those easy-to-cover branches. For ease of presentation, we call this step *preceding test generation*. Then, for the methods containing the branches that are still not covered, TELPA performs two types of program analyses: 1) object construction analysis: This analysis traces backward from the target method to identify sequences of method calls that lead to the construction of objects used as parameters. By capturing real usage scenarios, this analysis helps the LLM generate tests that construct valid objects with the necessary attributes; 2) branch dependency analysis: This analysis explores forward from the branch conditions of the target method to identify all methods that influence the outcome of the conditions. By providing precise contextual information about these dependencies, this analysis enables the LLM to generate tests that effectively cover hard-to-reach branches. To improve the testing efficiency, TELPA samples existing generated tests as counter-examples to guide LLMs to generate different tests, as they are shown ineffective to cover specific branches. These counter-examples, together with the program analysis results, are incorporated into the prompt for LLMs to generate new tests. The new tests will be executed and added into existing tests for future iterations. In the following, we will introduce TELPA in detail by using the example shown in Figure 4 for facilitating illustration.

3.1 Preceding Test Generation with Existing Tools

As our goal is to reach those hard-to-cover branches, TELPA uses existing test generation tools for the initial testing task (for those easy-to-cover branches). This helps improve the cost-effectiveness of the overall testing process due to the cost of utilizing LLMs. Specifically, for a project under test, TELPA runs an existing test generation tool until it is unable to cover new branches within a pre-defined timeframe. Following previous work [35], we conducted a pilot study on a small benchmark and then set the timeframe to 2 minutes based on the observed results in TELPA. When executing these generated tests, their corresponding coverage information is also recorded. In this section, we mainly describe the idea behind TELPA, and the implementation details will be presented in Section 4.1.1.

```
<irrelevant code>
class Class1():
    def dependent_method0(self, param):
        <other code>

    def dependent_method1(self):
        <other code>

    def method1(self):
        var0 = dependent_method1()

        if var0 and dependent_method0(var0):
            <other code>

class Class2():
    def method3(self):
        <other code>
        obj1.method1()
        obj1.method2()
<irrelevant code>
```

Fig. 4. Illustrative Example

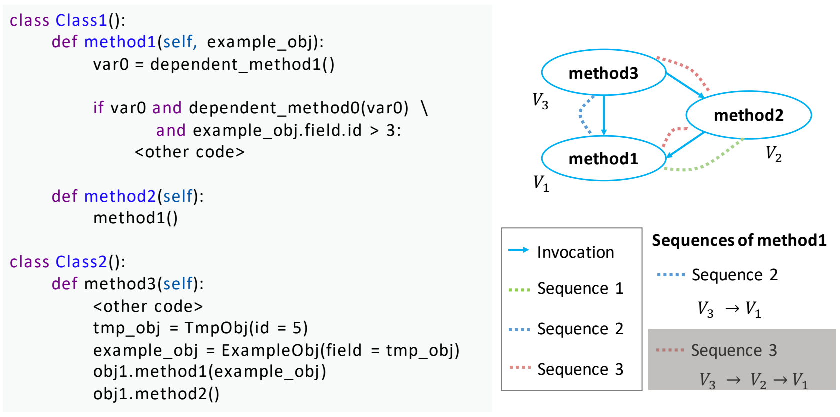
3.2 Method-Invocation Analyses

TELPA first collects the methods containing branches not covered by the preceding testing process, and then performs method invocation analyses on these methods to gather information needed for handling complex object construction and inter-procedural dependencies. This process is done in ascending order of coverage achieved for the methods, as we assume that methods with the lower coverage are more difficult to tackle and there is larger room for coverage improvement. Unlike existing LLM-based techniques (e.g., CODAMOSA [35]) that try to incorporate as much source code from the module as possible into prompts, we will only feed LLMs with the methods retrieved from method-invocation analyses as the context in order to reduce noise introduced by irrelevant code and alleviate computational burden of LLMs.

3.2.1 Object Construction Analysis. The goal of the object construction analysis is to extract method-invocation sequences ending with the target method. These sequences represent diverse real usage scenarios of the target method in the project and have a high chance to capture the whole process of constructing objects used in the target method. For example, a method v_0 may create an object o and invoke the target method v with o as an argument. A object construction analysis could collect the invocation sequence $v_0 \rightarrow v$ and trace back to the object construction process.

To achieve this goal, TELPA first constructs a method call graph within the module hosting the target method, and then extracts all paths terminating at the target method as the method invocation sequences.

Call Graph Construction. TELPA constructs a directed call graph to model method invocation relationships within the module containing the target method. The nodes in the graph represent the methods in the module, and the edges represent invocation relationships between pairs of methods. Figure 5 illustrates a simplified version of the code shown in Figure 4 and the constructed call graph. Given the simplified code snippet, the call graph will contain nodes `method1` (denoted by v_1) and `method2` (denoted by v_2) from `Class1`, and `method3` (denoted by v_3) from `Class2`. Clearly, v_1 is called by v_2 and v_3, while v_2 is only called by v_3.



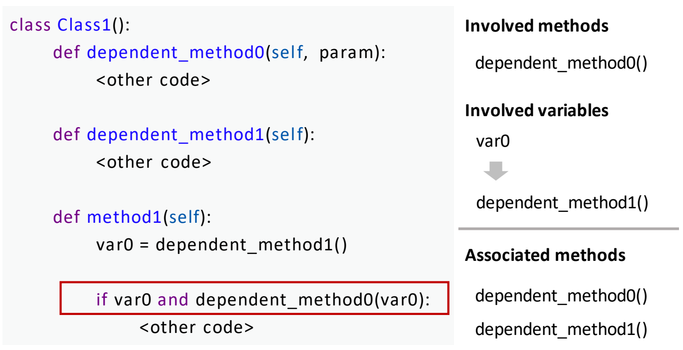
```
class Class1():
    def method1(self, example_obj):
        var0 = dependent_method1()

        if var0 and dependent_method0(var0) \
              and example_obj.field.id > 3:
            <other code>

    def method2(self):
        method1()

class Class2():
    def method3(self):
        <other code>
        tmp_obj = TmpObj(id = 5)
        example_obj = ExampleObj(field = tmp_obj)
        obj1.method1(example_obj)
        obj1.method2()
```

Fig. 5. Example of object construction analysis



```
class Class1():
    def dependent_method0(self, param):
        <other code>

    def dependent_method1(self):
        <other code>

    def method1(self):
        var0 = dependent_method1()

        if var0 and dependent_method0(var0):
            <other code>
```

Fig. 6. Example of branch dependency analysis

Sequence Extraction. With the directed call graph, TELPA proceeds to extract sequences of method invocations for the target method. Specifically, Depth-First Search (DFS) is used to identify all paths starting from entry nodes and terminating at the target method. If a cycle is encountered during path extraction, the traversal of that path will stop to prevent creating infinite paths. Going back to the example in Figure 5, for the target method v_1, three paths will be extracted: $p_1 = \{v_3 \longrightarrow v_2 \longrightarrow v_1\}$, $p_2 = \{v_3 \longrightarrow v_1\}$ and $p_3 = \{v_1\}$.

To alleviate computational load, only the shortest path from each entry point to the target method is retained. This strategy is used as such shorter and more direct call paths are easier for LLMs to understand and learn from. In our example, the path p_2 is selected (that is, p_1 is filtered out).

3.2.2 Branch Dependency Analysis. As illustrated in Section 2, it is challenging for test generation techniques to reach certain branches when the outcome of the branch condition depends on other invoked methods. To address it, TELPA performs a branch dependency analysis to recursively collect all methods that might be relevant to determining which branch to execute. The extracted methods provide rich information for interpreting what really entails in branch conditions. Below, we describe the process in detail with the example presented in Figure 6.

To extract the set of associated methods $\mathcal{B}_v$ for a target method v_1 (`method1` in Figure 6), TELPA first identifies branch conditions and captures all variables (`var0`) and methods (`dependent_method0`) that these condition expressions contain. All involved methods (`dependent_method0`) are added to $\mathcal{B}_v$. For each involved variable, TELPA recursively extracts relevant assignment statements. In our example, the value of `var0` is determined by `dependent_method1`. Therefore, TELPA considers `dependent_method1` is relevant to `var0` and adds the involved methods (`dependent_method1`) to $\mathcal{B}_v$. In this way, all methods that can impact the outcome of branch conditions are retrieved and a precise search scope is identified to understand the behavior of condition expressions.

Note that TELPA does not always use both branch dependency analysis and object construction analysis simultaneously. Specifically, it applies these analyses selectively based on the characteristics of the target method. Branch dependency analysis is used when the target method involves intricate inter-procedural dependencies in branch conditions, and object construction analyses is used when the target method involves complex objects in branch conditions. If the target method involves both complex objects and inter-procedural dependencies, TELPA applies both analyses to address these challenges comprehensively. For the simpler cases where the target method does not involve complex objects and inter-procedural dependencies, TELPA provides the method under test itself as its context to the LLM, like the existing LLM-based approaches (such as CHATTESTER). This ensures that TELPA remains effective for such cases.

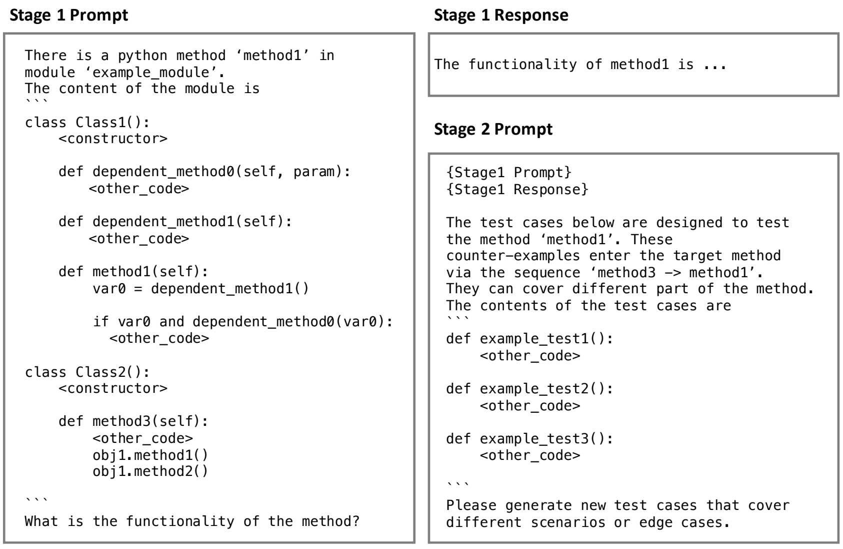
3.3 Counter-Examples Sampling

With the preceding testing process described in Section 3.1, we have acquired a set of tests that fail to reach specific hard-to-cover branches. These existing tests can be used as counter-examples to guide LLMs to generate divergent tests, further enhancing the overall test efficiency. However, incorporating all existing generated tests into the prompt is impractical, which will not only impose significant computational burden on LLMs but also easily exceed the input length limit of the prompt. To mitigate this, TELPA samples a diverse set of counter-example tests, striking a balance between effectiveness and efficiency. As our goal is to reach hard-to-cover branches, it is essential to let LLM understand which branches have already been covered by existing tests. Meanwhile, as mentioned before, we want to have a small (preferably minimal) set of tests as counter-examples.

