Automated Repair of Ambiguous Problem Descriptions for LLM-Based Code Generation

Haoxiang Jia*, Robbie Morris†, He Ye†, Federica Sarro† and Sergey Mechtaev*‡
*Key Laboratory of HCST (PKU), MoE, SCS, Peking University, Beijing, China
Emails: haoxiangjia@stu.pku.edu.cn, mechtaev@pku.edu.cn
†University College London, United Kingdom
Emails: {robbie.morris.22,he.ye,f.sarro}@ucl.ac.uk

***Abstract*—The growing use of large language models (LLMs) has increased the importance of natural language (NL) in software engineering. However, ambiguity of NL can harm software quality, as unclear problem descriptions may lead to incorrect program generation. Detecting and resolving such ambiguity is challenging, motivating our introduction of the automated repair of ambiguous NL descriptions, which we approach by reducing code generation uncertainty and better aligning NL with input–output examples. Ambiguity repair is difficult for LLMs because they must understand how their interpretation of a description changes when the text is altered. We find that directly prompting LLMs to clarify ambiguity often produces irrelevant or inconsistent edits. To address this, we decompose this task into two simpler steps: (1) analyzing and repairing the LLM's interpretation of the description — captured by the distribution of programs it induces — using traditional testing and program repair, and (2) refining the description based on distribution changes via a method we call contrastive specification inference. We implement this approach in a tool called SPECFIX and evaluate it using four state-of-the-art LLMs (GPT-4o, GPT-4o-mini, DeepSeek-V3, and Qwen2.5-Coder-32B-Instruct) on three popular code generation benchmarks (HumanEval+, MBPP+ and LiveCodeBench). Without human intervention or external information, SPECFIX modified 43.58% of descriptions, improving Pass@1 on the modified set by 30.9%. This yields a 4.09% absolute improvement across the entire benchmark. Repairs also transfer across models: descriptions repaired for one model improve other models' performance by 10.48%.**

***Index Terms*—Ambiguous Problem Description, Automated Repair, Code Generation, Large Language Model**

I. INTRODUCTION

Natural language (NL) has assumed an important role in software development due to the powerful capabilities of large language models (LLMs). LLMs use NL inputs, such as prompts or chat-based interactions, as a form of software requirements to generate code. However, the inherent ambiguity and susceptibility to misinterpretation in NL can impact the quality of generated code — introducing bugs or resulting in implementations that deviate from the intended purpose. For example, Vijayvargiya et al. [1] observed that the use of ambiguous task descriptions reduced the performance of models by 20%. Previous work [2] analyzed semantic discrepancies in programs sampled from an LLM to quantify the description's ambiguity, and then prompted the LLM to ask user clarifying questions. However, our experiments demonstrate that LLMs' clarifying questions are often redundant, i.e. irrelevant to LLMs' own interpretation of the problem, leading to verbose and ineffective descriptions and imposing an additional cognitive burden on the users.

To overcome the above limitations, this paper introduces the problem of *automated repair of ambiguous natural language problem descriptions*, defining it as the task of systematically modifying descriptions to ensure adherence to predefined quality criteria. Although in some cases human input is required to fix ambiguity, our insight is that many ambiguities can be resolved fully-automatically based on two key observations. First, we can modify problem descriptions to reduce LLM's code generation uncertainty by increasing the probability that the LLM generates the most likely interpretation of ambiguous language. Second, while humans often use input-output examples to clarify their intent [3], state-of-the-art (SOTA) LLMs struggle to interpret these examples in the presence of ambiguous language. In this context, we can automatically remove the ambiguity by aligning the natural language with the clarifying examples to facilitate the model's correct interpretation.

Our experience shows that SOTA LLMs struggle with directly repairing ambiguous problem descriptions, often producing irrelevant or inconsistent changes. We hypothesize that this is because for an LLM it is hard to reflect on how changes to natural language text affect the programs it would generate based on the altered text. To overcome this problem, we propose SPECFIX, an approach to repair problem description that addresses this challenge by decomposing the task into simpler subtasks: it first *repairs the distribution of programs* this description induces and then maps back the change to the description via *contrastive specification inference*.

When repairing the program distribution, SPECFIX uses two quality criteria. To lessen LLM's code generation uncertainty, SPECFIX aims to reduce *semantic entropy* [4], which is computed on the distribution of equivalence classes of programs clustered based on their input-output behavior, since high semantic entropy implies a high number of different interpretations. To align a problem description with clarifying examples, SPECFIX aims to increase *example consistency*, our novel measure that indicates to what degree programs in the distribution satisfy the examples. When all sampled

‡Sergey Mechtaev is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00038

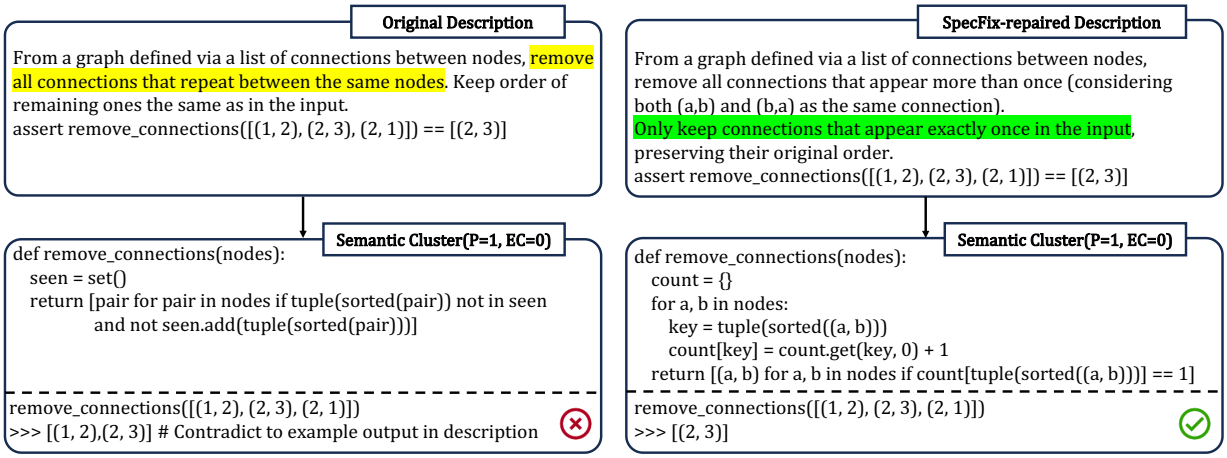

Fig. 1: The requirement to delete "repeating" connections, highlighted with , is ambiguous: it can be interpreted as either (1) deleting all connections that appear more than once, or (2) deleting all occurrences starting from the second one. Despite the presence of a clarifying example supporting the first interpretation, DeepSeek-V3 consistently outputs the second interpretation. A SPECFIX-generated repair, highlighted with , enables a consistent generation of the interpretation conforming to the example.

programs exhibit behavior contradictory to the examples, SPECFIX applies automated program repair [5] to fix programs in the distribution. The natural language description is then iteratively improved by SPECFIX's contrastive specification inference prompt, which minimally modifies the text to prioritize desirable interpretations over undesirable ones.

We conducted a comprehensive evaluation of SPECFIX using problem descriptions from there widely used code generation benchmarks: HumanEval+, MBPP+ and LiveCodeBench, and four SOTA LLMs: GPT-4o, GPT-4o-mini, DeepSeek-V3 and Qwen2.5-Coder-32b-Instruct. Our results reveal that solely based on the examples embedded within these descriptions — without relying on any external information or user feedback — SPECFIX modifies 43.58% of the descriptions, leading to a 30.9% improvement in model Pass@1 on the modified descriptions. Across the entire benchmark, this corresponds to an absolute increase of 4.09% in overall Pass@1. Importantly, we demonstrate cross-model generalizability: repairs generated by one model boost the performance of other models by 10.48%. Finally, we found that the SPECFIX 's repairs result in only modest increases in description length.

In summary, the paper makes the following contributions:

- The first approach to automated repair of natural language problem descriptions for LLM-based code generation.
- A novel problem description quality criterion, example-consistency, motivated by LLM's inability to extract the latent intent of clarifying input-output examples.
- A repair mechanism via reflection on the problem distribution using contrastive specification inference.
- An extensive evaluation showing the effectiveness of description repairs, and their cross-model generalizability.

All code, scripts, and data necessary to reproduce this work are available at https://github.com/msv-lab/SpecFix.

II. MOTIVATING EXAMPLES

In this section, we present three motivating examples to illustrate problems of ambiguous problem descriptions: (1) LLMs' inability to extract the latent intent of input-output examples to clarify ambiguous language, (2) limitations of existing LLM-based approaches that prevent them from rectifying ambiguity, and (3) the distinction between problem description repair and reasoning methods.

To characterize an LLM's interpretation of a problem description, we analyze the distribution of programs this description induces. Following previous work [6], [2], we compute this interpretation by sampling programs and partitioning them into equivalence classes based on their input-output behavior. We refer to these classes as *semantic clusters*.

