Large Language Model Enabled Symbolic Execution for Automated Functional Analysis

Meixi Liu*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
liumeixi@nudt.edu.cn

Zhenbang Chen*†
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Xudong Wang*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
wangxudong@nudt.edu.cn

Ziran He*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
hzr@nudt.edu.cn

Jiachen Gu
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
gujiachen21@nudt.edu.cn

Minghui Chen*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
chenminghui20@nudt.edu.cn

Lei Wang*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
wanglei@nudt.edu.cn

Wei Dong*
College of Computer Science and Technology, National University of Defense Technology
Changsha, China
wdong@nudt.edu.cn

Abstract

Symbolic execution shows promise in automated program analysis by systematically exploring the path space of the program. However, due to the inherent limitations of program analysis techniques, symbolic execution cannot understand the program's functionality, which hinders its application in functional analysis and defect detection. This paper introduces FuSE, a method that enhances symbolic execution with Large Language Models (LLMs) to automatically annotate program paths with functional specifications. A program path is annotated with a natural language description of its functionality, a precondition, and a postcondition. Our method significantly reduces the effort required for functional testing and defect detection. Based on UnitTestBot, FuSE can successfully annotate functional specifications for C, C++, Java, and Python programs, with success rates of 98.2%, 95.52%, 87%, and 95.37%, respectively. In our user study with professional developers and testers, FuSE helps users identify 21.67% more functional defects and improves functional coverage by 15.83%, demonstrating its practical utility.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

Large Language Models, Symbolic Execution, Formal Specifications, Automated Functional Analysis, Defect Detection

ACM Reference Format:
Meixi Liu, Zhenbang Chen, Xudong Wang, Ziran He, Jiachen Gu, Minghui Chen, Lei Wang, and Wei Dong. 2026. Large Language Model Enabled Symbolic Execution for Automated Functional Analysis. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837524

1 Introduction

Symbolic execution [7, 32] is a precise program analysis technique based on constraint solving [12]. It can systematically explore a program's paths by symbolizing its inputs and executing the program symbolically. Over the past decades, symbolic execution has been successfully applied to diverse software engineering tasks, such as automated testing [11, 44], bug detection [58], and bounded verification [31]. The development and widespread adoption of powerful tools like KLEE [11] and Symbolic Java PathFinder [55] have significantly advanced the technique's impact.

For a program $\mathcal{P}$, symbolic execution explores its paths, collecting a path constraint (*PC*) for each path. By solving the path constraints, it determines the feasibility (i.e., the existence of inputs satisfying the path constraint) of these paths, thereby enabling automated testing, vulnerability detection, and property verification. However, its understanding of $\mathcal{P}$ is inherently limited to data and control dependence [29]. Consequently, it lacks comprehension of the program's *functional semantics*—both at the high level (the *specification problem* [20, 53]) and at the path level, where the functionality of individual paths remains opaque. As a result, without a formal functional specification, symbolic execution is restricted to checking non-functional properties (e.g., division-by-zero, out-of-bounds access) and cannot identify deviations from intended

*Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China
†Zhenbang Chen is the corresponding author.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837524

functionality. To conclude, empowering symbolic execution to reason about functionality remains a key challenge.

Traditionally, tasks like automated functional testing [59], functional bug detection [41], and program verification [10] depend on formal functional specifications (e.g., pre/postconditions [24, 49] or finite-state machines [21]), which are labor-intensive to create manually. Translating natural language requirements into such formalisms demands significant expert effort. Although techniques such as differential testing [46], metamorphic testing [8], and specification mining [5] aim to alleviate this burden, they are often limited to specific scenarios or suffer from imprecision and incompleteness. Consequently, while these approaches offer valuable insights, the general problem of automatically comprehending program functionality remains largely open. Addressing this problem motivates augmenting symbolic execution for *automated functional analysis*, so that symbolic paths can serve as structured units for functional analysis in addition to path exploration and test generation.

A key observation is that individual paths explored by symbolic execution often embody specific sub-functions of a program. However, symbolic execution lacks the capability to automatically map these paths to their corresponding functionalities. Consequently, understanding the functional meaning of paths currently relies on *manual inspection*, a process that is inherently *time-consuming and error-prone*. Therefore, *automatically linking path constraints to their functional semantics* emerges as a critical requirement, which would empower symbolic execution to *directly reason about program functionality*.

Such path-level functional annotations enable functional interpretation of symbolic execution. By linking paths to requirement-derived functionalities, FuSE supports three tasks: (1) *functional analysis*, which identifies the functionality implemented by each feasible path; (2) *functional coverage assessment*, which shows which functionalities are covered by feasible paths or generated tests and which remain uncovered; and (3) *defect detection*, which reveals discrepancies between path behavior and the intended specification.

Since automated functional analysis remains challenging, we present an approach that takes a first step in this direction by leveraging Large Language Models (LLMs) [61] to connect symbolic execution paths with requirement-derived functional specifications. Concretely, given a program $\mathcal{P}$ and a natural language specification $\mathcal{D}$, our approach uses an LLM to *automatically* link the path constraints of $\mathcal{P}$ to $\mathcal{D}$. To the best of our knowledge, this is the first work to establish such a connection in the context of symbolic execution. Our core insight is to exploit LLMs' comprehension of both code and natural language [62] to generate *formal specifications*, which then bridge the gap between path constraints and their functional descriptions.

Guided by this insight, we propose FuSE as the first step toward LLM-enabled symbolic execution for *automated functional analysis*. Given a program $\mathcal{P}$ and its natural language specification $\mathcal{D}$, FuSE first collects path constraints by symbolically executing $\mathcal{P}$. Then, it leverages an LLM to generate formal pre/postconditions from $\mathcal{D}$ and uses them to bridge the gap between the paths and their corresponding functional descriptions. By combining the core capabilities of symbolic execution (formal reasoning) and LLMs (natural language understanding), FuSE enables symbolic execution to perform functional reasoning.

```
1 public char s2g(int s) {
2   if (s >= 80)
3     return 'A';  // path 1
4   else if (s >= 60)
5     return 'B';  // path 2
6   else
7     return 'E';  // path 3
8 }
```

Figure 1: An example Java program.

The FuSE framework addresses two key technical challenges: (1) reliably linking path constraints to LLM-generated specifications, and (2) handling path constraints that cannot be linked to any specification. For (1), we guide the LLM to generate specifications in an *executable* form, such as code that checks the pre/postconditions. The link is then verified by executing this code, making FuSE easily adaptable to different programming languages. Regarding (2), FuSE guides the LLM to generate specifications for these *unlinked* constraints, thereby enhancing its overall effectiveness.

We implement FuSE for C, C++, Java, and Python by extending the state-of-the-art symbolic execution engine UnitTestBot [25, 27, 60]. We evaluate its effectiveness through large-scale experiments and a user study in real-world functional testing scenarios, which confirm its practical utility. Our contributions are:

- We propose FuSE, an LLM-based framework that automatically associates symbolic execution paths with their corresponding natural language functional descriptions, enabling path-level functional annotation and discrepancy analysis.
- We present a method that links symbolic execution paths to LLM-generated functional specifications via *executable* code, enabling language-agnostic and verifiable linkage as well as discrepancy analysis between path behavior and functional specifications. We further enhance FuSE by generating new specifications for path constraints that cannot be linked to any existing specification.
- Our evaluation on four programming languages shows that FuSE automatically associates functional specifications with *87%-98.2%* of paths. A user study further demonstrates that, with FuSE's assistance, users detect *21.67%* additional functional bugs and increase functional coverage by *15.83%*.

2 Background and Motivation

2.1 Symbolic Execution and Its Limitations

Symbolic execution analyzes programs by executing them with symbolic inputs, where each input represents a set of possible concrete values. It maintains symbolic expressions for variables influenced by these inputs, along with path constraints (*PC*), which are quantifier-free first-order logic constraints over the inputs that determine feasible execution paths. The *PC* is initialized to *true*. For non-branching statements, the symbolic expressions are updated. For conditional branches, the satisfiability of $PC \wedge \varphi$ and $PC \wedge \neg\varphi$ is checked to determine path feasibility, enabling systematic exploration. Here, φ denotes the branch condition. Moreover, concrete test inputs can be generated automatically by solving the *PC* of a path using a constraint solver.

Figure 1 shows a Java program that converts scores to grades, and Figure 2 exhibits its symbolic execution tree. Symbolic execution can collect all three path constraints, thereby covering all branches

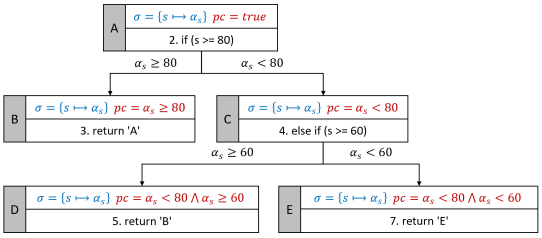

Figure 2: Symbolic execution tree of function s2g.

in the program. An example path constraint is:

$$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge ret = \text{'B'} \quad (1)$$

While symbolic execution is effective in achieving high path coverage and generating test inputs, it operates purely at the syntactic and semantic level of the program. It lacks the ability to interpret execution paths in terms of *high-level functional specifications* or user requirements. For example, it can determine that the second path returns 'B', but it cannot capture the intended high-level functionality of that path. This gap makes it challenging to automate tasks such as functional test oracle generation, requirements traceability, and detecting deviations between the implemented code and the intended functionality.

