SpecGen: Automated Generation of Formal Program Specifications via Large Language Models

Lezhi Ma
Nanjing University
China
lezhima@hotmail.com

Shangqing Liu†
Nanjing University
China
shangqingliu666@gmail.com

Yi Li
Nanyang Technological University, Singapore
yi_li@ntu.edu.sg

Xiaofei Xie
Singapore Management University, Singapore
xfxie@smu.edu.sg

Lei Bu†
Nanjing University
China
bulei@nju.edu.cn

***Abstract*—In the software development process, formal program specifications play a crucial role in various stages, including requirement analysis, software testing, and verification. However, manually crafting formal program specifications is rather difficult, making the job time-consuming and labor-intensive. Moreover, it is even more challenging to write specifications that correctly and comprehensively describe the semantics of complex programs. To reduce the burden on software developers, automated specification generation methods have emerged. However, existing methods usually rely on predefined templates or grammar, making them struggle to accurately describe the behavior and functionality of complex real-world programs.**

To tackle this challenge, we introduce *SpecGen*, a novel technique for formal program specification generation based on Large Language Models (LLMs). Our key insight is to overcome the limitations of existing methods by leveraging the code comprehension capability of LLMs. The process of *SpecGen* consists of two phases. The first phase employs a conversational approach that guides the LLM in generating appropriate specifications for a given program, aiming to utilize the ability of LLM to generate high-quality specifications. The second phase, designed for where the LLM fails to generate correct specifications, applies four mutation operators to the model-generated specifications and selects verifiable specifications from the mutated ones through a novel heuristic selection strategy by assigning different weights of variants in an efficient manner. We evaluate *SpecGen* on two datasets, including the SV-COMP Java category benchmark and a manually constructed dataset containing 120 programs. Experimental results demonstrate that *SpecGen* succeeds in generating verifiable specifications for 279 out of 385 programs, outperforming the existing LLM-based approaches and conventional specification generation tools like Houdini and Daikon. Further investigations on the quality of generated specifications indicate that *SpecGen* can comprehensively articulate the behaviors of the input program.

***Index Terms*—program verification, specification inference, large language model**

I. Introduction

Formal specifications play a central role in describing, understanding, and reasoning about program behaviors. They capture the intended or actual program behaviors, in terms of formal languages, with precise semantics. Formal specifications may take various forms, such as procedure pre-/post-conditions, loop invariants, and assertions at specific program locations. They are essential in a variety of software quality assurance tasks, including software testing [1, 2], model checking [3, 4], and program verification [5, 6].

Yet, a practical challenge is the absence of documented formal specifications in most real-world software projects, since manually writing high-quality specifications is highly nontrivial. To alleviate the burden on software developers, several tools have been introduced for generating program specifications automatically [7, 8, 9], including Houdini [7] and Daikon [9], two most representative ones for Java programs. However, these tools rely heavily on predefined templates or grammars during the specification generation process. As claimed by Molina et al. [10], the fixed templates involved result in a limited range of specifications covered, usually yielding overly simplistic specifications that struggle to capture the complex behaviors and functionalities of real-world programs accurately. This phenomenon poses non-negligible limitations for these tools, consequently hindering their applications in the actual software development process [11, 12, 13].

To address this challenge, we introduce *SpecGen*, an automated technique for Java program specification generation based on the Large Language Models (LLMs). With the rise of LLMs, extensive research has attempted to apply them in software engineering and LLMs exhibit outstanding performance in various tasks [14, 15, 16, 17, 18], where LLMs have demonstrated remarkable capabilities on code comprehension and summarization [19]. Inspired by this insight, we believe that LLMs can serve as a potent solution to overcome the limitations of existing program specification generation methods. The core idea of this work is to leverage LLMs to generate specifications that accurately capture the real behaviors of input programs, thus imbuing these specifications with richer semantics for further practical use.

The workflow of *SpecGen* comes in two phases. In the first phase, *conversation-driven specification generation*, we aim to query the output specifications by conducting a conversation with the LLM. To start the conversation, a prompt is constructed with several few-shot examples for the initial query. During the conversation process, we utilize the verification failure information from the specification verifier as

† Corresponding author.

the feedback prompt for the next round of the conversation. In this way, LLMs receive more cues, facilitating them to better generate accurate specifications. Nevertheless, despite the powerful code understanding and generation capabilities of large language models, they still struggle to handle complex programs effectively i.e., generating accurate specifications for complex programs. Through our repeated observation and testing of the model-generated results, we found that although the generated content is not highly accurate, it is already very close to the oracle, which motivates us to design the second phase, *mutation-based specification generation*. It focuses on generating accurate specifications where the LLM fails to provide verifiable results. Specifically, given a verification failure result by the LLM, *SpecGen* endeavors to combine four different kinds of mutation operators to modify it and obtain all potential variants. A selector adopting a heuristic selection strategy by assigning different weights of variants further repeatedly chooses a subset of these mutated variants deemed most likely to pass the verification until the results are successfully verified.

To evaluate *SpecGen*, we conduct experiments on two datasets. We first evaluate *SpecGen* on the benchmark for the Java category of SV-COMP [20]. To further evaluate the performance of *SpecGen* on different kinds of programs, we constructed another dataset containing 120 Java programs with manually written ground-truth specifications. The selected programs are highly representative, encompassing different control-flow structures and various data structures to avoid any bias in our evaluation. We compared the performance of *SpecGen* on the dataset against multiple baselines. The results of our evaluation demonstrate that *SpecGen* significantly outperforms the baseline methods. *SpecGen* successfully generated verifiable specifications for 279 out of the total 385 programs, outweighing 247 for AutoSpec [21], the best-performing LLM-based approach, and 98 for Houdini, the best-performing non-LLM method. An ablation study on mutations was also conducted, proving the effectiveness of all four types of mutation operators. Additionally, the results of evaluations on the heuristic selection strategy suggested that our strategy effectively improves the efficiency of *SpecGen* compared to the random selection strategy. Furthermore, a user study was conducted to evaluate the semantic quality of the generated specifications, illustrating the ability of *SpecGen* to accurately and comprehensively characterize program behaviors. The main contributions are summarized as follows:

- A novel approach for formal program specification generation and corresponding prototype tool [22], leveraging the Large Language Models to generate accurate and comprehensive specifications to describe program behaviors. Benefiting from the code comprehension ability of LLMs, our approach is capable of generating specifications with high quality, overcoming the limitations of existing methods in generating simplistic and basic specifications.
- A mutation-based generation approach to enrich the diversity of the LLM output, consisting of a set of mutation operators and a novel heuristic selection strategy proposed to improve the efficiency of the verification that existing works fail to consider.
- A dataset named *SpecGenBench*, with hand-written specifications by experts, facilitating follow-up research. Other than the established benchmark SV-COMP, we collected programs on a more diverse spectrum for deeper insights.
- A comprehensive evaluation to evaluate our approach in all aspects. We compare *SpecGen* against Purely LLM-based approaches and representative non-LLM approaches. *SpecGen* succeeds in 279 out of the 385 programs, significantly outperforming the baseline approaches.

II. Background and Motivation

A. Specification Generation and Verification

Program specifications encompass precise statements that describe the intended or actual behaviors of a particular program, either in its entirety or in distinct parts. In this work, we focus on generating specifications for the actual behaviors of input programs. A large proportion of program specifications are expressed in formal languages, such as mathematical expressions to describe the constraints on the behaviors of a program. There are different kinds of specifications such as pre-conditions, which establish constraints on function parameters, ensuring proper execution of the function, post-conditions, which delineate the properties of a set of variables that persist after a function is executed, and loop invariants, which represent a specialized form of specification, detailing properties that consistently hold before executing the loop body. For different programming languages, the specifications may have different implementation forms. For example, in Java, the specifications can be expressed in Java Modeling Language (i.e., JML) [23] where `requires` statements denote the pre-conditions of a function, `ensures` statements represent the post-conditions of a function, and `maintaining` statements specify the loop invariants.

