Enchanting Program Specification Synthesis by Large Language Models using Static Analysis and Program Verification

Cheng Wen[1], Jialun Cao[2,⋆], Jie Su[1], Zhiwu Xu[3], Shengchao Qin[1,5,⋆], Mengda He[4], Haokun Li[4], Shing-Chi Cheung[2], and Cong Tian[5]

[1] Guangzhou Institute of Technology, Xidian University, Guangzhou, China
[2] The Hong Kong University of Science and Technology, Hong Kong, China
[3] College of Computer Science and Software Engineering, Shenzhen University, China
[4] Fermat Labs, Huawei, Hong Kong, China
[5] ICTT and ISN Laboratory, Xidian University, Xi'an, China

Abstract. Formal verification provides a rigorous and systematic approach to ensure the correctness and reliability of software systems. Yet, constructing specifications for the full proof relies on domain expertise and non-trivial manpower. In view of such needs, an automated approach for specification synthesis is desired. While existing automated approaches are limited in their versatility, *i.e.*, they either focus only on synthesizing loop invariants for numerical programs, or are tailored for specific types of programs or invariants. Programs involving multiple complicated data types (*e.g.*, arrays, pointers) and code structures (*e.g.*, nested loops, function calls) are often beyond their capabilities. To help bridge this gap, we present AutoSpec, an automated approach to synthesize specifications for automated program verification. It overcomes the shortcomings of existing work in specification versatility, synthesizing satisfiable and adequate specifications for full proof. It is driven by static analysis and program verification, and is empowered by large language models (LLMs). AutoSpec addresses the practical challenges in three ways: (1) driving AutoSpec by static analysis and program verification, LLMs serve as generators to generate candidate specifications, (2) programs are decomposed to direct the attention of LLMs, and (3) candidate specifications are validated in each round to avoid error accumulation during the interaction with LLMs. In this way, AutoSpec can incrementally and iteratively generate satisfiable and adequate specifications. The evaluation shows its effectiveness and usefulness, as it outperforms existing works by successfully verifying 79% of programs through automatic specification synthesis, a significant improvement of 1.592x. It can also be successfully applied to verify the programs in a real-world X509-parser project.

1 Introduction

Program verification offers a rigorous way to assuring the important properties of a program. Its automation, however, needs to address the challenge of proof

⋆ Corresponding authors: Jialun Cao and Shengchao Qin.

construction [1,2]. Domain expertise is required for non-trivial proof construction, where human experts identify important program properties, write the *specifications* (*e.g.*, the pre/post-conditions, invariants, and contracts written in certain *specification languages*), and then use these specifications to prove the properties.

Despite the immense demand for software verification in the industry [3,4,5,6,7], ***manual verification by experts remains the primary approach in practice.*** To reduce human effort, ***automated specification synthesis*** is desired. Ideally, given a program and a property to be verified, we expect the specifications that are sufficient for a full proof could be synthesized automatically.

Research gap – Prior works are limited in versatility, *i.e.*, the ability to simultaneously handle *different types of specifications* (*e.g.*, invariants, preconditions, postconditions), *code structures* (*e.g.*, multiple function calls, multiple/nested loops), and *data structures* (*e.g.*, arrays, pointers), leaving room for improvement towards achieving full automation in proof construction. Existing works focus only on loop invariants [8,9,10], preconditions [11,12], or postconditions [13,14,15]. Moreover, most works on loop invariant synthesis can only handle numerical programs [16,2,17,18] or are tailored for specific types of programs or invariants [19,20,21,22,23,24]. To handle various types of specifications simultaneously and to process programs with various code and data structures, a versatile approach is required.

Challenges – Although the use of large language models (LLMs) such as ChatGPT may provide a straightforward solution to program specification generation, it is not a panacea. The generated specifications are mostly incorrect due to three intrinsic weaknesses of LLMs. First, ***LLMs can make mistakes.*** Even for the well-trained programming language Python, ChatGPT-4 and ChatGPT-3.5 only achieve 67.0% and 48.1% accuracy in program synthesis [25]. In comparison with programming languages, LLMs are much less trained in specification languages. Therefore, LLMs generally perform worse in synthesizing specifications than programs. Since the generated specifications are error-prone, we need an effective technique to detect incorrect specifications, which are meaningless to verify. Second, ***LLMs may not attend to the tokens we want them to***. Self-attention may pay no, less, or wrong attention to the tokens that we want it to. Recent research even pointed out a phenomenon called "lost in the middle" [26], observing that LLMs pay little attention to the middle if the context goes extra long. In our case, the synthesized specifications are desired to capture and describe as many program behaviors as possible. Directly adopting the holistic synthesis (*i.e.*, synthesizing all specifications at once) may yield unsatisfactory outcomes. Third, ***errors accumulate in the output of LLMs.*** LLMs are auto-regressive. If they make mistakes, these wrong outputs get added to their inputs in the next round, leading to way more wrong outputs. It lays a hidden risk when taking advantage of LLMs' dialogue features, especially in an incremental manner (*i.e.*, incrementally synthesizing specifications based on previously generated ones).

Insight – To address the above challenges, ***our key insight is to let static analysis and program verification take the lead, while hiring LLMs to synthesize candidate specifications.*** Static analysis parses a given

program into pieces, and passes each program piece in turn to LLMs by inserting a placeholder in it. Paying attention to the spotted part, LLMs generate a list of specifications as candidates. Subsequently, a theorem prover validates the generated specifications and keeps the validated ones in the next round of synthesis. The iteration process terminates when the property under verification has been proved, or the iteration reaches a predefined limit.

Solution – Bearing the insight, we present AutoSpec, an LLM-empowered framework for generating specifications. It tackles the three above-mentioned limitations of directly adopting LLM in three perspectives. First, ***it decomposes the program hierarchically*** and employs LLMs to generate specifications incrementally in a bottom-up manner. This allows LLMs to focus on a selected part of the program and generate specifications only for the selected context. Thus, the limitation of context fragmentation could be largely alleviated. Second, ***it validates the generated specifications*** using theorem provers. Specifications that are inconsistent with programs' behaviors and contradict the properties under verification will be discarded. This post-process ensures that the generated specifications are satisfiable by the source code and the properties under verification. Third, ***it iteratively enhances the specifications*** by employing LLMs to generate more specifications until they are adequate to verify the properties under verification or the number of iterations reaches the predefined upper bound.

We evaluate the effectiveness of AutoSpec by conducting experiments on 251 C programs across four benchmarks, each with specific properties to be verified. We compare AutoSpec with three state-of-the-art approaches: Pilat, Code2inv, and CLN2Inv. The result shows AutoSpec can successfully handle 79% (= 199 / 251) programs with various structures (*e.g.*, linear/multiple/nested loops, arrays, pointers), while existing approaches can only handle programs with linear loops. As a result, 59.2% (= (199 - 125) / 125) more programs can be successfully handled by AutoSpec. The result also shows that AutoSpec outperforms these approaches regarding effectiveness and expressiveness when accurately inferring program specifications. To further indicate its usefulness, we apply AutoSpec to a real-world X509-parser project, demonstrating its ability to automatically generate satisfiable and adequate specifications for six functions within a few minutes. In addition, the ablation study reveals that the program decomposition and the hierarchical specification generation components contribute most to performance improvement.

In summary, this paper makes the following contributions:

- **Significance.** We present an automated specification synthesis approach, AutoSpec, for program verification. AutoSpec is driven by static analysis and program verification, and empowered by LLMs. It can synthesize different types of specifications (*e.g.*, invariants, preconditions, postconditions) for programs with various structures (*e.g.*, linear/multiple/nested loops, arrays, pointers).
- **Originality.** AutoSpec tackles the practical challenges for applying LLMs to specification synthesis: It decomposes the programs hierarchically to lead LLMs' attention, and validates the specifications at each round to avoid error

```
1 #include <limits.h>
2
3 /*@
4 requires \valid(a);
5 requires \valid(b);
6 ensures *a == \old(*b);
7 ensures *b == \old(*a);
8 assigns *a,*b;
9 */
10 void swap(int *a, int *b) {   1. Swap
11   int temp = *a;
12   *a = *b;
13   *b = temp;
14 }
15
16 /*@
17 requires \valid(array+(0..n-1));
18 requires 0 < n < INT_MAX;
19 ensures \forall integer i; 0 < i < n ==> array[i-1] <= array[i];
20 */
21 void bubbleSort(int *array, int n) {   4. bubbleSort
22   if (n <= 0) return;
23   int i, j;
24   /*@
25   loop invariant 0 <= i < n;
26   loop invariant \forall integer k; i <= k < n-1 ==> array[k] <= array[k+1];
27   loop invariant \forall integer k; 0 <= k < i+1 <= n-1 ==> array[k] <= array[i+1];
28   loop assigns i, j, array[0..n-1];
29   */
30   for(i = n - 1; i > 0; i--) {      3. Outer loop
31     /*@
32     loop invariant 0 <= j <= i < n;
33     loop invariant \forall integer k; 0 <= k <= j ==> array[k] <= array[j];
34     loop invariant \forall integer k; 0 <= k < i+1 <= n-1 ==> array[k] <= array[i+1];
35     loop assigns j, array[0..i];
36     */
37     for(j = 0; j < i; j++) {        2. Inner loop
38       if (array[j] > array[j+1]) {
39         swap(&array[j], &array[j+1]);
40       }
41     }
42   }
43 }
44
45 void main() {
46   int array[5000] = {..., 5, 4, 3, 2, 1};
47   bubbleSort(array, 5000);
48   //@ assert \forall int i; 0 < i < 5000 ==> array[i-1] <= array[i];
49 }
```

Fig. 1: ACSL Annotations to Functional Proof of Bubble Sort

accumulation. By doing so, AutoSpec can incrementally and iteratively generate satisfiable and adequate specifications to verify the desired properties.