2.2 Problem Statement

How can we *automatically bridge the gap* between the concrete program paths explored by symbolic execution and the *informal functional specifications* described in natural language, to enable automated functional annotation, testing, and defect detection?

2.3 Overview of FuSE

To address the problem above, we propose FuSE, an LLM-enabled symbolic execution for *automated functional analysis*. FuSE augments traditional symbolic execution by using LLMs to generate candidate functional annotations and identify potential functional discrepancies. Figure 3 illustrates FuSE's workflow in four steps.

FuSE takes a program $\mathcal{P}$ and its functional description as inputs. First, it uses symbolic execution to generate path constraints for $\mathcal{P}$ (step ① in Figure 3). Second, an LLM is employed to derive candidate functional specifications from $\mathcal{P}$'s functional description (step ②). Specifically, the LLM decomposes $\mathcal{P}$'s overall functionality into candidate sub-functionalities, each consisting of a natural-language description and a pre/postcondition pair. Third, FuSE matches each pre/postcondition pair with the path constraints obtained from symbolic execution (step ③). A match is established when a path constraint logically entails both the precondition and the postcondition. If a path implies a precondition but not the corresponding postcondition, FuSE reports a potential discrepancy between the implementation and the specification as a defect candidate for user confirmation. Fourth, for paths that cannot be functionally annotated, the LLM is invoked to propose a new candidate functional specification in a step we refer to as Path Constraint Enhancement (step ④). Finally, all paths with validated functional annotations are automatically annotated by FuSE.

2.4 Motivating Example

We illustrate FuSE using the example program in Figure 1, with the following functional description:

Convert scores to grades. Valid inputs are 0-100; scores 80 and above receive 'A', scores 60 and above receive 'B', and scores below 60 receive 'E'.

1) Limitation of Traditional Symbolic Execution. As shown in Figure 4(a), symbolic execution explores all three paths, producing constraints such as:

$$s >= 60 \;\&\&\; s < 80.$$

However, it cannot derive that this path corresponds to "assigning grade 'B' to scores in [60, 80)".

2) How FuSE Bridges the Gap. FuSE operates in four steps:

(i) LLM Querying. It uses an LLM to decompose the natural-language requirement into four specification partitions (Figure 4(b)).

(ii) Matching Paths with Specifications. It matches these specifications to the path constraints. A path matches a sub-functionality only if its path constraint implies both the corresponding precondition and postcondition. For instance, the second path constraint implies both the pre- and the postcondition of the second partition. Consequently, the second path is annotated with the functional specification highlighted in blue in Figure 4(c). In contrast, path 1 does not match sub-functionality 1: its path constraint, $s \geq 80 \wedge ret = \text{'A'}$, does not imply the precondition $80 \leq s \leq 100$, since $s = 101$ satisfies the constraint but violates the precondition. The same holds for path 3. As a result, among these three path constraints and four functional specifications, only one constraint-specification pair is matched.

(iii) Path Constraint Enhancement. During the matching process, the LLM initially fails to produce functional specifications for the first and third paths. For such unmatched paths (e.g., the third), FuSE automatically infers and assigns a specification based on the path constraint. Consequently, the third path is annotated with the functional specification highlighted in blue in Figure 4(d). In this way, FuSE aims to annotate as many execution paths as possible with functional specifications.

(iv) Functional Specification Enhancement. Conversely, functional specifications that remain unmatched (e.g., Partition 4) indicate a potential gap between the requirements and the implementation. FuSE leverages this insight to support the automated identification of candidate missing functionality—for instance, by generating a test case targeting the potentially unimplemented behavior (e.g., "$s = -5$" with the expected outcome "$IllegalArgumentException$"). Consequently, the test case is annotated with the functional specification highlighted in blue in Figure 5. Executing this test can expose the discrepancy as a defect candidate. In this example, it is confirmed as a functional defect because the implementation fails to handle the invalid input required by the specification.

3) Summary. This example illustrates the key capabilities of FuSE, namely, to automatically connect low-level execution paths to high-level functional semantics, infer missing specifications, and reveal discrepancies between requirements and implementation. These capabilities collectively form the basis for the automated functional analysis we present in this work.

3 Method

Algorithm 1 details the workflow of FuSE. The algorithm takes a program $\mathcal{P}$ and its functional specification $\mathcal{D}$ as inputs, and outputs a set of path constraints annotated with functional specifications

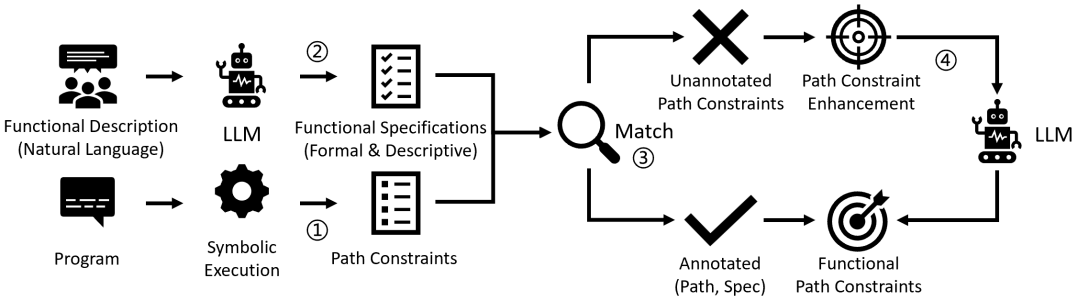

Figure 3: The design of FuSE framework.

① (a) The path constraints explored by symbolic execution for the example program

```
α_s ≥ 80 ∧ ret = 'A'          // pc of path 1
α_s < 80 ∧ α_s ≥ 60 ∧ ret = 'B'   // pc of path 2
α_s < 80 ∧ α_s < 60 ∧ ret = 'E'   // pc of path 3
```

② (b) The functional specifications generated by LLM for the example program

```
# Partition of sub-functionality 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80 && s <= 100
Postcondition: return == 'A'
# Partition of sub-functionality 2
Description: Valid inputs between 60 and 79 should return grade 'B'.
Precondition: s >= 60 && s < 80
Postcondition: result == 'B'
# Partition of sub-functionality 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s >= 0 && s < 60
Postcondition: return == 'E'
# Partition for sub-functionality 4
Description: Any input outside [0, 100] should be treated as invalid.
Precondition: s < 0 || s > 100
Postcondition: Throws an IllegalArgumentException.
```

③ (c) Path constraints with functional annotation

```
// pc of path 2
# Description: Valid inputs between 60 and 79 should return grade 'B'.
# Precondition: s >= 60 && s < 80
# Postcondition: result == 'B'
α_s < 80 ∧ α_s ≥ 60 ∧ ret = 'B'
```

④ (d) Enhancement for the path constraints without functional annotation

```
// pc of path 1
# Description: Inputs that are valid and ≥ 80 should return grade 'A'.
# Precondition: s >= 80
# Postcondition: result == 'A'
α_s ≥ 80 ∧ ret = 'A'
// pc of path 3
# Description: Inputs that are valid and less than 60 should return grade 'E'.
# Precondition: s < 60
# Postcondition: result == 'E'
α_s < 80 ∧ α_s < 60 ∧ ret = 'E'
```

Figure 4: Path-level functional analysis of the example program by FuSE.

```
# Partition of sub-functionality 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80 && s <= 100
Postcondition: return == 'A'
Input: s = 100, Output: return = 'A'
# Partition of sub-functionality 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s >= 0 && s < 60
Postcondition: return == 'E'
Input: s = 0, Output: return = 'E'
# Partition for sub-functionality 4
Description: Any input outside [0, 100] should be treated as invalid.
Precondition: s < 0 || s > 100
Postcondition: Throws an IllegalArgumentException.
Input: s = -5, Output: throw new IllegalArgumentException
```

Figure 5: Functional Specification Enhancement.

alongside a set of unmatched specifications. The process comprises three main steps.

Symbolic Execution and LLM Querying (Lines 1-3). First, FuSE employs SymbolicExecution($\mathcal{P}$) to generate a set of path constraints $\mathcal{PC}$ for program $\mathcal{P}$(Line 1). Each path constraint $pc \in \mathcal{PC}$ encodes the conditions to reach a specific path and its expected output. Next, it queries the LLM via QueryLLM$_1$($\mathcal{D}$) to decompose the functional description $\mathcal{D}$ into a set of functional specifications $\mathcal{FS}$(Line 2). Each specification $fs \in \mathcal{FS}$ is a triple (Pre, $Post$, $Description$), representing the precondition, postcondition, and a natural language description. Essentially, FuSE leverages the LLM's ability to decompose functionality and generate functional specifications, rather than to annotate all path constraints directly. Using LLMs to annotate every path would primarily capture what the implementation appears to do, whereas FuSE aims to determine whether the implementation conforms to the intended requirements. Therefore, FuSE generates path-agnostic functional specifications that define intended behavior independently of any execution path, which helps identify discrepancies between requirements and implementation. In contrast, generating specifications per path risks overfitting to potentially faulty implementations and thereby obscuring defects. The algorithm maintains four initially empty sets during matching: annotated path constraints $\mathcal{PC}_a$, matched path constraints $\mathcal{PC}_m$, covered specifications $\mathcal{FS}_c$, and the mapping between pc and fs, $\mathcal{R}$ (Line 3).