A series of automated program specification generation tools have been developed [7, 8, 9, 24, 25] to reduce the burden on software developers. Two representative works are Houdini [7] and Daikon [9]. Both rely on templates defined by human experts to generate a massive amount of candidate specifications, which verifiers then filter to eliminate incorrect candidate specifications until the remaining candidates are successfully verified. In particular, the template usually involves two or three variables and their corresponding operators i.e., `<var> <op> <var>`, where `<var>` should be filled in with variable names and `<op>` should be an operator. For example, if a function contains two integer parameters `x` and `y` and an integer return value, Houdini may generate candidate pre-conditions and post-conditions for this function in the form of `x > y`, `x < y`, `\result >= 0`, `\result <= 0`, etc. where `\result` is the defined variable denoting the return value in JML. For a program, all available variables within the scope such as class members, function parameters, and return values are taken to generate candidate specifications based on the defined templates and instrumented into corresponding

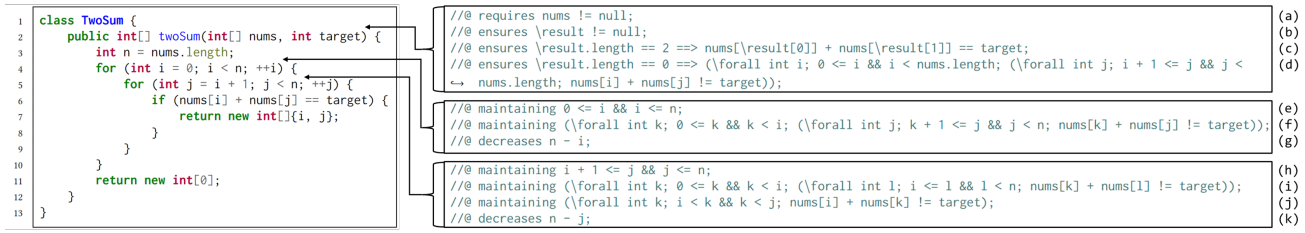

Fig. 1: An example program and corresponding specifications generated by *SpecGen*, for which existing tools cannot generate comprehensive specifications to describe the program behaviors.

points of the input program to verify the correctness of the specifications. The main difference between Houdini and Daikon lies in the design of the verifier where Houdini adopts a JML specification verifier, OpenJML [26], which is designed from constraint solving [27] for verifying. Yet, Daikon is based on the runtime checking which compares each candidate specification with the runtime execution traces.

B. Motivation

The existing automated program specification generation tools have limitations hindering real-world deployment and application. They rely on templates defined by human experts to generate specifications, which results in simple and trivial specifications. We present an example program on the left of Figure 1 for illustration. This program aims to search in the given integer array for the indexes of two separated elements of which the sum is exactly the given target value, and is implemented in two nested loops. If there do not exist such elements in the array, the program returns an empty array. To fully articulate the behaviors of the program, such properties must be specified, where Daikon and Houdini fail. In particular, both Houdini and Daikon can only generate trivial post-conditions such as `nums != null` and `\result[i] >= 0` for the method `TwoSum()` as a whole. As for the outer-layer loop, only some simple loop invariants are generated, describing trivial numerical relationships between variables, such as `i >= 0` and `i < arr.length`. For the inner-layer loop, the generated specifications are `j >= 1`, `i < j` and `j < nums.length`, which are similar to the out-layer. The generated specifications are too trivial, without detailed information to accurately capture the program's functionality.

Recently, large language models (i.e., LLMs) [28, 29] have exhibited powerful capacities in coding [18, 19]. The emergence of these models may greatly compensate for the limitations of traditional software analysis tools in code understanding. A significant amount of work attempts to leverage large language models in software engineering [30, 31, 32, 33, 34] and we have witnessed substantial progress brought about by the introduction of LLMs. Inspired by these works, in this work, we aim to leverage the large language models in the automated generation of formal program specifications to address the limitations of conventional template-based approaches. From this perspective, we innovate our approach *SpecGen*, which generated three parts of specifications for the example presented in the right part of Figure 1. The first part is to describe the method `TwoSum()` as a whole, specifying its pre-conditions and post-conditions. The specifications in lines a and b claim that the input array must not be null before and after the method is executed. The post-condition in line c specifies that the target value equals the sum of the two elements corresponding to the indexes stored in the returned array. The post-condition at line d specifies that there does not exist such a pair of elements that satisfies the constraint when the length of the returned array is zero. These generated specifications can fully articulate the functionality of method `TwoSum()`. Furthermore, the loop invariants in the second and third parts specify corresponding constraints that must be met within a certain range in the array. These specifications generated by *SpecGen* comprehensively describe the semantics of this function and their correctness is verifiable.

III. Approach

A. Overview

The overview of *SpecGen* is presented in Figure 2, which consists of two components i.e., conversation-driven specification generation and mutation-based specification generation. The former is designed to communicate with the large language model to query the output in a conversational manner. In particular, a prompt is constructed with some few-shot examples for the initial query. The verification failure information provided by the verifier is further used as the prompt for the next round of conversation if the model-generated results are incorrect. The conversation will be repeated iteratively until the generated specifications successfully pass the verifier or a maximum number of iterations is reached. The latter aims at generating the specifications of a program that the large language model fails to generate. Four kinds of mutation operators are adopted to mutate the specification that failed verification by the verifier and obtain all potential variants. A heuristic selector is further designed to efficiently choose a set of mutated variants most likely to pass verification.

B. Conversation-Driven Specification Generation

Engaging in conversation with large models can fully leverage their capabilities, better assisting them in generating the desired content and avoiding potential errors [35]. Inspired by Xia et al. [15], we propose our conversation-driven specification generation in *SpecGen* to interact with the LLM conversationally to generate specifications. There are two main benefits: firstly, the conversational manner aids

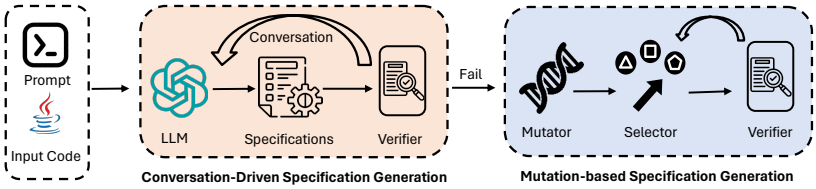

Fig. 2: Overview of our *SpecGen*.

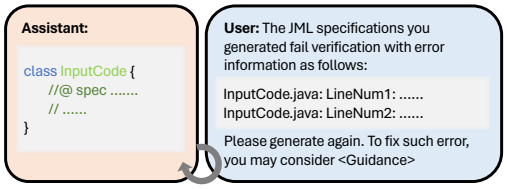

Fig. 3: Conversational generation.

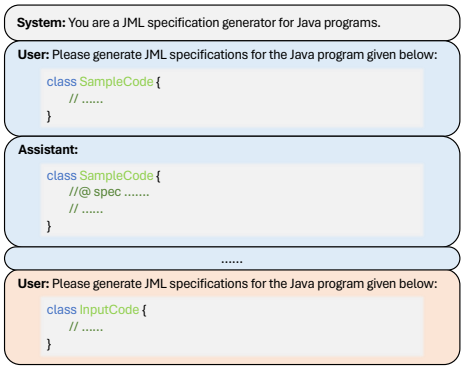

Fig. 4: Illustration of the initial prompt construction.

the large model in automatically correcting potential syntax errors in the generated content, secondly, providing the model with the verification failures information by the verifier in the conversation helps it generate more accurate specifications. The design mainly consists of two sequential components: initial prompt construction, which pre-defined an initial prompt to prepare for querying with the LLM, and conversational specification generation, which communicates to the LLM by incorporating verification failure information produced by the verifier in the conversation manner to generate verifiable specifications. The conversation will be repeated iteratively until the generated specifications pass the verifier or a maximum number of iterations is reached.

