Expecto: Extracting Formal Specifications from Natural Language Description for Trustworthy Oracles

DONGJAE LEE, KAIST, Republic of Korea
KIHONG HEO, KAIST, Republic of Korea

Specification-Driven Development (SDD) has emerged as a promising paradigm in software development. This trend is fueled by recent advances in leveraging large language models (LLMs) to generate code from user intents expressed in natural language. However, the reliance on natural language specifications introduces ambiguity and challenges in ensuring correctness. To address these problems, we present Expecto, a system that automatically extracts trustworthy formal specifications from natural language intents. Expecto employs a neuro-symbolic approach, combining the strengths of LLMs and program synthesis techniques. Our key idea is to adopt a top-down, modular specification synthesis algorithm that breaks down the complex task of specification extraction into manageable units. The specifications are written in a domain-specific language (DSL) designed to succinctly express formal specifications of functional requirements. This modular synthesis with a concise DSL reduces the reasoning complexity for LLMs and enhances the accuracy of the extracted specifications. Our results demonstrate that Expecto significantly improves the accuracy and reliability of extracted specifications compared to a monolithic, purely LLM-based approach. Furthermore, when applied to real-world buggy programs in Defects4J, Expecto successfully generates formal specifications that detect more bugs than the baselines.

CCS Concepts: • **Theory of computation** → **Program specifications**; • **Software and its engineering** → *Formal software verification*; • **Computing methodologies** → Natural language processing.

Additional Key Words and Phrases: Specification, Code Generation, Program Verification, Large Language Model

ACM Reference Format:
Dongjae Lee and Kihong Heo. 2026. Expecto: Extracting Formal Specifications from Natural Language Description for Trustworthy Oracles. *Proc. ACM Program. Lang.* 10, PLDI, Article 254 (June 2026), 23 pages. https://doi.org/10.1145/3808332

1 Introduction

Recent advances in large language models (LLMs) have been changing the landscape of software development. Instead of writing code manually, developers can now describe their intent in natural language and let LLM-based coding assistants generate code automatically. In this new paradigm, the importance of specifying requirements and ensuring the correctness of generated code has increased significantly. This shift has led to a growing interest in Specification-Driven Development (SDD). In SDD, developers are encouraged to write specifications in a systematic way, so that the coding assistants generate code that satisfies the specification. Representative examples include SpecKit [8] from GitHub and KIRO [13] from AWS.

However, current state-of-the-art systems still rely solely on specifications written in natural language. SDD systems take natural language specifications as input in a format such as markdown

Authors' Contact Information: Dongjae Lee, KAIST, Daejeon, Republic of Korea, dongjae.lee00@kaist.ac.kr; Kihong Heo, KAIST, Daejeon, Republic of Korea, kihong.heo@kaist.ac.kr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART254
https://doi.org/10.1145/3808332

and merely use them as part of the prompts for code generation models. Such natural language specifications are often ambiguous and not machine-checkable. This limitation continues to impose a substantial burden on developers to verify that the generated code faithfully reflects their intent. In fact, a recent developer survey by Stack Overflow reveals that a large portion of developers experience difficulties when using AI coding assistants [25].

To tackle these challenges, we propose Expecto, a system that automatically extracts formal specifications from user intents expressed in natural language. The extracted formal specifications are written in our domain-specific language (DSL). Our DSL is an extension of the specification language used in Dafny [17]. The DSL can concisely and unambiguously express a wide range of functional requirements. This allows users to easily check that their intents are accurately reflected in the formal specifications. Moreover, the correctness of model-generated code can be automatically validated in combination with conventional verification and testing techniques.

A straightforward approach is to directly prompt LLMs to generate formal specifications. However, such monolithic and purely LLM-based specification generation faces two main problems. First, generating a complete formal specification in one shot often overwhelms LLMs. This complexity often exceeds the models' reasoning capabilities, leading to mistakes and omissions in the specifications. Second, validating the correctness of a generated specification only after the entire process can be inefficient. Even a small error in the middle of the specification can render the entire specification useless or require significant effort to repair. It is highly inefficient to discover and fix such small errors after the entire specification is generated.

Expecto leverages a neuro-symbolic approach that combines the strengths of LLMs and program synthesis techniques. We introduce a top-down and modular algorithm that synthesizes specifications hierarchically, from high-level structures to low-level details. Instead of generating the entire specification monolithically, Expecto first formalizes a high-level outline of the specification using an LLM, and then recursively formalizes each subcomponent (i.e., functions or predicates) in a top-down manner. This approach treats the generation of a formal definition for each predicate or function as an independent unit of work. Such modularity significantly reduces the reasoning complexity required for LLMs, enabling them to produce more accurate specifications.

Moreover, this modular design allows the continuous validation of partially extracted specifications during the synthesis process. At each step in the specification synthesis, Expecto checks the well-formedness and type correctness of newly generated components. Furthermore, Expecto checks consistency with user-provided input-output examples (if any) using an SMT solver. Functions that have not yet been formalized are treated as uninterpreted functions. This allows continuous validation even with incomplete specifications and helps prune incorrect partial specifications early in the synthesis process.

We implemented Expecto, which extracts formal specifications from natural language descriptions of programming problems. We evaluated the effectiveness of Expecto on competitive programming benchmarks (HumanEval+ [5, 20] and APPS [10]) and real-world bug benchmarks (Defects4J [12]). The results show that Expecto significantly outperforms the baseline approaches that directly generate formal specifications using LLMs in a monolithic manner. Expecto generated 1.20–2.81× more correct specifications than the baselines for competitive programming problems. Additionally, when applied to real-world buggy Java programs in Defects4J, Expecto successfully extracted 3.0–7.0× more correct formal specifications that can detect bugs compared to the baseline.

We summarize our contributions below:

- We present Expecto, a system that automatically extracts formal specifications from natural language descriptions.