– **Usefulness.** We evaluate AutoSpec on four benchmarks and a real-world X509-parser. The four benchmarks include 251 programs with linear/multiple/nested loops, array structures, pointers, *etc.*. AutoSpec can successfully handle 79% of them, 1.592x outperforming existing works. The experiment result shows the effectiveness, expressiveness, and generalizability of AutoSpec.

2 Background and Motivation

Listing 1 illustrates a C program that implements the `bubble sort` (sorting a 5000-element array of integers in ascending order), where the *property to be verified* (line 48) prescribes that after sorting, any index `i` between 1 and 4999, the element at `array[i-1]` is no larger than the element at `array[i]`. To verify the property, we use a specification language for C programs, ACSL [27] (ANSI/ISO-C Specification Language) to write the proof. It appears in the form of code comments (annotated by `//@ ...` or `/*@ ... */`) and does not affect the program execution. The ACSL-annotated program can be directly fed to auto-active verification tool (Frama-C [28] in this paper) to prove the properties.

In the running example, specifications in the program prescribe the *preconditions* (begin with `\requires`), *postconditions* (begin with `\ensures`), and *loop*

invariants (begin with `loop invariant`)[6]. To prove the property in line 48, practitioners usually write specifications *in a bottom-up manner*, that is, from line 47 tracing to `bubbleSort` (line 21), then from line 39, tracing to `swap` (line 10). Starting from `swap`, practitioners identify the inputs and outputs of the `swap` function and write the pre/post-conditions (lines 4-8). In particular, the precondition (lines 4-5) requires the two input pointers to be valid (*i.e.*, they can be safely accessed), which is necessary to ensure the safe execution of the operations involving dereferencing. Additionally, the postcondition ensures that the values of `*a` and `*b` are swapped (lines 6-7) and assigned (line 8) during execution.

Then tracing back to where `swap` is called, *i.e.*, inside `bubbleSort`, it can be challenging because it contains nested loops. In a bottom-up manner, the ***inner loop*** of `bubbleSort` (lines 37-41) is first analyzed. In particular, to verify a loop, it is composed of (1) loop invariants (*i.e.*, general conditions that hold before/during/after the loop execution, begin with `loop invariant`), and possibly (2) the list of assigned variables (begin with `assign`). In the example, practitioners analyze the inner loop and write specifications in lines 31-36. Specifically, the index `j` should fall into the range of 0 to n (line 32), the elements from index `0` to `j` are not larger than the element at `j` (line 33), and all elements from index `0` to `i` are smaller than or equal to the element at `i+1` (line 34). Also, the variables to be assigned in this inner loop include `j` and first-`i` elements in the `array` (line 35). Similarly, for ***the outer loop*** (lines 30-42), lines 24-29 describe the range of index `i` (line 25), invariants (lines 26-27) and assigned variables (line 28).

Finally, practitioners analyze `bubbleSort` (lines 21-43), identifying that the first-`n` elements of `array` can be safely accessed (line 17), `n` must be greater than zero (line 18). After execution, the array is in ascending order (line 19). Once all the specifications are written, they are fed into a prover/verification tool), Frama-C [28] which supports ACSL to verify the ***satisfiability*** (*i.e.*, the specifications satisfy the program) and ***adequacy*** (*i.e.*, the specifications are sufficient to verify the desired properties) of all specifications until the desired property verification succeeds. If the verification fails, practitioners debug and refine the specifications.

From this example, we can see that the manual efforts to write specifications are non-trivial. Even for a simple algorithm such as bubble sort. In practice, the program under verification could be on a far larger scale, which brings a huge workload to practitioners, motivating the *automated specification synthesis*.

Motivation – Existing automated specification synthesis works can only synthesize loop invariants for programs with a single loop [2,16] or multiple loops [29] on the numerical program. These approaches are unable to generate satisfiable and adequate specifications to fully prove the correctness of basic programs such as bubble sort.

Motivated by the research gap, AutoSpec is presented. It synthesizes specification in a bottom-up manner, synthesizing versatile specifications (*i.e.*, not only loop invariants, but also precondition, postcondition, and assigned variables, which are necessary for the full proof). It validates the satisfiability of speci-

[6] ACSL has more keywords with rich expressiveness. Refer to the documentation [27].

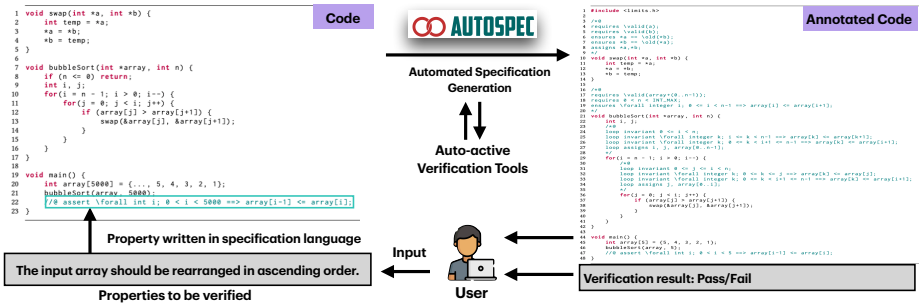

Fig. 2: User Scenario of AutoSpec

fications whenever specifications are synthesized, and verifies the adequacy of specifications after all specifications are synthesized.

User Scenario – We envision the user scenario of AutoSpec in Figure 2. Given a program and properties under verification, AutoSpec provides a fully automated verification process. It synthesizes the specifications for the program, validates the satisfiability of specifications, verifies the specifications against the desired properties, and outputs the verification result with proof if any.

Note that proof can be provided by AutoSpec if the program is correctly implemented (*i.e.*, the properties can be verified). When the given program is syntactically buggy, the program reports the syntactic error at the beginning before launching AutoSpec. If the given program is semantically buggy, then AutoSpec cannot synthesize adequate specifications for verification, the synthesis terminates when the maximum iteration number is reached.

3 Methodology

Figure 3 shows an **overview** of AutoSpec. The workflow comprises three main steps: ❶ **Code Decomposition** (Section 3.1). AutoSpec statically analyzes a C program by decomposing it into a call graph, where loops are also represented as nodes. The aim of the first step is to generalize the procedure that was previously discussed in Section 2 to include the implicit knowledge of simulating interactions between humans and verification tools. By decomposing the program into smaller components, LLMs can iteratively focus on different code components for a more comprehensive specification generation. ❷ **Hierarchical specification generation** (Section 3.2). Based on the call graph with loops, AutoSpec inserts *placeholders* in each level of the graph in a bottom-up manner. Taking the program in Listing 1 for example, AutoSpec inserts the first placeholder (`/*@ 1. SPEC PLACEHOLDER */`) before `swap`, and then inserts the second placeholder in the inner loop of `Sort`. Then, AutoSpec iteratively masks the placeholder one at a time with "`>>> INFILL <<<`" and feeds the masked code into LLMs together with few-shot examples. After querying LLMs, they reply with a set of specifications. AutoSpec then fills the generated specifications into the placeholder and proceeds to the next one. Once all the placeholders are filled with LLM-generated specifications, AutoSpec proceeds to the next step. ❸ ***Specification Validation*** (Section 3.3). AutoSpec feeds the verification conditions of each generated specification into a theorem prover to verify their

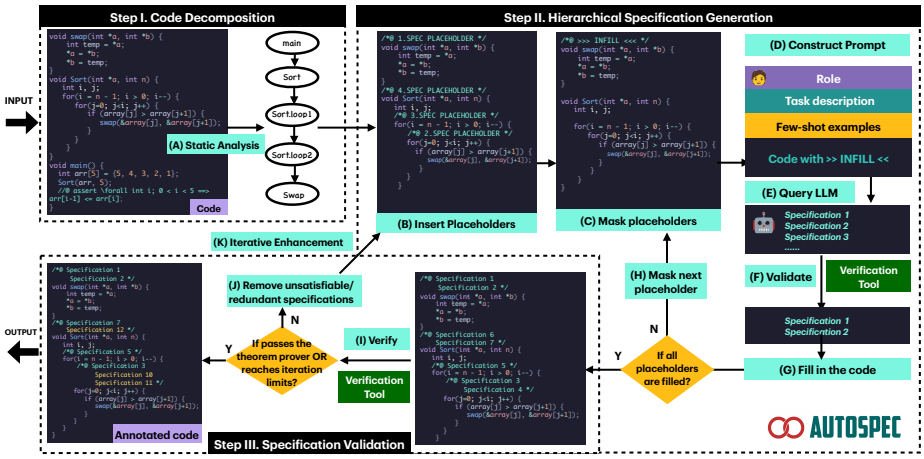

Fig. 3: Overview of AutoSpec

satisfiability. If the theorem prover confirms the satisfiability of the specifications, they will be annotated as a comment in the source code. Otherwise, if the theorem prover identifies any unsatisfiable specifications (*i.e.*, cannot be satisfied by the program), AutoSpec removes those specifications and annotates program with the remaining specifications. Then, AutoSpec returns to the second step to insert additional placeholders immediately after the specifications generated in the previous iteration and generate more specifications. This iterative process continues until all the specifications are successfully verified by the prover or until the number of iterations reaches the predefined upper limit (in our evaluation, this is set to 5). We will explain the methodology of each step in detail.