A. Natural Language Ambiguity Degrades LLM Performance

Consider the problem description in Figure 1, which specifies to remove all edges that repeat between nodes in a graph. This natural language description is ambiguous, admitting two interpretations: (1) delete all connections that appear more than once, or (2) delete all occurrences starting from the second one. Although the embedded example in this description clearly supports the first interpretation, sampling 20 programs from DeepSeek-V3 [7] generates a single semantic cluster implementing the second, incorrect interpretation. One of such programs is shown at the bottom left of Figure 1 as representative of the 20 programs. Despite matching one another ($P = 1$), every program failed the embedded example, consistently adopting the second interpretation and thus disregarding the clarifying I/O example. We measure this phenomenon using example consistency (EC)—the fraction of I/O examples that programs in a cluster satisfy. Here, the cluster's EC is 0, indicating a complete mismatch between the model's behavior and the intended semantics.

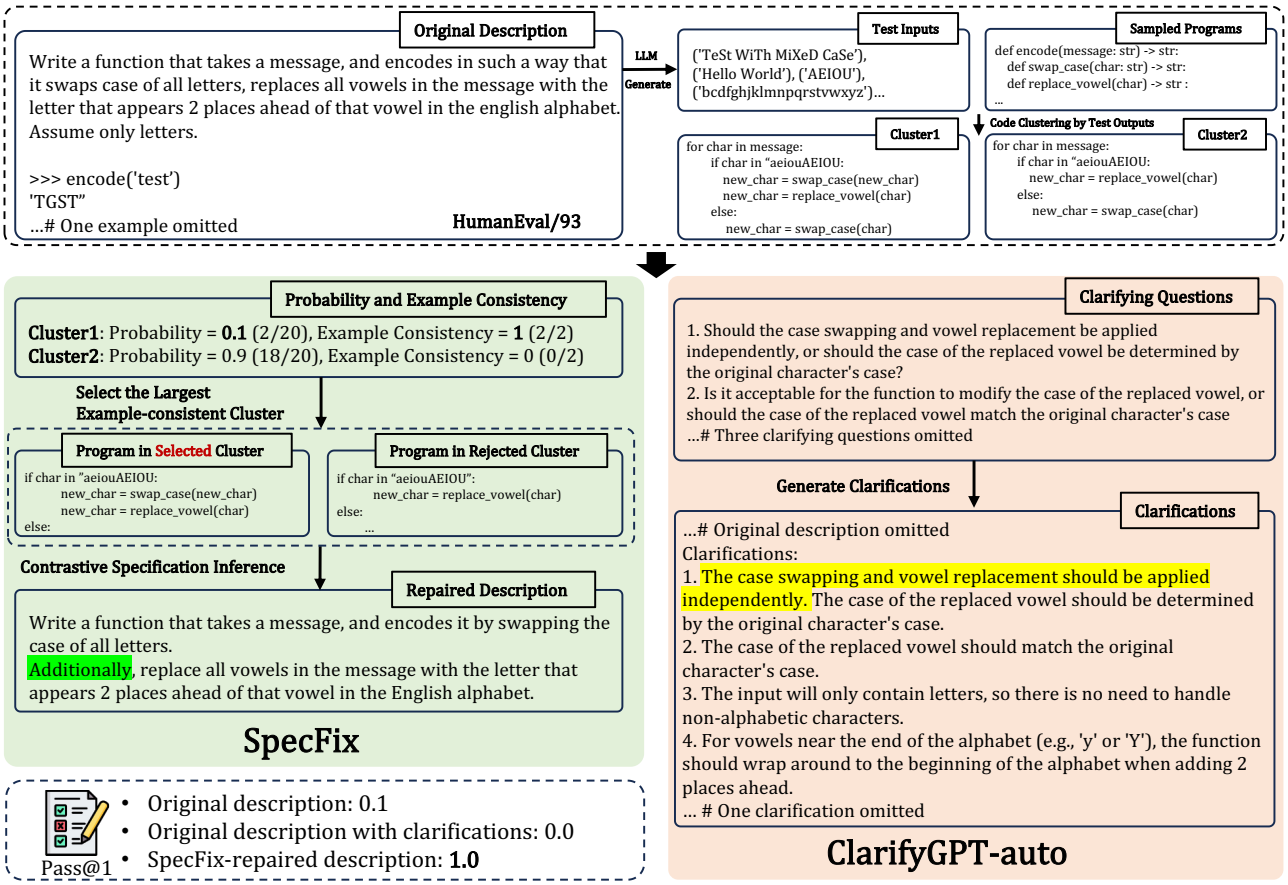

Fig. 2: A straightforward application of an LLM to repair ambiguity with ClarifyGPT-auto results in inconclusive (marked with) or irrelevant clarifications. SPECFIX decomposes the problem into *program distribution repair* and *contrastive specification inference* to produce a correct, minimal disambiguation highlighted with . The responses are generated with DeepSeek-V3.

This example demonstrates that subtle ambiguity can dominate the model's performance, which was also observed in previous work [1]. Our solution to the automated repair problem, SPECFIX, resolved the above ambiguity by explicitly stating: "Only keep connections that appear exactly once in the input." Under this disambiguated specification, DeepSeek-V3 generates the correct implementation with $P = 1$.

Observation 1: Subtle ambiguities in natural language problem descriptions can lead LLMs to generate incorrect code, even when clarifying I/O examples are present. Aligning the natural language description with the examples enables automatic, precise disambiguation.

B. Limitations of Existing LLM-Based Approaches

LLMs are obvious candidates for ambiguity repair. One prominent approach is ClarifyGPT [2], which marks a problem description as ambiguous whenever an LLM's interpretation of this description contains at least two semantic clusters of programs, and asks clarifying questions to distinguish the clusters. Although ClarifyGPT was not designed for automated repair, it can be adapted for repair by using its "user simulation prompt" to answer the generated clarifying questions based on the examples embedded in the description, and appending the resulting clarifications to the original description. We refer to this variant as ClarifyGPT-auto.

Figure 2 shows that the lack of a conjunction such as "and", "or" or "then" in the sentence "swap cases of all letters, replace all vowels" (top left) confuses DeepSeek-V3. It generates the correct interpretation, first swap cases of all letters and then replace vowels, only in 10% of samples, corresponding to the Cluster 1. In the remaining 90% of samples, it swaps case of only consonants, even though the clarifying example clearly indicates the first interpretation. When applying ClarifyGPT-auto to repair this ambiguity, it asks five questions, four of which are redundant, as they do not lead to alternative interpretations, and answers the only relevant question about operation dependency inconclusively. Consequently, the pass rate after clarification drops to 0.

This example shows the challenge of reasoning about ambiguity. To address it, SPECFIX first analyzes and repairs LLM's interpretation of a problem description embodied in the

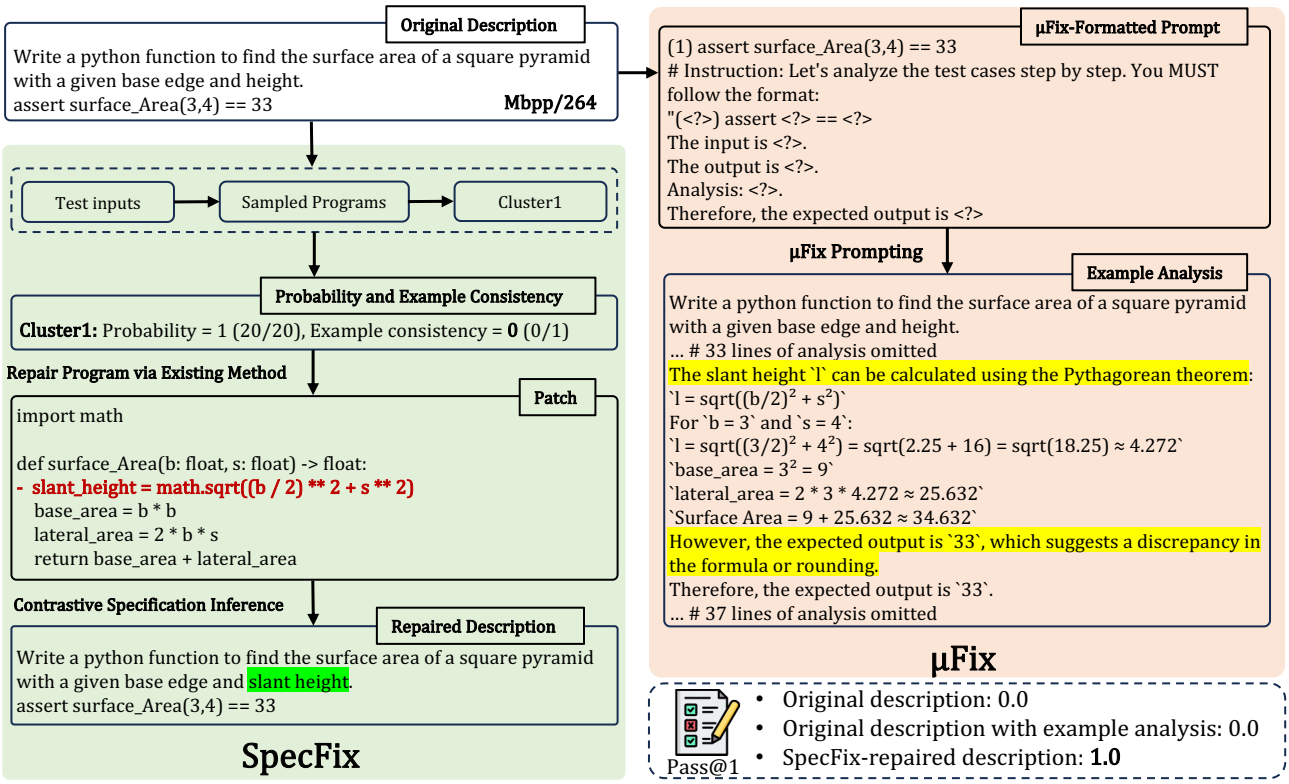

Fig. 3: SPECFIX repairs the description ambiguity by first using the example to repair programs in the distribution, and then applying contrastive specification inference on the programs before and after repair; μFix fails to correctly analyze examples. marks the incorrect example analysis, while marks the correct disambiguation. This example uses DeepSeek-V3.