To achieve this, TELPA employs a coverage-based approach. For each method invocation sequence of a target method v obtained from the object construction analysis, TELPA collects all the existing tests that invoke the first method of the sequence (denoted by v_0) as the candidate tests (denoted by $\mathcal{T}_{v_0}$). We only consider the first method of the sequence as it will in the end propagate to the target method through the chain of method calls. From $\mathcal{T}_{v_0}$, TELPA first selects the test that achieves the highest coverage for the target method v, then picks the next tests that can achieve the highest incremental coverage. This is an iterative process until no more test can increase the coverage. In this way, the minimal set of tests can be acquired to reach all already-covered branches. Note that the counter-examples are constructed for each method invocation sequence, this is because in the later stage of test generation, each time we will only feed one method invocation sequence to the LLM to alleviate the computational load and avoid exceeding the input length limit.

3.4 Feedback-based Test Generation Process with LLMs

TELPA integrates program analysis results and counter-examples into prompts for the LLM to generate new tests. Previous studies have shown large performance improvement for LLMs through Chain-of-Thought (CoT) [55, 62]. Therefore, we also adopt a typical CoT strategy and divide the process into two stages. Note that TELPA can also adopt different CoT techniques, as the contribution of TELPA lies in improving the prompting approach instead of designing CoT techniques.


Stage 1 Prompt
Stage 1 Response
Stage 2 Prompt

```
There is a python method ‘method1’ in
module ‘example_module’.
The content of the module is
```
class Class1():
    <constructor>

    def dependent_method0(self, param):
        <other_code>

    def dependent_method1(self):
        <other_code>

    def method1(self):
        var0 = dependent_method1()

        if var0 and dependent_method0(var0):
          <other_code>

class Class2():
    <constructor>

    def method3(self):
        <other_code>
        obj1.method1()
        obj1.method2()

```
What is the functionality of the method?
```

```
The functionality of method1 is ...
```

```
{Stage1 Prompt}
{Stage1 Response}

The test cases below are designed to test
the method ‘method1’. These
counter-examples enter the target method
via the sequence ‘method3 -> method1’.
They can cover different part of the method.
The contents of the test cases are
```
def example_test1():
    <other_code>

def example_test2():
    <other_code>

def example_test3():
    <other_code>

```
Please generate new test cases that cover
different scenarios or edge cases.
```

Fig. 7. Example of prompt construction

In the first stage, TELPA constructs the context of the target method v with two sources: 1) methods in a selected method-invocation sequence from object construction analysis, and 2) the associated methods from branch dependency analysis. For each method, TELPA incorporates the declarations and constructors of their associated classes and the content of the method itself as the context information. To select the method-invocation sequence for v in each iteration, TELPA first uses the sequence $p = \{v\}$ (i.e., the entry method is v itself), as this is the most straightforward sequence for the LLM to invoke and understand the semantics of v. If this sequence does not exist (e.g., the target method is not a public method), or the generated tests with it in the second stage do not cover new branches (note that our testing process is iterative with feedback), TELPA randomly selects an unused sequence. The context information is then fed to the LLM for summarizing the functionality of v, enabling the LLM to gain contextual awareness of method semantics. The test generation process for v terminates when no sequence is left. For example, given *method1* in Figure 4, if the selected sequence is $p = v_3 \longrightarrow v_1$, the prompt will be constructed using *method3* from the object construction analysis, *dependent_method0* and *dependent_method1* from the branch dependency analysis, and *method1* itself, as shown in Figure 7.

In the second stage, TELPA integrates the contents of the sampled counter-examples corresponding to the method invocation sequence into the prompt, along with a description: "These counter-examples enter the target method via the selected sequence of method invocations. They can cover different parts of the method. Please generate new test cases that cover different scenarios or edge cases. ", as shown in Figure 7. Such a description can facilitate the understanding of the LLM on the intent of the generation, namely to cover different branches in the target method following the method invocation sequence. The LLM is then instructed to generate divergent tests.

The new test generated by the LLM is executed and added to the existing tests along with its coverage information. The used counter-examples are removed from the candidates to ensure the diversity in future counter-example selection. This type of feedback helps TELPA generate more tests with different behaviors.

4 EVALUATION

To evaluate the performance of TELPA, we formulate the following research questions (RQs).

- **RQ1**: *To what extent can TELPA improve code coverage compared to the state-of-the-art techniques?*
- **RQ2**: *How does each of the main components in TELPA affect the performance?*
- **RQ3**: *How does the adoption of preceding test generation tools and LLMs impact the performance of TELPA?*
- **RQ4**: *How is the correctness of the unit tests generated by TELPA?*
- **RQ5**: *How generalizable is TELPA?*

4.1 Experimental Setup

4.1.1 Implementation of TELPA. While TELPA is not limited to specific programming languages, in this experiment, we target test generation for Python programs in RQ1-RQ4, given the popularity of Python and the existing Python benchmarks [35]. Additionally, we adapted TELPA to Java to evaluate its generalizability in RQ5.

Regarding the Python version of TELPA, to construct the call graph, we rely on PyCG [45], a state-of-the-art call graph generator for Python. PyCG provides a robust foundation for identifying method invocation relationships within the code. PyCG constructs an assignment graph that tracks the assignment relations between program identifiers, such as variables and classes. Furthermore, it uses namespace and scope resolution to distinguish between methods with the same name in different classes. To better align with TELPA's requirements for sequence extraction and prompt construction, we map the nodes in the PyCG-extracted call graph back to the corresponding AST nodes parsed using Python's *AST* module. Since TELPA focuses on generating tests for the target module, we primarily consider method invocations within the module under test. External functions (e.g., those from imported libraries) are not further analyzed for sequence extraction, as their internal implementations are outside the scope of the target module. For coverage analysis, TELPA uses *coverage.py* [2], a widely-used tool for measuring code coverage in Python programs. The LLM *Phind-CodeLlama-34B-v2* [11] from Hugging Face [5] is employed for test generation by default, as it is one of the most effective open-source code LLMs according to the LLM leaderboard maintained by Hugging Face [7]. *PyTorch 1.13* [12], *Hugging Face Transformers 4.35.2* [13], and *fastchat 0.2.30* [4] are used to run the LLM locally. Following the existing work [57, 59], we set the temperature to 0, enabling TELPA to obtain more deterministic results from the LLM.

To reduce the resource consumption, TELPA uses state-of-the-art techniques for initial test generation. In our study, we adopted two test generation tools separately: a state-of-the-art SBST tool Pynguin [39] and a state-of-the-art LLM-enhanced technique CODAMOSA [35]. That is, we constructed two instances of TELPA: Pynguin-based TELPA$_p$ and CODAMOSA-based TELPA$_c$. The adoption of these two different types of preceding test generation techniques ensures the conclusion generalizability. More information regarding Pynguin and CODAMOSA will be given in Section 4.1.2.

Specifically, to answer RQ5, we adapted TELPA to Java. Note that although TELPA is in theory language-independent, it still requires engineering efforts as the analysis and the parsing of test cases are highly specific to the chosen frameworks. Thus we made the engineering effort to extend TELPA to support Java projects. Specifically, we adapted TELPA's program analysis components to work with Java code by incorporating JavaParser [9], allowing to effectively parse Java code. We also implemented the test execution process using the JUnit framework (the standard framework for unit testing in Java) and incorporated Jacoco [8] for code coverage analysis (a widely-used tool for measuring Java code coverage). Additionally, we modified the processing of the LLM's output to extract Java tests and address potential issues such as missing external libraries.

Similarly, to reduce resource consumption, TELPA leverages the state-of-the-art search-based test generation tool for Java (i.e., EvoSuite [30]) for preceding test generation. Specifically, EvoSuite is used to generate initial tests, when EvoSuite no longer improves coverage within a set time window, TELPA is activated, takes over the test generation process entirely and leverages EvoSuite-generated tests as counter-examples. The LLM is used in the same way as the Python version of TELPA.

As mentioned before, TELPA is activated when the preceding tool fails to increase coverage within a pre-defined timeframe. As previous work [35], we conducted a pilot study on a small test benchmark to optimize the performance of TELPA. The timeframe is set to 2 minutes based on the observed results, namely TELPA will take over the test generation task if Pynguin, CODAMOSA or EvoSuite cannot increase the coverage for two minutes. This choice is further discussed in Section 4.7.

4.1.2 Baselines for RQ1 and RQ4. To answer RQ1 and RQ4, we compared TELPA with three baselines: **Pynguin**, **CODAMOSA**, and **CHATTESTER**.

Pynguin [39] is the state-of-the-art SBST tool for Python. It initiates the testing process by taking the Python code under test as input, and employs search-based algorithms (including MIO [19, 20], MOSA [41], and DynaMOSA [41]) to generate new tests by mutating values and statements of a seed test. Here, we used *Pynguin 0.19.0* with DynaMOSA as this algorithm has been demonstrated to be the most effective one among all supported algorithms in Pynguin [39].

CODAMOSA [35] is the state-of-the-art LLM-based test generation tool, which was proposed for improving SBST techniques. It prompts LLMs to generate seed tests for Pynguin in order to help SBST escape the local optimum. More specifically, when Pynguin falls into the local optimum, CODAMOSA switches to invoke LLMs to generate new seed tests by incorporating as much source code of the module under test into the prompt as possible, and then switches back to Pynguin for running on these new seed tests, until the given testing budget is reached. Specifically, we consider CODAMOSA as an integral tool integrating both Pynguin and the LLM, and then incorporate it into the preceding test generation component of TELPA. In the case of using CODAMOSA as the preceding test generation tool, when the switch (between Pynguin and the LLM) within CODAMOSA does not improve the coverage any more within the given timeframe (exceeding the switching time used within CODAMOSA), TELPA makes CODAMOSA terminate and then activates its program-analysis-enhanced LLM-based test generation component for further coverage improvement. CODAMOSA originally employed *Codex* as the LLM for generating seed tests. However, as the Codex service has been shut down [1], we employed *Phind-CodeLlama-34B-v2* as the LLM for CODAMOSA to have a fair comparison with our technique. We call it CODAMOSA^{phi}.