3.1 Code Decomposition

Using static analysis, AutoSpec constructs a comprehensive call graph for the given program to identify the specific locations where specifications should be added and determine the order in which these specifications should be added. This call graph is an extended version of the traditional one, where loops are also treated as nodes, in addition to functions. This is particularly useful for complex programs where loops can significantly affect the program's behavior.

The algorithm for constructing such a call graph is shown in Algorithm 1. Specifically, the algorithm selects a function that contains the target assertion to be verified as the entry point for the call graph construction. Then, it traverses the abstract syntax tree (AST) of the source code to identify all functions and loops and their calling relationships. For instance, the extended call graph generated for the program in Listing 1 is given in Figure 3(A).

Then, the specifications are generated step-by-step based on the nodes in the extended call graph. When generating the specification for a node, one only needs to consider the code captured by the node and the specifications of its callees in the extended call graph. Furthermore, modeling loops in addition to functions as separate nodes in the extended call graph allows AutoSpec to generate loop invariants, which are essential to program verification. Therefore,

Algorithm 1: Construct an Extended Loop/Call Graph

Input: The source code C and the location loc of the assertion to be verify
Output: A call graph G extended with loops

```
1  G ← get_function(loc)                                   // initialize a Graph G
2  WorkList ← get_function(loc)
3  while WorkList! = ∅ do
4      Fn ← select_and_remove_a_node(WorkList)             // transitively visit all reachable nodes
5      for each basicBlock bb in Fn do
6          if bb calls a function M then                   // If there exist a function call
7              if M is not already a node in G then
8                  Add a node M to G
9                  Add an edge from Fn to M in G
10                 WorkList ← WorkList ∪ M                 // Add this node (function) to the WorkList
11         if bb is a loop entry for loop L then           // If there exist a loop
12             if L is not already a node in G then
13                 Add a node L to G
14                 Add an edge from Fn to L in G
15                 WorkList ← WorkList ∪ L                 // Add this node (loop) to the WorkList
16 return G
```

code decomposition allows LLMs to focus on small program components to generate specifications, thus reducing the complexity of specification generation and making it more manageable and efficient. And traversing the extended call graph from bottom to top can simulate the programmers' verification process.

3.2 Hierarchical Specification Generation

AutoSpec generates specifications for each node in the extended call graph in a hierarchical manner. It starts from the leaf nodes and moves upward to the root node. This bottom-up approach ensures that the specifications for each function or loop are generated within the context of their callers. Algorithm 2 shows the algorithm of hierarchical specification generation. The algorithm takes as input an extended call graph G, an iteration bound t, a large language model, and the assertion to be verified; and outputs an annotated code C with generated specifications. In detail, the algorithm works as follows: First, the algorithm initializes a code template C with the original code without specifications (line 1). The code template, similar to Figure 3(B), includes placeholders. Each placeholder corresponds to a node in the call graph. These placeholders will be iteratively replaced with the valid specifications generated by the LLMs and validated by the theorem prover, within a maximum of t iterations (line 2).

In each iteration, the algorithm performs the following steps. *First*, AutoSpec initializes a stack S with the root node of graph G, which is the target function containing the assertion to be verified (line 3). *Second*, AutoSpec pushes the nodes that require specification generation into the stack S and traverses the stack S in a depth-first manner (lines 4-15). For each node f in the stack, the algorithm checks if all the callees of f have their specifications generated in this iteration (lines 7-8). If not, the algorithm pushes the callee nodes into the stack and marks f as not ready for specification generation (lines 9-10). If all of the functions called by f have had their specifications generated, the algorithm will then proceed to

Algorithm 2: Hierarchical Specification Generation

Input: A loop/call Graph G, an iteration bound t, a Large Language Model LLM, and the assertion ass to be verified
Output: Annotated Code C with generated specifications

```
1  C.init()                                                    // initialize the code (without specifications)
2  for i in range(0, t) do                                     // iteratively enhancing specifications
3      Initialize a stack S with the root node of G
4      while S is not empty do
5          f = S.top()                                         // get the element at the top of the stack
6          allgen = true
7          for each callee in f.callees() do
8              if spec generation for callee has not been done in i^th iteration then
9                  S.push(callee)                              // push all callee into the stack
10                 allgen = false
11         if allgen == true then
12             spec_tmp = spec_generation(C, f, LLM)           // query LLM to generate specification candidates
13             spec_f = spec_validation(C, spec_tmp)           // specification validation
14             C.insert(spec_f)                                // insert the specifications into the code template
15             S.pop()                                         // pop up the top element f of the stack
16     if spec_validation(C, ass) then                         // determine whether the whole verification task has been completed
17         simplify(C)                                         // eliminating redundant specifications
18         break
19 return C
```

generate the specifications for f. (lines 11-15). In particular, AutoSpec queries the LLMs to generate a set of candidate specifications $spec_{tmp}$ for f (line 12), and validates $spec_{tmp}$ by examining their syntactic and semantic validity. Any illegal or unsatisfiable specifications are eliminated, and the remaining valid specifications are referred to as $spec_f$ (line 13). The validation process may employ existing provers/verification tools to guarantee soundness. Then, AutoSpec inserts the validated specifications into the source code C at the placeholder of f (line 14), and pops up the node f from the stack, indicating that f has its specification generated in this iteration (line 15). *Third*, AutoSpec examines whether the whole verification task has been completed, that is, whether the generated specifications are adequate to verify the target assertion (line 16). If it does, AutoSpec proceeds to simplify the annotated code C by eliminating redundant or unnecessary specifications (line 17) and then terminates (line 18). Otherwise, it is assumed that the specifications generated so far are satisfiable, though they may be inadequate. And AutoSpec will start another iteration to generate additional specifications while retaining those already generated. *Finally*, the algorithm returns the annotated code C with the generated specifications as the output (line 19). After several iterations, if the whole verification task remains incomplete, the programmer can make a decision on whether to involve professionals to continue with the verification process for the annotated code C.

Consider the extended call graph in Figure 3(A) for example. AutoSpec first pushes the `main` function, the `sort` function, the `sort.loop1` loop, the `sort.loop2` loop, and the `swap` function into the stack in order, as all of them, except the `swap` function have some callee nodes that require specification generation. Then, it generates specifications for each function or loop in the stack in reserve order. This order echoes what is described in Section 2. AutoSpec

will leverage the power of LLMs to generate candidate specifications for each component/function (*i.e.*, `spec_generation` function in line 12). In the following, we discuss how AutoSpec utilizes LLMs for generating specifications.

Specification generation by LLMs. To employ LLMs in producing precise and reliable responses in the specified format, AutoSpec automatically generates a prompt for each specification generation task. This prompt is a natural language query that includes the role setting, task description, a few examples showing the desired specifications, and the source code with a highlighted placeholder (*e.g.*, Figure 3(C)). The prompt template used in AutoSpec is shown in Figure 3(D). Specifically, a prompt typically consists of the following elements: a system message, code with a placeholder, and an output indicator. The system message provides the specification generation task description and the specification language, which are called *context*. AutoSpec sets the **role** of LLMs as "*As an experienced C/C++ programmer, I employ a behavioral interface specification language that utilizes Hoare style pre/post-conditions, as well as invariants, to annotate my C/C++ source code*". The system message also indicates the task's instructions, such as "*Fill in the* `>>> INFILL <<<`". As explained in Section 3.1, when querying LLMs for the specifications of a component (*i.e.*, a function or loop), the code of this component and the specifications of its callees in the call graph are needed. That is to say, the irrelevant code that is not called by this component can be omitted, allowing LLMs to maintain their focus on the target component and reduces unnecessary token costs. Finally, AutoSpec uses the output format `/@ ... /` to indicate the generated specifications, which is crucial for programmatically processing the responses of LLMs.