Depot is a rectangular checkered field of n × m size. Each cell in a field can be empty (".") or it can be occupied by a wall ("*"). You have one bomb. If you lay the bomb at the cell (x, y), then after triggering it will wipe out all walls in the row x and all walls in the column y. You are to determine if it is possible to wipe out all walls in the depot by placing and triggering exactly one bomb. The bomb can be laid both in an empty cell or in a cell occupied by a wall.
Example
Input: (1,1,[["*"]])
Output: (True, 1, 1)

(a)

$$
\begin{aligned}
Spec(n, m, depot, possible, x, y) &\equiv \big(possible \longrightarrow \big(ValidPos(x, y, n, m) \land WipesAll(x, y, n, m, depot)\big)\big)\land \\
&\quad \big(\neg possible \longrightarrow (\forall r, c. ValidPos(r, c, n, m) \longrightarrow \neg WipesAll(r, c, n, m, depot)\big) \\
ValidPos(r, c, n, m) &\equiv (1 \le r \le n) \land (1 \le c \le m) \\
WipesAll(r, c, n, m, depot) &\equiv \forall i, j. IsWall(i, j, n, m, depot) \longrightarrow \big((i + 1 = r) \lor (j + 1 = c)\big) \\
IsWall(r, c, n, m, depot) &\equiv (0 \le r < n) \land (0 \le c < m) \land (depot[r][c] = \text{“}*\text{”})
\end{aligned}
$$

(b)

Fig. 1. A problem statement in natural language (a) and its formal specification extracted by EXPECTO (b)

- We design a neuro-symbolic specification synthesis algorithm based on a top-down and modular approach. The algorithm employs a continuous specification validation method to mitigate hallucinations of LLMs and improve the efficiency of specification extraction.
- We demonstrate the effectiveness of our approach on competitive programming benchmarks and real-world bug benchmarks. EXPECTO outperforms baseline approaches by a significant margin in generating correct formal specifications.
- Our tool and datasets are publicly available at GitHub (https://github.com/prosyslab/expecto-artifact) and Zenodo (https://doi.org/10.5281/zenodo.19450217).

2 Overview

2.1 Motivating Example

In this section, we illustrate the main idea of EXPECTO using a running example from a competitive programming problem in the APPS benchmark. Fig. 1(a) shows the problem statement written in natural language. Like other programming tasks for AI models, the problem statement describes the desired functionality of a program as well as a few input-output examples. At a high level, the functional specification (*Spec*) is defined as a combination of two sub-constraints: (1) the validity of the cell to place a bomb (*ValidPos*), and (2) the effect of placing a bomb on the walls (*WipesAll*). The effect of placing a bomb is further defined based on the presence of walls (*IsWall*). Each color highlights the corresponding part of the problem statement that describes each constraint.

Such problem statements often contain multiple hierarchical constraints with tricky logical relationships among them. For instance, according to the provided input-output example, the problem assumes the 1-based indexing for the depot cells like matrix notation in mathematics (*ValidPos*). However, the actual implementation and machine-checkable specifications typically use 0-based indexing, like array notation in programming languages (*WipesAll* and *IsWall*).

Existing work often fails to correctly formalize such complex specifications. State-of-the-art techniques, such as NL2POSTCOND [9], attempt to generate the entire formal specification in one shot. Such *monolithic* approaches typically incur significant reasoning burden on LLMs, leading to incorrect or incomplete specifications.

```
# Base prompt
assert possible == any(
all((cell[i][x - 1] == "*" or cell[y - 1][j] == "*")
all((x == j + 1 or y == i + 1)
  if cell[i][j] == "*" else True
    for i in range(n)
    for j in range(m))
  for y in range(1, n + 1)
  for x in range(1, m + 1)
) and (
  not possible or
  all((cell[i][x - 1] == "*" or cell[y - 1][j] == "*")
    if cell[i][j] == "*" else True
    for i in range(n)
    for j in range(m))
  for y, x in [(y, x)]
)

# Simple prompt
assert (not possible)
  or (1 <= y <= n and 1 <= x <= m)
```

(a)

```
def Spec(n: int, m: int, cell: list[string],
          possible: bool, x: int, y: int) {
  if possible then
    ValidPos(x, y, n, m) ∧ WipesAll(x, y, n, m, cell)
  else
    ∀(r: int, c: int) ::
      (ValidPos(r, c, n, m) ==>
       ¬WipesAll(r, c, n, m, cell))
}

def ValidPos(r: int, c: int, n: int, m: int) {
  (1 <= r <= n) ∧ (1 <= c <= m)
}

def WipesAll(r: int, c: int, n: int, m: int,
              cell: list[string]) {
  ∀(wr: int, wc: int) ::
    (IsWall(wr, wc, n, m, cell) ==>
      (wr + 1 == r ∨ wc + 1 == c))
}

def IsWall(r: int, c: int, n: int, m: int,
            cell: list[string]) {
    (0 <= r < n) ∧ (0 <= c < m) ∧ (cell[r][c] == '*')
}
```

(b)

Fig. 2. Formal specifications generated by NL2Postcond (a) and Expecto (b)

For example, Fig. 2(a) shows a formal specification generated by NL2Postcond [9] in the form of an assertion statement in Python. Their approach merely relies on prompt engineering to guide the model to generate the entire specification. The first assertion of Fig. 2(a) is generated by their Base prompting strategy. While effective for simple tasks, this monolithic approach struggles with complex specifications that involve multiple hierarchical constraints. The red-colored parts in the specification indicate errors where the generated specification does not align with the intended functionality. The first error should be replaced with the condition in the green-colored part, and the second error should be removed entirely. To handle the complexity, NL2Postcond introduces another prompting strategy called Simple, which is shown in the second assertion of Fig. 2(a). However, this strategy produces a too weak specification that fails to capture the intended functionality.

On the other hand, Expecto generates a correct formal specification shown in Fig. 2(b). The specification is written in our domain-specific language (DSL). The DSL is an extension of the specification language used in Dafny [17]. This allows Expecto to express complex functional requirements concisely and unambiguously.

Expecto generates specifications in a *top-down and modular* manner. Predicate `Spec` at the top level captures the overall functionality by composing two sub-constraints, `ValidPos` and `WipesAll`. Each of the sub-constraints is defined independently. This allows LLMs to focus on one small piece of the specification at a time, significantly reducing the reasoning complexity. The generated specification itself is modular, making it easy for humans to compare the generated specification with the original user intent.