CHATTESTER [61] is another state-of-the-art LLM-based test generation tool, which first extracts context information for ChatGPT to generate tests and then designs post-processing strategies to fix invalid tests. The main contribution of TELPA is the novel prompting method. Unlike CODAMOSA^{phi} that incorporates as much source code of the module under test as possible for prompting, or CHATTESTER that incorporates insufficient context for prompting, TELPA feeds the code information highly relevant to branch constraints to LLMs. However, CODAMOSA^{phi} and TELPA differ in both their LLM invocation workflows and their prompting methods. Specifically, CODAMOSA^{phi} iteratively switches between the search-based tool Pynguin and the LLM, while once TELPA switches, it does not revert back to the preceding tool. That is, TELPA continues to use the LLM with its program-analysis-enhanced prompting for subsequent test generation. Additionally, CODAMOSA$^{\text{phi}}$ uses a simple prompting method by incorporating the entire module's source code as context for the LLM. TELPA introduces a novel prompting method that leverages program analysis to provide precise and relevant context for the LLM. Therefore, the comparison with the LLM-enhanced CODAMOSA$^{\text{phi}}$ cannot clearly demonstrate the impact of this new prompting method due to the different LLM invocation workflows. Therefore, we used CHATTESTER as another LLM-based baseline. To highlight the effectiveness of advanced prompting methods for test generation, we directly replaced our novel prompting method with the one used by CHATTESTER and removed the post-processing strategy from CHATTESTER. Note that this component is orthogonal to the test generation process and can be also applied to TELPA. That is, we activated the prompting method of CHATTESTER for test generation when the preceding test generation tool reaches a coverage plateau, same as the workflow of TELPA. CHATTESTER originally employed ChatGPT as the LLM for generating tests, we replaced it with *Phind-CodeLlama-34B-v2* for fair comparison. We call it CHATTESTER$^{\text{phi}}$. Since both Pynguin and CODAMOSA$^{\text{phi}}$ are used as preceding test generation tools in our study, we also created two instances of CHATTESTER for comparisons: Pynguin-based **CHATTESTER$_p^{\text{phi}}$** and CODAMOSA-based **CHATTESTER$_c^{\text{phi}}$**.

To sum up, we have two sets of comparisons: 1) TELPA$_p$ vs. Pynguin vs. CHATTESTER$_p^{\text{phi}}$ and 2) TELPA$_c$ vs. CODAMOSA$^{\text{phi}}$ vs. CHATTESTER$_c^{\text{phi}}$. These comparisons aim to investigate whether TELPA can improve the existing test generation tools (Pynguin and CODAMOSA$^{\text{phi}}$) and examine whether our novel prompting method can outperform the state-of-the-art prompting method proposed by CHATTESTER regardless of the preceding test generation tools.

4.1.3 Variants for RQ2. To answer RQ2, we constructed seven variants of TELPA for the ablation experiment to investigate the contribution of each component in TELPA :

- **TELPA$_{nba}$** that removes the object construction analysis from TELPA;
- **TELPA$_{nfa}$** that removes the branch dependency analysis from TELPA;
- **TELPA$_{nce}$** that removes the counter-example guidance from TELPA;
- **TELPA$_{rce}$** that randomly selects the same number of counter-examples instead of coverage-based sampling in TELPA;
- **TELPA$_{nf}$** that removes the feedback process from TELPA. Specifically, the newly generated tests by the LLM, along with their coverage information, will not be added to the pool of existing tests by TELPA$_{nf}$.
- **TELPA$_{npf}$** that removes the path filtering process from TELPA. Specifically, instead of condensing the sequences with the same entry and ending method, we keep all sequences.
- **TELPA$_{ncot}$** that removes the functionality summarization process from TELPA.

Similar to RQ1, we constructed two sets of variants based on the preceding test generation tool: Pynguin-based variants (TELPA$_{nba}^p$, TELPA$_{nfa}^p$, TELPA$_{nce}^p$, TELPA$_{rce}^p$, TELPA$_{nf}^p$, TELPA$_{npf}^p$, TELPA$_{ncot}^p$); and CODAMOSA-based variants (TELPA$_{nba}^c$, TELPA$_{nfa}^c$, TELPA$_{nce}^c$, TELPA$_{rce}^c$, TELPA$_{nf}^c$, TELPA$_{npf}^c$, TELPA$_{ncot}^c$).

4.1.4 Variants for RQ3. TELPA is built on top of the adopted preceding test generation tools and LLM. To better understand how this design choice will impact the performance of TELPA, in this RQ, we construct another two variants. In the first variant, named **TELPA$_{scratch}$**, no preceding test generation tool is used. That is, TELPA is activated from the very beginning. In the second variant, we consider a different LLM. The default LLM in TELPA is *Phind-CodeLlama-34B-v2*, which is relatively large-scale and may require a lot of resources to run. To make TELPA more practical, we also investigated the effectiveness of TELPA when a relatively small-scale LLM is adopted. More specifically, we employed *DeepSeek-Coder-6.7B-instruct* [3], which is one of the most effective LLMs among the set of LLMs with comparable (small) scales on HuggingFace according to the LLM leaderboard maintained by Hugging Face [7]. Accordingly, we call the variant TELPA$_{ds}$. Similar to previous RQs, Pynguin-based and CODAMOSA$^{\text{phi}}$-based variants are created and named as **TELPA**$^{p}_{ds}$ and **TELPA**$^{c}_{ds}$. Furthermore, the commercial closed-source LLMs developed by OpenAI, such as ChatGPT and GPT-4, have shown superior performance on various benchmarks. Therefore, in addition to open-source LLMs, we also conducted an experiment to understand the effectiveness of TELPA when using the more advanced GPT-4. Due to the high cost of invoking GPT-4's APIs, we only constructed a Pynguin-based variant, i.e., TELPA$^{p}_{gpt}$.

4.1.5 Baseline for RQ5. To answer RQ5, we adapted TELPA to Java and constructed TELPA$_{e}$, which leverages the state-of-the-art search-based test generation tool for Java (i.e., **EvoSuite** [30]) for preceding test generation and compared it with EvoSuite.

EvoSuite is one of the most prominent search-based techniques. It employs evolutionary algorithms to generate unit tests aimed at maximizing code coverage. Additionally, it integrates advanced program analysis techniques to further enhance its effectiveness, including leveraging a constant pool for generating test inputs and applying testability transformations to improve guidance during the test generation process, etc.

4.1.6 Benchmark, Metrics, and Environment. To answer RQ1-RQ4, following the existing work [35], we evaluated TELPA on the widely-studied benchmark, which consists of 486 modules from 27 Python projects. The whole benchmark contains 42,897 source lines of code and 4,518 methods in total. Since TELPA focuses on improving coverage for hard-to-cover branches, we also analyzed the branches within the benchmark. The benchmark contains a total of 6,559 branches, of which 3,979 are classified as hard-to-cover. Specifically, 3,931 branches involve complex dependencies, while 256 branches are associated with complex objects.

To answer RQ5, we further evaluated TELPA on four real-world open-source projects using the JUnit framework [10] in the widely-used Defects4J benchmark [33] due to the significant popularity of Java and JUnit (i.e., Chart, Time, Lang, and Math). Unlike Python, Java is a statically typed language. Automated test case generation for dynamically typed languages often struggles with object creation, as obtaining detailed type information can be challenging. In contrast, Java's static typing allows for more precise and efficient handling of type information, making it an ideal choice for evaluating the effectiveness of TELPA on static type systems. Here, we excluded Closure since it does not use the JUnit framework and involves significant testing costs. In total, the benchmark contains 9,391 branches, of which 5,494 are classified as hard-to-cover. Specifically, 2,453 branches involve complex dependencies, while 4,842 branches are associated with complex objects.

Following the existing work on test generation [35, 39], we adopted both branch coverage and line coverage as metrics to measure the effectiveness of a test generation technique. Specifically, to evaluate the correctness of the generated tests cases in RQ4, we executed all generated test cases and measure the correctness using two metrics: (1) *Syntax Correctness* refers to the percentage of generated tests that are syntactically valid. Specifically, we checked this by parsing the generated tests using Python's built-in ast module. Any test that passed parsing without errors was considered syntactically correct. (2) *Execution Pass Rate* refers to the percentage of generated tests that executed successfully without runtime errors. Specifically, we ran the tests and recorded any runtime errors or failures during execution. For each module and each technique in a project, we allocated 20 minutes for test generation. All the techniques kept running until the time budget ran

Table 1. Comparison among TELPA, CHATTESTER, Pynguin and CODAMOSA in terms of branch coverage on all branches

Project	**Pynguin$_{bn}$**	**Pynguin**	**CHATTESTER$^{phi}_{p}$**	**TELPA$_p$**	**CODAMOSA$^{phi}_{bn}$**	**CODAMOSA^{phi}**	**CHATTESTER$^{phi}_{c}$**	**TELPA$_c$**
pysnooper	16.67%	18.07%	20.00%	**31.23%**	16.67%	18.25%	27.37%	**30.70%**
apimd	39.63%	43.18%	48.04%	**81.50%**	6.92%	39.81%	51.78%	**67.94%**
blib2to3	23.55%	26.09%	25.18%	**42.39%**	19.59%	25.58%	30.66%	**38.78%**
codetiming	65.00%	75.00%	80.00%	**95.00%**	70.00%	70.00%	60.00%	**95.00%**
cookiecutter	45.37%	54.15%	63.90%	**64.63%**	45.37%	57.80%	52.44%	**67.07%**
dataclasses_json	16.11%	16.37%	21.77%	**27.96%**	17.17%	17.61%	17.17%	**28.94%**
docstring_parser	43.58%	53.77%	72.64%	**85.85%**	83.77%	86.04%	89.62%	**91.13%**
flutes	68.61%	77.22%	81.94%	**84.17%**	68.06%	76.67%	76.11%	**82.50%**
flutils	46.52%	47.41%	59.70%	**78.89%**	66.22%	66.74%	67.26%	**78.00%**
httpie	28.14%	29.79%	38.35%	**45.52%**	26.49%	27.11%	40.05%	**44.90%**
isort	94.00%	99.00%	98.00%	**100.00%**	94.00%	98.00%	94.00%	**98.00%**
mimesis	76.97%	80.42%	80.25%	**89.75%**	68.57%	79.58%	83.45%	**89.66%**
py_backwards	28.62%	42.00%	50.00%	**63.69%**	31.69%	36.00%	33.23%	**60.31%**
pymonet	62.67%	65.33%	68.00%	**85.33%**	61.33%	63.33%	71.56%	**88.22%**
pypara	44.13%	56.35%	47.14%	**63.97%**	18.41%	48.57%	34.13%	**56.51%**
semantic_release	37.50%	38.33%	45.56%	**60.28%**	37.22%	37.22%	39.72%	**60.00%**
string_utils	83.75%	85.62%	93.12%	**97.97%**	64.84%	80.62%	85.31%	**87.97%**
pytutils	35.95%	37.03%	50.95%	**57.57%**	41.08%	45.14%	43.92%	**57.57%**
sanic	44.56%	44.91%	59.65%	**63.07%**	46.23%	48.60%	51.58%	**59.91%**
sty	87.14%	90.00%	91.43%	**95.71%**	90.00%	93.57%	89.29%	**95.00%**
thefuck	19.88%	20.71%	21.07%	**50.60%**	21.90%	23.45%	38.69%	**54.17%**
thonny	17.35%	36.63%	34.58%	**38.31%**	27.11%	36.51%	37.95%	**43.13%**
tornado	46.30%	47.89%	55.75%	**65.36%**	42.79%	44.05%	51.20%	**65.61%**
tqdm	15.00%	18.48%	40.45%	**44.70%**	35.30%	42.12%	38.33%	**49.24%**
typesystem	33.08%	37.50%	39.46%	**50.87%**	20.94%	56.09%	57.32%	**72.54%**
youtube_dl	15.28%	16.97%	19.46%	**25.03%**	20.70%	23.57%	24.56%	**29.01%**
ansible	28.00%	29.86%	30.88%	**38.02%**	29.73%	31.38%	30.76%	**37.78%**
Average Branch Cov.	43.09%	47.71%	53.23%	**63.98%**	43.41%	50.87%	52.50%	**64.06%**
Average Line Cov.	62.36%	64.72%	70.20%	**75.54%**	58.37%	67.92%	72.64%	**74.25%**