distribution of semantic clusters. It measures the probability (P) and example consistency (EC) for each cluster, and repairs the distribution accordingly. In this example, the repair aims to increase the probability of the largest cluster that is fully consistent with the examples (EC $= 1$). Second, SPECFIX employs *contrastive specification inference* by prompting the LLM to generate the minimal revision to the problem description which enables the selected cluster while disables the rejected cluster. This process results in a repair that adds a simple clarification "Additionally", showing that the two operations are sequential. Using the repaired description, DeepSeek-V3 generates a correct program in 100% of cases.

Observation 2: LLMs struggle with ambiguity repair, which results in redundant or misleading clarifications. By decomposing the task into simpler subtasks, SPECFIX achieves small, intent-aligned problem description repairs.

C. Problem Description Repair vs. Reasoning

Some issues described above can be addressed by enhancing LLMs' reasoning. For example, μFix, a state-of-the-art reasoning approach for code generation "repairs" its reasoning by analyzing why generated code fails provided examples [8]. Although problem description repair and reasoning are complementary problems, we observe μFix also struggles to reason correctly when natural language is ambiguous.

Figure 3 shows an ambiguous problem description that does not specify if the input height is vertical height or slant height of a pyramid. The clarifying example implies slant height, but inferring this is nontrivial as it requires "reverse engineering" various versions of the algorithm. To reason about these descriptions, μFix first constructs a structured analysis prompt to elicit a step-by-step derivation from the given example inputs to the expected outputs. If the model fails to generate correct programs using the original description and the analysis, μFix revises its analysis by summarizing incorrect code and derives an updated description. In this example, it correctly identifies that the total surface area equals base area plus lateral surface. However, when the computed results diverge slightly from the target values for the provide IO example, μFix mistakenly attributes the discrepancy to "rounding" rather than questioning its understanding of the height parameter, yielding a 0% test pass rate.

SPECFIX addresses this limitation by repairing the program distribution. Specifically, sampling programs from DeepSeek-V3 and partitioning them into semantic clusters results in a single cluster with the example consistency 0, i.e. failing the example test. Inspired by Fan et al. [9], SPECFIX applies program repair to fix a program from this cluster using the example as the correctness criteria. Next, SPECFIX employs contrastive specification inference to concisely modify the

problem description so that it mirrors the behavior of the repaired program while diverging from the original program. Here, the repaired description explicitly states that the height is the slant height. With the repaired description, DeepSeek-V3 generates a correct program in 100% of cases.

Observation 3: The SOTA approach for reasoning about problem descriptions struggles in the presence of natural language ambiguity, as it is hard to "reverse engineer" the clarifying example for an ambiguously defined problem. SPECFIX resolves by repairing programs in the distribution so that they align with the clarifying examples.

III. BACKGROUND AND NOTATION

A *specification* (or *requirement*) is defined as "the authoritative description of the behaviors, properties, or results that the automatically-generated program must satisfy" [10]. In practice, specifications may be expressed via formal logics, natural-language docstrings, or sets of input/output examples. In this work we focus exclusively on functional specifications (*i.e.*, observable program behavior) for LLM-based code generation prompts, abstracting away from non-functional concerns such as performance, resource usage, or coding style. Henceforth we use the term "problem descriptions".

Let $\mathcal{D}$ denote the space of problem descriptions and $\mathcal{P}$ the set of all programs in a chosen programming language. An LLM can be formalized as a conditional probability distribution that, for given problem description $D \in \mathcal{D}$, assigns probabilities to various programs $m(\cdot \mid D) : \mathcal{P} \to [0,1]$. $\mathrm{supp}(m(\cdot \mid y))$ is the support of this distribution, i.e. the set of values with non-zero conditional probability.

In practice, this distribution is not given explicitly, but it can be approximated by sampling N programs $\{p_i\}_{i=1}^{N} \overset{\text{i.i.d.}}{\sim} m(\cdot \mid D)$ and using their frequencies to approximate probabilities. We denote such an approximated distribution as $\hat{m}$.

To factor out superficial syntactic variation, we use semantic equivalence relation $\equiv$ on $\mathcal{P}$: programs P and Q are equivalent if they produce identical outputs on all valid inputs.

Definition 1 (Semantic Cluster): Let $\mathcal{P}$ be a set of programs sampled from an LLM. Semantic clusters are equivalence classes of the sampled programs denoted as $\mathcal{P}/\equiv$, where for each $P \in \mathcal{P}$ its semantic cluster is denoted as

$$[P] \triangleq \{\, Q \in \mathcal{P} \mid Q \equiv P \}.$$

Since the LLM induces a distribution over individual programs, it also induces a distribution over equivalence clusters:

$$m_{\equiv}\big([P] \mid D\big) \triangleq \sum_{Q \in [P]} m(Q \mid D).$$

Example 1: In Figure 2, there are two semantic clusters: one, $[P_1]$, contains programs that first swap cases of all letters, and the second, $[P_2]$, contains programs that swaps case of only consonants. Based on the generated sample, the estimated distribution of equivalence classes is

$$m_{\equiv}\big([P_1] \mid D\big) = 0.1 \quad m_{\equiv}\big([P_2] \mid D\big) = 0.9$$

A specification is *ambiguous* if it admits multiple plausible semantic interpretations [11]. Previous work demonstrated that task ambiguity increases model's uncertainty [12]. A popular measure of uncertainty is the *semantic entropy*, which is the entropy over the meaning-distribution:

Definition 2 (Semantic Entropy [4]): Let m be an LLM, $\equiv$ be a semantic equivalence relation over responses, $m_{\equiv}$ be the corresponding conditional distribution over equivalence classes. The semantic entropy is defined as

$$\mathrm{SE}(m_{\equiv}(\cdot \mid x)) \triangleq -\sum_{y} m_{\equiv}(y \mid x) \log m_{\equiv}(y \mid x).$$

ClarifyGPT [2] proposed to classify problem descriptions as ambiguous if their semantic entropy is above zero, as it means having more than one semantic clusters of programs:

$$\mathrm{ambiguous}_{\mathrm{ClarifyGPT}}(D) \triangleq \mathrm{SE}(m_{\equiv}(\cdot \mid R)) > 0.$$

Example 2: For the distribution in Example 1, the semantic entropy is $-(0.1\log(0.1) + 0.9\log(0.9)) = 0.469$. Since it is positive, ClarifyGPT classifies the description as ambiguous.

IV. SPECFIX

SPECFIX is a framework for automatically repairing ambiguous NL problem descriptions to enhance code generation performance. Inspired by ClarifyGPT's ambiguity measure, SPECFIX aims to eliminate code generation uncertainty by reducing the semantic entropy SE. Motivated by our observation that ambiguous language may result in the generation of programs inconsistent with input-output examples embedded in the description, which may encode latent intents not explicitly stated in the text (Section II-A), SPECFIX also maximizes alignment with example via a new measure of *example consistency* EC:

Definition 3 (Example Consistency): Let m be an LLM, D be a problem description with embedded input-output examples $\{(x_i, y_i)\}_{i=1}^{m}$, and $m_{\equiv}(\cdot \mid D)$ be the corresponding distribution of semantic clusters. For a program $P \in \mathrm{supp}(m_{\equiv}(\cdot \mid D))$, the example consistency is defined as

$$\mathrm{EC}(P,\, \{(x_i, y_i)\}_{i=1}^{m}) \;=\; \frac{1}{m} \sum_{i=1}^{m} \mathbb{I}\big(P(x_i) = y_i\big)$$

where $\mathbb{I}(\cdot)$ is the indicator function. For a distribution of clusters $m_{\equiv}(\cdot \mid D)$, the example consistency is defined as

$$\mathrm{EC}(m_{\equiv}(\cdot \mid D),\, \{(x_i, y_i)\}_{i=1}^{m}) \;= \sum_{[P] \in m_{\equiv}(\cdot \mid D)} m_{\equiv}([P] \mid D)\, \mathrm{EC}(P,\, \{(x_i, y_i)\}_{i=1}^{m})$$

Example 3: Consider clusters in Figure 2, for which we defined the distribution in Example 1. Programs in $[P_1]$ pass 2 out of 2 tests, while those in $[P_2]$ pass 0 out of 2 tests. Thus, the example consistency of this distribution is $\mathrm{EC} = 0.1 * (2/2) + 0.9 * (0/2) = 0.1$.