Moreover, Expecto continuously validates each extracted component during the extraction process. For example, when formalizing each predicate, Expecto checks the well-formedness, type correctness, and consistency with the provided input-output examples. This continuous validation

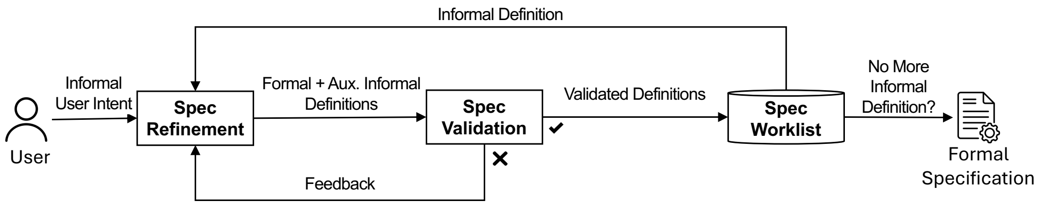

Fig. 3. Overview of EXPECTO

enables early detection of errors and provides feedback to refine the specification iteratively. We will provide the details of our approach in the following sections.

2.2 Our Approach

Fig. 3 presents an overview of EXPECTO. Given a functional requirement expressed in natural language along with input-output examples (if any), EXPECTO synthesizes a formal specification in our DSL. EXPECTO is based on a neuro-symbolic approach that combines the strengths of large language models (LLMs) and symbolic reasoning. At a high level, EXPECTO employs a top-down modular specification synthesis algorithm that iteratively refines and validates the specification. The algorithm consists of two main processes: SpecRefinement and SpecValidation.

SpecRefinement takes a natural language description and generates the formal definition of the corresponding function or predicate. If the formalization requires auxiliary functions, this process also generates natural language descriptions for these functions instead of formalizing them immediately.

SpecValidation validates whether the extracted specifications are correct. We check the well-formedness, type correctness, and consistency with the provided input-output examples. Because of the modular nature of our approach, SpecValidation can validate each component even when the entire specification is not yet fully formalized. When errors are detected in the formalization during SpecValidation, SpecValidation constructs feedback to SpecRefinement to correct the error. The validated definitions and the yet-to-be formalized descriptions are accumulated in a worklist.

The algorithm iteratively selects a candidate informal definition from the worklist and performs the whole process again. This process terminates when all functions or predicates are formally defined through iterative execution of these two procedures. In the rest of this section, we detail each of these two processes.

2.2.1 SpecRefinement. The SpecRefinement process maintains and updates two sets of definitions: (1) a set D_f of formally defined predicates and functions, and (2) a set D_{nl} of informally described predicates and functions in natural language, yet to be formalized. Initially, D_f is empty, and D_{nl} contains only the top-level user intent in natural language. For example, the initial set D_{nl} for the example in Fig. 1 is as follows:

Type	**Description**
`def Spec(n: int, m: int, ...)`	*"Depot is a rectangular ..."*

The type of the top-level specification Spec is inferred from input-output examples provided in the user intent. When no examples are given, EXPECTO infers the type by prompting an LLM with the user intent.

The key feature of our approach is to synthesize specifications in a *top-down and modular* manner. For each iteration, SpecRefinement selects a target from D_{nl} and generates its formal definition

You are an expert in formal methods and program specification. Your goal is to generate its formal definition using the provided DSL. *[DSL description]* Here is the target informal definition: *[target informal definition]* Here are the currently defined definitions: *[current D_f and D_{nl}]* Avoid introducing single complex definition. Instead, introduce new auxiliary definitions that are simpler and more composable.

Fig. 4. Prompt used in the SpecRefinement process

using an LLM. If the formalization requires auxiliary predicates or functions, Expecto does not attempt to formalize them immediately. Instead, we query the LLM to describe these auxiliary components in natural language and add them to D_{nl} for future refinement. Fig. 4 illustrates the prompt used in this process. The prompt is carefully designed to encourage the LLM to decompose complex specifications into simpler, more manageable components. In the example task, this step generates the formal definition of `Spec` as shown in Fig. 2(b), along with two auxiliary predicates, `ValidPos` and `WipesAll`, described in natural language:

Type	Description
`def ValidPos(r: int, c: int, ...)`	*"Checks if the cell ..."*
`def WipesAll(r: int, c: int, ...)`	*"Determines if placing a bomb ..."*

Expecto then validates the correctness of the generated definitions, which will be detailed in the next section. Once validated, Expecto adds the formally defined `Spec` to D_f and the two auxiliary predicates to D_{nl} (i.e., D_f = {`Spec`} and D_{nl} = {`ValidPos`, `WipesAll`}).

In the next iteration, Expecto selects one of the auxiliary predicates from D_{nl}, say `WipesAll`, and repeats the SpecRefinement process. Expecto generates the formal definition of `WipesAll` in a similar manner as shown in Fig. 2(b). This time, `WipesAll` requires another auxiliary predicate `IsWall`, which is again described in natural language and added to D_{nl}.

Expecto repeats this process until all functions and predicates are formally defined, or a predefined maximum number of refinement attempts is reached. For each SpecRefinement, the current D_f and D_{nl} are provided to the LLM to encourage the model to utilize previously defined components. Additionally, the initial user intent is provided throughout the processes so that the LLM can continuously consider the final goal.

2.2.2 *SpecValidation.* Throughout the extraction process, SpecValidation continuously validates the evolving specification against three properties: well-formedness, type correctness, and consistency with provided input-output examples (if any). A key advantage of our modular refinement approach is that validation can proceed even when the specification is only partially formalized. Expecto enables this by encoding informal definitions into uninterpreted functions, allowing early detection and correction of incorrect formalizations.

Consider the motivating example in Fig. 1. Suppose `Spec` is correctly defined as shown in Fig. 2(b) and `ValidPos` is incorrectly formalized as follows:

```
def ValidPos(n: int, m: int, x: int, y: int) { (0 <= x < n) ∧ (0 <= y < m) }
```

Note that this definition assumes that the depot uses 0-based indexing. Thus, a cell at position `(x,y)` is valid if `x` and `y` are in the range of $[0, \mathtt{n})$ and $[0, \mathtt{m})$ respectively. However, according to the input-output example, the user intent actually requires 1-based indexing; if a depot has size `(1, 1)`, placing a bomb at cell `(1, 1)` is valid.