To improve the quality of the generated specifications, AutoSpec employs the prompt engineering technique of few-shot prompting [30]. To achieve this, the prompts are designed to include a few relevant input-output examples. Feeding LLMs a few examples can guide them in leveraging previous knowledge and experiences to generate the desired outputs. This, in turn, enables LLMs to effectively handle novel situations. In particular, few-shot prompting allows LLMs to facilitate the learning of syntax and semantics of specification language through in-context learning. For example, consider a prompt that includes an input-output example with a loop invariant for an array that initializes all elements to 0, such as `\forall integer j; 0 <= j < i ==> ((char*)p)[j] == 0;`. With this example, AutoSpec is able to generate a valid loop invariant that involves using quantifiers for the inner loop of `bubbleSort` in a single query.

3.3 Specification Validation

The hierarchical specification generation algorithm also employs specification validation (*i.e.*, `spec_validation()` in line 13) and specification simplification (*i.e.*, `simplify()` in line 17) techniques to ensure the quality of the specifications.

Specification validation. Once candidate specifications have been generated for a component, AutoSpec will check their syntactic and semantic validity (*i.e.*, legality and satisfiability), as shown in Figure 3(F). Specifically, for a function, the legality and satisfiability of the generated specifications are checked immediately.

Table 1: Statistics of Benchmarks.

Benchmarks / Project	Description	Num of Prog	Types of Specifications	Ave LoC	Num of Spec
Frama-C-problems[31]	Programs with function calls, nested/multiple loops, arrays, pointers.	51	pre/post-conditions, loop invariants	17.43	1∼3
X509-parser [32]	A real-world software implements a X.509 certificate parser.	6	pre/post-conditions, loop invariants	82.33	3∼19
SyGuS [33]	Programs with a single loop.	133	loop invariants	22.56	1∼12
OOPSLA-13 [34]	Programs with a single loop or nested/multiple loops.	46	loop invariants	30.28	1∼3
SV-COMP [35]	Programs with more complex nested/multiple loops.	21	loop invariants	24.33	1∼5

While for a loop, the legality is checked immediately, but the satisfiability check is postponed until the outermost loop. This is because inner loops often use variables defined in some of their outer loops (*e.g.*, variable `i` in the `bubbleSort` example), and the satisfiability of all loop invariants needs to be verified simultaneously.

AutoSpec leverages the verification tool (*i.e.*, Frama-C) to verify the specifications. If the verification tool returns a compilation error, AutoSpec identifies the illegal specification where the error occurs and continues verifying without it if there are still some candidates. Otherwise, if the verification tool returns a verification failure, AutoSpec identifies the unsatisfiable specification which fails during verification and continues verifying without it if there are still some candidates. Finally, if the verification succeeds, the specifications will be correspondingly inserted into the code as a comment (Figure 3(G)).

In addition, AutoSpec also validates whether the generated specifications are adequate to verify the target assertion (line 16), which is the same as the validation above but with the target assertion. Note that, the validation phase is crucial to AutoSpec as it ensures that the generated specifications are not only legal and satisfiable but also adequate to verify the target assertion (Figure 3(I)).

Specification simplification (Optional). The objective of specification simplification is to provide users with a concise and elegant specification that facilitates manual inspection and aids in understanding the implementation. This process could be *optional* if one's goal is simply to complete the verification task without placing importance on the specifications. After successfully verifying the assertion, we proceed to systematically remove specifications that are not needed for their verification, one by one. Our main idea is that a specification is unnecessary if the assertion is still verifiable without it. We repeat this process until we reach the minimal set of specifications for manual reading.

There are two main reasons to eliminate specifications: (1) The specification is considered weak and does not capture relevant properties of the verification task. For example, both the loop invariant `i > 0` and `i > 1` are satisfiable, but `i > 0` can be safely removed. (2) The specification is semantically similar to another specification. For example, both `\forall integer i; 0 < i < n ==> array[i-1] <= array[i];` and `\forall integer i; 0 <= i < n-1 ==> array[i] <= array[i+1];` accurately describe the post-condition of `bubbleSort`. Removing either of them has no impact on the verification results.

4 Evaluation

The experiments aim to answer the following research questions:

RQ1. Can AutoSpec generate specifications for various properties effectively? We aim to comprehensively characterize the effectiveness of AutoSpec

Table 2: Effectiveness of AutoSpec in General Specification Generation

Benchmark Information				AutoSpec					
Type	**Program**	**LoC**	**Component (Func, Loop)**	**Success**	**Ratio**	**Iterations**	**Generated Spec**	**Correct Spec**	**Time(s) mean ± std**
general_wp_problems	absolute_value.c	15	1 (1,0)	✔	5/5	1,1,1,1,1	7,7,7,7,7	7/7	14.17 ± 8.74
	add.c	11	1 (1,0)	✔	5/5	1,1,1,1,1	3,3,3,3,3	2/2	14.36 ± 8.20
	ani.c	18	2 (1,1)	×	0/5	-,-,-,-,-	41,33,26,20,24	3/4	–
	diff.c	10	1 (1,0)	✔	5/5	1,1,1,1,1	2,2,2,2,2	1/1	8.85 ± 5.72
	gcd.c	22	1 (1,0)	×	0/5	-,-,-,-,-	3,6,8,5,5	2/5	–
	max_of_2.c	15	1 (1,0)	✔	5/5	1,1,1,1,1	4,3,5,3,5	2/2	17.64 ± 11.24
	power.c	18	2 (1,1)	N/A	–	–	–	–	–
	simple_interest.c	14	1 (1,0)	✔	5/5	1,1,1,1,1	5,5,5,5,5	5/5	15.34 ± 8.60
	swap.c	16	1 (1,0)	✔	5/5	1,1,1,1,1	3,3,3,3,3	2/2	15.36 ± 8.66
	triangle_angles.c	14	1 (1,0)	✔	5/5	1,1,1,1,1	7,5,6,4,5	4/4	23.99 ± 13.67
	triangle_sides.c	16	1 (1,0)	✔	5/5	1,1,1,1,1	3,3,3,2,2	2/2	20.11 ± 11.41
	wp1.c	14	1 (1,0)	×	0/5	-,-,-,-,-	1,4,1,4,2	3/3	–
pointers	add_pointers.c	19	1 (1,0)	✔	5/5	1,1,1,1,1	3,4,4,4,4	2/2	10.00 ± 5.85
	add_pointers_3_vars.c	20	1 (1,0)	✔	5/5	2,5,3,-,3	3,6,10,14,4	3/3	74.45 ± 47.78
	div_rem.c	12	1 (1,0)	✔	5/5	1,1,1,1,1	7,8,7,7,7	4/4	14.66 ± 8.39
	incr_a_by_b.c	13	1 (1,0)	✔	5/5	1,1,1,1,1	6,6,6,4,6	3/3	18.82 ± 13.43
	max_pointers.c	16	1 (1,0)	✔	5/5	1,2,1,1,1	5,4,4,5,5	4/4	41.73 ± 15.74
	order_3.c	36	1 (1,0)	×	0/5	-,-,-,-,-	16,19,15,6,12	3/4	–
	reset_1st.c	16	1 (1,0)	✔	5/5	1,1,1,1,1	7,5,4,6,5	4/4	15.23 ± 9.42
	swap_pointer.c	13	1 (1,0)	✔	5/5	1,1,1,1,1	5,5,5,5,5	2/2	10.72 ± 6.00
loops	1.c	9	1 (0,1)	✔	5/5	1,1,1,1,1	2,2,2,2,2	1/1	2.35 ± 2.23
	2.c	17	2 (1,1)	×	0/5	-,-,-,-,-	8,10,11,11,5	2/5	–
	3.c	18	2 (1,1)	×	0/5	-,-,-,-,-	10,6,7,5,6	3/4	–
	4.c	18	2 (1,1)	N/A	–	–	–	–	–
	fact.c	19	2 (1,1)	×	0/5	-,-,-,-,-	3,8,7,6,6	3/7	–
	mult.c	16	2 (1,1)	×	0/5	-,-,-,-,-	6,14,10,9,16	2/3	–
	sum_digits.c	17	2 (1,1)	×	0/5	-,-,-,-,-	7,13,13,12,11	-	–
	sum_even.c	16	2 (1,1)	×	0/5	-,-,-,-,-	9,14,16,18,19	2/3	–
immutable_arrays	array_sum.c	16	2 (1,1)	×	0/5	-,-,-,-,-	5,4,4,3,4	3/5	–
	binary_search.c	24	2 (1,1)	✔	1/5	3,-,-,-,-	16,16,30,36,19	7/7	739.62 ± 239.59
	check_evens_in_array.c	19	2 (1,1)	×	0/5	-,-,-,-,-	11,18,15,13,16	4/6	–
	max.c	20	2 (1,1)	✔	5/5	1,1,1,1,1	10,9,12,8,7	5/5	40.11 ± 29.49
	occurences_of_x.c	26	2 (1,1)	✔	5/5	2,1,1,1,2	16,12,10,14,12	3/3	121.83 ± 75.04
	sample.c	19	1 (0,1)	✔	5/5	1,1,1,1,1	3,3,4,3,3	1/1	16.89 ± 10.47
	search.c	17	2 (1,1)	×	0/5	-,-,-,-,-	12,14,16,16,12	5/8	–
	search_2.c	18	2 (1,1)	✔	4/5	1,3,2,-,3	13,18,19,10,16	5/5	155.32 ± 175.44
mutable_arrays	array_double.c	19	2 (1,1)	✔	4/5	-,2,2,2,3	12,17,19,16,17	4/4	81.44± 21.14
	bubble_sort.c	26	3 (1,2)	✔	3/5	-,2,3,3,-	9,12,15,12,15	10/10	448.76 ± 554.81
more_arrays	equal_arrays.c	15	2 (1,1)	×	0/5	-,-,-,-,-	9,14,15,13,8	5/7	–
	replace_evens.c	17	2 (1,1)	✔	5/5	1,1,1,1,1	13,12,15,20,14	3/3	52.33 ± 17.30
	reverse_array.c	23	2 (1,1)	×	0/5	-,-,-,-,-	10,7,14,13,18	5/-	–
arrays_and_loops	1.c	10	1 (1,0)	✔	5/5	1,1,1,1,1	3,2,3,2,3	1/1	3.22 ± 2.07
	2.c	18	2 (1,1)	✔	5/5	1,1,1,1,1	10,10,10,10,10	2/2	30.15 ± 27.79
	3.c	19	1 (1,0)	✔	5/5	1,1,1,1,1	9,10,10,4,9	2/2	12.05 ± 7.09
	4.c	18	2 (1,1)	✔	5/5	1,1,1,1,1	12,10,10,8,0	2/2	18.34 ± 13.46
	5.c	18	2 (1,1)	×	0/5	-,-,-,-,-	12,10,10,4,9	3/4	–
miscellaneous	array_find.c	20	2 (1,1)	×	0/5	-,-,-,-,-	7,7,7,7,7	4/7	–
	array_max_advanced.c	20	2 (1,1)	✔	5/5	1,1,1,1,1	5,6,5,6,5	2/2	31.99 ± 34.41
	array_swap.c	18	1 (1,0)	✔	5/5	1,1,1,1,1	8,4,7,8,4	3/3	26.05 ± 30.72
	increment_arr.c	17	2 (1,1)	×	0/5	-,-,-,-,-	2,2,2,2,2	3/6	–
	max_of_2.c	14	1 (1,0)	✔	5/5	1,1,1,1,1	2,3,3,2,3	1/1	9.98 ± 10.31
Overall				**31 / 51**					89.17 ± 172.75