Given the above measures, we formally define problem description repair as follows:

Definition 4 (Problem Description Repair): Let D be a problem description with embedded input-output examples EX, m be an LLM. The goal of automated problem description repair is to find a description D' such that

$$\mathrm{SE}(m_{\equiv}(\cdot \mid D')) < \mathrm{SE}(m_{\equiv}(\cdot \mid D)) \\ \wedge\ \mathrm{EC}(m_{\equiv}(\cdot \mid D'), \mathrm{EX}) > \mathrm{EC}(m_{\equiv}(\cdot \mid D), \mathrm{EX})$$

and the difference between descriptions $\mathrm{diff}(D, D') < \tau$ for some predefined threshold τ. For descriptions without embedded input-output examples, the EC is omitted.

The ultimate goal is to find a description with $\mathrm{SE} = 1$ — meaning the LLM interpretation forms a single semantic cluster — and $\mathrm{EC} = 0$ — meaning all generated programs pass the embedded examples. Since achieving such a perfect repair may not always be possible, we instead aim to improve these measures within k iterations. We call SPECFIX 's minimally edited output a *repaired description*, since improving these measures helps to resolve ambiguity.

SPECFIX's key innovation is in decomposing this task into simpler subtasks. First, it analyzes and repairs the distribution of programs the problem description induces. Second, it maps the change to the distribution back into the problem description via contrastive specification interference.

A. Distribution Repair

Algorithm 1 summarizes the workflow of SPECFIX. It accepts four arguments: a problem description D, an LLM m, the iteration bound K, and the sample size N. It first starts with extracting examples from problem descriptions using the function `Extract_Examples`, which we implement by prompting the same LLM m.

A key part of this algorithm is repairing the distribution of programs that D induces, as this distribution characterizes LLM's interpretation of the problem description. The algorithm approximates the distribution $\hat{m}_{\equiv}(\cdot \mid D)$ by sampling N programs and partitioning them into semantic clusters using the function `Interpret`. In the body of this function, the partitioning algorithm is encapsulated by the functions `Generate_Inputs` and `Partition`. In our implementation, we realize them by prompting the same LLM to generate test inputs $\{t_i\}_{i=1}^{j}$ with the aim of covering all functionality. Then, we partition programs based on the outputs they produce on these inputs — programs with identical outputs are grouped into the same cluster, indicating equivalent behavior. Alternative implementations such as differential fuzzing [13] or symbolic execution [14] can also be applied in this context.

To resolve ambiguities, SPECFIX employs two repair strategies: probability-guided repair and program repair based problem description repair. If one or more clusters pass all examples ($\mathrm{EC} = 1$), SPECFIX invokes the probability-guided repair strategy; otherwise — when no cluster can pass all examples — it resorts to the program repair based description repair.

Probability-guided Repair: To reduce semantic entropy, SPECFIX prioritizes the most likely interpretation. First, it prioritizes semantic clusters that pass all examples ($\mathrm{EC} = 1$) over other clusters. The process is described in Algorithm 1 in lines 15–17, and is illustrated in Figure 2, where the low-probability cluster (0.1) is prioritized over the high-probability cluster (0.9), because the former has higher example consistency. If there are multiple clusters with $\mathrm{EC} = 1$, the algorithm selects one with the highest probability, treating all others as rejected clusters, which is influenced by previous research on program selection via majority voting such as CodeT [15], which demonstrate that, in the absence of additional information, selecting the solution with the highest estimated probability improves the likelihood of passing the tests.

Program Repair Based Description Repair: If no semantic cluster matches all input-output examples, SPECFIX applies program repair to fix faulty programs, using the given example inputs and outputs. Specifically, we employ the self-refine method [16], which utilizes incorrect programs, together with the runtime feedback (e.g. test failures, expected output and actual output), and automatically generate a corrected program. SPECFIX selects programs from the most probable cluster for repair, as this cluster represents the predominant

Algorithm 1: SPECFIX Algorithm

Input : Problem description D, LLM m, iteration bound K, sample size N
Output: Repaired problem description D'

1 **Function** `Interpret`$(m,\ D,\ N)$**:**
2 $\quad \{p_i\}_{i=1}^{N} \sim m(\cdot \mid D)$;
3 $\quad \{t_i\}_{i=1}^{j} \leftarrow$ `Generate_Inputs`(D);
4 $\quad \hat{m}_{\equiv}(\cdot \mid D) \leftarrow$ `Partition`$\big(\{p_i\}_{i=1}^{N},\ \{t_i\}_{i=1}^{j}\big)$;
5 $\quad$ **return** $\hat{m}_{\equiv}(\cdot \mid D)$
6 **Function** `SpecFix`(D, m, K, N)**:**
7 $\quad \{(x_i, y_i)\}_{i=1}^{l} \leftarrow$ `Extract_Examples`(D);
8 $\quad \hat{m}_{\equiv}(\cdot \mid D) \leftarrow$ `Interpret`(m, D, N);
9 $\quad \mathrm{se} \leftarrow \mathrm{SE}(\hat{m}_{\equiv}(\cdot \mid D))$;
10 $\quad \mathrm{ec} \leftarrow \mathrm{EC}(\hat{m}_{\equiv}(\cdot \mid D), \{(x_i, y_i)\}_{i=1}^{l})$;
11 $\quad$ **for** $i \leftarrow 1$ **to** K **do**
12 $\quad\quad$ **if** $\mathrm{se} = 0 \wedge \mathrm{ec} = 1$ **then**
13 $\quad\quad\quad$ **return** D ; `// D is unambiguous`
14 $\quad\quad C \leftarrow \mathrm{supp}(\hat{m}_{\equiv}(\cdot \mid D))$;
15 $\quad\quad$ **if** $\exists\, c \in C$ *s.t.* $\mathrm{EC}(c) = 1$ **then**
16 $\quad\quad\quad [P_{\mathrm{select}}] \leftarrow \arg\max_{c \in C \text{ s.t. } \mathrm{EC}(c)=1} |c|$;
17 $\quad\quad\quad P_{\mathrm{reject}} \leftarrow \cup(C \setminus [P_{\mathrm{select}}])$;
18 $\quad\quad$ **else**
19 $\quad\quad\quad [P_{\mathrm{reject}}] \leftarrow \arg\max_{c \in C} |c|$;
20 $\quad\quad\quad P_{\mathrm{select}} \leftarrow$ `Program_Repair`$(P_{\mathrm{reject}}, \{(x_i, y_i)\}_{i=1}^{l})$
21 $\quad\quad D' \leftarrow$ `Contrast_Infer`$(D, P_{\mathrm{select}}, P_{\mathrm{reject}})$;
22 $\quad\quad \hat{m}_{\equiv}(\cdot \mid D') \leftarrow$ `Interpret`(m, D', N);
23 $\quad\quad \mathrm{se}' \leftarrow \mathrm{SE}(\hat{m}_{\equiv}(\cdot \mid D'))$;
24 $\quad\quad \mathrm{ec}' \leftarrow \mathrm{EC}(\hat{m}_{\equiv}(\cdot \mid D'), \{(x_i, y_i)\}_{i=1}^{l})$;
25 $\quad\quad$ **if** $\mathrm{ec}' > \mathrm{ec} \wedge \mathrm{se}' < \mathrm{se}$ **then**
26 $\quad\quad\quad D \leftarrow D', \mathrm{ec} \leftarrow \mathrm{ec}', \mathrm{se} \leftarrow \mathrm{se}'$;
27 $\quad$ **return** D;

interpretation by the LLM. SPECFIX treats the faulty program as a rejected program, and the repaired program that is example consistent as a selected program. This process is provided in Algorithm 1 in lines 19–20, and is illustrated in Figure 3, where a representative of the cluster with $\mathrm{EC} = 0$ is repaired by removing the slant height computation so that it is consistent with the example, and then this change informs the necessary edit of the problem description.

B. Contrastive Specification Inference

Once SPECFIX has identified a selected program and one or more rejected programs, it applies our *contrastive specification inference* prompt to disambiguate a given problem descriptions via a minimal natural language change. Contrastive specification inference leverages the insight that an explanation is most informative when it highlights why outcome A occurs instead of outcome B rather than describing outcome A in isolation. By contrasting the behavior of the selected program with that of rejected programs, the repaired description is prone to align with the selected program while intentionally diverge from the rejected ones. SPECFIX realizes this objective by prompting an LLM with both the supporting evidence (selected program & outputs) and the contradicting evidence (rejected programs & outputs), and asking the model to (i) diagnose the sources of ambiguity and (ii) rewrite the specification so that only the intended behavior remains admissible.