1) Initial Prompt Construction: We need to define the initial prompt to query with LLM to obtain the model output. After multiple attempts to assess the impact of different prompts on the quality of generated program specifications, we ultimately chose to follow Xia et al. [36] in designing our prompt. The prompt is presented in Fig. 4 illustrating the components of the initial prompt, which consists of three different parts: the system role, few-shot examples, and the queried program. The system's role aims to inform LLMs about our application scenario, which is to generate JML specifications. We further add some few-shot examples. The reasons are two-fold. On one hand, few-shot examples can help the model to generate more accurate outputs [37]. On the other hand, LLMs can generate the desired output format that is learned from these examples. Each example is a pair of a program and its corresponding specifications. We randomly select it from our collected dataset to construct the few-shot examples. The last component is the queried program which requires the model to generate the output.

2) Conversational Specification Generation: Given the initial prompt, LLM can obtain the initial output for the input program. As the output of the LLM in the first attempt may not successfully pass the validation of the verifier, we interleave the process of specification generation with verification failure feedback to prompt future generation in a conversational manner which is illustrated in Fig. 3. In particular, each generated specification by the model is verified by a JML verifier to test whether the generated result can pass the verifier. If the verification fails, we construct feedback information using the reported error message from the verifier as the prompt for the next generation. The verification error message can help the model understand the reason for failure and provide guidance for generating correct specifications. In addition, to avoid the verifier providing excessively long error messages, we configure the verifier to report only one verification failure message per attempt. Furthermore, through a massive amount of experiments, we summarize several types of common verification failures reported by the verifier. For each kind of error, we provide guidance in the natural language to facilitate the model in generating correct specifications. Upon encountering these types of verification failures reported by the verifier, we will insert corresponding guidance information into the prompt e.g., <Guidance> in Fig. 3 to assist the model in resolving the issues. The conversation will be repeated iteratively until the specifications are successfully verified or a maximum number of iterations is reached.

C. Mutation-based Specification Generation

In the conversation-driven generation process, some few-shot examples are provided in the initial prompt to start the query with the LLM. To further stimulate the potential of the LLM, multi-turn conversation continually guides the LLM in approaching the accurate specification more closely. Yet, they still struggle to generate fully correct specifications for some complex programs. The reasons are two-fold. On the one hand, in comparison to code generation [37, 38, 39], specification generation poses greater challenges for LLMs due to the limited corpora related to program specifications for the model to learn from. On the other hand, although the verification failure information provided by the verifier can assist LLMs in providing higher-quality responses to some extent, as the error messages are highly abstract and generalized, LLMs still struggle to accurately understand the semantic information within error messages for complex programs. While LLMs may not accurately generate specifications for complex programs, the generated results are already highly close to the oracle, inspiring us to design mutation-based generation.

In particular, the mutation-based specification generation component takes the output generated by the large language

model that fails to pass the verifier through the multi-round conversation as the input. We further define a set of mutation operators to modify these generated outputs to obtain more diverse results. Then a heuristic strategy is adopted for efficient verification. The workflow is presented in Algorithm 1. Specifically, we define the specifications generated by LLM that fail verification as the set of template specifications E_t, which consists of different specifications generated for different locations in a program, and a set of mutation operators as M. The `MutationBasedGen` takes E_t and M as the input and outputs a set of correct specifications as E. The function `SpecMutation` corresponds to the mutation operation of E_t, where each kind of mutation operator will be performed through the mutation function $Mutate()$ (Section III-C1) on a template specification e ($e \in E_t$) to obtain a set of candidates $E_{mutated}$ (lines 7 and line 8).

After the mutation operations are performed, we further design the specification selection algorithm to select a subset $E_{selected}$ of mutated specifications that can pass the verification. The selected subset $E_{selected}$ is initialized with E_t. We then iteratively require the verifier to check the correctness of $E_{selected}$ and obtain a set of refuted specifications denoted as $E_{refuted}$ from $E_{selected}$ that the verifier fails to verify. After that, we need to replace the failed specifications from $E_{refuted}$ with another mutated variant for the next iteration of verification. The $ReSelect$ function is presented from line 18 to line 25. For each refuted specification $e_r \in E_{refuted}$, we first remove it from the mutation set $E_{mutated}$ and the selected set $E_{selected}$. Then we replace e_r with another mutated variant e that comes from the same family by a heuristic selection strategy (Section III-C2) from line 22 to line 23. Here the family refers to a set of mutated specifications E_f that come from the same template specification. Finally, we add e to the selected set $E_{selected}$ to prepare for the next iteration of verification. The above process will be repeated until all candidates in $E_{selected}$ are successfully verified i.e., $E_{refuted}$ is empty (line 16). Note that if all candidates are refuted, the process will finally select an empty set of candidates, guaranteeing the termination of the whole process.

1) Template Specification Mutation: As shown in Table I, we define four kinds of mutation operators including predicative, logical, comparative, and arithmetic. Each type of mutation corresponds to one type of operator supported by JML. LLMs perform well in formulating the overall syntactical structure of specifications, but they often make mistakes in grasping the fine-grained relationships between variables, resulting in incorrect operators used to describe the relationships between variables, which is why our mutation design is centered around the operators. A mutation operation substitutes the operators of the corresponding type in the specification with another of the same type. For example, after applying a predicative mutation, a `\exists` predicate within a specification may be substituted with `\forall`. Note that the mutation for a certain type of operator does not necessarily create only one mutated candidate. For example, the expression `a <= b` may be mutated to `a < b` or `a - 1 <= b`. If multiple

Algorithm 1: Mutation-based Specification Generation

Input : Set of template specification E_t, set of mutations M
Output: Set of verified specifications E

1 **Function** `MutationBasedGen`(E_t, M)
2 $E_{mutated} = \text{SpecMutation}(E_t, M)$
3 $E = \text{SpecSelection}(E_{mutated}, E_t, M)$
4 **return** E
5 **Function** `SpecMutation`(E_t, M)
6 $E_{mutated} = \emptyset$
7 **for** $e \in E_t$ **do**
8 $E_{mutated} = E_{mutated} \cup \text{Mutate}(e, M)$
9 **return** $E_{mutated}$
10 **Function** `SpecSelection`$(E_{mutated}, E_t, M)$
11 $E_{selected} = E_t$
12 $E_{refuted} = \emptyset$
13 **repeat**
14 $E_{refuted} = \text{Verify}(E_{selected})$
15 $E_{selected} = \text{ReSelect}(E_{selected}, E_{mutated}, E_{refuted}, M)$
16 **until** $E_{refuted}$ *is* $\emptyset$;
17 **return** $E_{selected}$
18 **Function** `ReSelect`$(E_{selected}, E_{mutated}, E_{refuted}, M)$
19 **for** $e_r \in E_{refuted}$ **do**
20 $E_{mutated} = E_{mutated} \backslash \{e_r\}$
21 $E_{selected} = E_{selected} \backslash \{e_r\}$
22 $E_f = \text{GetFamilyOf}(e_r, E_{mutated})$
23 $e = \text{SelectByHeuristic}(E_f, M)$
24 $E_{selected} = E_{selected} \cup \{e\}$
25 **return** $E_{selected}$

TABLE I: The defined mutation operators.

Mutation Type	Original Operator	Mutated Operators
Predicative	\forall	\exists
	\exists	\forall
Logical	&&	||
	||	&&
	<==>	<==, ==>
	==>	<==
	<==	==>
Comparative	<=	<, "- 1 <="
	>=	>, "+ 1 >="
	<	<=
	>	>=
	==	!=
	!=	==
Arithmetic	+	-
	-	+

mutations can be applied to a specification at the same time, we try to exhaust each combination of different types of mutations to get all potential variants. Since the set of all potential variants of a certain template is determined, the exhaustive searching process is deterministic. For instance, the expression `x < n + 1` can be mutated to `x <= n + 1` from the comparative type, `x < n - 1` from the arithmetic type, or `x <= n - 1` by combining them.