against various types of specifications including pre/post-conditions, loop invariants.

RQ2. Can AutoSpec generate specifications for loop invariant effectively? Loop invariant, as a major specification type, is known for its difficulty and significance. We select three benchmarks with linear and nested loop structures and compare them with state-of-the-art approaches.

RQ3. Is AutoSpec efficient? We compare the AutoSpec's overhead incurred by LLM querying and theorem proving with the baselines.

RQ4. Does every step of AutoSpec contribute to the final effectiveness? We conduct an ablation study on each part of the AutoSpec's design, showing the distinct contribution made independently.

4.1 Evaluation Setup

Benchmark. We conducted evaluations on four benchmarks and a real-world project. The statistical details of these benchmarks can be found in Table 1.

The FRAMA-C-problems [31] benchmark and the X509-parser [32] comprises programs that involve multiple functions or loops, requiring the formulation of pre/post-conditions, loop invariants, *etc.*. The SyGuS [33] benchmark only includes programs with linear loop structures. While the OOPSLA-13 [34] and SV-COMP [35] benchmarks include programs with nested or multiple loops, making them suitable for evaluating the versatility and diversity of generated specifications. Please note that we assume the programs being verified are free of compilation errors, and the properties being verified are consistent with the programs. If there are any inconsistencies between the code and properties, AUTOSPEC is expected to fail the verification after the iterations end.

Baselines. For RQ1, as previous works have primarily relied on manually written specifications for the deductive verification of functional correctness for C/C++ programs [36,37], we then conduct our approach based on this baseline, and use the ablation study to demonstrate the contribution of different parts of the design in AUTOSPEC in RQ4. For RQ2, we compare with Code2Inv [2], a learning-based approach for generating linear loop invariants[7]. Although there are newer approaches built on Code2Inv such as CLN2INV [16], their replicable toolkit is only applicable to the benchmark they used (*i.e.*, SyGuS [33]) and incomplete, failing to apply to other benchmarks. Additionally, for RQ2, we also compared with Pilat [29] using the default settings.

Configuration. For implementation, we use ChatGPT's API *gpt-3.5-turbo-0613*. We configure the parameters in API as follows: `max_token`: 2048, `temperature`: 0.7. To show the generalizability of AUTOSPEC, we also utilize *Llama-2-70b* for conducting a comparable experiment (Section 5). Lastly, we employ FRAMA-C [28,38] and its WP plugin to verify the specifications.

4.2 RQ1. Effectiveness on General Specification

Table 2 shows the results of AUTOSPEC on the FRAMA-C-problems benchmark. This benchmark consists of 51 C programs, divided into eight categories (as indicated in the entry *Type*). Each type contains several programs. The size of the programs ranges from 9 to 36 lines of code (the entry *LoC*). We also list the number of functions and loop structures defined in the program. Most programs contain a main function, with one or more loop structures. Since we could not find other previous work that can automatically generate various types of specifications to complete the verification task on FRAMA-C-problems benchmark, we hereby show the effectiveness of AUTOSPEC in detail.

Overall, 31/51 of these programs can be successfully solved by AUTOSPEC. In particular, due to the randomness of LLMs, we ran the experiment five times for each program and reported the detailed results. The success rate is tabulated in

[7] We reproduce their implementation using the provided replicable package and run the tool on two additional benchmarks following their instructions. However, in their original setting, the maximal time limit for each program is set to 12 hours, which is far from affordable. So we lowered the threshold to 1 hour for efficiency.

Table 3: Effectiveness on a Real-world X.509 Certificate Parser Project

Function Information				AutoSpec					
Project	Function	Feature	LoC	Success	Ratio	Iterations	Generated Spec	Correct Spec	Time(s) mean ± std
X509-parser	`check_ia5_string`	loop; buffer pointer	60	✔	5/5	1,1,2,1,1	13,11,14,13,11	6/6	20.89 ± 10.51
	`verify_correct_time_use`	switch-case	90	✔	5/5	1,1,3,1,2	10,19,23,15,16	3/3	24.16 ± 11.73
	`bufs_differ`	loop; buffer pointer	55	✔	5/5	1,1,1,1,1	17,17,20,16,15	5/5	12.49 ± 5.11
	`parse_null`	call the function bufs_differ; buffer pointer	87	✔	2/5	-,-,1,1,-	25,36,44,33,34	13/13	260.96 ± 118.58
	`parse_algoid_params_none`	call the function `parse_null` and `bufs_differ`	136	✔	2/5	-,2,-,-,2	184,92,156,142,96	19/19	957.14 ± 446.56
	`time_components`	shift operation; multiple data type	63	✔	5/5	1,1,1,1,1	11,17,16,13,17	7/7	11.82 ± 5.34
Overall				**6 / 6**					214.58 ± 389.58

Table 2, column *Ratio*. It shows that the results are stable over five runs. Almost all passed cases can be successfully solved in five runs, with only a few exceptions (*e.g.*, $1/5, 4/5$). The stable result shows that the randomness of LLMs has little impact on the effectiveness of AutoSpec. Furthermore, AutoSpec enables an iterative enhancement on specification generation. We hereby show the number of iterations used for success generation (column *Iterations*). Most cases can be solved in the first iteration. While the iterative enhancement also contributes to certain improvements. For example, `add_pointers_3_vars.c` in the `pointers` category needs two more iterations to generate adequate specifications to pass the theorem prover. In addition, we also report the number of generated specifications that are correct by using the ground truth in the benchmark as a reference, as shown in column *Correct Spec*. We can see that for the failed cases, there is at least one generated specification that is correct. This shows that the generated specifications are not excessive, and still have the potential to improve. Finally, in terms of overhead (column *Time(s)*), AutoSpec processes a case in minutes, from 2.53 seconds to 12 minutes, with an average of 89.17 seconds.

A real-world X509 parser project. The X509-parser project, which aims to ensure the absence of runtime errors, has undergone verification by Frama-C and the ACSL specification language. Note that the specifications for this project were manually added throughout 5 months [3]. It is currently impractical to seamlessly apply AutoSpec to the entire project without human intervention. We manually extracted 6 representative functions without specifications. These functions handle pointer dereference, multiple data types, shift operations, *etc.*. For each function, we set a verification target that accurately describes its functional correctness properties. AutoSpec generates specifications for these functions, as shown in Table 3. Surprisingly, all 6 functions were solved by AutoSpec. Through our comprehensive manual examination of the generated specifications, we found that AutoSpec can generate a variety of specifications not previously written by the developer. These specifications play a crucial role in ensuring functional correctness. Considering that it takes five calendar months to write specifications for the whole X509-parser project [3], AutoSpec can automatically generate the required specifications for the functions in X509-parser in a few minutes. We believe that AutoSpec could be useful for real-world verification tasks.