Now, Expecto validates the formal definition of `ValidPos` with respect to the provided input-output example. In particular, Expecto assigns the input-output examples to the arguments of the top-level predicate `Spec`: `Spec(1, 1, [['*']], True, 1, 1)`. Note that we cannot directly compute the truth value of `Spec` even with the input-output example, because `WipesAll` is not yet

formally defined. Thus, we treat the informally defined predicate WipesAll as an uninterpreted function and formulate the validation problem as a satisfiability query. For example, we encode the ground instance of Spec for the example into the following formula:

$$(\text{true} \rightarrow (0 \leq 1 < 1) \wedge (0 \leq 1 < 1) \wedge \texttt{WipesAll}(1, 1, 1, 1, [[\text{`*'}]])) \wedge$$
$$(\neg\text{true} \rightarrow \forall r, c.(0 \leq r < 1) \wedge (0 \leq c < 1) \rightarrow \neg\texttt{WipesAll}(r, c, 1, 1, [[\text{`*'}]]))$$

We check the satisfiability of this formula using an SMT solver. Because of the incorrect formalization of ValidPos, the SMT query cannot be satisfied. Thus, Expecto invalidates the current formal definition of ValidPos without needing to wait for the entire specification to be formalized.

When an incorrect formalization is detected, Expecto invokes the SpecRefinement process again to correct the error. This time, the prompt provided to the LLM includes the error message indicating the inconsistency with the input-output example. When syntax errors or type errors are detected, the error location and error message are conveyed. This correction process repeats until validation succeeds or the maximum number of refinement attempts is reached.

If validation succeeds, Expecto adds the generated formal and informal definitions to a worklist for further processing. In the next iteration, Expecto selects another informal definition from the worklist and repeats the SpecRefinement and SpecValidation processes.

2.2.3 Top-down Specification Synthesis with Tree Search. We further incorporate a tree search strategy into our top-down modular specification synthesis. Instead of generating a single candidate formal specification at each refinement step, Expecto explores multiple candidate specifications in a tree structure. Due to the probabilistic nature of LLMs, generating multiple candidates increases the robustness of our approach against potential errors in individual generations. This strategy allows Expecto to systematically employ *test-time scaling* [31] for specification synthesis. This section describes how we extend our basic system to support the tree search strategy.

Consider the example in Fig. 1 again. With tree search, Expecto generates multiple candidate formal definitions each of which may introduce different auxiliary predicates. Suppose Expecto generates the following two candidate formal definitions for Spec at the first refinement step:

Formal Definition	**Informal Auxiliary Definitions**
$\texttt{Spec}_1$	$\{\texttt{ValidPos}_1, \texttt{WipesAll}_1\}$
$\texttt{Spec}_2$	$\{\texttt{ValidPos}_2, \texttt{WipesAll}_2\}$

where $\texttt{ValidPos}_i$ and $\texttt{WipesAll}_i$ are auxiliary predicates or functions described in natural language that are used in the corresponding formal definition $\texttt{Spec}_i$. Then we validate each candidate pair of formal definition and auxiliary definitions in SpecValidation as described in § 2.2.2. Finally, we store each validated candidate $\langle\{\texttt{Spec}_i\}, \{\texttt{ValidPos}_i, \texttt{WipesAll}_i\}\rangle$, into the worklist. Notice that each set in a pair represents a set of formal and informal definitions, respectively, which explores the same path in the tree search.

Expecto guides the tree search using a greedy strategy. We prioritize candidates that have fewer informal definitions and more formal definitions in the worklist. For example, if the worklist contains the following two candidates:

$$\{\langle\{\texttt{Spec}_1, \texttt{ValidPos}_1\}, \{\texttt{WipesAll}_1\}\rangle, \ \langle\{\texttt{Spec}_2\}, \{\texttt{ValidPos}_2, \texttt{WipesAll}_2\}\rangle\}$$

The first candidate has two formal definitions and one informal definition, while the second candidate has one formal definition and two informal definitions. In the next iteration, Expecto selects the first candidate for further refinement since it has fewer informal definitions to be formalized. Note that each refinement may generate different sets of auxiliary definitions. Thus, the tree search explores various combinations of formal and informal definitions.

This tree search strategy significantly enhances the robustness of Expecto. When one candidate formalization fails validation, Expecto can explore other candidates in the worklist. In our implementation, Expecto also employs a caching mechanism to avoid redundant computations during the tree search. We keep track of previously refined formal definitions and reuse them when the same informal definition is required to be formalized again.

3 Design

In this section, we formally define our design of Expecto. We first formulate the specification extraction problem (§ 3.1). Next, we introduce our domain-specific language for writing formal specifications (§ 3.2). Finally, we present the top-down specification synthesis algorithm (§ 3.3).

3.1 The Specification Extraction Problem

Specification extraction aims to generate formal specifications that accurately capture the user's intent expressed in natural language. Let $P(\phi)$ denote a set of programs that satisfy a specification ϕ. Given a user intent ϕ_{nl} in natural language, the problem is to find a corresponding formal specification ϕ_f expressed in a formal language $\mathcal{L}$, i.e., $P(\phi_f) = P(\phi_{nl})$. Based on this, we define two properties of a generated formal specification: *soundness* and *completeness*.

An extracted formal specification ϕ_f is *sound* if ϕ_f is more restrictive than ϕ_{nl}:

$$Sound(\phi_f, \phi_{nl}) \;\equiv\; P(\phi_f) \subseteq P(\phi_{nl}).$$

However, this property cannot be automatically verified since ϕ_{nl} is defined in natural language. Thus we can approximate the soundness by using a set of incorrect input-output pairs T^- that do not satisfy the user's intent. In other words, we can check approximate soundness by ensuring that ϕ_f rejects all incorrect input-output pairs in T^-:

$$\widehat{Sound}(\phi_f, T^-) \;\equiv\; \forall (i, o) \in T^-.\ \neg\phi_f(i, o)$$

where $\phi_f(i, o)$ denotes the instantiation of the specification ϕ_f with input-output pair (i, o).