2) Mutated Specification Selection: Typically, for a program, Houdini [7] verifies all generated specifications at one time. However, similar practice cannot be applied in *SpecGen* as we exhaust all potential combinations of mutations for a template specification. The set of the obtained specifications for verification is considerably large, posing a much greater burden for the verifier within a single verification process. To address this challenge, we innovate a heuristic selection strategy to improve the stability and efficiency of verification.

In general, the heuristic selection algorithm finds a specifi-

cation $\hat{e}$ such that

$$\hat{e} = \arg\max_{e \in E_f} \sum_{m \in M} (times(m, e, e_t) \cdot weight(m)) \quad (1)$$

where E_f denotes a family of mutated specifications that come from the same template specification e_t, and M denotes the set of all mutations. Given E_f, e_t, and M, we design the heuristic selection logic to prioritize selecting important candidates for verification. In particular, we assign scores for each mutated candidate $e \in E_f$ and select the candidate with the highest score as the output. To calculate the score of a candidate e, for all types of mutations $m \in M$, we sum up all the values of $times(m, e, e_t)$ multiplied by $weight(m)$, where $times(m, e, e_t)$ calculates how many times the mutation m is performed when e_t mutates into e, and $weight(m)$ denotes the corresponding weight of m.

IV. Experimental Setup

We design the following four research questions for evaluation:

- **RQ1:** How does *SpecGen* compare with the baseline approaches?
- **RQ2:** How does each type of mutation contribute to the effectiveness of *SpecGen*?
- **RQ3:** How do different candidate selection strategies affect the efficiency of *SpecGen*?
- **RQ4:** To what extent can the generated specification contain the semantic information of the input program?

A. Implementation

We use the API provided by OpenAI [40] to communicate with the large language model of `gpt-3.5-turbo-1106` for the experiments. Temperature is set to 0.4 to balance the diversity and rigorousness of the outputs of GPT. 4 few-shot examples are used during the prompt construction to balance the input length and response time. The maximum number of rounds of conversation is set to 10. The verifier is OpenJML [26], the most recent JML specification verification tool to check the consistency between Java source code and JML specifications. Due to the incompleteness in the implementations of OpenJML, we set a timeout limit of 30 minutes for a single verification in our implementation to avoid unexpected situations, such as the non-responding of OpenJML. All experiments are conducted on an 8-core workstation with Intel Core i7-12700H CPU @2.30GHz and 32GB RAM, running Ubuntu 22.04.3 LTS. The version of OpenJDK is 1.8.0_371 for all experiments except for Houdini, which has to run under OpenJDK 1.6.0_45. We set the weight of comparative, logical, arithmetic, and predicative mutation to -1, -2, -4, and -4 respectively as the comparative mutation is more likely to pass the verification followed by the logical mutation. The predicative and arithmetic mutations are the least important through our observations from extensive experiments. Note that the weights are defined with negative values, leading to negative calculated scores as well. The reason for the design is to prioritize the specification candidates with fewer mutations.

B. Dataset

To comprehensively evaluate the effectiveness of *SpecGen*, following previous work [41], we first use an established dataset, the benchmark of SV-COMP [20], for evaluation. Specifically, we used 265 class definitions in the Java category of SV-COMP benchmark and conducted the necessary modifications on part of these programs (referred to as Dataset SV-COMP hereinafter) for ease of evaluation. The remaining data in the benchmark cannot be applied for specification generation even with our modification. We made minimal modifications to SV-COMP programs to ensure they can be executed outside the SV-COMP environment. Specifically, the programs destined to trigger false assertions have to be modified so that the programs can exit properly. Also, those library calls specific to the competition settings (e.g. `Verifier.nondetInt()`) are replaced with equivalent Java library calls so that they can be successfully compiled. It is ensured that the semantics of the modified programs remain unchanged. As calculated by tool JaCoCo [42], these programs have an average line of code (LoC) of 22.51, along with an average cyclomatic complexity (CC) of 6.18. However, after a deep analysis of the characteristics of the data from SV-COMP, we find that 88.7% programs are loop-free, indicating that the samples with more complex program structures cannot be covered by this dataset, inducing limitations on the evaluation. Also, very few datasets have been established specifically for specification generation tasks so far.

To remedy this gap, we further collect another dataset, *SpecGenBench*, containing 120 samples as a supplement where 20 programs (including the corresponding specifications) from the dataset constructed by Nilizadeh et al. [43] and 100 programs from LeetCode [44]. The selected programs are assured of the feasibility of expressing their behaviors as JML-specified verifiable specifications. These programs involve a variety of control flow structures and encompass multiple data structures such as arrays, strings, and other data structures supported by the Java library. They also cover a diverse set of specifications including post-conditions and loop invariants, involving both linear and nonlinear relationships between variables, making them representative of a broad spectrum of scenarios. They can be categorized into five categories according to their types of control flow structures [45]. Specifically, *Sequential* denotes the programs without branches or loops. *Branched* represents the loop-free programs that will contain branches like if-else or switch structures. *Single-path Loop* contains the simplest type of loop, with only one layer of loop structure without branches in their loop bodies. In contrast, *Multi-path Loop* denotes the loops that have branches in the loop bodies. Lastly, *Nested Loop* denotes the programs with multiple layers of loop structure where each layer may have a branch. The quantity of programs for *Sequential*, *Branched*, *Single-path Loop*, *Multi-path Loop* and *Nested Loop* is 26, 23, 24, 26, and 21, respectively. Programs in *SpecGenBench* have an average LoC of 20.77 and an average CC of 6.60.

To obtain the ground truth specifications for the 100 Java programs from LeetCode, we follow a similar procedure

in Nilizadeh et al. [43] with the help of human experts. Three experts with rich experience in formal verification were employed, to manually write specifications for each program. Each expert is required to write the specifications that can be successfully verified to describe the functionality and behavior of the program as accurately and comprehensively as possible. For a single program, if multiple experts have written verifiable specifications, another expert is responsible for selecting one of them as the ground truth.

As Daikon [9] requires a set of test suites instrumented into the source code to execute the code, we manually write these test suites for it. A small test suite is initialized first, on which Daikon is invoked to generate specifications. If the results fail verification, the counterexample produced by the verifier will be added to the test suite. The procedure is repeated until no new specifications are generated. The test suites achieve an average instruction coverage of 90.16%, branch coverage of 87.98%, and line coverage of 91.36%. Lastly, we instrument dummy function calls at the top of each loop body in a program to ensure Houdini and Daikon can generate the specifications at these program points.

C. Baselines

We select two conventional approaches and several LLM-based approaches as the baselines for comparison.

Houdini [7]. It is a template-based JML annotation generator that relies on a series of pre-defined templates to generate candidate specifications. Given the input program, it first generates candidate specifications by filling in the templates with available variables and all kinds of operators. Afterward, It iteratively invokes a JML specification verifier to check their correctness and removes the refuted ones. The process will be repeated until the remaining candidates are all verified.

Daikon [9]. It is a classic tool for the dynamic detection of program specifications which relies on the dynamic execution trace of the target program to infer likely specifications. Given the input program, it first instruments the target program to trace certain variables and extracts execution traces. Then the inference engine reads the trace data and infers the potential invariants with a generate-and-check algorithm. Daikon supports dynamic detection for Java, C/C++, C#, and Perl programs along with various formats such as DBC format [46], JML format [23], and CSharpContract format [47].

Apart from the conventional approaches for specification generation, we further add the LLM-based approaches.

Few-shot LLM. They refer to the large language model i.e., `gpt-3.5` with few-shot settings in our work to generate specifications. Under the few-shot settings, the LLM is queried only once to obtain the final result. We set 0-shot, 2-shot, and 4-shot for few-shot comparison.

Conversational. It refers to the generation technique described in Section III-B. Conversational generation iteratively queries the LLM to refine its results, with error information provided to the LLM as feedback on each iteration. The conversational setting is based on 4-shot examples. Other settings remain the same with *SpecGen*.

AutoSpec [21]. It is a recent technique for specification generation combining LLMs and static analysis. AutoSpec first decomposes the input program into its components, upon which a hierarchy graph is built. For each component, AutoSpec queries the LLM for corresponding specifications respectively. Eventually, specifications for all components are combined to obtain the overall result, which is presented to a verifier for correctness validation. The progress is repeated iteratively until the result is successfully verified.