Matching Path Constraint with Functional Specification (Lines 4-12). FuSE then matches each path constraint $pc \in \mathcal{PC}$ with a functional specification $fs \in \mathcal{FS}$. A match is established if pc logically implies both the precondition ($pc \Rightarrow Pre$ at Line 7) and

Algorithm 1 LLM-Enabled Path-Level Functional Analysis

Input: $\mathcal{P}$ is a program, and $\mathcal{D}$ is the functional specification of $\mathcal{P}$ in natural language.
Output: $(\mathcal{R}, \mathcal{FS}_u)$, where $\mathcal{R}$ is a set of annotated path constraints and $\mathcal{FS}_u$ is a set of unmatched functional specifications.

1: $\mathcal{PC} \leftarrow \text{SymbolicExecution}(\mathcal{P})$
2: $\mathcal{FS} \leftarrow \text{QueryLLM}_1(\mathcal{D})$
3: $\mathcal{PC}_a, \mathcal{PC}_m, \mathcal{FS}_c, \mathcal{R} \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
4: **for** each $pc \in \mathcal{PC}$ **do**
5: **for** each $fs \in \mathcal{FS}$ **do**
6: ▷ $fs = (Pre, Post, Description)$
7: **if** $pc \Rightarrow Pre$ **then**
8: $\mathcal{R} \leftarrow \mathcal{R} \uplus \{pc \mapsto \{fs\}\}$
9: $\mathcal{PC}_a \leftarrow \mathcal{PC}_a \cup \{pc\}$
10: **if** $pc \Rightarrow Post$ **then**
11: $\mathcal{PC}_m \leftarrow \mathcal{PC}_m \cup \{pc\}$
12: $\mathcal{FS}_c \leftarrow \mathcal{FS}_c \cup \{fs\}$
13: **for** each $pc_f \in \mathcal{PC} \setminus \mathcal{PC}_a$ **do**
14: $fs_n \leftarrow \text{QueryLLM}_2(\mathcal{D}, pc_f)$
15: ▷ $fs_n = (Pre_n, Post_n, Description_n)$
16: **if** $pc_f \Rightarrow Pre_n \wedge pc_f \Rightarrow Post_n$ **then**
17: $\mathcal{R} \leftarrow \mathcal{R} \uplus \{pc_f \mapsto \{fs_n\}\}$
18: **return** $(\mathcal{R}, \mathcal{FS} \setminus \mathcal{FS}_c)$

the postcondition ($pc \Rightarrow Post$ at Line 10) of fs. Upon a successful match, pc is annotated with fs's functionality (Line 11), and fs is marked as covered (Line 12). If pc implies the precondition but not the postcondition, it is annotated with the corresponding specification and flagged for user inspection, indicating a potential discrepancy between the program and the LLM-derived specification (Lines 8-9). Here, $\uplus$ denotes adding fs to the annotation set of pc. The set $\mathcal{R}$ records all successful (pc, fs) mappings.

Path Constraint Enhancement (Lines 13-17). If a path constraint cannot be initially annotated, FuSE performs an enhancement step. Specifically, when the LLM fails to produce a precondition implied by the constraint, FuSE prompts the LLM to generate a new functional specification fs_n, which is then used to annotate the path. The annotation succeeds only if the path constraint implies both the new pre- and postcondition, and accepted annotations must pass executable validation. Thus, FuSE avoids directly using the LLM to annotate all paths, since unvalidated LLM-generated annotations may be unreliable.

Following this enhancement, the algorithm yields the final annotated path constraints $\mathcal{R}$ and the set of unmatched specifications $\mathcal{FS}_u = \mathcal{FS} \setminus \mathcal{FS}_c$ (Line 18). The LLM is re-queried if its output contains invalid syntax. Methodologically, Algorithm 1 does not restrict the search scope and serves as a general-purpose approach applicable to different levels of symbolic execution, such as unit-level and system-level analysis. It employs two LLM queries (Lines 2 and 14), each with a tailored prompt: one for high-level functional partition and specification generation, and the other for generating a functional specification for a single path constraint.

3.1 Matching Path Constraint with Functional Specification

The primary challenge is automatically translating natural language functional descriptions into precise, verifiable formal specifications. For the first LLM query, $\text{QueryLLM}_1(\mathcal{D})$ (Line 2 of Algorithm 1), we designed a structured prompt template based on prompt engineering research [37, 45, 66]. Our template consists of four parts: (1) a role description, (2) relevant information, (3) task requirements, and (4) the expected output format, as detailed in Figure 6.

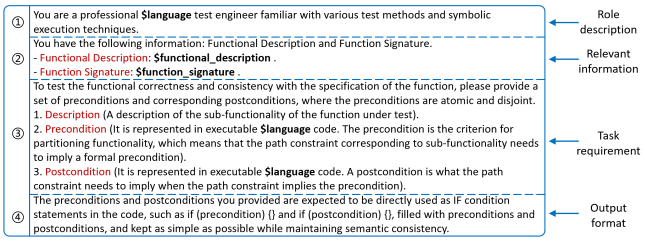

Figure 6: Prompt template for generating functional specifications from natural language descriptions.

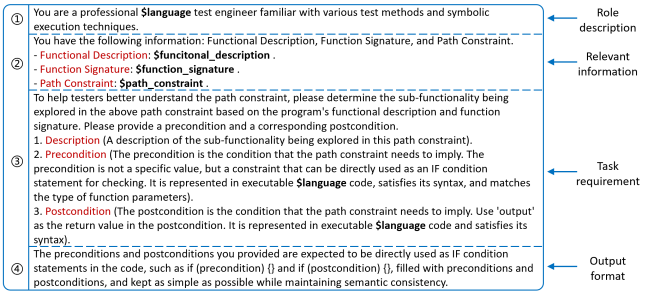

Figure 7: Prompt template for generating functional specifications for unannotated path constraints.

Key design choices include: providing only the function signature to ensure reasoning from the natural language description, and tailoring instructions to the target language (e.g., including explicit guidance on handling exceptions for Java). The prompt instructs the LLM to reason solely from the function signature (excluding the implementation code) and to generate path-agnostic, executable pre/postconditions. To enhance the generalizability of FuSE and avoid designing a language-specific intermediate constraint representation, we prompt the LLM to generate executable formal specifications directly in the target programming language. These executable checks enable direct validation of path constraints against the generated specifications.

Consider the example in Section 2, where the symbolic execution generates three path constraints (Figure 4(a)). Only one of them is successfully matched against the LLM-derived specifications (Figure 4(b)). This result directly underscores the necessity of our enhancement step.

3.2 Path Constraint Enhancement

A further challenge arises because LLMs, trained on finite corpora, may not cover all corner cases or exception-handling scenarios [23]. Consequently, some path constraints may initially lack a matching specification. To address this, we introduce a second LLM query $\text{QueryLLM}_2(\mathcal{D}, pc_f)$ (Line 14 of Algorithm 1). This query provides the LLM with additional context (i.e., the unannotated path constraint) and instructs it to generate a tailored functional specification on demand.

The prompt template for this query consists of four parts: (1) role definition, (2) details about the unannotated path constraint, including relevant functional specifications, the function signature, and the path constraint itself, (3) task requirements, and (4) the

Algorithm 2 Functional Specification Enhancement

Input: $\mathcal{D}$ is the functional specification of program $\mathcal{P}$ in natural language, and $\mathcal{FS}_u$ is a set of unmatched functional specifications.
Output: $\mathcal{M}$ is a set of annotated functional test cases.

1: $\mathcal{M} \leftarrow \emptyset$
2: **for** each $fs_f \in \mathcal{FS}_u$ **do**
3: ▷ $fs_f = (Pre_f, Post_f, Description_f)$
4: $uc_n \leftarrow \text{QueryLLM}_3(\mathcal{D}, fs_f)$
5: ▷ $uc_n = (I_n, O_n)$
6: **if** $I_n \vdash Pre_f \wedge (I_n, O_n) \vdash Post_f$ **then**
7: $\mathcal{M} \leftarrow \mathcal{M} \cup \{uc_n \mapsto fs_f\}$
8: **return** $\mathcal{M}$

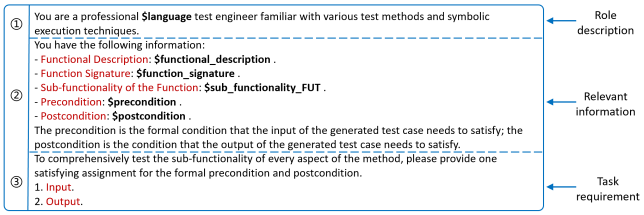

Figure 8: Prompt template for generating satisfying assignments for uncovered functional specifications.

required output format, as shown in Figure 7. The LLM is prompted to generate executable pre/postconditions, which are then used to validate the path constraint. If the LLM's output contains syntax errors or fails the validation check, the query is repeated, with a maximum of five attempts. If validation continues to fail after this threshold, the path constraint remains unannotated. For example, this enhancement successfully generates the functional specification in Figure 4(d) for the third path in Figure 4(a).