We extend this definition and reformulate the approximate soundness in terms of satisfiability for practical reasons:

$$\widehat{Sound}(\phi_f, T^-) \;\equiv\; \forall (i, o) \in T^-.\ \phi_f(i, o) \text{ is UNSAT}$$

This formulation allows us to naturally handle partial specifications that include uninterpreted functions. In addition, when there are no incorrect input-output pairs, we can at least check that the specification is not trivially true for all inputs:

$$\widehat{Sound}(\phi_f, \emptyset) \;\equiv\; \neg\phi_f \text{ is SAT}$$

An extracted formal specification ϕ_f is *complete* if ϕ_f captures all programs satisfying the user intention ϕ_{nl}:

$$Complete(\phi_f, \phi_{nl}) \;\equiv\; P(\phi_{nl}) \subseteq P(\phi_f)$$

Similarly to soundness, completeness can be approximated using a set of correct input-output pairs T^+ that satisfy the user's intent. We extend this definition and formulate the approximate completeness in terms of satisfiability:

$$\widehat{Complete}(\phi_f, T^+) \;\equiv\; \forall (i, o) \in T^+.\ \phi_f(i, o) \text{ is SAT}$$

Even if there are no correct input-output pairs, we can at least check that the specification is not contradictory (i.e., always false):

$$\widehat{Complete}(\phi_f, \emptyset) \;\equiv\; \phi_f \text{ is SAT}$$

Specification	ϕ	::=	δ^+
Definition	δ	::=	def $id(\overrightarrow{id\colon \tau})\colon \tau \quad \{e\}$
		|	def $id(\overrightarrow{id\colon \tau})\colon (id\colon \tau) \quad \{r\}$
		|	def $id(\overrightarrow{id\colon \tau})\colon \tau \quad \{desc\}$
Contract	r	::=	$\langle$req e, ens $e\rangle$
Type	τ	::=	int | real | bool | string | char
		|	option[τ] | list[τ] | tuple[$\vec{\tau}$] | map[τ, τ]
		|	record[$\overrightarrow{id\colon \tau}$]
Expression	e	::=	$\forall x.e \mid \exists x.e \mid id(e) \mid e \wedge e \mid e \vee e \mid \neg e \mid e \Rightarrow e \mid e \Leftrightarrow e$
		|	$e + e \mid e - e \mid e \times e \mid e \,/\, e$
		|	map(e, e) | filter(e, e) | fold(e, e, e) | len(e) | ...

Fig. 5. Our Domain-Specific Language for Writing Formal Specifications

Expecto starts from a trivially complete specification and refines it iteratively by adding constraints to the specification. This design choice is motivated by the observation that users typically provide only correct input-output pairs as test cases. For example, many widely used AI programming benchmark datasets such as HumanEval [5], MBPP [1], and APPS [10] provide only correct input-output pairs. This makes it easier to check completeness than soundness. Thus, our algorithm iteratively strengthens the specifications while maintaining completeness. In the rest of this paper, we describe our approach based on this design choice.

3.2 Domain-Specific Language for Specification

Fig. 5 presents our domain-specific language (DSL) for writing formal specifications. Our DSL is an extension of the specification language used in Dafny [17]. In particular, we extend Dafny's specification language with a new definition syntax for informal definitions (*desc*). This allows us to systematically connect informal specifications to formal specifications within a unified framework.

A specification (Specification) is a set of function/predicate definitions. A specification contains a special top-level predicate *Spec* that takes input and output variables as arguments. The *Spec* predicate defines the overall behavior of the target program and is used as the starting point of the specification synthesis process.

A function or predicate definition (Definition) can be written in three different ways: (1) directly defined with expressions (e), (2) specified with preconditions and postconditions (r), or (3) described with only types and natural language descriptions (*desc*). Notice that each style of definition serves a different purpose in the specification synthesis process. The first style is for fully formalized definitions using logical expressions. The second style also provides formal definitions but allows more abstract definitions by using contracts (preconditions and postconditions). This style is useful for defining functions that are complex to express directly but can be succinctly captured with their preconditions and postconditions. For example, sorting functions typically have simpler contracts than their full implementations. The third style is designed to support informal definitions that are not yet formalized. This feature enables language models to naturally define functions in a top-down manner without being distracted by low-level details at each refinement step.

3.3 Top-Down Specification Synthesis with Tree Search

To effectively explore various refinement paths using LLMs, we incorporate a tree search algorithm into the basic refinement and validation processes of Expecto. Instead of generating a single formal definition candidate at each refinement step, this tree search generates multiple formal definition candidates and explores them in a prioritized manner. This section details the algorithm.

Algorithm 1: Top-Down Spec Synthesis Algorithm with Tree Search

Input: User intent ϕ_{nl} in natural language, and sets of correct and incorrect test cases T^+ and T^-
Output: Most refined specification $\langle D_f, D_{nl} \rangle$

```
1  Q ← {⟨∅, {φ_nl}⟩}
2  repeat
3      ⟨D_f, D_nl⟩ ← PriorityDequeue(Q)                          ▹ Most refined specification
4      if D_nl = ∅ then
5          return ⟨D_f, ∅⟩
6      R ← SpecRefinement(φ_nl)                                   ▹ Refinement using LLM
7      foreach ⟨φ_f, D_nl'⟩ ∈ R do
8          repeat
9              isValid, feedback ← SpecValidation(φ_f, D_f, D_nl', T+, T−)
10             if isValid then
11                 D_f ← D_f ∪ {φ_f}
12                 D_nl ← D_nl ∪ D_nl'
13                 Q ← Q ∪ {⟨D_f, D_nl⟩}
14                 break
15             ⟨φ_f, D_nl'⟩ ← SpecRepair(φ_f, D_nl', feedback)          ▹ Repair using LLM
16         until maximum repair iterations
17 until maximum refinement iterations
18 ⟨D_f, D_nl⟩ ← PriorityDequeue(Q)                              ▹ Most refined specification
19 return ⟨D_f, D_nl⟩
```

Algorithm 1 presents the top-down specification synthesis algorithm with tree search. Expecto takes a user intent ϕ_{nl} in natural language and a (possibly empty) set of correct and incorrect input-output pairs T^+ and T^-. The algorithm initializes a priority queue Q with the *root state* consisting of an empty formal definition set and a singleton set of user intent in natural language (line 1). Each state in the priority queue is represented as a tuple $\langle D_f, D_{nl} \rangle$, where D_f is a set of formal definitions and D_{nl} is a set of informal definitions in natural language. The formal and informal definitions in a state together form a *specification candidate*, which shares the same refinement history.