D. Evaluation Metrics

Following the previous works [41, 48], we use the metric of *Number of Passes* for assessment. We further add more metrics for comprehensive evaluation.

Number of Passes. It defines the number of programs for which the generated specifications of an approach pass the validation by the verifier. For a program, we consider the specifications that pass the verifier as the correct specifications.

Success Probability. It is used to evaluate the model-based approaches. The randomness inherent in the content generated by large language models may introduce a certain level of contingency in a successful generation. Thus, we use the success probability for the measurement. For a test program, it is calculated by $\frac{N_{sucess}}{N_{attempt}}$ where N_{sucess} denotes the number of successful generations of verifiable specifications and N_{attemp} denotes a fixed number of trials in total (10 times in *SpecGen*).

Number of Verifier Calls. It is used to evaluate the efficiency of our approach. We propose a heuristic selection algorithm to prioritize the important candidates for the verifier to verify. To evaluate the effectiveness of the proposed selection algorithm, we use the number of verifier calls as the evaluation metric.

User Rating. It aims to measure the semantic quality of the generated specifications. We invited 15 Ph.D. students who are experts in Java programming language to rate the specifications generated by different approaches. The research is conducted using the Likert Scale [49], where students are required to give a rating from one point to five points for each case according to a reference rating criteria.

V. Experimental Results

A. RQ1: Comparison with Baselines

The experimental results are presented in Table II where *Num.* denotes the number of successfully handled programs, and *Prob.* denotes the average success probability. Each LLM-based approach is granted 10 trials for each program.

Performance on SV-COMP. From Table II, we can find that on the SV-COMP dataset (265 programs in total), Houdini handled 56 programs, which is more than Daikon. However, both underperform LLM-based approaches even in LLM's simplest setting i.e. 0-shot setting, which handled 81 programs, demonstrating the feasibility of employing LLMs for formal program specification generation. Among the LLM-based approaches with few-shot settings, as the number of given few-shot examples increases, the number of programs that LLM can generate verifiable specifications is also increased, substantiating the effectiveness of the few-shot examples. Based

TABLE II: Number of programs that successfully pass the verifier and average success probability.

Approach		SV-COMP (265)		*SpecGenBench*										Overall (385)	
				Sequential (26)		Branched (23)		Single-path Loop (24)		Multi-path Loop (26)		Nested Loop (21)			
		Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.
Daikon		51	-	10	-	10	-	0	-	1	-	0	-	72	-
Houdini		56	-	14	-	11	-	10	-	4	-	3	-	98	-
Few-shot LLM	0-shot	81	18.28%	23	74.19%	17	58.55%	5	7.08%	7	18.13%	2	3.33%	135	22.93%
	2-shot	83	18.79%	20	61.06%	17	53.91%	8	19.29%	13	26.58%	4	3.81%	145	23.48%
	4-shot	94	19.40%	23	73.85%	20	57.33%	10	23.40%	12	24.95%	5	6.24%	164	25.25%
Conversational		146	30.95%	23	82.49%	20	75.43%	12	27.02%	13	35.38%	4	9.20%	218	35.95%
AutoSpec		156	**42.26%**	24	85.38%	**21**	**85.20%**	22	57.00%	16	35.38%	8	12.38%	247	46.13%
SpecGen		**179**	40.41%	**24**	**92.31%**	20	79.57%	**23**	**73.75%**	**20**	**60.38%**	**13**	**36.55%**	**279**	**59.97%**

on the 4-shots examples in the initial prompt, the multi-turn conversational manner (in Section III-B) can further improve the performance, with 146 programs handled, compared to 94 programs of 4-shot LLM. Combining LLM-generated results and static analysis techniques, AutoSpec achieves enhanced results with 10 more programs handled. Lastly, *SpecGen* outperforms all baseline approaches, with the number of programs that generated verifiable specifications increasing to 179.

Performance on *SpecGenBench*. For deeper insights into the performance of different approaches on different types of programs, we further investigate the results on *SpecGenBench*. Similar to SV-COMP, Houdini outperforms Daikon, especially in generating specifications for complex program structures such as loops, Houdini exhibits certain abilities. In terms of the LLM-based approaches, we find that *SpecGen*, AutoSpec, and the conversational approach are all competent in generating specifications for relatively simple programs such as sequential and branched. AutoSpec also demonstrates impressive ability on programs with Single-path Loops, benefiting from the code decomposition technique adopted. However, when it comes to generating specifications for programs with more complex structures such as Multi-path Loops and Nested Loops, *SpecGen* has a clear advantage. The benefits are from our designed mutation-based specification generation (Section III-C), which can correct the erroneous output of the large language model to generate the verifiable specifications. Although *SpecGen* can generate more accurate specifications for different loop structures, it has a relatively poor performance in generating verifiable specifications for programs with nested loops.

For LLM-based approaches, we further use success probability to evaluate the probability of successful generation for a program in 10 times trials due to the randomness inherent in the generated content by LLMs. We can observe that *SpecGen* achieves an overall success probability of 59.97%, which is significantly higher than the values of conversational generation and AutoSpec (35.95% and 46.13% respectively). AutoSpec is not equipped with any feedback-refine mechanism. Although conversational generation can refine the intermediate results through conversation with LLM, there still exists detailed errors that cannot be fixed, since root causes of errors are difficult for LLMs to reason. Compared to these approaches, *SpecGen*, equipped with conversational generation and mutation-based generation, is more reliable with higher probabilities and lower chances of randomness

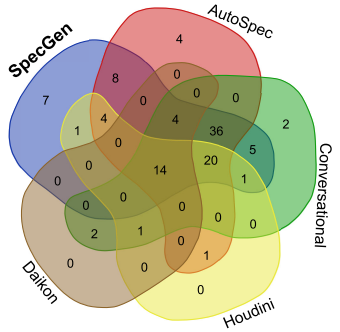

Fig. 5: Venn diagram of verifiable programs.

to generate verifiable specifications. From the results from SV-COMP and *SpecGenBench*, we can find that our proposed approach is orthogonal to different datasets.

Presented in Fig. 5 is the Venn diagram of the programs in *SpecGenBench* for which *SpecGen* and baseline approaches successfully generated verifiable specifications. It is noteworthy that *SpecGen* generates verifiable specifications for 7 programs that other baselines fail to yield, where 5 are from the Nested Loop, with the rest 2 from the Single-path Loop. Further investigation of these programs reveals that they are relatively complicated and challenging to handle.

Overall, it takes on average 7.07 verifier calls for the conversational approach per execution, whereas the figure is 16.19 and 15.51 for AutoSpec and *SpecGen*, respectively. Under our experimental settings, the maximum number of rounds for conversation is set to 10, so conversational generation naturally finishes execution faster than the other two approaches, but with relatively poor performance. In comparison, AutoSpec and *SpecGen* take more verifier calls to filter the LLM-generated results and produce more reliable results. Specifically, *SpecGen* displays a slight advantage over AutoSpec, indicating that *SpecGen* can achieve better performance compared to AutoSpec in the same or even shorter period of time.

RQ1: *SpecGen* outperforms all baselines on two different datasets, generating verifiable specifications for 279 out of 385 programs with the highest success probability among all LLM-based approaches. In comparison, the number of programs handled by Daikon, Houdini, Conversational generation, and AutoSpec is 72, 98, 218, and 247, respectively.

TABLE III: Effectiveness of different types of mutations.

Approach	*SpecGenBench*					SV-COMP (265)	Total (385)
	Sequential (26)	Branched (23)	Single-path (24)	Multi-path (26)	Nested (21)		
w/o Predicative	24	20	20	19	9	167	259
w/o Logical	24	18	14	18	10	151	235
w/o Comparative	24	19	13	12	7	148	223
w/o Arithmetic	23	19	18	**21**	11	170	262
SpecGen	**24**	**20**	**23**	20	**13**	**179**	**279**

B. RQ2: Ablation Study on Mutation Types

We conduct an ablation study to evaluate the effectiveness of different mutation types in *SpecGen*. The results are shown in Table III where w/o {*} denotes the disabled mutation type.