4.3 RQ2. Effectiveness on Loop Invariants

Table 4 shows the effectiveness of AutoSpec in generating specifications of loop invariants compared with three baselines. In particular, the SyGuS bench-

Table 4: Effectiveness on Loop Invariants Synthesis

SyGuS [33] (133 C Programs with One Loop)

Info		AutoSpec		Pailt		Code2Inv		CLN2Inv	
ID	LoC	Sucess	Time(s)	Success	Time(s)	Success	Time(s)	Success	Time
1	29	✔	6.65	✕	–	✔	4950.78	✔	4.32
2	20	✔	5.97	✕	–	✕	–	✔	4.11
3	18	✕	36.24	✕	–	✕	–	✔	0.22
4	13	✕	32.28	✕	–	✕	–	✔	0.21
5	17	✔	19.22	✕	–	✕	–	✔	2.48
6	21	✕	142.95	✕	–	✕	–	✔	1.7
7	17	✔	57.15	✕	–	✔	128.03	✔	3.47
8	15	✔	92.75	✕	–	✔	72.74	✔	3.13
9	25	✔	39.66	✕	–	✕	–	✔	3.04
10	21	✕	295.83	✕	–	✔	53.39	✔	3.14
11	18	✔	77.48	✕	–	✔	145.02	✔	3.33
12	30	✔	106.137	✕	–	✔	71.97	✔	3.37
13	31	✔	240.24	✕	–	✔	39.82	✔	3.07
14	18	✔	79.59	✕	–	✔	21.9	✔	3.23
15	20	✔	13.66	✕	–	✔	274.79	✔	2.43
16	22	✔	30.31	✕	–	✕	–	✔	6.16
17	15	✔	24.97	✕	–	✕	–	✔	2.31
18	22	✔	23.7	✕	–	✕	–	✔	6.47
19	33	✔	21.41	✕	–	✕	–	✔	2.47
20	22	✔	16.59	✕	–	✔	53.57	✔	9.78
113 more cases are omitted due to space limitation									
Total		**114/133**		0/133		73/133	248±982.2	**124/133**	2.1 ± 2.32

OOPSLA-13 [34] (46 C Programs with Various Loop Types)

Info				AutoSpec		Code2Inv		CLN2INV	
ID	LoC	Type	Loop Num.	Success	**Time(s) mean ± std**	Success	Time(s)	Success	Time(s)
1	23	Linear	1	✔	6.25 ± 7.22	✔	337.8	●	–
2	27	Linear	1	✔	7.17 ± 7.38	✔	74.36	●	–
3	22	Linear	1	✔	113.47 ± 88.10	✔	46.22	✕	–
4	28	Linear	1	✔	6.04 ± 7.64	✕	–	●	–
5	30	Linear	1	✔	8.59 ± 9.80	✕	–	●	–
6	31	Linear	1	✔	41.62 ± 24.06	✕	–	✕	–
7	30	Linear	1	✔	12.66 ± 15.12	✕	–	●	–
8	24	Linear	1	✔	3.40 ± 3.25	✕	–	●	–
9	27	Linear	1	✕	–	✕	–	✕	–
10	26	Linear	1	✔	9.40 ± 7.29	✕	–	●	–
11	26	Linear	1	✔	16.82 ± 16.90	✕	–	✕	–
12	25	Linear	1	✔	51.27 ± 34.43	✕	–	✕	–
13	25	Linear	1	✔	12.57 ± 10.57	✕	–	●	–
14	29	Linear	1	✔	19.42 ± 11.52	✕	–	✕	–
15	37	Linear	1	✔	58.85 ± 13.83	✕	–	●	–
16	37	Linear	1	✕	–	✕	–	●	–
17	27	Linear	1	✔	4.80 ± 3.62	✔	66.4	✕	–
18	24	Linear	1	✔	19.86 ± 12.17	✕	–	✕	–
19	23	Linear	1	✔	10.34 ± 7.59	✕	–	●	–
20	25	Linear	1	✕	–	✕	–	✕	–
21	24	Linear	1	✔	12.09 ± 7.97	✕	–	●	–
22	22	Linear	1	✔	11.79 ± 11.14	✔	29.96	●	–
23	28	Linear	1	✔	32.71 ± 24.14	✕	–	✕	–
24	23	Linear	1	✔	12.84 ± 7.20	✕	–	✕	–
25	62	Linear	1	✔	6.64 ± 4.62	✕	–	✕	–
26	32	Linear	1	✔	44.89 ± 49.35	✕	–	✕	–
27	34	Linear	1	✕	–	✕	–	✕	–
28	26	Linear	1	✔	11.92 ± 9.78	✕	–	✕	–
29	35	Linear	1	✔	16.21 ± 14.83	✕	–	✕	–
30	29	Linear	1	✔	140.51 ± 67.06	✕	–	●	–
31	51	Multiple	4	✕	–	✕	–	✕	–
32	36	Multiple	2	✔	71.74 ± 60.13	✕	–	✕	–
33	27	Multiple	2	✔	94.02 ± 58.68	✔	26.97	✕	–
34	33	Multiple	2	✔	57.32 ± 39.28	✕	–	●	–
35	27	Nested	3	✔	58.26 ± 54.18	✕	–	●	–
36	30	Nested	2	✔	84.23 ± 57.52	✔	17.42	✕	–
37	23	Nested	2	✕	–	✔	0.26	●	–
38	22	Nested	3	✔	105.28 ± 60.01	✕	–	●	–
39	35	Nested	2	✔	96.50 ± 68.99	✕	–	✕	–
40	26	Nested	3	✔	139.88 ± 92.53	✔	147.89	✕	–
41	30	Nested	2	✔	177.43 ± 106.79	✕	–	✕	–
42	29	Nested	3	✔	228.71 ± 218.43	✕	–	✕	–
43	39	Nested	3	✕	–	✕	–	✕	–
44	61	Nested	4	✕	–	✕	–	✕	–
45	46	Nested	3	✔	202.78 ± 185.39	✕	–	✕	–
46	24	Nested	3	✔	86.61 ± 50.22	✕	–	●	–
Total				**38/46**	68.92 ± 171.13	9/46	83.0 ± 104.8	0/46	

SV-COMP (21 C programs with multiple/nested loops)

Benchmark Information				AutoSpec	
Type	**Program**	**LoC**	**Loop**	**Success**	**Time(s)**
quantifier-free	afnp2014_true-unreach-call.c	17	1	✔	40.69 ± 34.37
	bhmr2007_true-unreach-call.c	31	1	✔	4.50 ± 2.61
	cggmp2005_true-unreach-call.c	18	1	✔	128.04 ± 53.74
	count_up_down_true-unreach-call...	22	1	✔	18.44 ± 15.77
	css2003_true-unreach-call.c	19	1	✔	88.07 ± 74.35
	ddlm2013_true-unreach-call.c	33	1	✕	–
	down_true-unreach-call.c	21	2	✔	269.98 ± 224.05
	half_2_true-unreach-call.c	25	2	✕	–
	hhk2008_true-unreach-call.c	26	1	✔	8.49 ± 5.87
	jm2006_variant_true-unreach-call.c	30	1	✔	22.52 ± 14.55
	jm2006_true-unreach-call.c	25	1	✔	123.61 ± 100.31
	large_const_true-unreach-call.c	37	2	✕	–
	nest-if3_true-unreach-call.c	19	2	✔	107.05 ± 73.11
	nested6_true-unreach-call.c	31	3	✕	–
	nested9_true-unreach-call.c	23	3	✔	274.87 ± 156.77
	seq_true-unreach-call.c	33	3	✕	–
	sum01_true-unreach-call...	20	1	✔	8.12 ± 6.41
	terminator_03_true-unreach-call...	20	1	✔	26.09 ± 22.98
	up_true-unreach-call.c	30	2	✔	374.38 ± 291.44
quantifier	array_true-unreach-call1.c	13	1	✔	5.88 ± 3.78
	array_true-unreach-call2.c	18	1	✔	32.53 ± 27.11
Total				**16/21**	217.9 ± 451.7

mark consists of 133 C programs. Each program contains only one loop structure. We compare AutoSpec with three baselines: Pilat [29], Code2Inv [2] and CLN2INV [16] on this benchmark. The result is shown in Table 4 under *SyGuS* entry. Pailt fails to generate valid specifications for all cases in this benchmark, as all the specifications it generates are either unsatisfiable or irrelevant. On the other hand, Code2Inv and CLN2INV perform better, solving 73 and 124 programs, respectively. AutoSpec can handle a comparable number of cases, namely, 114 programs in this benchmark. Although CLN2INV can solve 10 more cases in this benchmark, it cannot handle any cases in the OOPSLA-13 benchmark. Although CLN2INV can successfully parse 19 out of 46 cases (denoted as ●), CLN2INV fails to construct satisfiable invariants that are adequate to verify the programs, resulting in a score of 0/46. This could be due to the overfitting of machine learning methods to specific datasets. Code2Inv, on the other hand, can handle 9 out of 46 cases. In comparison, AutoSpec can solve 38/46 (82.60%), which significantly outperforms existing approaches.