Then, Expecto iteratively refines informal definitions using a greedy tree search strategy. For each iteration, we select the *most refined* specification candidate $\langle D_f, D_{nl} \rangle$ from the priority queue Q (line 3). The degree of refinement is determined by the following partial ordering:

$$\langle D_f, D_{nl} \rangle \sqsubseteq \langle D_f{}', D_{nl}{}' \rangle \iff |D_{nl}{}'| < |D_{nl}| \lor (|D_{nl}| = |D_{nl}{}'| \land |D_f| \leq |D_f{}'|)$$

That is, we prioritize specification candidates with fewer informal definitions. Among those with the same number of informal definitions, we prioritize those with more formal definitions. If there are no remaining informal definitions in a candidate, we return the candidate as the final result.

For the most refined specification candidate $\langle D_f, D_{nl} \rangle$, we refine one informal definition ϕ_{nl} from D_{nl} (line 6). In this process, we query an LLM to generate multiple refined candidates $\langle \phi_f, D_{nl}{}' \rangle$ from ϕ_{nl}, where each ϕ_f is a formalization candidate of ϕ_{nl} and each $D_{nl}{}'$ is an auxiliary natural language definition required to define ϕ_f. This multiple candidate generation allows Expecto to explore various refinement paths in parallel via tree search. In our implementation, we generate 3 different refined candidates. This process is implemented using structured prompt engineering and lightweight parsing.

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{S}} & : & \mathcal{S} \to C \\
\mathcal{T}_{\mathcal{S}}(\{f\} \cup F) & = & \mathcal{T}_{\mathcal{D}}(f) \land \mathcal{T}_{\mathcal{S}}(F) \\
\mathcal{T}_{\mathcal{S}}(\emptyset) & = & \mathsf{true} \\
\\
\mathcal{T}_{\mathcal{D}} & : & \mathcal{D} \to C \\
\mathcal{T}_{\mathcal{D}}(\mathtt{def}\ f(x : \tau_1) : \tau_2\ \{e\}) & = & \forall x. f(x) = e \\
\mathcal{T}_{\mathcal{D}}(\mathtt{def}\ f(x : \tau_1) : \tau_2\ \{r\}) & = & \forall x. e_1 \Rightarrow e_2 \quad \text{where } r = \langle \mathtt{req}\ \ e_1, \mathtt{ens}\ \ e_2 \rangle \\
\mathcal{T}_{\mathcal{D}}(\mathtt{def}\ f(x : \tau_1) : \tau_2\ \{desc\}) & = & \mathsf{true}
\end{array}
$$

Fig. 6. Transformation rules from our DSL to SMT Expressions. For brevity, we assume that functions have only one argument; the rules can be easily extended to support multiple arguments.

Then, we validate each refined candidate $\langle \phi_f, D_{nl}' \rangle$ using the SpecValidation process (line 9). We check the well-formedness and well-typedness using our DSL parser and type checker. If a set of input-output pairs is provided, we also check the consistency of the candidate with the input-output pairs. For consistency checking, we instantiate the formal definitions with each input-output pair, while encoding the informal definitions as uninterpreted functions in SMT. At this point, we check the approximated soundness and completeness as defined in § 3.1. Through this process, we can determine whether the current partial specification still has a chance of being a valid specification according to the given test cases. If the candidate passes all validation checks, we add the candidate to the priority queue Q. The details of the validation and the SMT encoding are presented in § 3.4.

If the candidate fails any validation check, we repair the candidate using the SpecRepair process (line 15). The repair process is based on LLMs that utilize the feedback from the SpecValidation process. For the syntax and type checker, we provide the error locations and causes as feedback. For the consistency check with input-output examples, the failed input-output pairs are provided to the LLM. In our implementation, we repair the candidate up to three times.

In the implementation, we optimize the tree search process by employing a caching mechanism. EXPECTO stores previously refined informal definitions along with their refinement results. Before refining a new informal definition, we check whether it has already been refined before. If it has, we directly reuse the cached refinement results instead of querying the LLM again (line 6). We use the type information of the functions and the embeddings of their natural language descriptions as keys for the cache. We first search for functions with completely matching types in the cache. Then, we select the refinement result with the highest embedding similarity among the functions with matching types, where the natural language description embedding exceeds the threshold τ.

3.4 Continuous Specification Validation

For each validation step, EXPECTO encodes specification candidates written in our DSL into SMT formulas. Fig. 6 presents the translation rules. Given a newly refined formal definition ϕ_f and a set of already-formalized definitions D_f, the translation $\mathcal{T}_{\mathcal{S}}(\{\phi_f\} \cup D_f)$ encodes the entire specification into a single SMT constraint. The encoding for each definition is handled by $\mathcal{T}_{\mathcal{D}}$ which follows the standard encoding for function definitions and contracts. For informal definitions, we do not attempt to formalize the definitions; instead, we encode them as uninterpreted functions in SMT.

In the presence of input-output pairs, we instantiate the encoded SMT constraints with each pair. Given a set of correct input-output pairs $T^+ = \{(i_1, o_1), \ldots, (i_n, o_n)\}$, and a set of incorrect input-output pairs $T^- = \{(i_1', o_1'), \ldots, (i_m', o_m')\}$, we check whether the current specification candidate is consistent with all the input-output pairs:

$$\forall (i, o) \in T^+ . \mathcal{T}_{\mathcal{S}}(\{\phi_f\} \cup D_f)(i, o) \text{ is SAT} \qquad \forall (i', o') \in T^- . \neg \mathcal{T}_{\mathcal{S}}(\{\phi_f\} \cup D_f)(i', o') \text{ is SAT}$$

where $\mathcal{T}_{\mathcal{S}}(F)(i, o)$ denotes the SMT formula obtained by instantiating the input and output variables in $\mathcal{T}_{\mathcal{S}}(F)$ with i and o, respectively. The first part ensures that all correct input-output pairs potentially satisfy the specification, while the second part ensures that all incorrect input-output pairs potentially do not satisfy the specification.

If no input-output pairs are provided, we check the satisfiability of $\mathcal{T}_{\mathcal{S}}(\{\phi_f\} \cup D_f)$ (the specification is not contradictory) and $\neg\mathcal{T}_{\mathcal{S}}(\{\phi_f\} \cup D_f)$ (the specification is not trivial). This check guarantees that there exists at least one input-output pair that satisfies the specification and vice versa. In other words, it ensures that the specification is not contradictory and not trivial.