We can find that *SpecGen* successfully generates verifiable specifications for 279 out of a total of 385 programs. *SpecGen* w/o Comparative addresses the least number of programs i.e., 223, indicating that the comparative mutation is the most important in the defined mutation operations. The main reason is the frequent usage of numerical variables in programs and the recurring need to bound their range in the specifications. *SpecGen* w/o Logical has the second least number of programs i.e., 235, indicating that the logical operators are also important to generate verifiable specifications. This is due to the necessity of combining two or more expressions for different properties with logical operations when specifying complex behaviors. *SpecGen* w/o Predicative and *SpecGen* w/o Arithmetic have the most number of programs (259 and 262 respectively), which means both of them are less important than the comparative and logical mutation. We still consider them as they are applicable in certain situations. In some complex programs, specifications generated by the LLM are prone to have predicate errors. Hence, the predicative mutation will be useful. Similar cases exist when there are complicated numerical constraints on variables, where mutations on arithmetic operators turn out to be helpful.

Further analyzing the effectiveness of the mutation type for different kinds of programs, we can find that *SpecGen* w/o Comparative handles an especially lower number of programs in the loop category including single-path, multi-path, and nested loop. It is due to the rigid demand of scope bounding for loop variables when loops are involved. Scope bounding for loop variables is an intricate work where LLMs frequently make mistakes, substantiating the importance of comparative mutations. The performance of *SpecGen* w/o Predicative also drops on programs with nested loop, because of the relatively higher quantity and complexity of `\forall` and `\exists` statements involved in these nested programs.

> **RQ2:** Each type of mutations contributes differently to *SpecGen*. The comparative mutation contributes the most to the performance while the predicative and arithmetic are less important. When combining them together, *SpecGen* achieves the best performance.

C. RQ3: Effectiveness of Selection Strategy

In Section III-C2, we design the heuristic selection strategy to improve the efficiency of verification. We also conduct an experiment to compare with the random selection strategy. Specifically, when a candidate specification is refuted by the verifier, we randomly select another specification from E_f for replacement. To compare with different strategies, for each program that successfully generates verifiable specifications by *SpecGen*, we run *SpecGen* 5 times to obtain the average number of verifier calls as the evaluation metric.

TABLE IV: Average numbers of verifier calls in a single run under different specification selection strategies in Section III-C2.

Strategy	SV-COMP	*SpecGenBench*					Total
		Sequential	Branched	Single-path	Multi-path	Nested	
Random	9.41	2.70	2.32	32.62	55.16	41.97	18.44
Heuristic	**8.91**	**2.59**	**2.16**	**24.21**	**43.58**	**34.99**	**15.51**

Performance on SV-COMP. Using the random selection strategy makes 9.41 verifier calls on average while the heuristic selection strategy takes 8.91 calls, resulting in an improvement of 5.30%. The improvement is relatively modest and the values denote that only less than 10 verifier calls on average are used to generate verifiable specifications for programs in SV-COMP. The main reason is that the number of rounds for conversation is set to 10 in *SpecGen*, the specifications for these programs tend to be successfully generated within the conversation module (Section III-B). It is before the selection strategy used in the mutation-based specification generation (Section III-C) comes into effect.

Performance on *SpecGenBench*. Using the random selection strategy takes 36.20 verifier calls on average while the heuristic selection strategy takes 28.51 calls in terms of five categories in *SpecGenBench*. Hence, the heuristic selection strategy achieves an improvement of 21.23%. Furthermore, we can observe that the improvements in different categories of programs vary significantly. The improvements in the loop categories including single-path, multi-path, and nested loop are more significant than sequential and branched. The main reason is that generating specifications for loop-containing programs is challenging, and usually requires more iterations to obtain verifiable specifications. In this case, a good selection strategy often highlights advantages more effectively. However, the improvements in sequential and branched categories are fewer. The reason is similar to SV-COMP, where the high efficiency of conversational generation on sequential and branched programs makes the selection strategies invalid. Nevertheless, loop structures are common in programs, thus a heuristic selection strategy to improve the validation efficiency is still helpful and necessary.

> **RQ3:** The heuristic selection strategy effectively improves the efficiency of *SpecGen*. It is especially useful when generating specifications for programs with more complex structures such as loops.

D. RQ4: User Study on the Quality of Specifications

A user study is conducted to evaluate the semantic quality of the generated specifications. 15 Ph.D. students are invited

TABLE V: Average rating scores on the generated specifications by different approaches.

Test case	Houdini	Daikon	SpecGen	Oracle
Absolute	3.50	3.36	**4.85**	5.00
AddLoop	2.40	1.33	**4.57**	5.00
Conjunction	4.50	3.50	**5.00**	5.00
ConvertTemperature	2.33	2.50	**5.00**	5.00
Disjunction	2.50	3.50	**5.00**	5.00
FizzBuzz	2.63	2.86	**5.00**	5.00
IsCommonFactor	2.00	4.13	**4.14**	4.71
IsPalindrome	1.83	1.17	**4.75**	5.00
IsSubsequence	2.43	1.13	**4.14**	4.00
IsSuffix	2.20	1.50	**4.33**	4.63
MulLoop	1.88	1.25	**3.33**	5.00
MySqrt	2.00	2.80	**3.75**	4.25
Perimeter	1.00	2.80	**4.78**	5.00
SmallestEvenMul	2.57	1.00	**4.50**	5.00
Swap	1.00	2.00	**5.00**	4.88
Average	2.32	2.32	**4.54**	4.83

to rate the specifications generated by different approaches. A detailed description of the rating process is given in Section IV-D. We selected the 15 programs from the dataset of *SpecGenBench* that can be handled by all of Houdini, Daikon, and *SpecGen*. Apart from the specifications generated by these approaches, we also add the ground truth as a reference. The specifications are kept anonymous to the students, disclosing no information about the sources of the specifications. The rating scores are presented in Table V, where the score for full marks is 5.

We can observe that the ground truth specifications (oracle) receive an average rating score of 4.83, indicating that the semantics of these programs can be described comprehensively through JML specifications. Furthermore, the specifications generated by *SpecGen* received a rating score of 4.54, which is close to the oracle, indicating that the generated specifications by *SpecGen* can also describe the real behaviors of the input program more fully. Among the 15 programs, all rating scores given to *SpecGen* are above 3, with the lowest rating being 3.33, meaning that in the worst case, *SpecGen* can still generate non-trivial specifications about the properties of the input program. In comparison, the specifications generated by Houdini and Daikon received an average rating score of 2.32, reflecting the semantic weakness in these specifications. Houdini and Daikon rely on pre-defined templates, which are in fact independent from the input program and can only cover a limited number of specification patterns. Consequently, the specifications produced are often simplistic and trivial, involving only a narrow range of variables and operators, making it difficult to capture the actual behavior and functionality of the input program precisely. Unlike traditional approaches that rely on a fixed set of templates, *SpecGen* utilizes the code comprehension capabilities of LLMs, which can cover a larger range of scenarios and generate targeted specifications that more closely match the semantics of the input program.

RQ4: *SpecGen* received an average rating score of 4.54, which is close to the 4.83 of the oracle specifications, demonstrating the ability to accurately characterize the real program behaviors and generate specifications with comprehensive program semantics.

VI. Discussion

A. Performance on Real-world Programs

To further evaluate the performance of *SpecGen* on real-world programs, we collect programs involved in Defects4J [50], a well-known dataset of reproducible bugs within open-source repositories. During the collection process, we only consider individual files with no dependency on third-party libraries or other files in the repository. This ensures that all the collected files can be properly executed and verified outside the repository. Eventually, 50 Java source files from 9 repositories are collected. The average line of code and cyclomatic complexity of the collected programs are 374.78 and 18.29, respectively. We follow the same experimental settings in Section IV-A for experiments. Note that we only aim to evaluate the verifiability of the generated specifications in the same way Section V-A does, so the ground truth specifications of the programs are not prepared.