3.3 Functional Specification Enhancement

A symmetric challenge arises when some functional specifications are not covered by any path constraint, indicating potential gaps in test coverage. To address this, we leverage unmatched specifications for functional test generation, as outlined in Algorithm 2.

For each unmatched specification fs_f, we construct a prompt incorporating fs_f and its original natural language description $\mathcal{D}$. The LLM is then queried ($\text{QueryLLM}_3(\mathcal{D}, fs_f)$ at Line 4) to generate a satisfying assignment uc_n, consisting of an input I_n and an expected output O_n. The consistency of this assignment with fs_f is verified (Line 6) by checking that the input satisfies the precondition ($I_n \vdash Pre_f$) and that the input–output pair satisfies the postcondition ($(I_n, O_n) \vdash Post_f$). If consistent, the assignment is annotated as a functional test case (Line 7). The algorithm ultimately returns a set $\mathcal{M}$ of such annotated test cases. If the LLM's output contains syntax errors or fails the consistency check, the query is repeated up to a maximum of five attempts. If the failure persists, no test case is generated for that specification.

The prompt template consists of three components: 1) role definition and language-specific guidance, 2) the requirement specification, function signature, sub-functionality descriptions, and formal specifications, and 3) the required output content and format, as shown in Figure 8. For instance, this enhancement annotates the invalid-input sub-functionality with a new satisfying assignment in Figure 5, thereby enhancing the robustness of testing against the functional specifications.

3.4 Discussion

FuSE supports functional analysis by generating path-level annotations, including preconditions, postconditions, and natural-language descriptions. LLM-derived specifications and annotations are not treated as ground truth; instead, LLMs serve only as candidate generators, and their outputs are validated against symbolic execution results through pre/postcondition implication checks. As a result, FuSE reports validated annotations and defect candidates, while the final judgment of semantic correctness remains with developers or testers.

4 Experimental Design

We evaluate the effectiveness and utility of FuSE in *automated functional testing* scenarios. Our investigation addresses the following research questions:

RQ1: Effectiveness in Automated Generation. How well can FuSE *automatically generate* functional annotations for paths?

RQ2: Practical Utility. Does FuSE improve *automated functional testing* (quantified by improvements in functional coverage and the identification of confirmed functional defects)?

RQ3: Quality of Automated Annotations. To what extent do the annotations generated by FuSE meet the required quality?

4.1 Evaluation Subjects

We evaluate FuSE across two complementary categories of subjects.
Benchmarks. Our evaluation employs two established benchmarks: `HumanEval-X`[71] for code generation and `logic_bombs`[67] for symbolic execution. `HumanEval-X` provides natural language intents and reference solutions for each problem, enabling direct construction of functional testing and defect candidate identification tasks. In contrast, `logic_bombs` is designed to capture core challenges in symbolic execution rather than to provide functional descriptions. Consequently, we manually craft a functional description for each program in this benchmark to support our evaluation.
Real-world Programs. To further assess the practical applicability of FuSE, we select four popular, high-star-count open-source projects that contain functional descriptions in their source code comments. Our selection spans multiple languages and domains, including the numerical C library `FDLIBM` [1], the Python algorithm repository `The Algorithms` [2], the Java libraries `Apache Commons Math` [3] and `Apache Commons CLI` [4].

4.2 Evaluation Metrics

We evaluate FuSE in two phases: the *match* phase and the *enhancement* phase. Our metrics are derived from the following foundational sets.
Foundational Sets. For a target program $\mathcal{P}$, let (1) $\mathcal{PC}$ be the set of all path constraints and (2) $\mathcal{FS}$ be the set of formal specifications (pre/postcondition pairs) derived from the functional description. We partition $\mathcal{PC}$ and $\mathcal{FS}$ according to the following two phases.
Match Phase. This phase matches the paths $\mathcal{PC}$ against the formal specification $\mathcal{FS}$, where (1) $\mathcal{PC}_m$ denotes the path constraints fully matched by a specification (both pre/postconditions hold); (2) $\mathcal{PC}_a$ denotes the path constraints annotated by at least one specification's precondition (may not be fully matched); (3) $\mathcal{PC}_d$ denotes the path constraints annotated but not fully matched, indicating

potential implementation-specification discrepancies and reported as defect candidates for user confirmation; and (4) $\mathcal{FS}_m$ denotes the specifications covered by at least one fully matched path constraint. Notations are formalized as follows.

$$\mathcal{PC}_m = \{pc \in \mathcal{PC} \mid \exists fs \in \mathcal{FS}, (pc \Rightarrow fs_{Pre}) \wedge (pc \Rightarrow fs_{Post})\} \quad (2)$$

$$\mathcal{PC}_a = \{pc \in \mathcal{PC} \mid \exists fs \in \mathcal{FS}, pc \Rightarrow fs_{Pre}\} \quad (3)$$

$$\mathcal{PC}_d = \mathcal{PC}_a \setminus \mathcal{PC}_m \quad (4)$$

$$\mathcal{FS}_m = \{fs \in \mathcal{FS} \mid \exists pc \in \mathcal{PC}, (pc \Rightarrow fs_{Pre}) \wedge (pc \Rightarrow fs_{Post})\} \quad (5)$$

Enhancement Phase. In this phase, FuSE utilizes a set of *newly generated candidate specifications* $\mathcal{FS}_n$ and a set of *newly generated test cases* $\mathcal{UC}_n$ to improve matching.

$$\mathcal{PC}^e_m = \{pc \in \mathcal{PC} \setminus \mathcal{PC}_a \mid \exists fs \in \mathcal{FS}_n, (pc \Rightarrow fs_{Pre}) \wedge (pc \Rightarrow fs_{Post})\} \quad (6)$$

$$\mathcal{FS}^e_m = \{fs \in \mathcal{FS} \setminus \mathcal{FS}_m \mid \exists uc \in \mathcal{UC}_n, uc_I \vdash fs_{Pre} \wedge uc_O \vdash fs_{Post}\} \quad (7)$$

Here, $\mathcal{PC}^e_m$ is the set of previously unmatched paths that are newly matched by $\mathcal{FS}_n$ and $\mathcal{FS}^e_m$ is the set of previously uncovered specifications that are newly covered by $\mathcal{UC}_n$.

Metrics. Based on the sets defined above, we derive three core metric groups: the *Path Match Rate* ($\mathcal{R}_m$), the *Path Annotation Rate* ($\mathcal{R}_a$), and the *Functional Coverage* ($\mathcal{R}_c$). These metrics are reported in three variants: *initial* ($\mathcal{R}_*$), *enhancement* ($\mathcal{R}^e_*$), and *final* ($\mathcal{R}^f_*$).

1. Path Match Rate ($\mathcal{R}_m$): The proportion of path constraints that are fully matched to a specification.

$$\mathcal{R}_m = \frac{|\mathcal{PC}_m|}{|\mathcal{PC}|} \quad \mathcal{R}^e_m = \frac{|\mathcal{PC}^e_m|}{|\mathcal{PC}|} \quad \mathcal{R}^f_m = \frac{|\mathcal{PC}_m \cup \mathcal{PC}^e_m|}{|\mathcal{PC}|} \quad (8)$$

2. Path Annotation Rate ($\mathcal{R}_a$): The proportion of path constraints annotated with at least one specification.

$$\mathcal{R}_a = \frac{|\mathcal{PC}_a|}{|\mathcal{PC}|} \quad \mathcal{R}^f_a = \frac{|\mathcal{PC}_a \cup \mathcal{PC}^e_m|}{|\mathcal{PC}|} \quad (9)$$

3. Functional Coverage ($\mathcal{R}_c$): The proportion of formal specifications (derived from discretizing natural language requirements via an LLM) that are covered by at least one test case.

$$\mathcal{R}_c = \frac{|\mathcal{FS}_m|}{|\mathcal{FS}|} \quad \mathcal{R}^e_c = \frac{|\mathcal{FS}^e_m|}{|\mathcal{FS}|} \quad \mathcal{R}^f_c = \frac{|\mathcal{FS}_m \cup \mathcal{FS}^e_m|}{|\mathcal{FS}|} \quad (10)$$

4.3 Baseline and LLMs

Baseline. We adopt UnitTestBot [25–27, 60] as the baseline. For each subject, we generate two sets of test cases: the original UnitTestBot output and those annotated by FuSE.

LLMs. Since FuSE relies on an underlying LLM, we use two widely-used models: OpenAI's GPT-4o and Alibaba Cloud's Qwen3-Max. Our evaluation focuses on assessing FuSE rather than benchmarking the LLMs themselves.

4.4 Implementation and Environment

Implementation. Our implementation extends UnitTestBot [25–27, 60], a multi-language symbolic execution engine that generates unit test cases and therefore naturally supports function-level analysis. Building on this foundation, the core of FuSE is a prompt that instructs the LLM to generate *executable formal specifications* (pre/postconditions) directly in the target programming language as check code. For each function, FuSE matches the generated specifications against execution paths. Our approach bypasses the conventional pipeline (natural-language → intermediate representation → code), eliminating the semantic gaps inherent in translation.