Note that our validation guarantees only necessary conditions for the correctness of the specification. A candidate specification is accepted when it is consistent with the provided test cases. However, these test cases may be insufficient to fully capture the user's intent. As a result, the generated specifications may fail to generalize to unseen test cases that are also consistent with the intended behavior.

4 Evaluation

We evaluate Expecto to answer the following four research questions:

RQ1 How effective is Expecto in generating formal specifications from informal descriptions?
RQ2 How do the top-down specification synthesis and tree search contribute to the performance of Expecto?
RQ3 How does continuous specification validation contribute to the performance of Expecto?
RQ4 Can Expecto be practically applied to detect functional bugs in real-world software?

4.1 Experimental Setup

4.1.1 Implementation. We implemented Expecto using Z3 [18] as the SMT solver, GPT-4.1-mini [23] as the LLM, and text-embedding-3-large [24] as the embedding model for caching. Our implementation of Expecto consists of 18K of code in Python including the parser and type checker for our DSL. All the experiments were conducted on a machine with 64 cores of Xeon Gold 6226R CPU and 512 GB RAM. We set the timeout for each SMT query to 30 seconds.

4.1.2 Baselines. We compare Expecto with NL2Postcond, a state-of-the-art tool that translates natural language descriptions into postconditions of desired functions [9]. To the best of our knowledge, NL2Postcond is the only prior work that directly produces specifications using natural language descriptions. NL2Postcond uses LLMs to generate postconditions in the form of assertions in the target programming language (e.g., Python, Java). We use two variants of NL2Postcond, each of which employs a different prompting strategy described in their paper, and a simplified version of Expecto with minimal validation, called Expecto$^-$:

- Base instructs the LLM to generate detailed postconditions to fully capture the user's intent.
- Simple instructs the LLM to generate simple postconditions that capture an aspect of the intent.
- Expecto$^-$ is a variant of Expecto without the top-down specification synthesis and continuous specification validation with test cases. It generates the entire specification monolithically and performs only syntax, type, and logical consistency checking, with up to three repair attempts.

For NL2Postcond, we used their publicly available artifact and prompts for our experiments.

4.1.3 Benchmarks. We evaluated Expecto on three widely used benchmarks:

- HumanEval+ [20]: A testcase-enhanced version of HumanEval [5], which is one of the most widely used benchmarks for code generation tasks. This benchmark contains 164 problems that require implementing basic algorithms and data structures in Python.

- APPS [10]: A dataset of competitive programming problems that require complex algorithmic reasoning, data structure manipulation, and mathematical computations. We selected 127 problems from APPS that have at least 100 test cases.
- Defects4J [12]: A benchmark containing real-world Java software bugs from open-source projects. We selected 501 methods from 336 bugs that can be reproduced in Java 8 or later because NL2Postcond only supports these versions.

For HumanEval+ and APPS, we used the problem descriptions as natural language specifications. For Defects4J, we extracted the method-level Javadoc comments as natural language specifications.

4.1.4 Evaluation Criteria. Since it is challenging to directly evaluate the correctness of generated formal specifications, we use test cases to evaluate the correctness, as in prior work [9]. However, the AI programming benchmarks, HumanEval+ and APPS, only provide correct input-output pairs. To measure the quality of the generated specifications against incorrect input-output pairs, we generated (likely) incorrect pairs from the correct pairs using a simple heuristic. Given a set of correct input-output pairs, we randomly shuffled the outputs to create incorrect pairs and selected 5 such pairs for each input. We did not use random mutations because they often produce invalid outputs that do not match the expected formats. Also, we generated multiple incorrect outputs for each input because a specification may accept multiple correct outputs for a given input. In this case, a single incorrect output may not be sufficient to determine whether the specification is sound with respect to the input.

We classify the results into four categories: sound and complete (**S&C**), complete only (**C**), sound only (**S**), and wrong (**W**) with respect to the input-output pairs. Following the definitions in § 3.1, we define these categories as follows:

- A generated specification is *complete* if it is consistent with all correct input-output pairs. For Expecto, we check whether the specification is satisfiable with each pair. For NL2Postcond, we check whether the assertion holds for each pair.
- A generated specification is *sound* if it is inconsistent with more than $X\%$ of the incorrect input-output pairs. For Expecto, we check whether the specification is unsatisfiable with each incorrect pair. For NL2Postcond, we check whether the assertion is false for each pair.
- A generated specification is *wrong* if it is neither complete nor sound.

When measuring completeness and soundness of Expecto, we exclude test cases where the SMT solver times out. Note that we define soundness in a probabilistic manner because of the inherent uncertainty in the generated incorrect input-output pairs. We set X to 50 in our primary experiments and will discuss the impact of different values of X in § 4.2.

For each AI programming task, we chose three correct input-output pairs and used them for validation during the specification generation process. If no input-output pairs are provided, we simply check that the specification is neither contradictory nor trivially true. We will discuss the impact of continuous specification validation in § 4.4.

4.2 RQ1. Effectiveness of Expecto in Formal Specification Generation

We evaluate the effectiveness of Expecto in extracting formal specifications from natural language descriptions on the HumanEval+ and APPS benchmarks. We compare the quality of specifications (i.e., soundness and completeness) generated by Expecto with that of specifications generated by Expecto$^-$ and the variants of NL2Postcond. Table 1 shows the experimental results.