V. EVALUATION

Our study addresses the following research questions:

- **RQ1**: To what extent do SPECFIX-repaired descriptions improve code generation compared to baseline methods?
- **RQ2**: Do problem descriptions repaired by SPECFIX enable cross-model improvement in code generation?
- **RQ3**: How much does SPECFIX change problem descriptions in comparison with baseline methods?

A. Experimental Setup

Models: We selected four widely-used LLMs: GPT-4o, GPT-4o-mini, DeepSeek-V3 and Qwen2.5-Coder-32b-Instruct. GPT-4o [17] is OpenAI's advanced commercial model with high accuracy across diverse tasks, and GPT-4o-mini is its smaller variant. DeepSeek-V3 [7] is an open-source, efficiency-focused model optimized for mathematics and coding. Qwen2.5-Coder-32B-Instruct [18], developed by Alibaba, specializes in multilingual code generation, debugging, and software development with robust problem solving capabilities. For brevity, we refer to DeepSeek-V3 and Qwen2.5-Coder-32b-Instruct simply as "DeepSeek" and "Qwen2.5".

Benchmarks: We selected three widely-used code generation benchmarks: HumanEval+, MBPP+ and LiveCodeBench. HumanEval+ consists of 164 programming problems, each including a function signature, docstring, canonical solutions, and several test cases, designed to evaluate language comprehension, algorithms, and simple mathematics. MBPP+ includes 378 hand-verified Python programming problems with task descriptions, canonical solutions, and multiple test cases, covering programming fundamentals and standard library functionalities. HumanEval+ and MBPP+ are enhanced versions of the original HumanEval [19] and MBPP [20] benchmarks augmented by the EvalPlus framework [21]. Humaneval+ includes 2.81 examples per problem on average and 161 problem descriptions embed at least one example. MBPP+ includes one example in each problem. LiveCodeBench [22] is a dynamic, contamination-aware benchmark that continuously harvests recent competitive-programming problems from LeetCode, AtCoder, and Codeforces. To minimize data leakage, we select livecodebench_v6, which contains 175 problems released between January 2025 and May 2025. Hidden tests used for evaluation differ from the examples in the descriptions.

Following prior work [2], we set the model temperature at 0 for all tasks to ensure deterministic outputs, except for program sampling, where the temperature is set to the default value specified by each LLM to encourage diversity. Since our approach, like most works on uncertainty estimation, relies on output diversity, it requires non-zero sampling temperature. To avoid implying that disambiguation was always successful, we use the term "modified" rather than "repaired" in certain descriptions. To account for generation variability, we repeated all experiments three times and reported averages.

To estimate entropy, we sample N programs for reliable estimation. We conducted an experiment on a subset of problems with various $N \in$ 5, 10, 15, ..., 40, which showed that entropy increases rapidly from 0.096 (N=5) to 0.1123 (N=20), after which additional samples result only in small fluctuations. Therefore, we adopt N=20 as our default configuration, ensuring both statistical reliability and computational cost.

B. Evaluated Approaches

To the best of our knowledge,SPECFIX is the first fully automated method designed specifically to repair problem descriptions for LLM-based code generation. Thus, we adapted the following approaches as baselines for problem description repair evaluation:

Vanilla Repair prompts an LLM to classify each description as ambiguous or unambiguous. If deemed ambiguous, the model resolves the ambiguity by adopting the most likely interpretation.

ClarifyGPT-auto adapts ClarifyGPT [2] for fully automated problem description repair. Whereas the original approach simulates human clarifications using hidden test cases, we replace this mechanism with examples drawn directly from the problem descriptions to eliminate data-leakage risks. The resulting "repaired" description is formed by concatenating the original specification with the model's clarifications. Semantic clustering again uses $N = 20$ samples.

μFix [8] analyzes program failures on input–output examples and generates reasoning chains to iteratively refine its interpretation of problem descriptions. Although μFix does not produce explicit description edits, its focus on improving problem understanding makes it a relevant baseline.

TABLE I: The effect of repairing problem descriptions from HumanEval+, MBPP+, and LiveCodeBench on code generation performance of four LLMs. Each approach is denoted as "{description repair method} + {code generation method}". "Original" refers to the descriptions before repair. SPECFIX consistently outperforms other approaches in code generation metrics and uncertainty reduction. "Mod%" refers to the ratio of modified descriptions.

Model	Approach	HumanEval+						MBPP+						LiveCodeBench					
		Mod%	Pass@1	APR	NZP@1	M@20	SE	Mod%	Pass@1	APR	NZP@1	M@20	SE	Mod%	Pass@1	APR	NZP@1	M@20	SE
DeepSeek	Original	–	87.8%	94.1%	**93.9%**	89.6%	0.1	–	79.4%	88.9%	**83.6%**	80.2%	0.1	–	37.5%	65.3%	44.6%	40.0%	0.5
	Original + μFix	–	88.9%	94.6%	93.3%	89.0%	0.1	–	79.1%	89.4%	81.7%	79.7%	0.1	–	37.4%	65.7%	38.9%	37.5%	**0.1**
	Vanilla Repair	34.2%	84.8%	92.2%	89.0%	86.6%	0.1	44.4%	79.1%	88.5%	81.2%	79.4%	0.1	33.5%	35.2%	60.6%	42.3%	37.7%	0.5
	ClarifyGPT-auto	17.7%	89.0%	95.2%	90.9%	89.8%	0.1	12.7%	79.9%	89.9%	81.7%	80.0%	0.1	61.7%	38.1%	67.1%	47.0%	40.0%	0.4
	SPECFIX	20.1%	**92.0%**	**95.9%**	93.3%	**92.7%**	**0.0**	19.3%	**82.1%**	**91.8%**	83.2%	**82.2%**	**0.0**	66.3%	**40.8%**	**69.9%**	**47.0%**	**41.9%**	0.3
Qwen2.5	Original	–	84.4%	92.7%	87.7%	85.2%	0.1	–	79.2%	89.7%	81.5%	79.4%	0.1	–	29.4%	54.3%	40.0%	31.4%	0.5
	Original + μFix	–	83.9%	93.1%	85.0%	84.3%	0.0	–	79.0%	89.1%	79.7%	78.9%	0.0	–	29.6%	54.9%	33.5%	30.1%	**0.3**
	Vanilla Repair	15.2%	83.3%	92.3%	86.0%	84.1%	0.1	36.8%	76.1%	86.8%	77.8%	76.2%	0.1	1.8%	29.3%	54.1%	39.4%	30.9%	0.5
	ClarifyGPT-auto	15.9%	85.2%	93.9%	86.9%	85.6%	0.1	13.5%	79.2%	89.5%	80.3%	79.1%	0.0	69.1%	28.9%	54.5%	37.3%	31.0%	0.5
	SPECFIX	20.7%	**87.7%**	**96.5%**	**88.6%**	**87.7%**	**0.0**	16.9%	**82.3%**	**91.9%**	**83.5%**	**82.7%**	**0.0**	77.1%	**33.1%**	**60.1%**	**40.6%**	**34.9%**	0.4
GPT-4o	Original	–	82.0%	93.0%	**91.5%**	83.5%	0.2	–	78.2%	88.9%	**84.9%**	78.8%	0.2	–	32.3%	54.2%	41.1%	33.1%	0.6
	Original + μFix	–	82.2%	93.0%	88.6%	83.9%	0.2	–	78.0%	88.7%	81.1%	78.4%	0.2	–	30.5%	52.9%	35.0%	31.0%	**0.3**
	Vanilla Repair	62.8%	80.1%	91.5%	84.1%	79.3%	0.2	82.5%	76.0%	87.5%	82.0%	77.0%	0.2	42.1%	32.0%	54.7%	40.6%	32.6%	0.5
	ClarifyGPT-auto	36.0%	82.3%	92.7%	87.8%	83.7%	0.1	30.2%	77.6%	88.9%	80.7%	78.0%	0.1	76.0%	32.2%	55.0%	40.6%	33.7%	0.5
	SPECFIX	39.6%	**87.6%**	**96.4%**	90.2%	**87.4%**	**0.0**	31.8%	**80.6%**	**91.2%**	**82.7%**	**80.8%**	**0.0**	82.3%	**35.6%**	**58.5%**	**41.7%**	**37.9%**	0.3
GPT-4o-mini	Original	–	81.6%	92.0%	92.7%	81.7%	0.2	–	75.8%	86.6%	80.4%	76.7%	0.2	–	28.6%	53.6%	39.4%	31.4%	0.6
	Original + μFix	–	80.2%	90.4%	86.4%	81.7%	0.1	–	76.0%	86.7%	79.0%	76.5%	0.1	–	28.2%	53.3%	32.0%	29.0%	**0.4**
	Vanilla Repair	11.8%	81.5%	91.6%	91.5%	81.5%	0.2	31.3%	75.5%	86.4%	79.6%	76.5%	0.2	2.6%	28.7%	53.4%	39.4%	31.4%	0.6
	ClarifyGPT-auto	30.5%	80.1%	91.6%	86.0%	81.1%	0.1	31.0%	75.1%	87.1%	77.3%	75.7%	0.1	77.1%	28.2%	53.3%	38.5%	30.3%	0.6
	SPECFIX	34.1%	**87.6%**	**95.3%**	**93.5%**	**88.0%**	**0.1**	33.1%	**78.5%**	**89.5%**	**80.5%**	**79.2%**	**0.1**	81.7%	**32.6%**	**57.7%**	**40.8%**	**35.4%**	0.4

TABLE II: For each baseline (Vanilla Repair, ClarifyGPT-auto and μFix), this table compares performance of SPECFIX's repairs with the baseline's performance on the subset of problem descriptions that both SPECFIX and the baseline modified. "#" denotes the number of such descriptions. Since μFix does not modify descriptions, the corresponding subsets includes all descriptions SPECFIX modified. SPECFIX-repaired descriptions outperform baselines on most subsets.