Furthermore, we consider a more difficult benchmark, SV-COMP. Due to the unsatisfactory results of the existing approaches, we have opted to exclusively present the results obtained using AutoSpec. As shown in Table 4 under *SV-COMP* entry, AutoSpec can solve 16 out of 21 programs with an average time of 3 minutes. Note that there are three programs with 3-fold nested loop structures in this benchmark. AutoSpec can solve one of them, while for the other two programs, AutoSpec can generate several satisfiable specifications, but there are still one or two specifications that cannot be generated after five iterations.

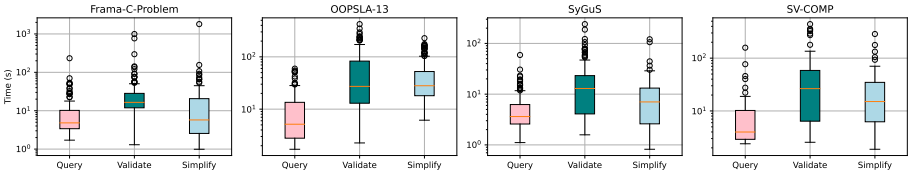

Fig. 4: **Overhead of AutoSpec on Four Benchmarks**.

4.4 RQ3. Efficiency of AutoSpec

In the first RQs, we can observe that AutoSpec can generate satisfiable and adequate specifications for the proof ranging from 2.35 to 739.62 seconds (*i.e.*, 0.04 to 12.33 minutes). In this RQ, we illustrate the composition of the overhead in AutoSpec across sub-tasks, *i.e.*, the time required for querying the LLM for specifications, validating and verifying the specifications against the theorem prover, and simplifying the specifications (optional). The results for these four benchmarks are presented in Figure 4.

We can see that for the four benchmarks, validating and verifying the specifications (*Validate*) takes the most time, ranging from 1.3 to 994.9 seconds. Querying the LLMs (*Query*) takes the least time, averaging less than 10 seconds. It is noteworthy that, unlike existing works that tend to generate a lot of candidate specifications and check their validity for one hour [16] to 12 hours [2], AutoSpec takes far less time in validating (*e.g.*, 1.2 seconds to 3.88 minutes). This is because AutoSpec generates fewer but higher-quality specifications. The efficiency of AutoSpec makes it both practical and cost-effective for various applications.

In addition, the time required for simplifying the specifications (*Simplify*) may vary depending on the number of generated specifications. A larger number of specifications leads to a longer simplification process. Nonetheless, given the fact that the simplification step is optional in AutoSpec, and considering the benefit of faster solving brought about by the concise and elegant specifications, the cost of simplification is justified.

4.5 RQ4. Ablation Study

Finally, we evaluate the contribution made by each part of AutoSpec's design. The results are shown in Table 5. We conduct the evaluation on Frama-C-problems benchmark [31] under seven settings: (1) - (4) settings under *Base ChatGPT* entry directly feed the C program together with the desired properties to be verified into ChatGPT, with zero-/one-/two-/three-shot. These settings are designed to compare with the decomposed manner adopted by AutoSpec. Setting (5) under entry *Decomposed* adopts the code decomposition (*i.e.*, Step 1 of AutoSpec) with three-shot, because it shows the best result according to the results of the previous settings. Settings (6) and (7) are respectively configured with only one pass (*i.e.*, without enhancement) and five iterations (*i.e.*, with enhancement), showing the improvement brought by the iterative enhancement

Table 5: Experiment Result of Ablation Study

Type	Base ChatGPT				Decomposition	Iterative Enhancement	
	(1) 0-shot	(2) 1-shot	(3) 2-shot	(4) 3-shot	(5) 3-shot	(6) Pass@1	(7) Iter@5
Loops	1	1	1	1	1	1	1
Immutable_arrays	0	0	0	0	3	4	5
Mutable_arrays	0	0	0	0	0	0	2
Arrays_and_loops	1	0	1	1	4	4	4
More_arrays	0	0	0	0	1	1	1
General_wp_problems	2	4	3	5	8	8	8
Pointers	0	2	0	1	6	6	7
Miscellaneous	1	1	1	1	3	3	3
Total	5	8	6	9	26	27	**31**

(Step (K) in Figure 3). The last row shows the total number of programs that can be successfully solved under the corresponding settings.

Table 5 shows an ascending trend in the number of solved programs, from 5 to 31 over 51. On the one hand, it is hardly possible to directly ask ChatGPT to generate specifications for the entire program. The input-output examples bring only a limited improvement (from 5 to 9) in the performance. On the other hand, code decomposition and hierarchical specification generation bring a significant improvement (from 9 to 26). This shows the contribution made by the first two steps of AutoSpec. Furthermore, the contribution of iterative enhancement can be observed in the last two columns, from 27 to 31. Overall, the ablation study shows that every step in AutoSpec has a positive impact on the final result and that the idea of code decomposition and hierarchical specification generation brings the biggest improvement.

4.6 Case Studies

We discuss two representative cases to show how iterative enhancement contributes (Fig. 5), and a situation where AutoSpec fails to handle (Fig. 6).

Case 1. A success case made by validation and iterative Enhancement. We show how specification validation and iterative enhancement help AutoSpec to generate satisfiable and adequate specifications. The program presented below computes the sum of three values stored in pointers. In the first iteration, only two specifications (lines 2 and 3) are generated, which respectively require three pointers should be valid (line 2), and the result of `add` is the sum (line 3). However, these two specifications alone are inadequate to verify the property due to the lack of a specification describing whether the values of the pointers have been modified within the `add` function. AutoSpec then inserts placeholders immediately after the two generated specifications and continues to the second iteration of enhancement. The subsequently generated specification (line 4) states the `add` function has no assignment behavior, making the verification succeed.

Case 2. A failing case due to missing context. We present an example where AutoSpec fails to generate adequate specifications due to *the lack of necessary context*. The code for the `pow` in `<math.h>` is not directly accessible. Currently, AutoSpec does not automatically trace all the dependencies and include their code in the prompt. LLMs can hardly figure out what `pow` is expected to do. As a

```
/*@
requires \valid(a) && \valid(b) && \valid(r);
ensures \result == *a + *b + *r;
assigns \nothing;
*/
int add(int *a, int *b, int *r)
{
  return *a + *b + *r;

}
```

Fig. 5: Case 1

```
#include <math.h>
int fun(int n) {
  double y = 0, i = 0;
  /*@ loop invariant y == (\pow(2,i))-1;
      loop invariant i <= n; */
  while(i <= n){
    y = y + pow(2.0, i);
    i = i + 1; }
  return y;
}
```

Fig. 6: Case 2

result, the specification for this function cannot be generated despite all attempts made by AutoSpec. This case shows a possible improvement by adding more dependencies in the prompt.

5 Threats to Validity

There are three major validity threats. The first concerns **the data leakage problem**. We addressed this threat in two folds. First, we directly apply LLMs to generate the specifications (Section 4.5). The unsatisfactory result (success rate: 5/51) shows that the chance of overfitting to the benchmark is low. Second, we followed a recent practice [39] for the data leakage threat. We randomly sampled 100 programs from three benchmarks in RQ2 (*i.e.*, SyGuS, OOPSLA-13, and SV-COMP) in a ratio of 50:25:25 and mutated these programs by variable renaming (*e.g.*, renaming `x` to `m`) and statement/branch switching (*e.g.*, negotiating the if-condition, and switching the statements in `if` and `else` branches) without changing the semantics of the program manually. Then we applied AutoSpec over the 100 mutated programs. The experiment shows that *98% results hold after the programs are mutated*. It further confirmed that the validity threat of data leakage is low. The second concerns **the generalizability to different LLMs**. To address this concern, we implemented AutoSpec to a popular and open-source LLM called *Llama-2-70b* and ran it on the same benchmark used in RQ1. Similar results were observed, with AutoSpec (Llama2) achieving a score of 25/51 compared to the score of 31/51 achieved by AutoSpec (ChatGPT). The third concerns **the scalability of AutoSpec**. We have evaluated a real-world X509-parser project and achieved unexpectedly good performance. However, completing the whole verification task on the entire project remains challenging. The evidence suggests that AutoSpec has the potential to assist participants in writing specifications for real-world programs.

6 Related Work

Specification Synthesis. While there exist various approaches and techniques for generating program specifications from natural language [40,41,42], this paper primarily focuses on specification generation based on the programming language. There has been work using data mining to infer specifications [43,44,45,46]. Several of these techniques use dynamic traces to infer possible invariants and preconditions from test cases, and static analysis to check the validity and completeness of the inferred specifications [47,48,49]. While others apply domain knowledge and statically infer specifications from the source code [50,43,51].