Under-approximation Validation. To match path constraints with generated specifications in functional testing, we adopt an under-approximation strategy based on executable validation rather than logical implication checking. This approach is well-suited for functional testing, where the objective is to obtain correct test cases rather than to perform exhaustive verification. For each path, we verify that concrete solutions (i.e., values satisfying the constraint) also satisfy the specification's pre- and postconditions. Since these solutions satisfy the path constraint, any input derived from them is guaranteed to follow the intended execution path.

This under-approximation ensures that the generated test cases satisfy both the path constraints and the formal specifications, although it may miss some functional issues because logical implication is not guaranteed. We use this design for two reasons. First, such satisfaction is sufficient for behavior validation in our setting. Second, checking whether path constraints logically imply specifications is difficult in practice, as it would require a unified formal representation: path constraints are logical formulas, whereas specifications may involve library calls, helper functions, or other constructs that are hard to encode uniformly in logic across tools and languages. To evaluate the correctness of the matching, we manually inspected a random sample of matches.

Experimental Setup. All experiments were conducted on a server running Ubuntu 20.04, equipped with 128 3.1GHz cores and 256G memory. To mitigate the stochasticity of LLM generation, we conducted 5 independent runs and report the mean results.

4.5 Manual Inspection Design

To evaluate the quality of generated specifications and the impact of under-approximation (RQ3), we conducted a manual inspection on 8 randomly selected programs per language (C++, Java, and Python). Three inspectors with formal methods background independently assessed each specification along three dimensions:

- **Consistency:** The extent to which the path constraints logically imply the preconditions and postconditions. Specifically, we calculate the proportion of test cases (satisfying the path constraints) that also satisfy the pre/postconditions.
- **Correctness:** The semantic accuracy of each LLM-generated component (preconditions, postconditions, descriptions) in capturing the intended requirements. It is quantified as the ratio of correct component specifications.
- **Completeness:** This property holds for a program if the disjunction of all its generated preconditions covers the entire input domain. The ratio is the proportion of programs for which completeness holds.

4.6 User Study Design

To evaluate the practical utility (RQ2) and perceived quality of the generated annotations (RQ3), we conducted a controlled user study with software testers. The study was designed to compare the effectiveness of FuSE against a state-of-the-art baseline and to collect subjective ratings on annotation quality.

Participants. We recruited 40 participants (33 from academia and 7 from industry) with backgrounds in software development and testing. Their average experience was 3.88 years in software development and 1.3 years in software testing. The participants were

grouped by their primary programming language expertise: C/C++ (18 participants), Java (10), and Python (12). Each participant was assigned to evaluate programs written in their declared language of expertise, ensuring familiarity with the code under review.

Subject Programs and Assignment. We selected 18 programs (2 C, 4 C++, 6 Java, 6 Python) spanning a range of complexity. They were partitioned into two groups (Set A and B) of comparable complexity and defect distribution. The study used a within-subjects, counterbalanced design. Each participant evaluated 6 programs. The independent variable was the type of test information provided: (1) *Experimental Condition:* Programs evaluated with FuSE's functionally annotated test cases. (2) *Baseline Condition:* Programs evaluated with UnitTestBot's non-annotated test cases.

Procedure and Tasks. Each session lasted one hour. For each program, participants were given its source code, a functional specification, and a set of test cases (the format of which depended on the experimental condition). Their task was to identify as many functional defects as possible within a time limit.

To ensure comparability, both groups were required to demonstrate their understanding of program functionality in different forms. Participants in the FuSE group wrote new test cases based on the annotated tests, whereas those in the UnitTestBot group wrote functional descriptions based on the unannotated tests. This design allowed us to assess functional understanding while controlling task difficulty across conditions. Effectiveness was measured by the number of confirmed functional defects identified and the functional coverage achieved during the session. Table 1 details the presentation format and instructions for each condition.

Quality Assessment Survey. To complement the task-based evaluation, we administered a survey to assess the perceived quality of the functional annotations generated by FuSE. After completing the experimental tasks, participants rated the annotations on several dimensions using a 5-point Likert scale, with scores of 5, 4, 3, 2, and 1 indicating "extremely well", "very well", "moderately well", "slightly well", and "not well", respectively. The rated dimensions, explained to participants beforehand, were:

- **Correctness (FS_{cor}/FD_{cor}):** Whether the specification/ description accurately captures the intended requirement.
- **Accuracy (FS_{acc}/FD_{acc}):** The precision and lack of ambiguity in the specification/description.
- **Completeness (FS_{com}/FD_{com}):** whether the set of specifications/ descriptions collectively covers the complete functionality.
- **Usefulness (FA_{use}):** Whether the annotations reduce the effort required to understand test case functionality.
- **Necessity (FA_{nec}):** Whether each annotation is essential, as opposed to redundant (e.g., for trivial test cases).

This survey provided direct subjective feedback on the annotation quality, addressing RQ3.

5 Experimental Results

5.1 Results of RQ1: Effectiveness

This section evaluates the effectiveness of FuSE in automatically generating functional annotations for program paths. We present benchmark results reporting the initial match rate ($\mathcal{R}_m$), enhancement match rate ($\mathcal{R}_m^e$), and final annotation success rate ($\mathcal{R}_a^f$). Table 2 summarizes the core findings. Columns 1, 2, and 3 present the following information, respectively: (1) the programming language and the number of subject programs; (2) the number of symbolic execution paths explored, with the average per program ranging from 2.1 to 5.6 across languages; and (3) the large language model (LLM) used. (Note: Due to UnitTestBot's limitations in handling complex data types, the C/C++ evaluation uses a subset of the benchmark.)

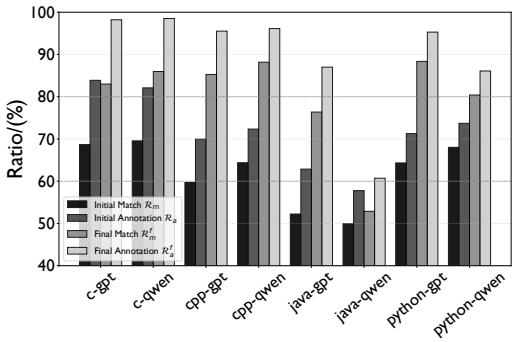

Figure 9: Match and Annotation Success Rates Across Languages and LLMs (Initial vs. Final).

Overall Effectiveness. FuSE achieves high final annotation rates ($\mathcal{R}_a^f$), ranging from 0.61 to 0.99 across the four programming languages. This demonstrates successful functional annotation for the majority of analyzed program paths. Both LLMs (GPT and Qwen) produce competitive results, with GPT consistently outperforming Qwen, particularly on Java programs (0.87 vs. 0.61).

Effectiveness of the Two-Phase Process. The decomposition into matching ($\mathcal{R}_m$) and enhancement ($\mathcal{R}_m^e$) phases elucidates each component's contribution. The initial match phase achieves substantial rates (0.50–0.70), confirming the core mechanism's efficacy. The enhancement phase then provides significant absolute improvements, increasing match rates by 0.03 to 0.26. This highlights the value of the enhancement step in recovering matches initially missed. For example, on C++ programs, enhancement improved GPT's match rate by 0.26, raising the final annotation rate to 0.96. Figure 9 visually illustrates this two-phase progression, showing the consistent gains from initial to final match rates across languages and LLMs.

Causes of Unmatched or Unannotated Paths. A path is counted as annotated once it receives an annotation, and as matched only if it passes executable validation after the two-stage process. Accordingly, the annotation rates (0.61–0.99) reflect annotated paths, whereas the match rates (0.53–0.88) reflect validated matches rather than raw LLM outputs. Paths that remain unmatched or unannotated may arise from several factors, including (1) underspecified requirements; (2) implementation-specific behavior, such as helper functions, defensive checks, or exception handling, that is not captured by the requirements; (3) specifications that do not pass executable validation in either the matching stage or the enhancement stage; (4) complex path constraints involving boundary cases, compound predicates, or library-dependent behavior; and (5) discrepancies between the implementation and the intended requirements.

Actionable Output: The Candidate Set $\mathcal{PC}_d$. Paths that satisfy a precondition but lack a matching postcondition are collected in the candidate set $\mathcal{PC}_d$ (Column 6 in Table 2). This set is a high-priority target for analysis. It may include: (1) potential specification

Table 1: User Study Design: Task Breakdown by Group

Group	Materials Provided	Participant Tasks
FuSE	1. Functional Specification 2. Program Source Code 3. Test Cases *with* Functional Annotations	1. **Functional Defect Identification:** Annotate code locations inconsistent with the specification. 2. **Coverage Assessment:** Annotate the degree to which test cases cover the specification. 3. **Comprehension & Extension:** Write new test cases covering the same functionality.
UnitTestBot	1. Functional Specification 2. Program Source Code 3. Test Cases *without* Functional Annotations	1. **Functional Defect Identification:** Annotate code locations inconsistent with the specification. 2. **Coverage Assessment:** Annotate the degree to which test cases cover the specification. 3. **Comprehension & Description:** Write a natural language description of each test case's functionality.