Model	Baseline	HumanEval+				MBPP+				LiveCodeBench			
		#	Pass@1			#	Pass@1			#	Pass@1		
			Original	Baseline	SPECFIX		Original	Baseline	SPECFIX		Original	Baseline	SPECFIX
Deepseek	Vanilla Repair	17	34.12%	29.41%	**51.76%**	48	38.75%	39.58%	**46.74%**	40	9.06%	7.00%	**12.93%**
	ClarifyGPT-auto	29	48.62%	55.63%	**71.15%**	64	42.66%	45.83%	**52.08%**	108	15.53%	16.81%	**20.28%**
	Original + μFix	33	45.76%	54.34%	**66.57%**	73	37.67%	43.47%	**52.05%**	116	16.08%	16.50%	**21.09%**
Qwen2.5	Vanilla Repair	8	**36.25%**	35.00%	29.17%	29	53.21%	42.50%	**62.02%**	3	6.67%	0.00%	**22.22%**
	ClarifyGPT-auto	26	49.66%	54.23%	**64.36%**	51	50.20%	50.63%	**57.89%**	121	13.01%	12.36%	**18.30%**
	Original + μFix	34	47.97%	63.91%	**64.71%**	64	40.51%	51.30%	**60.17%**	135	12.41%	13.57%	**17.23%**
GPT-4o	Vanilla Repair	50	50.82%	53.76%	**70.20%**	104	54.93%	53.45%	**61.15%**	60	14.41%	12.96%	**21.07%**
	ClarifyGPT-auto	59	61.21%	62.08%	**73.61%**	114	55.46%	53.37%	**61.05%**	133	18.19%	18.94%	**22.98%**
	Original + μFix	65	58.17%	63.81%	**72.30%**	120	52.69%	52.53%	**60.03%**	144	21.33%	19.74%	**25.43%**
GPT-4o-mini	Vanilla Repair	7	45.71%	44.29%	**61.43%**	25	32.80%	30.00%	**46.40%**	2	35.00%	50.00%	**50.00%**
	ClarifyGPT-auto	50	57.36%	52.46%	**74.56%**	117	47.01%	44.44%	**53.68%**	135	16.64%	16.12%	**21.82%**
	Original + μFix	56	56.75%	58.33%	**74.54%**	125	44.00%	47.70%	**52.16%**	143	16.29%	15.76%	**21.10%**

C. Evaluation Metrics

We apply five measures to comprehensively evaluate the correctness and interpretive clarity of repaired descriptions.

Pass@1 is the probability that a generated program passes all tests. We estimate Pass@1 by sampling ten independent programs per problem according to the best practices [19].

AvgPassRate (APR) measures the expected ratio of tests passed across generated programs. APR provides a more granular view of how disambiguating problem descriptions improves partial correctness.

%Pass@1>0 (NZP@1) denotes the proportion of problems for which Pass@1 exceeds zero. An increase in NZP@1 when comparing original versus repaired problem descriptions indicates that repair enables the model to produce at least one fully correct program for cases previously unsolvable.

Majority@20 (M@20) measures the accuracy of the most frequent correct program within 20 generated samples (i.e. the majority vote) [15], [23]. This metric evaluates the model's consistency in converging on a consensus solution.

Semantic Entropy (SE) [4] quantifies the distribution of generated programs across distinct semantic clusters. Higher semantic entropy suggests greater ambiguity [2].

D. *RQ1: Repair Effectiveness of* SPECFIX

To evaluate repair effectiveness, we measured how problem descriptions modified by an LLM affect the same LLM's performance. Table I presents the ratio of modified descriptions and performance metrics on all benchmark problems. For all model–benchmark pairing, SPECFIX achieves the

TABLE III: Pass@1 (%) on HumanEval+, MBPP+, and LiveCodeBench before ("Orig.") and after ("Repair") ambiguity repair. Rows are the description-repair models that perform the repair; columns are the models used for evaluation. Each cell shows Orig./Repair. Underlined cells indicate the same model is used for both repair and evaluation. "#" is the number of modified descriptions. Underlined cells indicate the same model performed both repair and evaluation.

HumanEval+

Repair Model	#	DeepSeek	Qwen2.5	GPT-4o	GPT-4o-mini
DeepSeek	33	45.76/66.57	51.55/61.10	44.88/56.18	45.69/60.71
Qwen2.5	34	60.59/65.00	43.86/59.12	49.15/58.05	46.11/52.65
GPT-4o	65	72.62/71.13	67.88/74.04	58.17/72.30	61.20/69.44
GPT-4o-mini	56	71.79/72.26	66.69/73.14	65.91/73.34	56.75/74.54

MBPP+

Repair Model	#	DeepSeek	Qwen2.5	GPT-4o	GPT-4o-mini
DeepSeek	73	37.67/52.05	49.95/46.68	46.30/49.96	42.05/46.58
Qwen2.5	64	47.03/54.74	40.00/58.12	46.45/54.72	44.69/46.30
GPT-4o	120	55.75/60.03	58.41/61.51	52.69/60.03	48.58/55.36
GPT-4o-mini	125	55.36/51.03	57.11/51.42	52.10/48.70	44.00/52.16

LiveCodeBench

Repair Model	#	DeepSeek	Qwen2.5	GPT-4o	GPT-4o-mini
DeepSeek	116	16.08/32.09	9.82/12.91	12.60/15.13	9.14/11.61
Qwen2.5	135	22.35/23.00	12.26/17.06	17.80/18.29	12.66/15.28
GPT-4o	144	26.91/27.39	18.73/23.52	21.33/25.43	17.68/20.62
GPT-4o-mini	143	27.64/28.17	17.28/20.86	22.12/20.96	16.29/21.10

highest Pass@1 score among all approaches. For instance, on the HumanEval+ benchmark with DeepSeek, SPECFIX achieves a Pass@1 of 92.0% surpassing the performance of the original descriptions (87.8%), and the results of ClarifyGPT-auto (89.0%). Similar gains hold for Qwen2.5, GPT-4o and GPT-4o-mini. SPECFIX also achieves the best average pass rate, with DeepSeek/HumanEval+ yielding an AvgPassRate of 95.6%. This suggests SPECFIX repairs guide models toward outputs passing a higher ratio of tests. Notably, SPECFIX also substantially reduces semantic entropy, showing that repaired descriptions yield less semantically diverse code. Finally, SPECFIX is the only approach that consistently improved Majority@20, showing that SPECFIX not only increases the probability of generating a correct program but also enhances the consistency with which a consensus solution emerges under repeated sampling.

An alternative to repairing problem descriptions is to strengthen the code generator. Thus, we investigate how the performance improvement due to descriptions repaired by SPECFIX in the context of zero-shot code generation compares to the performance of original descriptions with a SOTA code generation method, μFix, which employs advanced reasoning techniques. Table I present the results. Since μFix reasons over all benchmark problems, it does not modify a subset, hence no "Modified%" is shown for this method in Table I. SPECFIX outperformed μFix in most settings. Moreover, μFix showed only minimal enhancements in code generation measures compared to the zero-shot approach. We attribute this to the fact that benchmarks like HumanEval and MBPP are reaching their saturation [24], where SOTA models already exhibit high success rate, making additional improvements through prompt tuning challenging. SPECFIX's repaired descriptions demonstrated substantial performance gains, showing that problem description repair is a promising pathway for further pushing the boundaries of code generation.

We also investigated Pass@1 only on the modified descriptions shown in Table III. On average, SPECFIX modifies 43.58% of descriptions. Underlined values indicate the Pass@1 results when the same model performs both repair and evaluation. All models improve, with an average Pass@1 increase of 30.9%. Since each method uses a different ambiguity-detection heuristic, they operate on partially disjoint sets of problem descriptions. As Monperrus [25] notes, fair comparisons should account for the defect classes each method addresses. To eliminate the influence from different ambiguity detection strategies, we performed pairwise evaluations of SPECFIX against each baseline, restricted to the intersection of problem descriptions they both modify. Table II shows the Pass@1 on these subsets of problem descriptions. SPECFIX outperforms both the original descriptions and the baseline repairs on all subsets except those jointly repaired with Vanilla Repair on HumanEval+ with Qwen2.5. That subset is small (8 cases), so the observed drop likely reflects statistical noise.