out, even if the coverage did not grow before that. Note that the time costs spent on all components in TELPA (including the preceding test generation, program analysis, test generation, and the coverage analysis) were included in this time budget. The 20-minute time budget was selected to ensure that all the studied techniques have sufficient time to explore the test space and demonstrate their potential as much as possible. While the previous studies set shorter time budgets for evaluation (e.g., 10 minutes for evaluating Pynguin and CODAMOSA), a longer time budget is helpful to assess the sustained effectiveness of each technique over a sufficient period.

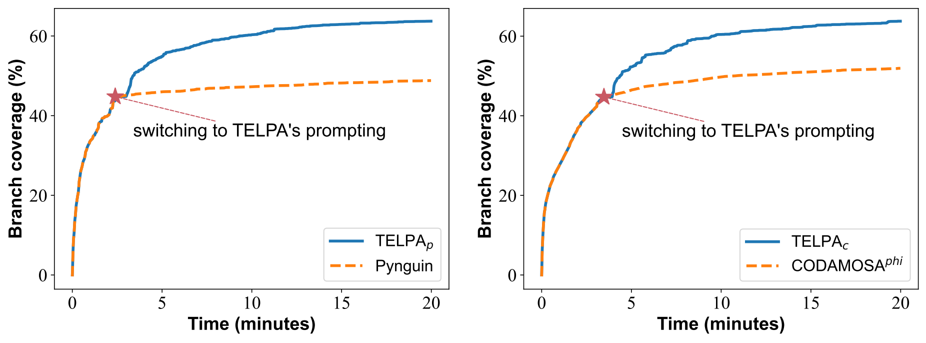
To avoid the influence of randomness, we repeated our experiments for 20 times considering the evaluation cost and used the average value of the final results. The experiments were conducted on a workstation with 128-core CPU, 504G memory, 4 NVIDIA A100 GPUs, and Ubuntu 20.04 OS.

We released our implementation and all experimental data at the project homepage [14] to facilitate replication, future research, and practical use.

4.2 RQ1: Effectiveness of TELPA

Table 2. Comparison among TELPA, CHATTESTER$^{\text{phi}}$, Pynguin and CODAMOSA$^{\text{phi}}$ in terms of branch coverage on different types of branches

Branch Type		Pynguin$_{bn}$	Pynguin	CHATTESTER$_p^{\text{phi}}$	TELPA$_p$	CODAMOSA$_{bn}^{\text{phi}}$	CODAMOSA$^{\text{phi}}$	CHATTESTER$_c^{\text{phi}}$	TELPA$_c$
Easy-to-cover		64.50%	72.51%	77.17%	78.84%	73.10%	78.18%	80.65%	82.32%
Hard-to-cover	Branches with complex dependencies	29.00%	31.16%	32.15%	39.10%	18.75%	20.17%	28.80%	34.90%
	Branches with complex objects	52.73%	57.03%	58.98%	63.67%	39.06%	42.97%	53.90%	58.59%
	All hard-to-cover branches (union of the above two types)	29.20%	31.62%	32.67%	39.53%	19.13%	20.58%	29.10%	35.23%



(a) Branch coverage trend of TELPA$_p$ and Pynguin (b) Branch coverage trend of TELPA$_c$ and CODAMOSA$^{\text{phi}}$

Fig. 8. Branch coverage trend

The goal of TELPA is to reach hard-to-cover branches, and this RQ verifies its ability in this regard. Table 1 presents the comparison results for TELPA$_p$ vs. Pynguin vs. CHATTESTER$_p^{\text{phi}}$ and TELPA$_c$ vs. CODAMOSA$^{\text{phi}}$ vs. CHATTESTER$_c^{\text{phi}}$ on each project in terms of branch coverage. To gain a better understanding of the performance of the preceding test generation step, we also show the branch coverage achieved by Pynguin and CODAMOSA$^{\text{phi}}$ right before switching to LLMs in "Pynguin$_{bn}$" and "CODAMOSA$_{bn}^{\text{phi}}$" columns[3], respectively. The last two rows present the average branch coverage and line coverage across all projects. Due to the space limit, we do not show the line coverage for each project but instead include these detailed results in our replication package [14]. To provide a clearer comparison of those techniques on hard-to-cover branches, Table 2 specifically shows the branch coverage achieved by each technique on different types of branches.

To better understand the effectiveness of TELPA$_p$ and TELPA$_c$, we also measured the time spent on running Pynguin and CODAMOSA$^{\text{phi}}$ before switching to LLMs. On average across all projects, Pynguin and CODAMOSA$^{\text{phi}}$ ran for 2.39 and 3.47 minutes, respectively, before switching. That is, TELPA is activated at an early stage of the entire testing process (given a time budget of 20 minutes) regardless of taking Pynguin or CODAMOSA$^{\text{phi}}$ as the preceding test generation tool. When comparing Pynguin$_{bn}$, Pynguin, CODAMOSA$_{bn}^{\text{phi}}$, and CODAMOSA$^{\text{phi}}$, from Table 1, we found that on average, Pynguin achieves a branch coverage of 43.09% within only 2.39 minutes, but just achieves 47.71% coverage after continuing running for 17.61 minutes. Similarly, CODAMOSA$^{\text{phi}}$ achieves 43.41% branch coverage within only 3.47 minutes, but in the end just achieves 50.87% coverage after continuing running for 16.53 minutes. From Table 2, this trend is consistent for hard-to-cover

[3]Here, "bn" represents "bottleneck".

branches as well, where Pynguin and CODAMOSA$^{\text{phi}}$ quickly encounter bottlenecks in achieving further coverage improvements on hard-to-cover branches within a short time period. Figure 8 shows the branch-coverage trend achieved by TELPA$_p$, Pynguin, TELPA$_c$, and CODAMOSA$^{\text{phi}}$. From this figure, Pynguin and CODAMOSA$^{\text{phi}}$ can quickly cover easily reachable branches, and then encounter the coverage bottleneck to some degree, while TELPA$_p$ and TELPA$_c$ continue to achieve new coverage. This highlights the challenge posed by hard-to-cover branches and underscores the motivation behind our work. This also confirms that the two-minute timeframe set for switching to LLMs is sufficient for Pynguin and CODAMOSA$^{\text{phi}}$, which can reflect their potential in covering hard-to-cover branches to a large extent, as extending their running time does not bring significant coverage improvement. Additionally, over the 20-minute period, TELPA's coverage continues to increase, though at a slower rate after the first 10 minutes, underscoring the importance of sufficient testing time to fully harness the testing capabilities. In contrast, Pynguin and CODAMOSA$^{\text{phi}}$ quickly plateau within the first few minutes, while TELPA steadily achieves higher coverage. Notably, TELPA also surpasses both Pynguin and CODAMOSA$^{\text{phi}}$ within the 10-minute time budget, demonstrating its effectiveness.

Takeaway I: The state-of-the-art test generation methods (Pynguin and CODAMOSA$^{\text{phi}}$) encounter the bottleneck in coverage improvement at an early stage of testing, possibly due to those hard-to-cover branches.

From Table 1, after switching to TELPA, the branch coverage can be significantly improved according to results of TELPA$_p$ and TELPA$_c$. Specifically, after reaching the testing time budget, TELPA$_p$ achieves an average improvement of 34.10% over Pynguin across all projects in terms of branch coverage, and TELPA$_c$ achieves an average improvement of 25.93% over CODAMOSA$^{\text{phi}}$. In terms of line coverage, TELPA$_p$ achieves an average improvement of 16.72% over Pynguin and TELPA$_c$ achieves an average improvement of 9.32% over CODAMOSA$^{\text{phi}}$. Additionally, TELPA$_p$ outperforms CODAMOSA$^{\text{phi}}$ in terms of both branch and line coverage. Specifically, TELPA$_p$ achieves an average branch coverage of 63.98% while CODAMOSA$^{\text{phi}}$ achieves an average branch coverage of 50.87%. This also demonstrates the superiority of the prompting method and the test generation process designed in TELPA. Regarding hard-to-cover branches, as shown in Table 2, both TELPA$_p$ and TELPA$_c$ outperform Pynguin and CODAMOSA$^{\text{phi}}$ on both scenarios, including those with complex dependencies and those with complex objects. For example, TELPA$_p$ covers 39.53% hard-to-cover branches, outperforming Pynguin, which covers 31.62% branches with an improvement of 25.02%. Similarly, TELPA$_c$ covers 35.23% hard-to-cover branches, compared to CODAMOSA$^{\text{phi}}$'s 20.58% branches, achieving an improvement of 71.18%. Furthermore, we conducted a paired Wilcoxon signed-rank test [56] at a significance level of 0.05 between TELPA$_p$/TELPA$_c$ and Pynguin/CODAMOSA$^{\text{phi}}$, confirming TELPA's statistical superiority over the two state-of-the-art methods with all p-values below 0.05.

Regardless of the preceding test generation tools (Pynguin or CODAMOSA$^{\text{phi}}$), TELPA can achieve similar effectiveness, i.e., TELPA$_p$ achieves similar branch coverage and line coverage to TELPA$_c$ on average across all projects. The results demonstrate the stable effectiveness of TELPA. We conducted a paired Wilcoxon signed-rank test [56] at a significance level of 0.05 between TELPA$_p$ and TELPA$_c$ in terms of both branch coverage and line coverage. Both p-values exceed 0.05, demonstrating no statistically significant difference between TELPA$_p$ and TELPA$_c$ in improving coverage.