Table VI shows the performance of *SpecGen* and other baseline methods on the programs extracted from Defects4J. Although Daikon underperforms the LLM-based approaches, it still exhibits certain abilities in processing real-world programs. This is due to the existence of some simplistic methods within real-world class definitions, such as those retrieving the value of a certain class member without doing anything else, which Daikon is capable of handling. Compared to Daikon, the LLM-based approach with the simplest setting, i.e. 4-shot, achieved 5 more programs handled. Based on the few-shot learning technique, the conversational approach further achieved 28 programs handled. Lastly, *SpecGen* succeeds in handling 38 out of the 50 programs, with an average success probability of 55.20%, displaying decent capabilities in handling real-world programs.

B. Threats to Validity

Internal Validity. First, the prompts we used to communicate with the LLM may affect our results. To mitigate it, we refer to Xia et al. [15] to design the prompt. We plan to investigate the effect of different prompts in the future. Second, a potential threat lies in the risk of data leakage. Our constructed dataset *SpecGenBench* consists of 100 programs with expert-written specifications and 20 programs with their corresponding specifications from Nilizadeh et al. [43]. The former does not have the issue of data leakage as the specifications are written by experts in our research. However, since `gpt-3.5` does not release its model as well as the training data, the latter 20 programs from the existing dataset may have the risk. The used dataset SV-COMP also has this risk. Nevertheless, through our observation of *SpecGen* on these programs, we have never spotted a situation where the output of *SpecGen* is the same as the existing oracle. Hence, we believe this threat is limited. Furthermore, even if we remove these potentially risky programs, *SpecGen* still successfully handles 87 programs in the remaining 100 programs, which is also the best.

External Validity. One of the external threats lies in the accuracy of the verifier (OpenJML). Due to the implementation flaws in OpenJML, there may be cases where some correct

TABLE VI: Performance on programs collected from 9 repositories in Defects4J.

Approaches	chart (7)		cli (5)		codec (4)		compress (6)		jackson (7)		jxpath (6)		lang (7)		math (4)		time (4)		Total (50)	
	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.
Daikon	3	-	3	-	0	-	1	-	3	-	2	-	2	-	1	-	0	-	15	-
4-shot LLM	1	7.14%	2	7.33%	2	9.71%	3	11.67%	3	9.52%	1	2.78%	4	11.67%	3	17.50%	1	5.00%	20	8.97%
Conversational	4	47.62%	4	60.00%	3	41.67%	1	11.11%	2	19.05%	5	55.56%	3	28.57%	3	66.67%	3	41.67%	28	39.33%
SpecGen	**6**	**68.57%**	**4**	**68.00%**	**4**	**65.00%**	**4**	**36.67%**	**4**	**42.86%**	**5**	**63.33%**	**5**	**65.71%**	**3**	**45.00%**	**3**	**35.00%**	**38**	**55.20%**

specifications fail to pass verification. This is an inevitable problem that other verifiers [51] have to face as well. The reason lies in the undecidability of automatic software verification [52, 53]. Even in such a situation, *SpecGen* achieves impressive performance with the majority of testcases successfully generated verifiable specifications. Another threat is the potential bias of the hand-written specifications by experts. To mitigate this, we follow the procedure in Nilizadeh et al. [43]. First, the selected experts should have rich experience in writing specifications. Second, the initially chosen specifications should be verifiable by the verifier. Last, if multiple experts have written the specifications that pass the verifier, another expert is responsible for selecting one.

VII. Related Work

Large Language Models. With the advancement of generative AI, Large Language Models (LLMs) have emerged as a formidable force and have quickly found widespread applications. LLMs are characterized by their immense parameter scale and training dataset size [54]. An important feature of LLMs is their ability for in-context learning [37], which enhances the coherence between the context and the output of LLMs. The learning ability gives rise to a unique usage of LLMs known as prompting [55], where a natural language description of the intended downstream task is provided to the LLM before assigning it the task. LLMs initially demonstrated remarkable capabilities in the field of Natural Language Processing (NLP) [56], excelling in tasks such as document classification [57], text summarization [58], and machine translation [59]. They are also widely deployed in various software engineering tasks [14, 18], including software testing [30, 31], code generation [17, 33] and code summarization [16]. Compared with these works, our goal is to employ LLMs for the automated generation of program specifications, which is important in formal methods.

Program Specification Generation. The research on program specification generation can be categorized into two types: natural language specification generation and formal specification generation. Natural language specification generation primarily manifests as code summarization [60], a process of automatically generating accurate, human-readable descriptions of code functionality. Numerous efforts have been made to utilize machine learning methods for code summarization [61, 62, 63, 64]. Formal specification generation primarily takes the form of the generation of program invariants, the formal language representations of properties that a program is guaranteed to satisfy at a certain program point. In invariant generation, a large amount of research focuses on the generation of loop invariants [65, 66, 67, 68, 69, 70, 71], while the rest of the works attempt to generate invariants of other forms, e.g. pre-conditions [48, 72], post-conditions [10, 13, 41, 73, 74], assertion-based invariants [75] and finite automata [76, 77, 78]. With the development of Large Language Models, there have also been efforts employing LLMs to generate program specifications. Wen et al. [21] combine LLMs with static analysis techniques, including code decomposition, to generate verifiable program specifications. Pei et al. [79] utilize fine-tuning to enhance the performance of LLMs on specification generation tasks. Concerning the difficulty of selecting correct specifications from the massive LLM-generated results, Chakraborty et al. [69] propose a ranking algorithm that can distinguish correct inductive invariants from incorrect attempts based on the problem definition. Among these works, artifacts of Ghosal et al. [48] and Pei et al. [79] are not publicly available, the artifact of Alshnakat et al. [41] is built for C code and Frama-C contracts, and the grammar for specifications of Molina et al. [10, 74] involves specific features, which cannot be trivially translated into equivalent JML, thus we cannot include them for comparison. Compared to these works, *SpecGen* utilizes the code comprehension capability of LLMs for program specification generation in a conversational manner, further followed by the mutation-based approach for enhancement.

VIII. Conclusion

In this paper, we introduced *SpecGen*, a novel approach that utilizes the Large Language Model for formal program specification generation. Leveraging the code comprehension ability of LLMs as well as the well-designed mutation-based specification generation component, our approach is capable of accurately capturing the behaviour and functionality of input programs to generate accurate specifications. A comprehensive evaluation between *SpecGen* and other baselines is conducted on two different datasets, the benchmark for the Java category of SV-COMP, and a more diverse and manually constructed dataset containing 120 programs. The extensive experimental results haver demonstrated that our approach significantly outperforms the baseline approaches, with the ability to effectively articulate program behaviors.

Acknowledgment

We are grateful for the constructive feedback of all the anonymous reviewers to improve this manuscript. The authors from Nanjing University are supported in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001), the National Natural Science Foundation of China (No. 62232008, 62172200), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (No. KYCX24_0237).

REFERENCES

[1] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-based automatic testing of modern web applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 35–53, 2012.

[2] T.-H. Nguyen and D.-H. Dang, "Tc4mt: A specification-driven testing framework for model transformations," *International Journal of Software Engineering and Knowledge Engineering*, pp. 1–39, 2023.

[3] G. Cabodi, S. Nocco, and S. Quer, "Strengthening model checking techniques with inductive invariants," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 28, no. 1, pp. 154–158, 2008.

[4] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 622–640.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 234–245.

[6] E. Rodríguez-Carbonell and D. Kapur, "Program verification using automatic generation of invariants," in *International Colloquium on Theoretical Aspects of Computing*. Springer, 2004, pp. 325–340.

[7] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 500–517.

[8] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 4, pp. 229–239, 2002.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[10] F. Molina, M. d'Amorim, and N. Aguirre, "Fuzzing class specifications," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1008–1020.