SPECFIX's average repair time is 36.7 seconds, and the average token cost is 6770.7 tokens per description.

RQ1: Across benchmarks, SPECFIX yields a 30.9% average Pass@1 improvement on the modified subset, corresponding to a 4.09% improvement over complete benchmarks. Baseline methods commonly decreased performance or produced negligible gains (<0.5%).

E. *RQ2: Cross-Model Generalization*

To show that SPECFIX addresses inherent problems of problem descriptions rather than merely model-specific misunderstandings, we evaluate how descriptions repaired with one model affect another model's code generation performance. Table III shows the results of our experiments. Although in few instances, which are highlighted in red, the other model's performance dropped after repair, specifically MBPP+ repaired by GPT-4o-mini, on average using a different model to repair problem descriptions increases performance by 10.48%.

RQ2: Problem descriptions repaired by SPECFIX using one model improve another model's Pass@1 by 10.48% on average, showing that its repairs generalize across models.

F. *RQ3: Comparing Lengths of Repaired Descriptions*

To further illustrate limitations of baseline methods, we measured the relative increase in description length induced by each method. Table IV reports the average percentage growth of the repaired problem descriptions. ClarifyGPT-auto's clarification and μFix's reasoning chains substantially inflate length. With GPT-4o on MBPP+, ClarifyGPT-auto and μFix increase

TABLE IV: Relative increment in description length between before-repair and after-repair for HumanEval+, MBPP+ and LiveCodeBench. LCB refers to LiveCodeBench.

Model	Method	HumanEval+	MBPP+	LCB
DeepSeek	Vanilla Repair	−3.9%	57.4%	−46.4%
	ClarifyGPT-auto	158.8%	360.7%	64.4%
	μfix	255.0%	354.0%	339.9%
	SPECFIX	12.8%	120.2%	0.7%
Qwen2.5	Vanilla Repair	−44.9%	−31.5%	−62.5%
	ClarifyGPT-auto	82.5%	184.8%	32.9%
	μfix	405.3%	795.2%	339.3%
	SPECFIX	9.0%	58.4%	8.0%
GPT-4o	Vanilla Repair	−3.9%	2.2%	−37.7%
	ClarifyGPT-auto	262.2%	576.6%	94.3%
	μfix	298.6%	426.6%	375.8%
	SPECFIX	65.8%	237.4%	43.9%
GPT-4o-mini	Vanilla Repair	13.3%	10.2%	−58.0%
	ClarifyGPT-auto	114.0%	281.1%	23.1%
	μfix	263.5%	317.4%	243.4%
	SPECFIX	22.0%	113.7%	10.9%

description length by 576.61% and 425.56%, respectively. Although Vanilla Repair yields the smallest length increase, it also achieves the weakest repair effectiveness. In contrast, SPECFIX achieves the best repair performance while keeping descriptions concise. ClarifyGPT-auto often adds irrelevant clarifying questions, leading to redundant clarifications. μFix's format-specific prompts embed full reasoning in the repaired descriptions. Excessive explanation can dilute the original task and cause models to overfit to added clarifications.

RQ3: SPECFIX 's repairs result in only modest increases in description length compared with ClarifyGPT-auto's clarifications and μFix's reasoning chains, while still achieving superior repair effectiveness.

G. Ablation Study

SPECFIX uses two measures to guide repair: semantic entropy (SE) and example consistency (EC). To investigate their contributions, we compared SPECFIX with SPECFIX$_{woSE}$ (removing SE) and SPECFIX$_{woEC}$ (removing EC). Because SPECFIX and its variants modify different descriptions under their own criteria, we report Pass@1 on the full datasets. Table V reports the Pass@1 across three datasets and four models. First, SPECFIX consistently outperforms both SPECFIX$_{woSE}$ and SPECFIX$_{woEC}$, indicating both two components are beneficial. On average, SPECFIX improves 2.2% and 0.92% higher Pass@1 than SPECFIX$_{woSE}$ and SPECFIX$_{woEC}$, respectively.

Finally, SPECFIX has one hyperparameter k (the number of repair iteration). We evaluated $k \in$ 1, 2,..., 10 on LiveCodeBench and Qwen2.5. Pass@1 increases from 31.17% (k=1) to 36.51% (k=3), after which additional iterations produce only minor fluctuations.

TABLE V: Contribution of semantic entropy (SE) and example consistency (EC) to Pass@1 (%) of SPECFIX's repairs.

Dataset	Model	SPECFIX$_{woSE}$	SPECFIX$_{woEC}$	SPECFIX
HumanEval+	DeepSeek-V3	90.55	89.95	**91.99**
	Qwen2.5	87.23	86.77	**87.74**
	GPT-4o	86.57	84.07	**87.62**
	GPT-4o-mini	86.22	84.33	**87.65**
MBPP+	DeepSeek-V3	81.53	79.87	**82.14**
	Qwen2.5	81.06	78.02	**82.31**
	GPT-4o	80.22	78.28	**80.56**
	GPT-4o-mini	77.41	76.98	**78.54**
LCB	DeepSeek-V3	39.80	38.77	**40.84**
	Qwen2.5	32.88	32.14	**33.12**
	GPT-4o	34.84	34.23	**35.62**
	GPT-4o-mini	31.34	30.63	**32.55**

H. Analysis and Mitigation of Repair Errors

We analyze cases where SPECFIX generated incorrect modifications. Table VI reports the incorrect modification ratio (IMR), i.e. the fraction of modifications that reduce Pass@1, and their corresponding Pass@1. On average, IMR is 3.23%, and Pass@1 on incorrectly modifications is 20.93%. Manual inspection shows most errors stem from majority voting: when clusters with correct and incorrect programs have similar probabilities, majority vote often selects an incorrect one, leading to faulty edits. As a mitigation, we apply modified z-score for majority voting: a cluster is selected only if its probability substantially exceeds all others [26]. Otherwise, SPECFIX defers to user input. Compared to ClarifyGPT, which asks users to answer clarifying questions for every description, we ask users to choose between two programs in only 10.8% of modifications. We simulated the user using hidden tests by selecting the cluster that passes more hidden tests. which reduces IMR to 1.58% and raises Pass@1 to 47.93%.

Beyond the limitation of majority voting, SPECFIX assumes the input–output examples are correct, a standard assumotion in test-based automated program repair research. In HumanEval+ and MBPP+, we identified 5 out of 542 instances where the examples conflicted with the reference solutions ($<$ 1%), causing SPECFIX to produce incorrect repairs. This is small relative to the description modified by SPECFIX (24%).

VI. THREATS TO VALIDITY

SPECFIX assumes that code generated from problem descriptions can be independently executed, and that some candidate programs are correct or near-correct. This may not hold when the code is part of a larger system or when the task's complexity prevents the model from producing meaningful solutions. In such cases, ambiguity could be detected through behavioral proxies, (*e.g.*,formal models), which requires further investigation.

Construct validity may be threatened by how we define ambiguity and its repairs through metrics such as Pass@1, semantic entropy, example consistency, etc. We used this approach because our goal was to investigate a fully automated

TABLE VI: Incorrect-Modification Rate (IMR) and corresponding Pass@1 (%) across models and benchmarks. Entries are reported as *before mitigation→after mitigation*; arrows indicate the direction of change after mitigation (↓ = lower IMR, ↑ = higher Pass@1). IMR is the fraction of modified descriptions that are incorrect.

Model	Benchmark	IMR	Pass@1
DeepSeek-V3	HumanEval	3.0% → 2.4% ↓	0.0% → 10.0% ↑
	MBPP	3.3% → 2.9% ↓	3.8% → 24.8% ↑
	LCB	2.5% → 1.3% ↓	31.1% → 50.3% ↑
GPT-4o	HumanEval	3.4% → 0.9% ↓	17.3% → 64.5% ↑
	MBPP	7.2% → 4.7% ↓	9.0% → 31.4% ↑
	LCB	2.3% → 1.4% ↓	33.3% → 51.6% ↑
GPT-4o-mini	HumanEval	2.2% → 0.8% ↓	33.3% → 53.6% ↑
	MBPP	2.6% → 1.6% ↓	20.1% → 36.6% ↑
	LCB	3.0% → 1.0% ↓	40.0% → 57.4% ↑
Qwen2.5	HumanEval	3.9% → 0.7% ↓	10.4% → 60.0% ↑
	MBPP	2.3% → 0.0% ↓	25.7% → 91.7% ↑
	LCB	3.0% → 1.2% ↓	27.1% → 43.3% ↑

method for ambiguity resolution. However, some cases may require human input. In future research, we will investigate how to efficiently involve humans in the SPECFIX workflow.

External validity is limited by our focus on HumanEval+, MBPP+ and LiveCodeBench, and relying on the input-output examples embedded in their problem descriptions. These benchmarks are well-established and representative, and using input-output examples is a common practice, e.g. in programming-by-example [3], in Stack Overflow posts [27] and GitHub issues [28]. However, studying ambiguities in real-world applications such as in interactions with AI coding assistants remains an important future research direction.