Table 2: Main experimental results on benchmarks ($\sharp PC$: number of pc, $\mathcal{R}_m$: initial match rate, $\mathcal{R}_a$: initial annotation rate, $\sharp \mathcal{PC}_d$: number of defect candidates, $\mathcal{R}_m^e$: enhanced match rate, $\mathcal{R}_m^f$: final match rate, $\mathcal{R}_a^f$: final annotation rate)

Lang. (No.)	$\sharp PC$	LLM	$\mathcal{R}_m$	$\mathcal{R}_a$	$\sharp \mathcal{PC}_d$	$\mathcal{R}_m^e$	$\mathcal{R}_m^f$	$\mathcal{R}_a^f$
C(32)	67	gpt	0.69	0.84	10	+0.14	0.83	0.98
		qwen	0.70	0.82	8	+0.16	0.86	0.99
C++(65)	201	gpt	0.60	0.70	21	+0.26	0.85	0.96
		qwen	0.64	0.72	16	+0.24	0.88	0.96
Java(164)	843	gpt	0.52	0.63	90	+0.24	0.76	0.87
		qwen	0.50	0.58	66	+0.03	0.53	0.61
Python(164)	932	gpt	0.64	0.71	65	+0.24	0.88	0.95
		qwen	0.68	0.74	53	+0.12	0.80	0.86

violations that can be confirmed as functional defects after user inspection, and (2) legal program behaviors not yet described by the generated formal specifications. For instance, when using GPT as the LLM backend, FuSE reported 90 such candidate paths in Java programs. This output significantly reduces manual inspection effort by directing developers to potential implementation-specification discrepancies and missing requirements—a key practical benefit.

Language-Specific Trends. The results reveal language-dependent patterns. FuSE performs best on C programs, achieving final annotation rates of 0.98 (GPT) and 0.99 (Qwen). Performance is strong on Python and C++, but lower on Java, where initial match rates are reduced and the performance gap between LLMs is most pronounced, indicating potential language- or benchmark-specific challenges.

RQ1 (Effectiveness). FuSE effectively automates functional annotation, achieving high final success rates (0.87–0.98) across four languages through its robust two-phase process. The method also produces a high-priority candidate set $\mathcal{PC}_d$ that efficiently guides inspection toward potential discrepancies for user confirmation.

5.2 Results of RQ2: Utility

To answer RQ2, we evaluate FuSE's utility in improving functional testing outcomes using two metrics: functional coverage and the number of functional defects.

We evaluate utility in two settings. First, on benchmarks and real-world programs, we measure the functional coverage achieved by FuSE and the number of functional defects it exposes. Functional coverage is computed automatically, while reported defects are manually inspected. Second, we conduct a controlled user study to assess whether these benefits transfer to human testers. There, we measure the functional coverage achieved and the number of functional defects detected with FuSE's assistance based on participants' actual testing results.

Table 3: Main experimental results on benchmarks ($\sharp FS$: number of fs, $\mathcal{R}_p$: specification parsing rate, $\mathcal{R}_c$: initial functional coverage, $\mathcal{R}_c^e$: enhanced functional coverage, $\mathcal{R}_c^f$: final functional coverage)

Lang. (No.)	$\sharp PC$	LLM	$\sharp FS$	$\mathcal{R}_p$	$\mathcal{R}_c$	$\mathcal{R}_c^e$	$\mathcal{R}_c^f$
C(32)	67	gpt	136	0.89	0.27	+0.60	0.87
		qwen	144	0.88	0.27	+0.55	0.82
C++(65)	201	gpt	344	0.85	0.36	+0.36	0.72
		qwen	369	0.89	0.36	+0.38	0.74
Java(164)	843	gpt	1028	0.93	0.42	+0.42	0.84
		qwen	850	0.85	0.43	+0.26	0.69
Python(164)	932	gpt	953	0.99	0.50	+0.43	0.93
		qwen	929	0.99	0.53	+0.40	0.93

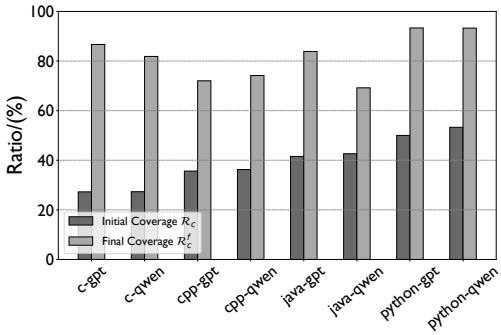

Figure 10: Functional Coverage Across Languages and LLMs (Initial vs. Final).

Functional Coverage on Benchmarks. The performance of FuSE is first quantified on benchmarks. Our specification generation phase produces a substantial number of formal specifications $\sharp FS$ that typically exceeds the scenarios covered by baseline unit tests $\sharp PC$, indicating broader testability. The subsequent matching and enhancement phases significantly improve the functional coverage, as measured by $\mathcal{R}_c^f$. As shown in Table 3, the final functional coverage ranges from 69% to 93% across the four languages. The improvement from the initial matching stage $\mathcal{R}_c$ to the final coverage $\mathcal{R}_c^f$ is substantial, as also visualized in Figure 10. This demonstrates that FuSE effectively generates and leverages functional specifications to achieve high coverage.

Defect Candidate and False Positive Analysis. Beyond coverage, the generated specifications support in-depth analysis of inconsistencies where test cases satisfy the precondition but do not satisfy the postcondition. We introduce a classification rule to interpret these defect candidates, manually labeling each as one of three root causes. We treat cases labeled as Functional Defect as confirmed defects, while cases labeled as LLM Hallucination or

Table 4: Root cause analysis of defect candidates reported by FuSE ($pc \Rightarrow Pre$ and $\neg(pc \Rightarrow Post)$). Root-cause categories are not mutually exclusive, since a defect candidate may involve multiple root causes. Percentages are calculated relative to the total number of unique defect candidates per language.

Root cause	C++	Java	Python
Functional Defect	10 (47.62%)	61 (67.03%)	31 (44.93%)
LLM Hallucination	13 (61.90%)	21 (23.08%)	36 (52.17%)
Ambiguous Requirement	2 (9.52%)	9 (9.89%)	11 (15.94%)
Total Unique Cases	21	91	69

Table 5: Results on real-world programs (cf. metrics in Table 2 and 3)

Lang.	LLM	$\mathcal{R}_m$	$\mathcal{R}_m^e$	$\mathcal{R}_m^f$	$\sharp PC_d$	$\mathcal{R}_c$	$\mathcal{R}_c^e$	$\mathcal{R}_c^f$
C	gpt	0.78	0.11	0.89	1	0.13	0.52	0.65
	qwen	0.33	0.11	0.44	5	0.06	0.27	0.33
Java	gpt	0.77	0.11	0.88	1	0.64	0.22	0.86
	qwen	0.75	0.09	0.84	1	0.50	0.25	0.75
Python	gpt	0.88	0.12	1.00	2	0.79	0.14	0.93
	qwen	0.65	0.27	0.92	2	0.61	0.06	0.67

Ambiguous Requirement are considered false positives from the perspective of defect reporting:

- **Functional Defect:** The program behavior violates a clear, unambiguous requirement and is confirmed by manual inspection.
- **LLM Hallucination:** The postcondition adds content not in the original requirement.
- **Ambiguous Requirement:** The inconsistency stems from vague or missing requirements.

Three authors independently inspected each case from the experimental round that yielded the highest number of inconsistencies, and all disagreements were resolved by consensus.

Table 4 reveals that a substantial portion of the reported defect candidates (up to 67.03% for Java) are confirmed as Functional Defects after manual inspection (e.g., unhandled edge cases). The remaining cases mainly stem from LLM Hallucinations (especially in C++/Python) or Ambiguous Requirements. Notably, Ambiguous Requirements result in underspecified prompts, which can lead LLMs to produce interpretations misaligned with the original intent. This misalignment renders final diagnosis a matter of careful adjudication, especially for cases with multiple root causes. This analysis clarifies the reliability boundary of FuSE: it reports validated candidates rather than final defect judgments. The results show that FuSE narrows the inspection space by producing actionable defect candidates that still require user confirmation.

Validation on Real-World Programs. To evaluate generalization, we applied FuSE to real-world programs. Table 5 shows that FuSE maintains high final functional coverage ($\mathcal{R}_c^f$) and matching success rate ($\mathcal{R}_m^f$), averaging 81% and 92%, respectively. This suggests that FuSE can be extended to more complex, practical scenarios.

Controlled User Study. A controlled, one-hour study with 40 experienced testers further shows the practical utility of FuSE. Participants were given testing tasks to compare FuSE against the state-of-the-art baseline, UnitTestBot. Table 6 presents the results. Testers using FuSE achieved significantly higher functional coverage (66.67% vs. 50.83%) and identified nearly twice as many confirmed functional defects (44.17% vs. 22.50%) compared to the baseline. This clear advantage in a realistic, time-constrained setting highlights the potential utility of the method for practitioners.