Takeaway II: TELPA significantly outperforms the state-of-the-art Pynguin and CODAMOSA$^{\text{phi}}$ in improving both branch coverage and line coverage, regardless of the used preceding test generation tools.

Compared to Pynguin, all the LLM-based/LLM-enhanced techniques (including TELPA$_p$, TELPA$_c$, CHATTESTER$_p^{\text{phi}}$, CHATTESTER$_c^{\text{phi}}$, CODAMOSA$^{\text{phi}}$) achieve higher branch coverage and line coverage on average across all projects. This demonstrates the effectiveness of LLMs in improving test coverage. However, different prompting methods can largely affect the effectiveness of LLMs. By comparing TELPA$_p$/TELPA$_c$ with CHATTESTER$_p^{\text{phi}}$/CHATTESTER$_c^{\text{phi}}$, our designed prompting method specific to the challenge posed by hard-to-cover branches performs much better than the general prompting method employed for LLM-based test generation. Specifically, the average improvement of TELPA$_p$ over CHATTESTER$_p^{\text{phi}}$ is 20.19% in terms of branch coverage and 7.61% in terms of line coverage. Similarly, the average improvement of TELPA$_c$ over CHATTESTER$_c^{\text{phi}}$ is 22.02% in terms of branch coverage and 2.22% in terms of line coverage. The improvement is also consistent on hard-to-cover branches. Specifically, the improvement of TELPA$_p$ over CHATTESTER$_p^{\text{phi}}$ is 21.00% and the improvement of TELPA$_c$ over CHATTESTER$_c^{\text{phi}}$ is 21.06% on hard-to-cover branches. This implies the importance of designing task-specific prompting, i.e., extracting relevant information for improving the coverage of hard-to-cover branches in TELPA. We conducted a paired Wilcoxon signed-rank test [56] at a significance level of 0.05 and found that TELPA$_p$/TELPA$_c$ significantly outperforms CHATTESTER$_p^{\text{phi}}$/CHATTESTER$_c^{\text{phi}}$ by obtaining all p-values smaller than 0.05.

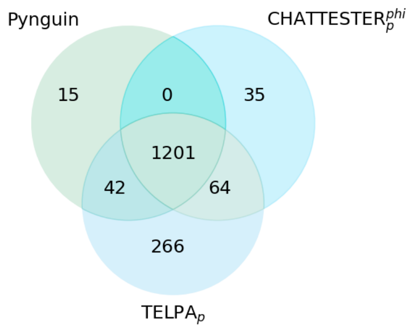
Takeaway III: Incorporating LLMs into test generation helps improve test coverage compared to traditional SBST. Designing task-specific prompting (the one in TELPA specific to hard-to-cover branches) can further improve the effectiveness of LLM-based test generation compared to general prompting.

To gain deeper insights into the specific branches covered by each method, we analyzed the overlap in covered branches across different methods, as visualized in Figure 9. The figure demonstrates that all studied techniques perform well on easy-to-cover branches. However, TELPA not only covers nearly all the hard-to-cover branches covered by other methods but also covers a significant number of unique hard-to-cover branches. This underscores TELPA's contribution to enhancing coverage of hard-to-cover branches and complementing existing methods.

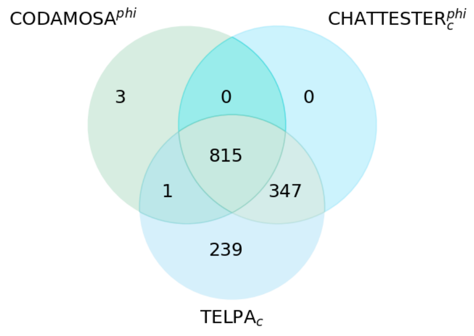
Takeaway IV: TELPA excels in covering hard-to-cover branches, including those missed by other methods, which highlights its potential to complement existing test generation techniques.

4.3 RQ2: Contribution of Each Component in TELPA

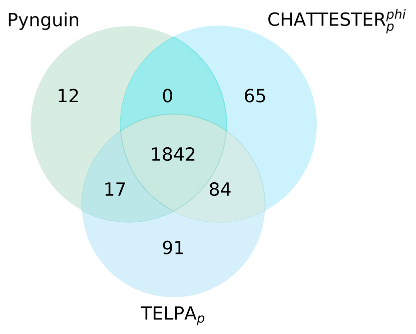
To investigate the contribution of each component in TELPA, we compared TELPA with a set of its variants (introduced in Section 4.1.3). Table 3 presents the comparison results in terms of average branch coverage and average line coverage across all the projects. As can be seen from Table 3, both TELPA$_p$ and TELPA$_c$ perform better than their corresponding variants. We further performed a paired Wilcoxon signed-rank test at the significance level of 0.05 to investigate whether



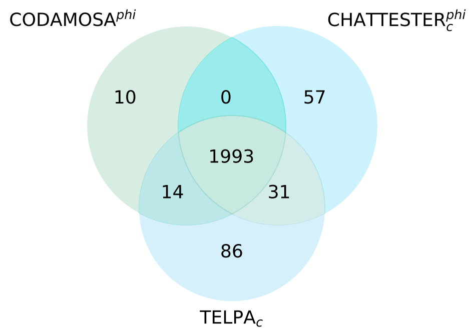
(a) Hard-to-cover branches covered by TELPA$_p$, CHATTESTER$_p^{\text{phi}}$, and Pynguin



(b) Hard-to-cover branches covered by TELPA$_c$, CHATTESTER$_c^{\text{phi}}$, and CODAMOSA$^{\text{phi}}$



(c) Easy-to-cover branches covered by TELPA$_p$, CHATTESTER$_p^{\text{phi}}$, and Pynguin



(d) Easy-to-cover branches covered by TELPA$_c$, CHATTESTER$_c^{\text{phi}}$, and CODAMOSA$^{\text{phi}}$

Fig. 9. Different branches covered by different techniques

Table 3. Comparison between TELPA and its variants in terms of branch and line coverage on average across all the projects

Cov.	**TELPA$^p_{nba}$**	**TELPA$^p_{nfa}$**	**TELPA$^p_{nce}$**	**TELPA$^p_{nf}$**	**TELPA$^p_{rce}$**	**TELPA$^p_{npf}$**	**TELPA$^p_{ncot}$**	**TELPA$_p$**
Branch Cov.	55.87%	58.74%	55.17%	58.34%	58.69%	52.80%	59.84%	**63.98%**
Line Cov.	70.67%	71.80%	70.46%	71.65%	71.07%	69.17%	72.29%	**75.54%**
Cov.	**TELPA$^c_{nba}$**	**TELPA$^c_{nfa}$**	**TELPA$^c_{nce}$**	**TELPA$^c_{nf}$**	**TELPA$^c_{rce}$**	**TELPA$^c_{npf}$**	**TELPA$^c_{ncot}$**	**TELPA$_c$**
Branch Cov.	56.68%	60.45%	56.12%	57.77%	57.32%	48.89%	60.12%	**64.06%**
Line Cov.	72.52%	73.78%	72.24%	72.30%	71.89%	66.88%	72.48%	**74.25%**

TELPA$_p$/TELPA$_c$ significantly outperforms each variant across all projects in terms of branch coverage and line coverage. All p-values are smaller than 0.05, demonstrating the statistically significant contribution of each component in TELPA, regardless of the used preceding test generation tools.

TELPA$_{npf}$ performs the worst among all variants. This is because the unfiltered invocation sequences can be excessively long and complex, posing significant challenges for LLMs to comprehend the complex sequence, which in turn reduces the effectiveness of test generation. The result

Table 4. Effectiveness of TELPA under different configurations in terms of average branch and line coverage

Cov.	Pynguin	CODAMOSA	TELPA$_{scratch}$	TELPA$^{p}_{ds}$	TELPA$_{p}$	TELPA$^{c}_{ds}$	TELPA$_{c}$
Branch Cov.	47.71%	50.87%	**52.75%**	60.34%	**63.98%**	59.79%	**64.06%**
Line Cov.	64.72%	67.92%	**68.26%**	73.96%	**75.54%**	74.08%	**74.25%**

demonstrates the contribution of sequence filtering, which could also balance the sequence content and computational burden well. In particular, except TELPA$_{npf}$, all other variants still outperform CHATTESTER even when some component is removed from TELPA. This demonstrates the effectiveness of using part of information specific to the challenges posed by hard-to-cover branches, compared to directly integrating unnecessary or insufficient context for prompting, which also reflects the negative influence of introducing irrelevant information on LLMs.

We also investigated whether our coverage-based counter-example sampling approach contributes to the overall effectiveness of TELPA by comparing with TELPA$_{rce}$ that randomly selects counter-examples. Table 3 shows that TELPA$_{rce}$ outperforms TELPA$_{nce}$ regardless of the preceding test generation tools. This result further confirms the importance of using counter-examples even if they are just selected randomly. Moreover, TELPA significantly outperforms TELPA$_{rce}$ regardless of the preceding test generation tools. For example, TELPA$_{p}$ improves TELPA$^{p}_{rce}$ by 9.01% in terms of average branch coverage achieved, while TELPA$_{c}$ improves TELPA$^{c}_{rce}$ by 11.76%. The results demonstrate the importance of a diverse set of counter-examples, which can be well achieved by our coverage-based sampling strategy. Additionally, TELPA$_{ncot}$ reduced branch coverage by 6.3% on average. The results empirically demonstrate that summarizing the functionality beforehand enhances test generation effectiveness.

Takeaway V: Each component in TELPA contributes to the overall effectiveness significantly, regardless of the preceding test generation tools used by TELPA.

4.4 RQ3: TELPA's Effectiveness with Different Configurations

TELPA uses existing tools for initial test generation, and utilizes an LLM to generate new tests. In this RQ, we examine how the adoption of existing tools and the choice of the LLM can impact the performance. We first investigated the effectiveness of TELPA$_{scratch}$ that removes the preceding test generation tool and activates TELPA from scratch. By comparing TELPA$_{scratch}$ with CODAMOSA^{phi} and Pynguin in Table 4, we can see that the TELPA$_{scratch}$ can improve CODAMOSA^{phi} and Pynguin by 3.70% and 10.56% in terms of average branch coverage and 0.50% and 5.47% in terms of average line coverage. The improvements are not huge, especially compared to the differences between TELPA$_{c}$ and CODAMOSA^{phi} or between TELPA$_{p}$ and Pynguin. This is expected, as the time needed for LLMs to generate each test cases is normally much more than that required by SBST techniques. TELPA$_{scratch}$ leverages LLMs for even easy-to-cover branches, preventing it from unleashing its full potential to reach hard-to-cover branches. Instead, CODAMOSA^{phi} alternately invokes Pynguin and LLMs for test generation, reducing the time needed by LLMs to a large extent, while Pynguin is also efficient as it does not rely on LLMs. Hence, within the same limited testing time, TELPA$_{scratch}$ does not achieve significant coverage improvement over CODAMOSA^{phi} and Pynguin. This justifies our design choice of adopting preceding test generation tools to balance effectiveness and efficiency.