[11] Z. Y. Ding, Y. Lyu, C. Timperley, and C. Le Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 2019, pp. 2–9.

[12] F. Rahman and Y. Labiche, "A comparative study of invariants generated by daikon and user-defined design contracts," in *2014 14th International Conference on Quality Software*. IEEE, 2014, pp. 174–183.

[13] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 191–200.

[14] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[15] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[16] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[17] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[18] W. Ma, S. Liu, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, and Y. Liu, "The scope of chatgpt in software engineering: A thorough investigation," *arXiv preprint arXiv:2305.12138*, 2023.

[19] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[20] sosy lab, "Sv-comp - international competition on software verification," 2024, https://sites.google.com/view/specgen.

[21] C. Wen, J. Cao, J. Su, Z. Xu, S. Qin, M. He, H. Li, S.-C. Cheung, and C. Tian, "Enchanting program specification synthesis by large language models using static analysis and program verification," *arXiv preprint arXiv:2404.00762*, 2024.

[22] Github, "Specgen-artifact," 2024, https://github.com/Lezhi-Ma/SpecGen-Artifact.

[23] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of jml tools and applications," *International journal on software tools for technology transfer*, vol. 7, pp. 212–232, 2005.

[24] E. I. Leonard and C. L. Heitmeyer, "Automatic program generation from formal specifications using apts," *Automatic Program Development: A Tribute to Robert Paige*, pp. 93–113, 2008.

[25] M. Pradel and T. R. Gross, "Automatic generation of object usage specifications from large method traces," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 371–382.

[26] D. R. Cok, "Openjml: Jml for java 7 by extending openjdk," in *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer, 2011, pp. 472–479.

[27] C. Barrett and C. Tinelli, *Satisfiability modulo theories*. Springer, 2018.

[28] OpenAI, "Gpt-3.5," 2023, https://platform.openai.com/docs/models/gpt-3-5.

[29] ——, "Chatgpt," 2023, https://chat.openai.com/.

[30] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023, pp. 830–842.

[31] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv preprint arXiv:2308.04748*, 2023.

[32] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.

[33] 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," *arXiv preprint arXiv:2305.01210*, 2023.

[34] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[35] Y. Zhang, Y. Li, L. Cui, D. Cai, L. Liu, T. Fu, X. Huang, E. Zhao, Y. Zhang, Y. Chen *et al.*, "Siren's song in the ai ocean: A survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023.

[36] 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.

[37] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[38] 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.

[39] OpenAI, "Gpt-4 technical report," 2023.

[40] ——, "Api reference - openai api," 2023, https://platform.openai.com/docs/api-reference.

[41] A. Alshnakat, D. Gurov, C. Lidström, and P. Rümmer, "Constraint-based contract inference for deductive verification," *Deductive Software Verification: Future Perspectives: Reflections on the Occasion of 20 Years of KeY*, pp. 149–176, 2020.

[42] EclEmma, "Eclemma - jacoco java code coverage library," 2024, https://www.eclemma.org/jacoco/.

[43] A. Nilizadeh, G. T. Leavens, X.-B. Le, C. S. Pasareanu, and D. Cok, "Exploring true test overfitting in dynamic automated program repair using formal methods (in press)," in *2021 14th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2021.

[44] LeetCode, "The world's leading online programming learning platform," 2023, https://leetcode.com/.

[45] X. Xie, B. Chen, L. Zou, Y. Liu, W. Le, and X. Li, "Automatic loop summarization via path dependency analysis," *IEEE Transactions on Software Engineering*, vol. 45, no. 6, pp. 537–557, 2017.

[46] Parasoft, "Ai-powered java testing tool," 2023, https://www.parasoft.com/products/parasoft-jtest/.

[47] Microsoft, "Code contracts - microsoft research," 2023, https://www.microsoft.com/en-us/research/project/code-contracts/.

[48] S. Ghosal, B. Jonsson, and P. Rümmer, "An active learning approach to synthesizing program contracts," in *International Conference on Software Engineering and Formal Methods*. Springer, 2023, pp. 126–144.

[49] T. Nemoto and D. Beglar, "Likert-scale questionnaires," in *JALT 2013 conference proceedings*, 2014, pp. 1–8.

[50] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[51] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager *et al.*, "The key tool: integrating object oriented design and formal verification," *Software & Systems Modeling*, vol. 4, pp. 32–54, 2005.

[52] P. A. Abdulla and B. Jonsson, "Undecidable verification problems for programs with unreliable channels," *Information and Computation*, vol. 130, no. 1, pp. 71–90, 1996.

[53] U. Mathur, P. Madhusudan, and M. Viswanathan, "What's decidable about program verification modulo axioms?" in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 158–177.

[54] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[55] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[56] B. Min, H. Ross, E. Sulem, A. P. B. Veyseh, T. H. Nguyen, O. Sainz, E. Agirre, I. Heintz, and D. Roth, "Recent advances in natural language processing via large pre-trained language models: A survey," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–40, 2023.

[57] S. Hegselmann, A. Buendia, H. Lang, M. Agrawal, X. Jiang, and D. Sontag, "Tabllm: Few-shot classification of tabular data with large language models," in *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, F. Ruiz, J. Dy, and J.-W. van de Meent, Eds., vol. 206. PMLR, 25–27 Apr 2023, pp. 5549–5581. [Online]. Available: https://proceedings.mlr.press/v206/hegselmann23a.html

[58] X. Yang, Y. Li, X. Zhang, H. Chen, and W. Cheng, "Exploring the limits of chatgpt for query or aspect-based text summarization," *arXiv preprint arXiv:2302.08081*, 2023.

[59] B. Zhang, B. Haddow, and A. Birch, "Prompting large language model for machine translation: A case study," *arXiv preprint arXiv:2301.07069*, 2023.

[60] Y. Zhu and M. Pan, "Automatic code summarization: A systematic literature review," *arXiv preprint arXiv:1909.04352*, 2019.

[61] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2016, pp. 2073–2083.

[62] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[63] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[64] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," *arXiv preprint arXiv:1811.01824*, 2018.

[65] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," *ACM Sigplan Notices*, vol. 51, no. 1, pp. 499–512, 2016.

[66] G. Ryan, J. Wong, J. Yao, R. Gu, and S. Jana, "Cln2inv: learning loop invariants with continuous logic networks," *arXiv preprint arXiv:1909.11542*, 2019.

[67] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[68] C. Janßen, C. Richter, and H. Wehrheim, "Can chatgpt support software verification?" *arXiv preprint arXiv:2311.02433*, 2023.

[69] S. Chakraborty, S. K. Lahiri, S. Fakhoury, M. Musuvathi, A. Lal, A. Rastogi, A. Senthilnathan, R. Sharma, and N. Swamy, "Ranking llm-generated loop invariants for program verification," *arXiv preprint arXiv:2310.09342*, 2023.

[70] A. Kamath, A. Senthilnathan, S. Chakraborty, P. Deligiannis, S. K. Lahiri, A. Lal, A. Rastogi, S. Roy, and R. Sharma, "Finding inductive loop invariants using large language models," *arXiv preprint arXiv:2311.07948*, 2023.

[71] V. J. Hellendoorn, P. T. Devanbu, O. Polozov, and M. Marron, "Are my invariants valid? a learning approach," *arXiv preprint arXiv:1903.06089*, 2019.

[72] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, "Automatic inference of necessary preconditions," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2013, pp. 128–148.

[73] Y. Moy and C. Marché, "Modular inference of subprogram contracts for safety checking," *Journal of Symbolic Computation*, vol. 45, no. 11, pp. 1184–1211, 2010.

[74] F. Molina, P. Ponzio, N. Aguirre, and M. Frias, "Evospex: An evolutionary algorithm for learning postconditions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1223–1235.

[75] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1178–1189.

[76] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 5–14.

[77] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. Vaandrager, "Automata learning through counterexample guided abstraction refinement," in *FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings 18*. Springer, 2012, pp. 10–27.

[78] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for finding api usage errors," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 151–162.

[79] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.