VII. RELATED WORK

LLM Reasoning About Problem Descriptions: μFIX [8] enhances problem description understanding by integrating thought-eliciting prompting with feedback-based code generation. Fan et al. [29] apply LLM reasoning to elaborate the meaning of low-frequency keywords in problem descriptions. In contrast, SpecFix aims to disambiguate the problem descriptions themselves so that the models interpret them correctly. Our experiments show that SpecFix is more effective than μFIX at interpreting non-trivial input-output examples. Li et al. [30] propose maintaining equivalent representations, such as natural language comments and pseudocode, that preserve semantics via a two-LLM reflection mechanism. This approach may enhance SpecFix' reflection prompt used for contrastive inference. CodeMind [31] introduces a "specification reasoning" task that evaluates model's ability to reason about combinations of NL specifications and test execution. Our study reveals that SOTA LLMs often perform poorly on this task, often generating program that contradict explicitly specified examples.

Detecting NL Ambiguity with LLMs: Ambiguity in problem descriptions undermines downstream performance and has long attracted attention. A problem description for LLM-based code generation is a special case of software requirements. Pre–ChatGPT work includes NLP-based detection of inconsistency and vagueness [32], [33], [34], [35] in software requirements, while recent efforts address semantic uncertainty in the ChatGPT era [36]. Vijayvargiya et al.[1] quantify the impact of ambiguous LLM inputs, showing up to a 20% degradation in LLM performance. ClarifyGPT[2] measures uncertainty in generated code to trigger clarifying questions; although promising, it often asks irrelevant questions and fails to use embedded examples. SPECFIX overcomes these shortcomings by repairing the induced program distribution first and then inferring concise description edits via contrastive specification inference.

Prompt Optimization: Prompt optimization is closely related to problem descriptions repair. Ma et al. [37] show that reflection-based prompt tuning often fails to identify root causes of prompt errors. SPECFIX addresses this by analyzing and repairing program distributions.

Confidence and Uncertainty: Hou et al. [12] show a connection between task ambiguity with aleatoric uncertainty, i.e. uncertainty from inherent randomness in the data-generating process. The uncertainty can be estimated via semantic entropy [4], which in code generation can measured by partitioning generated program into equivalence classes via differential testing [14], [6], [2], a method adapted by SpecFix.

Specification Inference with LLMs: SpecRover [38] infers program specifications expressed in natural language to generate bug fixes. Other work applies LLMs to infer formal specifications from text [39], [40]. SPECFIX's key novelty is contrastive specification inference, which infers specifications that differentiate two programs.

Automated Program Repair: Program repair [5] modifies a given program to meet given correctness criteria. Approaches include heuristic-based [41], semantics-based [42], and ML/LLM-based techniques [43], [44], [45]. Our innovation is to repair problem descriptions instead of programs, however, inspired by Fan et al. [9], we use program repair as a part of our algorithm.

VIII. CONCLUSION

This paper introduces the problem of automated repair of ambiguous problem descriptions for LLM-based code generation. It shows that descriptions can be repaired in a fully-automated fashion by aligning natural language with input-output examples and reducing uncertainty of code generation. SPECFIX accomplishes this by analyzing and repairing the distribution of programs the description induces, and then mapping the change back to the description. Our experimental evaluation shows that repaired problem descriptions significantly improve code generation performance of SOTA LLMs.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments. We also thank Dimitrios Bouras and Tatsan Kantasit for insightful discussions.

REFERENCES

[1] S. Vijayvargiya, X. Zhou, A. Yerukola, M. Sap, and G. Neubig, "Interactive agents to overcome ambiguity in software engineering," *arXiv preprint arXiv:2502.13069*, 2025.

[2] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: Empowering llm-based code generation with intention clarification," *arXiv preprint arXiv:2310.10996*, 2023.

[3] V. Le and S. Gulwani, "Flashextract: A framework for data extraction by examples," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 542–553.

[4] L. Kuhn, Y. Gal, and S. Farquhar, "Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation," *arXiv preprint arXiv:2302.09664*, 2023.

[5] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[6] Z. Fan, H. Ruan, S. Mechtaev, and A. Roychoudhury, "Oracle-guided program selection from large language models," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 628–640.

[7] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[8] Z. Tian, J. Chen, and X. Zhang, "Fixing large language models' specification misunderstanding for better code generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 645–645.

[9] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[10] S. Srivastava, S. Gulwani, and J. S. Foster, "Template-based program verification and program synthesis," *International Journal on Software Tools for Technology Transfer*, vol. 15, pp. 497–518, 2013.

[11] B. Gleich, O. Creighton, and L. Kof, "Ambiguity detection: Towards a tool explaining ambiguity sources," in *Requirements Engineering: Foundation for Software Quality: 16th International Working Conference, REFSQ 2010, Essen, Germany, June 30–July 2, 2010. Proceedings 16*. Springer, 2010, pp. 218–232.

[12] B. Hou, Y. Liu, K. Qian, J. Andreas, S. Chang, and Y. Zhang, "Decomposing uncertainty for large language models through input clarification ensembling," *arXiv preprint arXiv:2311.08718*, 2023.

[13] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 2017, pp. 615–632.

[14] A. Sharma and C. David, "Assessing correctness in llm-based code generation via uncertainty estimation," *arXiv preprint arXiv:2502.11620*, 2025.

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

[16] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "Cycle: Learning to self-refine the code generation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 392–418, 2024.

[17] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.

[18] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei *et al.*, "Qwen2.5 technical report," *arXiv preprint arXiv:2412.15115*, 2024.

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

[20] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[21] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[22] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," in *The Thirteenth International Conference on Learning Representations*, 2025.

[23] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[24] C. S. Xia, Y. Deng, and L. Zhang, "Top leaderboard ranking= top coding proficiency, always? evoeval: Evolving coding benchmarks via llm," *arXiv preprint arXiv:2403.19114*, 2024.

[25] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[26] B. Iglewicz and D. C. Hoaglin, *Volume 16: how to detect and handle outliers*. Quality Press, 1993.

[27] G. Uddin, F. Khomh, and C. K. Roy, "Mining api usage scenarios from stack overflow," *Information and Software Technology*, vol. 122, p. 106277, 2020.

[28] T. Hirsch and B. Hofer, "Identifying non-natural language artifacts in bug reports," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 191–197.

[29] L. Fan, M. Chen, and Z. Liu, "Self-explained keywords empower large language models for code generation," *arXiv preprint arXiv:2410.15966*, 2024.

[30] J. Li, G. Li, L. Wang, H. Zhu, and Z. Jin, "Generating equivalent representations of code by a self-reflection approach," *arXiv preprint arXiv:2410.03351*, 2024.

[31] C. Liu, S. D. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Codemind: A framework to challenge large language models for code reasoning," *arXiv preprint arXiv:2402.09664*, 2024.

[32] S. Ezzini, S. Abualhaija, C. Arora, and M. Sabetzadeh, "Automated handling of anaphoric ambiguity in requirements: a multi-solution study," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 187–199.

[33] D. Luitel, S. Hassani, and M. Sabetzadeh, "Using language models for enhancing the completeness of natural-language requirements," in *International working conference on requirements engineering: foundation for software quality*. Springer, 2023, pp. 87–104.

[34] A. Ferrari and A. Esuli, "An nlp approach for cross-domain ambiguity detection in requirements engineering," *Automated Software Engineering*, vol. 26, no. 3, pp. 559–598, 2019.

[35] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, and S. Gnesi, "Detecting requirements defects with nlp patterns: an industrial experience in the railway domain," *Empirical Software Engineering*, vol. 23, no. 6, pp. 3684–3733, 2018.

[36] A. Fantechi, S. Gnesi, L. Passaro, and L. Semini, "Inconsistency detection in natural language requirements using chatgpt: a preliminary evaluation," in *2023 IEEE 31st International Requirements Engineering Conference (RE)*. IEEE, 2023, pp. 335–340.

[37] R. Ma, X. Wang, X. Zhou, J. Li, N. Du, T. Gui, Q. Zhang, and X. Huang, "Are large language models good prompt optimizers?" *arXiv preprint arXiv:2402.02101*, 2024.

[38] H. Ruan, Y. Zhang, and A. Roychoudhury, "Specrover: Code intent extraction via llms," *arXiv preprint arXiv:2408.02232*, 2024.

[39] L. Ma, S. Liu, Y. Li, X. Xie, and L. Bu, "Specgen: Automated generation of formal program specifications via large language models," *arXiv preprint arXiv:2401.08807*, 2024.

[40] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1889–1912, 2024.

[41] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[42] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[43] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.
[44] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.
[45] N. Parasaram, H. Yan, B. Yang, Z. Flahy, A. Qudsi, D. Ziaber, E. Barr, and S. Mechtaev, "The fact selection problem in llm-based program repair," *arXiv preprint arXiv:2404.05520*, 2024.