Table 5. Effectiveness of TELPA using GPT-4 evaluated on 100 randomly sampled target methods

Cov.	**Pynguin**	**CODAMOSA**	**TELPA$_{ds}^{p}$**	**TELPA$_p$**	**TELPA$_{gpt}^{p}$**
Branch Cov.	26.86%	27.93%	35.68%	37.76%	38.43%
Line Cov.	42.13%	45.75%	53.66%	55.97%	56.08%

Table 6. Comparison among TELPA, CHATTESTER^{phi}, Pynguin and CODAMOSA^{phi} in terms of correctness

Metric	**Pynguin**	**CHATTESTER$_p^{phi}$**	**TELPA$_p$**	**CODAMOSA^{phi}**	**CHATTESTER$_c^{phi}$**	**TELPA$_c$**
Execution Pass Rate	97.44%	78.57%	62.48%	97.66%	78.82%	62.07%

By examining the data, we found that 603 methods were not covered by CODAMOSA^{phi} and Pynguin at all within the testing time budget. For these methods, TELPA$_{scratch}$ instead can effectively improve the coverage for 259 of them. On average across these methods, TELPA$_c$ achieves 21.86% branch coverage and 39.99% line coverage, while TELPA$_p$ achieves 17.14% branch coverage and 43.80% line coverage. The results further confirm the ability of TELPA in improving the coverage of hard-to-coverage branches.

Takeaway VI: Applying TELPA to easy-to-cover branches is not recommended due to the non-negligible time required by LLMs. Our designed usage scenario for TELPA balances effectiveness and efficiency well.

We then investigated the effectiveness of TELPA$_{ds}^{p}$ and TELPA$_{ds}^{c}$ that employ *DeepSeek-Coder-6.7B-instruct* [3] instead of *Phind-CodeLlama-34B-v2* [11]. From Table 4, we found although TELPA$_{ds}^{p}$/TELPA$_{ds}^{c}$ exhibits small performance reduction compared to TELPA$_p$/TELPA$_c$, the former still largely outperforms Pynguin/CODAMOSA^{phi}. For example, TELPA$_{ds}^{p}$/TELPA$_{ds}^{c}$ improves Pynguin/CODAMOSA^{phi} by 26.47% and 17.53% in terms of average branch coverage. To investigate the effectiveness of TELPA when using more advanced LLMs, we evaluated TELPA on GPT-4. We randomly sampled 100 target methods from the benchmark used in previous RQs for evaluation due to the high cost of invoking GPT-4's APIs. From Table 5, TELPA$_{gpt}^{p}$ exhibits similar effectiveness to TELPA$_p$ in terms of both average branch coverage and average line coverage on the same 100 sampled methods. This is attributed to the strength of Phind-CodeLlama-34B-v2, a powerful LLM that is specifically trained on code corpus and exhibits close performance to GPT-4 on many code-related tasks [6]. The results demonstrate the generalizability of TELPA under different LLMs to some degree.

Takeaway VII: Leveraging small-scale LLMs in TELPA can also largely improve test coverage, demonstrating the generalizability of TELPA. By incorporating more advanced LLMs, the effectiveness of TELPA is further improved.

4.5 RQ4: Correctness of Unit Tests Generated by TELPA

First, we found that all techniques achieve 100% syntax correctness, demonstrating their ability to generate syntactically valid tests without errors. Then, Table 6 compares the execution pass

Table 7. Comparison between TELPA and EvoSuite in terms of branch coverage on all branches

Project	**EvoSuite**	**TELPA**$_e$
Chart	48.64%	52.45%
Lang	85.00%	88.71%
Math	73.57%	77.51%
Time	43.49%	55.40%
Average Branch Cov.	62.68%	68.52%
Average Line Cov.	73.09%	76.85%

rate of unit tests generated by Pynguin, CODAMOSA^{phi}, CHATTESTER^{phi} and TELPA. From the table, both the LLM-based TELPA and CHATTESTER^{phi} exhibit lower execution pass rates than the search-based Pynguin, indicating the challenge of LLMs in generating semantically valid and executable tests. Note that although CODAMOSA^{phi} also relies on LLMs, it only leverages LLMs to generate seed tests for Pynguin, and most of the tests are ultimately generated by Pynguin, which results in comparable pass rates between CODAMOSA^{phi} and Pynguin. TELPA exhibit lower execution pass rates than CHATTESTER^{phi}. This is because TELPA specifically targets hard-to-cover branches, which often require constructing complex objects and resolving intricate inter-procedural dependencies. These tests are inherently more challenging to execute successfully due to their complexity, involving scenarios prone to runtime errors like incorrect object construction or missing dependencies. In contrast, CHATTESTER^{phi} generates simpler tests that are less effective at covering complex branches. For example, CHATTESTER^{phi}'s tests often cover straightforward branches with minimal object construction and fewer inter-procedural dependencies, resulting in higher pass rates but lower overall branch coverage. Specifically, TELPA$_p$ and TELPA$_c$ cover 21.00% and 21.06% more hard-to-cover branches than CHATTESTER$_p^{phi}$ and CHATTESTER$_c^{phi}$, respectively.

We further analyzed the distribution of errors in the generated tests. For TELPA, the tests predominantly fail due to AssertionError (32.0%) and AttributeError (33.5%). AssertionError occurs when the test fails an assertion, which is common in tests targeting complex branches where the expected behavior is harder to predict or verify. AttributeError often happens when the test attempts to access an attribute or method on an object that is not properly constructed or initialized, which is a common issue when dealing with complex object dependencies. Overall, these results confirm that while TELPA generates more complex tests that may have lower execution pass rates, these tests are often essential for achieving higher branch coverage, particularly for hard-to-cover branches involving complex objects and inter-procedural dependencies.

Takeaway VIII: TELPA generates more complex unit tests to target hard-to-cover branches, which, while resulting in slightly lower execution pass rates, enables it to achieve significantly higher branch coverage.

4.6 RQ5: Generalizability of TELPA

Table 8. Comparison between TELPA and EvoSuite in terms of branch coverage on hard-to-cover branches

Branch Type	EvoSuite	TELPA$_e$
Branches with complex dependencies	66.53%	77.42%
Branches with complex objects	56.79%	74.90%
All hard-to-cover branches (union of the above two types)	57.55%	74.59%

Table 7 presents the comparison results for TELPA$_e$ vs. EvoSuite on each project in terms of branch coverage on all branches. The last two rows present the average branch coverage and line coverage across all projects. Table 8 presents the comparison results on hard-to-cover branches. From Table 7 and Table 8, TELPA$_e$ achieves consistently higher branch coverage than EvoSuite. Specifically, on average, TELPA improves EvoSuite by 9.32% on all branches and 29.60% on hard-to-cover branches in terms of branch coverage. Note that the improvement on all branches is smaller than the improvement TELPA achieves over Pynguin. One possible explanation is that EvoSuite already incorporates more advanced static analysis techniques than Pynguin, potentially diminishing the additional benefits offered by TELPA. Moreover, other factors, such as differences in programming languages, testing strategies, and even tool implementation details, may also contribute to this discrepancy. Even though, they are still limited in handling intricate inter-procedural dependencies and dynamically computed values, which is confirmed by the larger improvement achieved by TELPA on hard-to-cover branches. The constant pool, for instance, may struggle when required values are not explicitly defined as constants but are derived from complex object construction logic. Similarly, testability transformations excels in cases where flags create coarse fitness landscapes in search-based approaches, making it difficult to find paths leading to high-fitness regions. However, when the program contains complex inter-dependencies or complex object constructions, testability transformation alone may not provide sufficient guidance. The transformation does not inherently handle the semantics of such complex dependencies, which are crucial for generating tests in challenging branches. The results reflects the fact that EvoSuite is already a highly-optimized tool, but TELPA still provides meaningful improvements in terms of code coverage, particularly for those much more challenging branches.

Takeaway IX: Even when compared to a highly optimized tool (i.e., EvoSuite) on a statically typed language (i.e., Java), TELPA consistently improves coverage, particularly on hard-to-cover branches, demonstrating its generalizability across different languages.

4.7 Threats to Validity

The threats mainly lie in parameter settings in TELPA , metrics and benchmarks used, and the randomness. By default, we set the timeframe to two minutes for switching from preceding test generation tools to our LLM-based test generation, the LLM temperature to 0, and the testing time budget to 20 minutes on each software module. Both the timeframe and the testing time budget are longer than the settings in the existing studies [35, 39], demonstrating higher sufficiency of reaching coverage bottlenecks and evaluating test effectiveness. The results shown in Figure 8 also confirm it. Lower temperature allows TELPA to receive more deterministic responses from LLMs, which is commonly employed in various tasks [24, 58]. In the future, we will investigate the influence of different settings on the effectiveness of TELPA. Regarding metrics, we have employed

Table 9. Comparison between TELPA$_{ct}$ and CHATTESTER$^{\text{phi}}$ in terms of branch coverage on all branches

Project	**CHATTESTER$^{\text{phi}}$**	**TELPA$_{ct}$**
MicroService	26.47%	64.71%
PATool	89.08%	90.78%
DAService	42.71%	53.65%
Average Branch Cov.	52.75%	69.71%
Average Line Cov.	59.43%	72.71%

the widely-used branch coverage and line coverage, which are also aligned with our goal (i.e., reaching hard-to-cover branches). We also analyzed the overhead of our static program analysis: on average, the time spent on each module is 1.5 seconds. This overhead is acceptable given its effectiveness. Moreover, this process is performed offline and it could be further accelerated via parallel executions.

Following the existing work [35], we did not investigate the bug detection effectiveness because test cases must include proper assertions to reveal bugs. However, TELPA focuses on improving branch coverage rather than generating assertions, and various assertion generation approaches [28, 52] can be applied to the test cases generated by TELPA. In the future, we will use more metrics to evaluate generated tests more sufficiently.

Regarding benchmarks, one potential threat to the validity of our results is the dynamically typed nature of Python, which presents unique challenges for automated test generation. In dynamically typed languages, type information is not explicitly available at compile time. This makes it difficult for automated test generation tools to infer correct types for object construction and method invocations, potentially leading to ineffective or invalid tests. Furthermore, automated test generation tools for Python, such as Pynguin and CODAMOSA, lack the advanced static analysis techniques used in more mature tools like EvoSuite. This may lead to ineffective or invalid tests and bias the evaluation results. To mitigate this threat, we adapted TELPA to statically typed languages (i.e., Java) and evaluate its effectiveness against EvoSuite on a widely-used Java benchmark Defects4J [33] in Section 4.6. The results demonstrate that TELPA consistently outperforms EvoSuite on Java projects, which helps mitigating this threat to some extent.