Table 6: Functionality coverage and functional defects confirmed by participants in the user study

Language	Functional Coverage			Confirmed Functional Defects		
	FuSE	UnitTestBot	Increased	FuSE	UnitTestBot	Increased
C&C++	77.78%	44.44%	**+33.33%**	44.44%	20.37%	**+24.07%**
Java	56.67%	70.00%	-13.33%	60.00%	40.00%	**+20.00%**
Python	58.33%	44.44%	**+13.89%**	30.56%	11.11%	**+19.44%**
Sum	66.67%	50.83%	**+15.83%**	44.17%	22.50%	**+21.67%**

Table 7: Quality Assessment via Manual Inspection

Metric	C++	Java	Python
Consistency	0.92	0.85	0.95
Correctness	0.84	0.90	0.90
Completeness	0.63	0.75	0.75

RQ2 (Utility). FuSE significantly enhances functional testing, reaching 69%-93% coverage on benchmarks and 81% on real-world programs. It reports defect candidates, of which 44%-67% are confirmed as real errors after manual inspection. In a user study, it helped testers achieve 15.8% higher coverage and find twice as many confirmed functional defects as the baseline.

5.3 Results of RQ3: Quality

To assess the quality of annotations automatically generated by FuSE, we evaluated them from two complementary perspectives: an internal manual inspection and an external user study. The results from both assessments confirm the high quality of the annotations.

Internal Manual Inspection. As summarized in Table 7, FuSE achieves high consistency (0.85-0.95) and correctness (0.84-0.90), indicating that the generated annotations are semantically accurate and internally consistent. The high consistency ratios further suggest that our under-approximation is effective in practice. Although executable validation does not guarantee logical implication, manual inspection shows that in 0.85-0.95 of the inspected cases, the path constraints of the generated test cases also logically imply the corresponding formal specifications. Completeness is lower (0.63-0.75), leaving room for improvement in covering the full input domain. Overall, FuSE produces reliable and logically sound specifications with reasonable input coverage.

We find that the consistency of specification-matched annotations (0.93 for C++, 0.85 for Java, and 0.93 for Python) is very close to the overall consistency (0.92, 0.85, and 0.95, respectively), suggesting that PC Enhancement does not introduce an obvious quality gap in the current evaluation. However, PC-enhanced annotations have a different risk profile: they rely more directly on the concrete implementation and may generate implementation-specific or overly strong conditions, such as `input == x`. This risk is particularly relevant for exceptional paths, where inferred exception conditions may become too narrow or tied to implementation-specific checks.

External User Study. To evaluate the annotation quality from a practitioner's perspective, we conducted a user study with 40 experienced software testers. Participants were asked to rate various attributes of the generated annotations on a 5-point Likert scale (1=Not Well, 5=Extremely Well). The rated attributes include: Formal Specifications: Correctness FS_{cor}, Accuracy FS_{acc}, and Completeness FS_{com}. Functional Descriptions: Correctness FD_{cor}, Accuracy FD_{acc}, and Completeness FD_{com}. Functional Annotations: Usability FA_{use}, Necessity FA_{nec}.

Table 8: User Study Ratings (5-point Likert Scale)

Language	FS_{cor}	FS_{acc}	FS_{com}	FD_{cor}	FD_{acc}	FD_{com}	FA_{use}	FA_{nec}
C&C++	4.39	**4.50**	3.94	**4.89**	**4.56**	**4.17**	**4.11**	**4.11**
Java	3.90	3.70	**4.10**	4.20	4.10	3.90	3.60	3.50
Python	**4.42**	4.25	4.08	4.42	4.33	4.08	4.00	4.08
Average	4.28	4.22	4.03	4.58	4.38	4.08	3.95	3.95

The user study results are presented in Table 8. The average ratings across all attributes range from 3.95 to 4.58, corresponding to a perception between "Very Well" and "Extremely Well". These consistently high ratings provide empirical evidence that practitioners perceive the automatically generated annotations as highly usable, accurate, and valuable in real-world scenarios.

RQ3 (Quality). FuSE generates high-quality annotations. Manual inspection shows high consistency (up to 0.93) and correctness (up to 0.90). A user study further validates high practical acceptance, with average ratings of 3.95-4.58 (5-point scale). FuSE produces practitioner-endorsed annotations.

5.4 Threats to Validity

Threats to validity are categorized as internal and external. A core internal threat stems from our reliance on LLM-derived specifications, which depend on the quality and ambiguity of the input requirements. Ambiguous or underspecified requirements can lead to inconsistent or incorrect LLM outputs, which subsequently affect the functional annotations and defect detection. FuSE mitigates this threat by treating LLM outputs as candidate specifications rather than ground truth and using executable checks to determine whether feasible symbolic execution paths imply the specifications. It therefore reports defect candidates for user confirmation rather than final correctness judgments.

The main external threat is scalability to larger, more complex systems. FuSE inherits the scalability limits of symbolic execution, as its cost grows with the number and complexity of paths. However, it does not require modifying the symbolic executor or exhaustively exploring all paths, and it can be applied modularly or with bounded path exploration. To mitigate this threat, we constructed our evaluation suite from established benchmarks and real-world programs across domains to improve the validity and breadth of the study. Nonetheless, our evaluation is based on a specific set of programs and languages; applying the approach to more complex systems or other programming languages requires further investigation.

6 Related Work

LLMs for Symbolic Execution. Existing work primarily uses LLMs to enhance symbolic execution. For example, LLMSym [30] guides test input generation, LangSym [68] mitigates path explosion via context-aware prompting, and LLM-Sym [63] assists SMT solvers in resolving path constraints. SymPrompt [56] decomposes test generation into staged prompts per path, while AutoBug [38] introduces "LLM-powered symbolic execution" to interpret and steer the process. Together, these approaches employ LLMs to produce tests or guide execution. In contrast, FuSE uses LLMs as requirement-aware specification annotators: it explicitly connects requirement-derived functional specifications with explored symbolic paths, shifting the focus from guiding path exploration to interpreting paths in terms of intended functionality.

LLMs for Test Generation. Research explores using LLMs for test generation. Approaches such as intention-guided IntUT [54] and path-sensitive JUnitGenie [39] directly synthesize unit tests. Others address challenges: SlicePromptTest4J [64] improves coverage via decomposition; CodaMosa [36] overcomes stagnation in search-based testing; UniTSyn [19] enhances accuracy with static analysis; and TestPilot [57] auto-generates JavaScript unit tests. UTGen [13] further integrates search-based testing with LLMs to produce more comprehensible unit tests. Collectively, these works focus on generating executable test cases. In contrast, FuSE does not use LLMs primarily to produce test data; instead, it attaches requirement-derived pre-/postconditions to symbolic paths, making each path a unit for functional annotation, functional coverage assessment, and discrepancy analysis.

Automated and LLM-based Specification Generation. Research on specification generation falls into two categories. The first derives specifications from code, as in tools such as Atlas [65], TOGA [14], EvoSpex [51], SpecFuzzer [50], SpecGen [43], and SpecEval [42]. The second generates specifications from language artifacts. This includes works that translate documentation (e.g., Toradocu [18], Jdoctor [9], C2S [70]), use LLMs to produce postconditions (e.g., nl2postcond [15]), or employ documentation (e.g., Javadoc comments) to generate test oracles (e.g., Doc2OracLL [22] and Tratto [52]). Unlike prior work that treats specification generation as the end result, FuSE uses LLMs to decompose natural language requirements into candidate pre-/postconditions and align them with symbolic execution paths, enabling path-level functional analysis.

Inferred and Executable Specifications. Specification mining and invariant inference infer implementation-level properties from executions, traces, or program states [6, 16, 17, 69]. Since these properties are derived from observed behavior, they are faithful to the implementation rather than to the intended requirements. As a result, they do not determine whether a program realizes requirement-level functionality. Executable specification languages and contract systems, such as Alloy [28], TLA+ [33, 34], JML [35], and design-by-contract frameworks [47, 48], enable property checking and behavioral validation but require formal specifications in advance. In contrast, FuSE derives candidate pre-/postconditions from natural language requirements and aligns them with symbolic execution paths, enabling path-level functional annotation and discrepancy analysis without pre-existing formal specifications.

7 Conclusion

This paper presents FuSE, an LLM-enabled symbolic execution for automated functional analysis, which bridges the gap between informal requirements and formal specifications by leveraging an LLM to decompose natural language descriptions into preconditions and postconditions. These conditions are then linked to test inputs and outputs, enabling functional annotation of program paths. We implement the framework for C/C++, Java, and Python based on UnitTestBot. Our evaluation shows that FuSE significantly enhances the functional analysis and practical utility of symbolic execution, as evidenced by extensive experiments and qualitative user feedback.

8 Data-Availability Statement

Our artifact is available on Zenodo [40].

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429 and U2341212). We also thank the anonymous reviewers for their valuable feedback.