Several works have been conducted to address the challenging sub-problem of loop invariant inference, including CLN2INV [16], Code2Inv [2], G-CLN [17] and Fib [52]. Additionally, there are also studies dedicated to termination specification inference [23]. A recent study, SpecFuzzer [53], combines grammar-based fuzzing, dynamic invariant detection, and mutation analysis to generate class specifications for Java methods in an automated manner. Our approach differs from these techniques as it statically generates comprehensive contracts for each loop and function, yielding reliable outcomes necessary for verification.

Assisting Program Analysis and Verification with LLMs. In recent years, there has been a growing interest in applying LLMs to assist program analysis tasks [54], such as fuzz testing [55,56], static analysis [57,58,59], program verification [60,61,62], bug reproduction [63] and bug repair [64,65,66]. For example, Baldur [60] is a proof-synthesis tool that uses transformer-based pre-trained large language models fine-tuned on proofs to generate and repair whole proofs. In contrast, AutoSpec focuses on generating various types of program specifications and leveraging the auto-active verification tool to complete the verification task, while Baldur focuses on automatically generating proofs for the theorems. Li *et al.* [59] investigated the potential of LLMs in enhancing static analysis by posing relevant queries. They specifically focused on UBITest [67], a bug-finding tool for detecting use-before-initialization bugs. The study revealed that those false positives can be significantly reduced by asking precisely crafted questions related to function-level behaviors or summaries. Ma *et al.* [68] and Sun *et al.* [58] explore the capabilities of LLMs when performing various program analysis tasks such as control flow graph construction, call graph analysis, and code summarization. Pei *et al.* [69] use LLMs to reason about program invariants with decent performance. These diverse applications underline the vast potential of LLMs in program analysis. AutoSpec complements these efforts by showcasing the effectiveness of LLMs in generating practical and elegant program specifications, thereby enabling complete automation of deductive verification.

7 Conclusion

In this paper, we presented AutoSpec, a novel approach for generating program specifications from source code. Our approach leverages the power of Large Language Models (LLMs) to infer the candidate program specifications in a bottom-up manner, and then validates them using provers/verification tools and iteratively enhances them. The evaluation results demonstrate that our approach to specification generation achieves full automation and cost-effectiveness, which is a major bottleneck for formal verification.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (Nos. 62372304, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

1. Reiner Hähnle and Marieke Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, 2019.
2. Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31, 2018.
3. Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. Journey to a rte-free x. 509 parser. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC 2019)*, 2019.
4. Denis Efremov, Mikhail Mandrykin, and Alexey Khoroshilov. Deductive verification of unmodified linux kernel library functions. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II 8*, pages 216–234. Springer, 2018.
5. Frank Dordowsky. An experimental study using acsl and frama-c to formulate and verify low-level requirements from a do-178c compliant avionics project. *arXiv preprint arXiv:1508.03894*, 2015.
6. Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre, and Frédéeric Loulergue. A case study on formal verification of the anaxagoros hypervisor paging system with frama-c. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 15–30. Springer, 2015.
7. Nikolai Kosmatov, Matthieu Lemerre, and Céline Alec. A case study on verification of a cloud hypervisor by proof and structural testing. In *Tests and Proofs: 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings 8*, pages 158–164. Springer, 2014.
8. Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.
9. Yingwen Lin, Yao Zhang, Sen Chen, Fu Song, Xiaofei Xie, Xiaohong Li, and Lintan Sun. Inferring loop invariants for multi-path loops. In *2021 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 63–70. IEEE, 2021.
10. Shiwen Yu, Ting Wang, and Ji Wang. Loop invariant inference through smt solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 175–187, 2023.
11. Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.
12. Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.
13. Corneliu Popeea and Wei-Ngan Chin. Inferring disjunctive postconditions. In *Annual Asian Computing Science Conference*, pages 331–345. Springer, 2006.
14. Jingyi Su, Mohd Arafat, and Robert Dyer. Using consensus to automatically infer post-conditions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 202–203, 2018.
15. John L Singleton, Gary T Leavens, Hridesh Rajan, and David Cok. An algorithm and tool to infer practical postconditions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 313–314, 2018.

16. Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: Learning loop invariants with continuous logic network. In *International Conference on Learning Representations*, 2020.
17. Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–120, 2020.
18. Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, pages 634–640. Springer, 2009.
19. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 52–68. Springer, 2014.
20. Qiuye Wang, Mingshuai Chen, Bai Xue, Naijun Zhan, and Joost-Pieter Katoen. Synthesizing invariant barrier certificates via difference-of-convex programming. In *International Conference on Computer Aided Verification*, pages 443–466. Springer, 2021.
21. Yijun Feng, Lijun Zhang, David N Jansen, Naijun Zhan, and Bican Xia. Finding polynomial loop invariants for probabilistic programs. In *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*, pages 400–416. Springer, 2017.
22. Ting Gan, Bican Xia, Bai Xue, Naijun Zhan, and Liyun Dai. Nonlinear craig interpolant generation. In *International Conference on Computer Aided Verification*, pages 415–438. Springer, 2020.
23. Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *The 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 489–498, 2015.
24. Marcell Vazquez-Chanlatte and Sanjit A Seshia. Maximum causal entropy specification inference from demonstrations. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, pages 255–278. Springer, 2020.
25. OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.
26. Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*, 2023.
27. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specification. 2021.
28. Frama-C. Frama-c, software analzer, Accessed: 2024-01-15.
29. Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. Polynomial invariants by linear algebra. In *Automated Technology for Verification and Analysis: 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings 14*, pages 479–494. Springer, 2016.
30. Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever,

and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

31. Frama-C. A repository dedicated for problems related to verification of programs using the tool frama-c. Accessed: 2024-01-15.
32. a rte-free x.509 parser, Accessed: 2024-01-15.
33. Rajeev Alur, Dana Fisman, Saswat Padhi, Rishabh Singh, and Abhishek Udupa. Sygus-comp 2018: Results and analysis. *CoRR*, abs/1904.07146, 2019.
34. Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 443–456, New York, NY, USA, 2013. Association for Computing Machinery.
35. Dirk Beyer. Progress on software verification: Sv-comp 2022. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402, Cham, 2022. Springer International Publishing.
36. Patrick Baudin, François Bobot, Loïc Correnson, Zaynah Dargaye, and Allan Blanchard. Wp plug-in manual. *Frama-c. com*, 2020.
37. Allan Blanchard, Frédéric Loulergue, and Nikolai Kosmatov. Towards full proof automation in frama-c using auto-active verification. In *NASA Formal Methods Symposium*, pages 88–105. Springer, 2019.
38. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal aspects of computing*, 27:573–609, 2015.
39. Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1282–1294. ACM, 2023.
40. Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 383–396, Cham, 2023. Springer Nature Switzerland.
41. Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.
42. Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. Generation of formal requirements from structured natural language. In *Requirements Engineering: Foundation for Software Quality: 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24–27, 2020, Proceedings 26*, pages 19–35. Springer, 2020.
43. Nels E Beckman and Aditya V Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 211–221, 2011.

44. David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *The 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, 2007.
45. Tien-Duy B Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 106–117, 2018.
46. Hong Jin Kang and David Lo. Adversarial specification mining. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–40, 2021.
47. Glenn Ammons, Rastislav Bodik, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.
48. Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, pages 282–291, 2006.
49. Jeremy William Nimmer. *Automatic generation and checking of program specifications*. PhD thesis, Massachusetts Institute of Technology, 2002.
50. Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. *ACM SIGPLAN Notices*, 42(6):123–134, 2007.
51. Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 174–184, 2007.
52. Shang-Wei Lin, Jun Sun, Hao Xiao, Yang Liu, David Sanán, and Henri Hansen. Fib: Squeezing loop invariants by interpolation between forward/backward predicate transformers. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 793–803, 2017.
53. Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. Fuzzing class specifications. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1008–1020, 2022.
54. Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *CoRR*, abs/2308.10620, 2023.
55. Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, 2023*, pages 423–435. ACM, 2023.
56. Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 2023*, pages 919–931. IEEE, 2023.
57. Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. Automatically inspecting thousands of static bug warnings with large language model: How far are we? *ACM Transactions on Knowledge Discovery from Data*, 2024.
58. Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. Automatic code summarization via chatgpt: How far are we? *CoRR*, abs/2305.12865, 2023.
59. Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. Poster: Assisting static analysis with large language models: A chatgpt experiment. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, 2023*. IEEE, 2023.

60. Emily First, Markus N Rabe, Talia Ringer, and Yuriy Brun. Baldur: whole-proof generation and repair with large language models. In *ESEC/FSE '23: 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2023.
61. Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. *arXiv preprint arXiv:2310.04870*, 2023.
62. Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 2023.
63. Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 2312–2323. IEEE, 2023.
64. Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Examining zero-shot vulnerability repair with large language models. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2339–2356. IEEE, 2023.
65. Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1482–1494. IEEE, 2023.
66. Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1469–1481. IEEE, 2023.
67. Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul L. Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 221–232. ACM, 2020.
68. Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu. The scope of chatgpt in software engineering: A thorough investigation. *CoRR*, abs/2305.12138, 2023.
69. Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? 2023.