Furthermore, the projects used for evaluation may appear in the training data of the LLMs used in our study, which can cause the data leakage problem and bias the results [46]. Particularly, following the guidelines provided by previous work [46], we also adopted three internal Java projects provided by our industrial partner (Huawei, a global leader company in IT), which can largely reduce the threat from the potential data leakage of open-source projects. The three industrial projects have different functionalities, i.e., a program analysis toolkit, an online micro-service system, and a data analysis framework involving parallel computing and adaptation of design patterns. For ease of presentation, we refer to the three projects as PATool, Microservice and DAService in the following sections. Due to the company policy, we are unable to disclose further details. We constructed TELPA$_{ct}$, which leverages CHATTESTER$^{\text{phi}}$ for preceding test generation and compared it with CHATTESTER$^{\text{phi}}$. We did not leverage EvoSuite since the three industrial projects use Java 17, but EvoSuite supports up to Java 11.

Table 9 shows the comparison results for TELPA$_{ct}$ vs. CHATTESTER$^{\text{phi}}$ on each project in terms of branch coverage. From the table, TELPA$_{ct}$ achieves significantly higher branch coverage than CHATTESTER$^{\text{phi}}$ across all three industrial projects. On average, TELPA$_{ct}$ improves the branch coverage from 52.75% to 69.71% and the line coverage from 59.43% to 72.71%. The improvements

```
def format_body(self, content: str, input: str) ->str:
    if is_valid_expression(input):
        for p in self.enabled_plugins:
            content = p.format_body(content, input)
    return content

def is_valid_expression(input):
    pattern = r"^[a-zA-Z0-9._%+-]*_[a-zA-Z0-9._%+-
        ]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return re.match(pattern, input) is not None
```

Focal Method and Its Context

```
def test_case_0():
    try:
        str_0 = 'u:'
        list_0 = []
        formatting_0 = module_0.Formatting(list_0)
        str_1 = formatting_0.format_body(str_0,
str_0)
    except BaseException:
        pass
```

Test Generated by CODAMOSA

```
def test_case_0():
    list_0 = []
    formatting_0 = module_0.Formatting(list_0)
    str_0 = "example_user@gmail.com"
    str_1 = formatting_0.format_body(str_0, str_0)
```

Test Generated by TELPA

```
def test_case_0():
    str_0 = "UP/VJeH~5&Db"
    list_0 = []
    formatting_0 = module_0.Formatting(list_0)
    str_1 = formatting_0.format_body(str_0, str_0)
```

Test Generated by Pynguin

Fig. 10. Case for complex dependencies

of TELPA on these unseen projects highlight its generalizability and effectiveness in real-world scenarios, independent of potential data leakage from open-source training data.

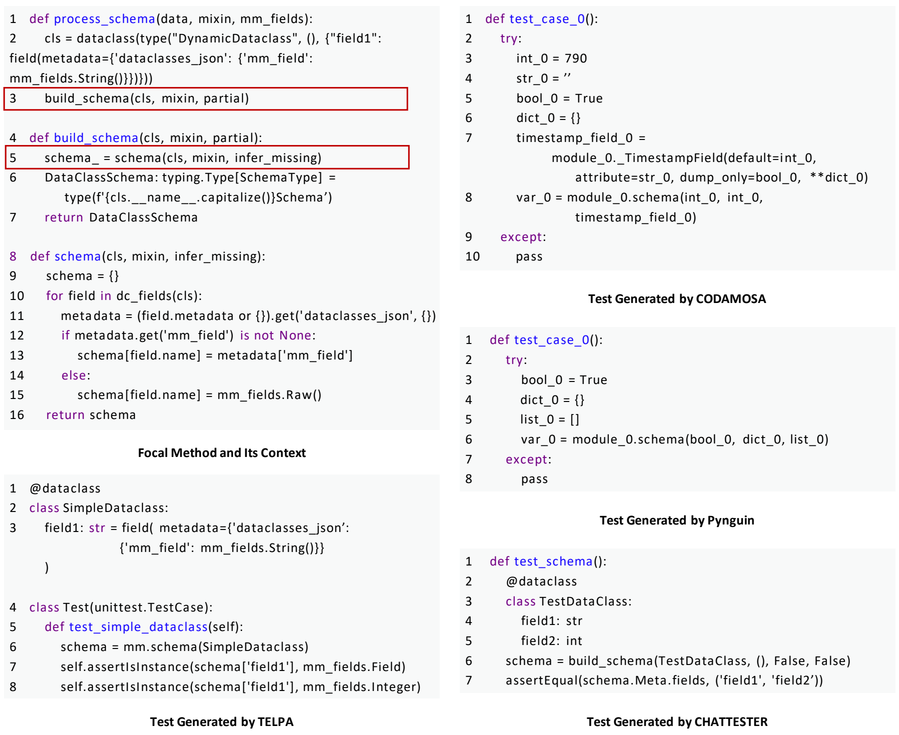
Regarding randomness, following the previous guideline [21], we repeated all the experiments for 20 times, including experiments on Java and those addressing data leakage threats. With 20 runs, TELPA exhibited a standard deviation of 1.8%, demonstrating consistency. Additionally, we conducted a statistical significance test using the Kruskal–Wallis H test [34]. The result showed no significant differences across runs ($p > 0.05$), indicating stable performance and mitigating this threat to some extent.

5 CASE STUDY

To illustrate the scenarios that TELPA specifically targets and is capable of covering, we present two cases that highlight its ability to address inter-procedural dependencies and complex object construction.

The first case is shown in Figure 10, which demonstrates how TELPA addresses the challenge posed by complex dependencies using branch dependency analysis. The target method `format_body` processes input content using enabled plugins only if the input matches a specific date pattern validated by `is_valid_expression`. To cover the branch where plugins are applied, the generated test must provide an input that satisfies `is_valid_expression`, which requires understanding the regex pattern. Baseline tools, such as Pynguin, generate random inputs (e.g., "UP/VJeH 5&Db") that fail the regex check, leaving the target branch uncovered. In contrast, TELPA leverages branch dependency analysis to identify the dependency between `format_body` and `is_valid_expression`. By including the source code of `is_valid_expression` in the prompt, TELPA guides the LLM to infer that valid inputs must adhere to the regex pattern, which requires an underscore in the account part of an email address (e.g., "example_user@gmail.com"). This allows TELPA to generate tests that invoke the plugin branch, while baseline tools fail due to their inability to infer valid inputs. Notably, the test generated by CHATTESTER is omitted here, as it differs from the Pynguin-generated test only in a single random value that still fails the regex check.

The second case is shown in Figure 11, which demonstrates how TELPA addresses the challenge posed by complex object construction using object construction analysis. In this case, the focal method is the `schema` function, which processes a dataclass (`cls`) and constructs a schema based



```
def process_schema(data, mixin, mm_fields):
    cls = dataclass(type("DynamicDataclass", (), {"field1":
field(metadata={'dataclasses_json': {'mm_field':
mm_fields.String()}})}))
    build_schema(cls, mixin, partial)

def build_schema(cls, mixin, partial):
    schema_ = schema(cls, mixin, infer_missing)
    DataClassSchema: typing.Type[SchemaType] =
        type(f'{cls.__name__.capitalize()}Schema')
    return DataClassSchema

def schema(cls, mixin, infer_missing):
    schema = {}
    for field in dc_fields(cls):
        metadata = (field.metadata or {}).get('dataclasses_json', {})
        if metadata.get('mm_field') is not None:
            schema[field.name] = metadata['mm_field']
        else:
            schema[field.name] = mm_fields.Raw()
    return schema
```

Focal Method and Its Context

```
@dataclass
class SimpleDataclass:
    field1: str = field( metadata={'dataclasses_json':
                  {'mm_field': mm_fields.String()}}
    )

class Test(unittest.TestCase):
    def test_simple_dataclass(self):
        schema = mm.schema(SimpleDataclass)
        self.assertIsInstance(schema['field1'], mm_fields.Field)
        self.assertIsInstance(schema['field1'], mm_fields.Integer)
```

Test Generated by TELPA

```
def test_case_0():
    try:
        int_0 = 790
        str_0 = ''
        bool_0 = True
        dict_0 = {}
        timestamp_field_0 =
              module_0._TimestampField(default=int_0,
                  attribute=str_0, dump_only=bool_0, **dict_0)
        var_0 = module_0.schema(int_0, int_0,
                  timestamp_field_0)
    except:
        pass
```

Test Generated by CODAMOSA

```
def test_case_0():
    try:
        bool_0 = True
        dict_0 = {}
        list_0 = []
        var_0 = module_0.schema(bool_0, dict_0, list_0)
    except:
        pass
```

Test Generated by Pynguin

```
def test_schema():
    @dataclass
    class TestDataClass:
        field1: str
        field2: int
    schema = build_schema(TestDataClass, (), False, False)
    assertEqual(schema.Meta.fields, ('field1', 'field2'))
```

Test Generated by CHATTESTER

Fig. 11. Case for complex object construction

on the metadata of its fields. Specifically, it checks if the `metadata` dictionary of each field contains a key `dataclasses_json` and uses the associated `mm_field` to populate the schema. Constructing a valid `cls` object with the correct metadata structure is critical for covering the branch where `metadata.get(mm_field)` is not None. Baseline tools struggle with this task: Pynguin and CODAMOSA generate invalid inputs (e.g., integers, booleans) that do not match the expected `cls` type, while CHATTESTER constructs a valid dataclass but omits the required `dataclasses_json` metadata. TELPA addresses this challenge by performing object construction analysis to trace the call chain `process_schema` → `build_schema` → `schema`. It extracts the logic from `process_schema`, where `cls` is dynamically instantiated with the required metadata, and incorporates this context into the prompt. As a result, TELPA generates a valid test case with the required metadata structure, ensuring coverage of the target branch.

These cases demonstrate how TELPA leverages program analysis to address challenges proposed by complex dependencies and object constructions. By understanding inter-procedural dependencies and the complex object construction logic, TELPA generates tests that can cover those challenging hard-to-cover branches.

6 DISCUSSION

Hallucination. Although TELPA can significantly improve test coverage compared to the state-of-the-art SBST and LLM-based techniques, it is still unable to reach all the hard-to-cover branches within the given testing time budget. Through our analysis, the major reason lies in the hallucination problem, namely LLMs confidently produce incorrect responses. For example, some generated tests by TELPA use non-existent parameters for invoking some methods. Currently, several approaches have been proposed to relieve the hallucination problem in various software engineering tasks, such as fine-tuning LLMs with PPO [48] or DPO [42] algorithms based on high-quality data. In the future, we can incorporate them to further improve the effectiveness of TELPA.