References

[1] 2025. . https://github.com/freemint/fdlibm
[2] 2025. . https://github.com/TheAlgorithms
[3] 2025. . https://github.com/apache/commons-math
[4] 2025. . https://github.com/apache/commons-cli
[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. *SIGPLAN Not.* 37, 1 (Jan. 2002), 4–16. doi:10.1145/565816.503275
[6] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. Association for Computing Machinery, New York, NY, USA, 4–16. doi:10.1145/503272.503275
[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. doi:10.1145/3182657
[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785
[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 242–253. doi:10.1145/3213846.3213872
[10] J. C. Browne. 1996. Object-oriented development of real-time systems: verification of functionality and performance. *SIGPLAN OOPS Mess.* 7, 1 (Jan. 1996), 59–62. doi:10.1145/227986.227996
[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.
[12] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. doi:10.1145/2408776.2408795
[13] Amirhossein Deljouyi, Roham Koohestani, Maliheh Izadi, and Andy Zaidman. 2025. *Leveraging Large Language Models for Enhancing the Understandability of Generated Unit Tests*. IEEE Press, 1449–1461. https://doi.org/10.1109/ICSE55347.2025.00032
[14] 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* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. doi:10.1145/3510003.3510141
[15] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE, Article 84 (jul 2024), 24 pages. doi:10.1145/3660791
[16] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123. doi:10.1109/32.908957
[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. Special issue on Experimental Software and Toolkits. doi:10.1016/j.scico.2007.01.015
[18] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 213–224. doi:10.1145/2931037.2931061
[19] Yifeng He, Jiabo Huang, Yuyang Rong, Yiwen Guo, Ethan Wang, and Hao Chen. 2024. UniTSyn: A Large-Scale Dataset Capable of Enhancing the Prowess of Large Language Models for Program Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1061–1072. doi:10.1145/3650212.3680342
[20] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using formal specifications to support testing. *ACM Comput. Surv.* 41, 2, Article 9 (Feb. 2009), 76 pages. doi:10.1145/1459352.1459354
[21] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
[22] Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE084 (June 2025), 22 pages. doi:10.1145/3729354
[23] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2024). Just Accepted. doi:10.1145/3695988
[24] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems* (2 ed.). Cambridge University Press.
[25] Dmitry Ivanov, Alexey Babushkin, Saveliy Grigoryev, Pavel Iatchenii, Vladislav Kalugin, Egor Kichin, Egor Kulikov, Aleksandr Misonizhnik, Dmitry Mordvinov, Sergey Morozov, Olga Naumenko, Alexey Pleshakov, Pavel Ponomarev, Svetlana Shmidt, Alexey Utkin, Vadim Volodin, and Arseniy Volynets. 2023. UnitTestBot: Automated Unit Test Generation for C Code in Integrated Development Environments. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 380–384. doi:10.1109/ICSE-Companion58688.2023.00107
[26] Dmitry Ivanov, Alexey Menshutin, Denis Fokin, Yury Kamenev, Sergey Pospelov, Egor Kulikov, and Nikita Stroganov. 2023. UTBot Java at the SBST2022 tool competition. In *Proceedings of the 15th Workshop on Search-Based Software Testing* (Pittsburgh, Pennsylvania) *(SBST '22)*. Association for Computing Machinery, New York, NY, USA, 39–40. doi:10.1145/3526072.3527529
[27] Dmitry Ivanov, Alexey Menshutin, Maxim Pelevin, Daniil Stepanov, Denis Fokin, Yury Kamenev, Egor Kulikov, Artemiy Kononov, Sergey Pospelov, Ivan Volkov, Alena Lisevych, Timur Yuldashev, Nikita Stroganov, and Andrey Tarbeev. 2023. UTBot at the SBFT 2023 Java Tool Competition. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. 68–69. doi:10.1109/SBFT59156.2023.00019
[28] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (April 2002), 256–290. doi:10.1145/505145.505149
[29] Daniel Jackson and Martin Rinard. 2000. Software analysis: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering* (Limerick, Ireland) *(ICSE '00)*. Association for Computing Machinery, New York, NY, USA, 133–145. doi:10.1145/336512.336545
[30] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1408–1420. doi:10.1145/3691620.3695513
[31] Sarfraz Khurshid, Corina S. PĂsĂreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, Hubert Garavel and John Hatcliff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 553–568.
[32] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. doi:10.1145/360248.360252
[33] Leslie Lamport. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923. doi:10.1145/177492.177726
[34] Leslie Lamport. 2002. *Specifying systems*. Vol. 388. Addison-Wesley Boston.
[35] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38. doi:10.1145/1127878.1127884
[36] 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)*. 919–931. doi:10.1109/ICSE48619.2023.00085
[37] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2024. Structured Chain-of-Thought Prompting for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (Aug. 2024). Just Accepted. doi:10.1145/3690635
[38] Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 385 (Oct. 2025), 29 pages. doi:10.1145/3763163
[39] Dianshu Liao, Xin Yin, Shidong Pan, Chao Ni, Zhenchang Xing, and Xiaoyu Sun. 2026. Navigating the Labyrinth: Path-Sensitive Unit Test Generation with Large Language Models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Seoul, Korea, Republic of). IEEE Press, 687–699. doi:10.1109/ASE63991.2025.00063
[40] Meixi Liu, Zhenbang Chen, Xudong Wang, Ziran He, Jiachen Gu, Minghui Chen, Lei Wang, and Wei Dong. 2026. Artifact for: Large Language Model Enabled Symbolic Execution for Automated Functional Analysis. doi:10.5281/zenodo.21773963

[41] Shaoying Liu and Shin Nakajima. 2022. Automatic Test Case and Test Oracle Generation Based on Functional Scenarios in Formal Specifications for Conformance Testing. *IEEE Transactions on Software Engineering* 48, 2 (2022), 691–712. doi:10.1109/TSE.2020.2999884

[42] Lezhi Ma, Shangqing Liu, Lei Bu, Shangru Li, Yida Wang, and Yang Liu. 2025. SpecEval: Evaluating Code Comprehension in Large Language Models via Program Specifications. arXiv:2409.12866 [cs.SE] https://arxiv.org/abs/2409.12866

[43] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 16–28. doi:10.1109/ICSE55347.2025.00129

[44] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE'07)*. 416–426. doi:10.1109/ICSE.2007.41

[45] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. 2024. Prompt Engineering in Large Language Models. In *Data Intelligence and Cognitive Informatics*, I. Jeena Jacob, Selwyn Piramuthu, and Przemyslaw Falkowski-Gilski (Eds.). Springer Nature Singapore, Singapore, 387–402.

[46] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.

[47] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40–51. doi:10.1109/2.161279

[48] Bertrand Meyer. 1997. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs.

[49] Richard Mitchell, Jim McKim, and Bertrand Meyer. 2001. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., USA.

[50] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2023. SpecFuzzer: A Tool for Inferring Class Specifications via Grammar-Based Fuzzing. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2094–2097. doi:10.1109/ASE56229.2023.00024

[51] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2023. EvoSpex: A Search-Based Tool for Postcondition Inference. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1519–1522. doi:10.1145/3597926.3604928

[52] Davide Molinelli, Alberto Martin-Lopez, Elliott Zackrone, Beyza Eken, Michael D. Ernst, and Mauro Pezzè. 2025. Tratto: A Neuro-Symbolic Approach to Deriving Axiomatic Test Oracles. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA083 (June 2025), 23 pages. doi:10.1145/3728960

[53] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). Wiley Publishing.

[54] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. *Test Intention Guided LLM-Based Unit Test Generation*. IEEE Press, 1026–1038. https://doi.org/10.1109/ICSE55347.2025.00243

[55] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 15–26. doi:10.1145/1390630.1390635

[56] 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. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages. doi:10.1145/3643769

[57] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[58] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. doi:10.1109/SP.2016.17

[59] P. Stocks and D. Carrington. 1996. A framework for specification-based testing. *IEEE Transactions on Software Engineering* 22, 11 (1996), 777–793. doi:10.1109/32.553698

[60] Ekaterina Tochilina, Vyacheslav Tamarin, Dmitry Mordvinov, Valentyn Sobol, Sergey Pospelov, Alexey Menshutin, Yury Kamenev, and Dmitry Ivanov. 2024. UTBot Python at the SBFT Tool Competition 2024. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing* (Lisbon, Portugal) *(SBFT '24)*. Association for Computing Machinery, New York, NY, USA, 41–42. doi:10.1145/3643659.3643934

[61] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[62] 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 Trans. Softw. Eng.* 50, 4 (Feb. 2024), 911–936. doi:10.1109/TSE.2024.3368208

[63] Wenhan Wang, Kaibo Liu, An Ran Chen, Ge Li, Zhi Jin, Gang Huang, and Lei Ma. 2024. Python Symbolic Execution with LLM-powered Code Generation. arXiv:2409.09271 [cs.SE] https://arxiv.org/abs/2409.09271

[64] 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* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1258–1268. doi:10.1145/3691620.3695501

[65] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3377811.3380429

[66] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[67] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu. 2020. Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2020), 1243–1256. doi:10.1109/TDSC.2018.2866469

[68] Jiahe Xu, Jingwei Xu, Taolue Chen, and Xiaoxing Ma. 2024. Symbolic Execution with Test Cases Generated by Large Language Models. In *2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS)*. 228–237. doi:10.1109/QRS62785.2024.00031

[69] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 282–291. doi:10.1145/1134285.1134325

[70] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 25–37. doi:10.1145/3368089.3409716

[71] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5673–5684. doi:10.1145/3580305.3599790