Orthogonality with Prompting Enhancement Methods. TELPA is a novel prompting technique for addressing the challenge of hard-to-cover branches in software testing, which improves the prompt contents specific to this task. Besides improving prompt contents, there are also some other prompting enhancement methods, including various CoT techniques (e.g., SCoT [36] and CCoT [26]) and post-processing techniques (e.g., self-repairing [25]). CoT techniques mainly elicit LLMs to produce intermediate reasoning steps based on the provided prompt for improving LLMs' effectiveness, while post-processing techniques mainly fix invalid code produced by LLMs based on the given prompt through static analysis or providing error messages to LLMs for self-repairing. They are actually orthogonal to TELPA (improving prompt contents), thus combining them may further improve test coverage.

Assertion Generation. TELPA focuses on improving branch coverage, a critical precursor to assertion generation. Assertions generated may reveal bugs only if the buggy code is covered by tests. While LLMs are capable of generating complete tests including both test prefixes and assertions [47, 60, 61], the quality of these assertions varies. That is, some assertions can effectively determine whether the focal method behaves as expected with specific inputs, while some others are relatively simplistic (e.g., just checking if an object is *None*). Some works [28, 52] have specifically focused on assertion generation as part of test creation, and these techniques could potentially enhance the quality of LLM-generated assertions. In the future, we plan to refine these assertions by integrating assertion generation techniques and evaluating their effectiveness using metrics like bug detection and mutation scores [32].

Trade-off in File-Level Analysis. TELPA currently limits its analysis of method invocation sequences to the file hosting the method under test. This design represents a trade-off between comprehensiveness and computational efficiency. The current file typically contains the most relevant usage scenarios for the target method, as it is both the definition location and often the location where it is the most frequently used [54, 61]. This local context provides sufficient information for generating effective tests in most cases. This approach of focusing on the local context is also widely used in many test generation approaches, such as CHATTESTER [61], ChatUniTest [25], and CODAMOSA [35]. While analyzing the entire codebase (e.g., looking at the other classes that might use the target class) could provide more comprehensive results, it would also introduce substantial computational costs. Additionally, cross-file analysis can be particularly challenging in dynamically typed languages like Python, where imports and method invocations are often resolved at runtime. While limiting the analysis to the current file works well in many cases, we acknowledge that it may miss some usage scenarios, particularly when the target method is primarily used in other files. In the future, we can explore project-wide usage patterns to further improve our approach.

7 RELATED WORK

We present traditional and deep-learning-based test generation techniques as related work. The most widely-studied traditional test generation technique category is SBST, including Pynguin [39] investigated in our work. Pynguin is a state-of-the-art technique for Python projects, and there are also SBST techniques for Java projects, such as Randoop [40] and EvoSuite [29]. Randoop performs random generation, while EvoSuite leverages more advanced search algorithms to guide test generation. They have also been improved by incorporating string literals from human-written tests [27] and object construction graphs [38].

Some other techniques, instead, employ symbolic execution to improve test coverage [22, 23, 31, 37, 50]. For example, Galeotti et al. [31] integrated dynamic symbolic execution (DSE) into the Genetic Algorithm (GA) adopted by EvoSuite. Their approach mutates tests, where primitive values can influence the fitness, to cover specific corner cases without sacrificing general coverage. Braione et al. [23] and Li et al. [37] transformed path conditions into optimization problems and solved them with SBST and machine learning techniques, respectively. Baluda et al. [22] combined symbolic execution and symbolic reachability analysis to improve the effectiveness by testing rare execution conditions and eliminating infeasible branches. However, it is well-known that symbolic execution has difficulties in dealing with complex data types and objects, thus unable to address our specific challenges.

Our approach TELPA utilizes program-analysis-enhanced prompting to exploit the code comprehension ability of LLMs and facilitate the effective test generation for hard-to-cover branches. The above-mentioned techniques, while not effective for this task, can still be incorporated into TELPA as preceding testing tools and benefit from our new prompting method.

There are also several deep-learning-based test generation techniques [16, 51]. Before prompting LLMs for test generation, ATHENATEST [51] built a transformer model based on a large dataset of target methods and tests for test generation. A3Test [16] first built a pre-trained language model for assertions in a self-supervised manner based on PLBART [15], and then fine-tuned it with test generation data. CAT-LM [43] trained a GPT-style LLM on Java and Python repositories, using a unique objective that maps source code to corresponding test files. Recently, some work leveraged LLMs for test generation [17, 25, 35, 47, 61], including CODAMOSA and CHATTESTER used in our study. For example, TestGen-LLM [17] utilized assured offline LLM-based software engineering [18] to integrate language models as a service within a comprehensive software engineering workflow, ultimately recommending unit tests with higher coverage. ChatUniTest [25] leveraged LLMs to generate tests with similar prompt contents used in CODAMOSA, but designed post-processing strategies to fix invalid tests generated by LLMs.

TELPA is also a LLM-based test generation technique, which improves prompt contents with the aid of program analysis (i.e., contextual information purification specific to the challenges of hard-to-cover branches). On one hand, those test-repair strategies can be combined with TELPA to further improve its performance. On the other hand, these deep-learning-based techniques can be also used as preceding test generation tools in TELPA for orthogonal integration.

There are also some empirical evaluations focusing on LLM-based test generation. Wang et al. [53] surveyed 102 recent papers on using LLMs for software testing, providing a comprehensive overview. Schäfer et al. [47] conducted an extensive study evaluating the effectiveness of using API signatures, documentation, and related information for prompting LLMs to generate unit tests. Tang et al. [49] presented a systematic comparison of test suites generated by ChatGPT and EvoSuite. Yang et al. [60] conducted the first empirical study to investigate the unit test generation effectiveness of open-source LLMs. Different from these empirical studies, TELPA is a novel LLM-based test generation technique with the aid of program analysis.

8 CONCLUSION

Existing test generation techniques face difficulties in covering branches involving specific complex objects and intricate inter-procedural dependencies. To address this issue, we propose a novel technique, named TELPA. TELPA leverages program analysis to assist LLMs in constructing complex objects and understanding code semantics, thus improving the test generation performance. Additionally, TELPA employs counter-example sampling and coverage-based feedback to guide

LLMs to effectively and efficiently generate tests. Our experiments on 27 open-source Python projects , four open-source Java Projects, and three internal Java projects demonstrate that TELPA significantly outperforms both the state-of-the-art SBST and LLM-based techniques. We released our implementation of TELPA and experimental data for replication and practical use. Please find them at **https://zenodo.org/records/15410112**.

ACKNOWLEDGMENTS

We thank the editor and the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China (Grant Nos. 62322208, 62232001), and Huawei Fund.

REFERENCES

[1] 2024. Codex shutdown. https://platform.openai.com/docs/deprecations.
[2] 2024. Coverage.py. https://coverage.readthedocs.io/en/7.6.0.
[3] 2024. Deepseek. https://github.com/deepseek-ai/DeepSeek-Coder.
[4] 2024. FastChat. https://github.com/lm-sys/FastChat.
[5] 2024. Hugging Face. https://huggingface.co.
[6] 2024. Hugging Face Big Code Model Leaderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.
[7] 2024. Huggingface LLM Learderboard. https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard.
[8] 2024. Jacoco. https://www.eclemma.org/jacoco.
[9] 2024. JavaParser. https://javaparser.org.
[10] 2024. Junit. https://junit.org/junit5.
[11] 2024. Phind-CodeLlama-34B-v2. https://huggingface.co/Phind/Phind-CodeLlama-34B-v2.
[12] 2024. PyTorch. http://pytorch.org.
[13] 2024. Transformers. https://github.com/huggingface/transformers.
[14] 2025. Project Homepage. https://zenodo.org/records/15410112.
[15] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.
[16] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3test: Assertion-augmented automated test case generation. *arXiv preprint arXiv:2302.10352* (2023).
[17] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.
[18] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering. *arXiv preprint arXiv:2402.04380* (2024).
[19] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*. Springer, 3–17.
[20] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.
[21] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd international conference on software engineering*. 1–10.
[22] Mauro Baluda, Giovanni Denaro, and Mauro Pezzè. 2016. Bidirectional Symbolic Analysis for Effective Branch Testing. *IEEE Transactions on Software Engineering* 42, 5 (2016), 403–426. https://doi.org/10.1109/TSE.2015.2490067
[23] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 90–101. https://doi.org/10.1145/3092703.3092715 https://doi.org/10.1145/3092703.3092715.
[24] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[25] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.

[26] Yew Ken Chia, Guizhen Chen, Luu Anh Tuan, Soujanya Poria, and Lidong Bing. 2023. Contrastive chain-of-thought prompting. *arXiv preprint arXiv:2311.09277* (2023).

[27] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. 2017. Saying 'hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 44–49.

[28] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K Lahiri. 2022. Toga: A neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering*. 2130–2141.

[29] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[30] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[31] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving search-based test suite generation with dynamic symbolic execution. In *2013 ieee 24th international symposium on software reliability engineering (issre)*. IEEE, 360–369.

[32] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[33] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*. 437–440.

[34] William H Kruskal and W Allen Wallis. 1952. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association* 47, 260 (1952), 583–621.

[35] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[36] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2023. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599* (2023).

[37] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic execution of complex program driven by machine learning based constraint solving. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 554–559. https://doi.org/10.1145/2970276.2970364 https://doi.org/10.1145/2970276.2970364.

[38] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.

[39] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[40] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[42] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems* 36 (2024).

[43] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. 2023. CAT-LM training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 409–420.

[44] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. *arXiv preprint arXiv:2402.00097* (2024).

[45] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*.

IEEE, 1646–1657.

[46] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 102–106.

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[48] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[49] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. 2024. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering* (2024).

[50] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *International conference on tests and proofs*. Springer, 134–153.

[51] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers. *CoRR* abs/2009.05617 (2020).

[52] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 54–64.

[53] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[54] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1258–1268.

[55] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[56] Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials* (2007), 1–3.

[57] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748* (2023).

[58] Ming Yan, Junjie Chen, Jie M Zhang, Xuejie Cao, Chen Yang, and Mark Harman. 2023. Coco: Testing code generation systems via concretized instructions. *arXiv preprint arXiv:2308.13319* (2023).

[59] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991* (2023).

[60] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. 2024. An Empirical Study of Unit Test Generation with Large Language Models. arXiv:2406.18181 [cs.SE] https://arxiv.org/abs/2406.18181

[61] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[62] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493* (2022).