Expecto outperforms Expecto$^-$ and NL2Postcond on both benchmarks. For HumanEval+, Expecto generates 39.2%, 19.7% and 24.1% more sound-and-complete (**S&C**) results compared to Expecto$^-$ and each variant of NL2Postcond, respectively. For APPS, which contains more complex problems, Expecto more significantly outperforms Expecto$^-$ and NL2Postcond. Expecto

Table 1. Performance of Expecto on the HumanEval+ and APPS benchmarks

Benchmark	Result	Expecto	Expecto⁻	Base	Simple
HumanEval+	**S&C**	103	74	86	83
	S	15	19	17	16
	C	42	12	14	53
	W	4	59	47	12
APPS	**S&C**	59	28	24	21
	S	11	19	61	26
	C	51	8	17	62
	W	6	72	25	18

You are given two squares, one with sides parallel to the coordinate axes, and another one with sides at 45 degrees to the coordinate axes. Find whether the two squares intersect. The interior of the square is considered to be part of the square, i.e. if one square is completely inside another, they intersect. If the two squares only share one common point, they are also considered to intersect.
Example
Input: `([(0, 0), (6, 0), (6, 6), (0, 6)], [(1, 3), (3, 5), (5, 3), (3, 1)])`
Output: “YES”

(a)

```
def Spec(sq1: list[tuple[int, int]], sq2: list[tuple[int, int]], res: string) {
  var intersect =
    ∃v :: (v in sq1 ∧ InPolygon(sq2, v)) ∨
    ∃v :: (v in sq2 ∧ InPolygon(sq1, v)) ∨
    EdgesIntersect(sq1, sq2)
  if intersect then (res == "YES") else (res == "NO")
}

def InPolygon(poly: list[tuple[int, int]], pt: tuple[int, int]) {
  var n = len(poly)
  ∃i :: (0 <= i < n ∧ OnSegment(poly[i], pt, poly[((i + 1) % n)])) ∨ (RayIntersections(poly, pt) % 2 == 1)
}

def OnSegment(p: tuple[int, int], q: tuple[int, int], r: tuple[int, int]) { ... }
def RayIntersections(poly: list[tuple[int, int]], pt: tuple[int, int]) -> (count: int) { ... }
def RayIntersectsSegment(p1: tuple[int, int], p2: tuple[int, int], q: tuple[int, int]) { ... }
def EdgesIntersect(p1: list[tuple[int, int]], p2: list[tuple[int, int]]) { ... }
def EdgePairIntersect(p1: tuple[int, int], p2: tuple[int, int], q1: tuple[int, int], q2: tuple[int, int]) { ... }
def Orientation(p: tuple[int, int], q: tuple[int, int], r: tuple[int, int]) { ... }
```

(b)

```
# Unsound specification generated by the Base and Simple strategy of NL2Postcond
assert result.upper() in {"YES", "NO"}
```

(c)

Fig. 7. A problem from APPS (a) and the simplified specification generated by Expecto (b) and NL2Postcond (c). The full definition of the generated specification is available in the appendix.

generates 110.7%, 145.8% and 181.0% more sound-and-complete results compared to Expecto⁻ and each variant of NL2Postcond, respectively.

Simple produces a high number of complete-only (**C**) specifications for both benchmarks. This is mainly because Simple often generates weak specifications that accept most input-output pairs. Fig. 7 illustrates an example from APPS that shows this point. This problem requires sophisticated geometric reasoning to determine whether two squares intersect. Notice that the prompting strategy of Simple instructs the LLM to generate simple postconditions. This in turn leads to the generation

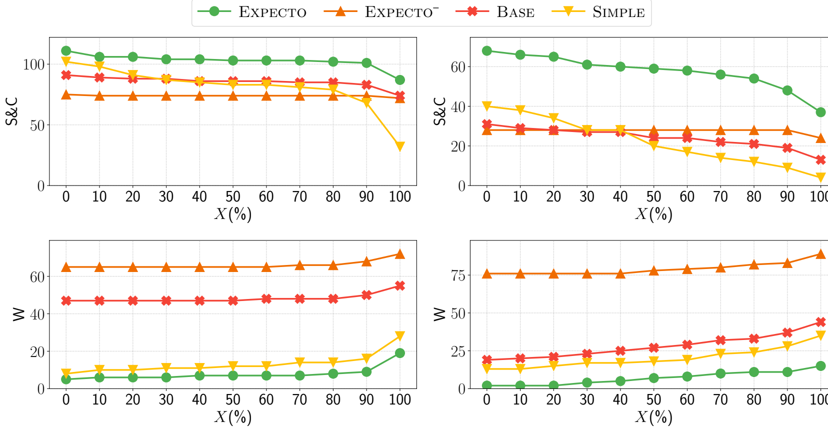

Fig. 8. The quality of specifications generated by each tool under different soundness threshold values X

of a trivial specification that always returns true. In contrast, Expecto successfully generates a specification that captures the intended behavior of the problem.

Expecto$^-$ produces more wrong (**W**) specifications than both Expecto and NL2Postcond. Expecto$^-$ requires the LLM to generate the full specification in one shot, which imposes a substantial reasoning burden on the model. Using a dedicated DSL further increases this burden, as LLMs often struggle with unfamiliar formal languages [4, 14]. As a result, Expecto$^-$ often produces ill-formed outputs, even after three repair attempts. In contrast, Expecto uses a top-down specification synthesis algorithm to decompose the task, reducing the LLM's reasoning burden and improving reliability. We further discuss the effectiveness of the top-down specification synthesis in § 4.3.

Expecto produced fewer sound-only (**S**) specifications than Expecto$^-$ and NL2Postcond. This is mainly because of our top-down specification synthesis approach combined with continuous validation. The top-down approach starts from a maximally permissive specification (i.e., trivially complete) and incrementally refines the specification to obtain more soundness. After each refinement, we verify that the refined specification is still consistent with all provided test cases. Consequently, it is more likely for Expecto to generate complete-only specifications than sound-only specifications.

Instead, Base often generates overly restrictive specifications. Recall that the prompting strategy of Base instructs the LLM to generate detailed postconditions. This often leads the LLM to include too many constraints in the specification, some of which may conflict with each other. This problem is especially pronounced in APPS, which contains complex programming tasks.

We also evaluate the performance of Expecto, Expecto$^-$ and NL2Postcond under different soundness threshold values X. Fig. 8 shows that Expecto outperforms NL2Postcond across all thresholds on both benchmarks. The Expecto$^-$ and Base strategies of NL2Postcond basically show a large number of **W** specifications even when the threshold is low (e.g., $X = 0$). This is because the specifications are often overly restrictive, so they also reject correct input-output pairs. They also contain errors in the specification itself (syntax errors, type errors, etc.). Simple shows relatively better performance than Base at low thresholds. However, as the threshold increases, the number of **S&C** specifications sharply decreases while the number of **W** specifications sharply increases. This indicates that the generated specifications are not strong enough to reject incorrect input-output pairs. In contrast, Expecto generates fewer **W** specifications than both variants of NL2Postcond across all thresholds. Even at high thresholds (e.g., $X = 100$), the number of sound-and-complete specifications generated by Expecto remains relatively stable. This indicates

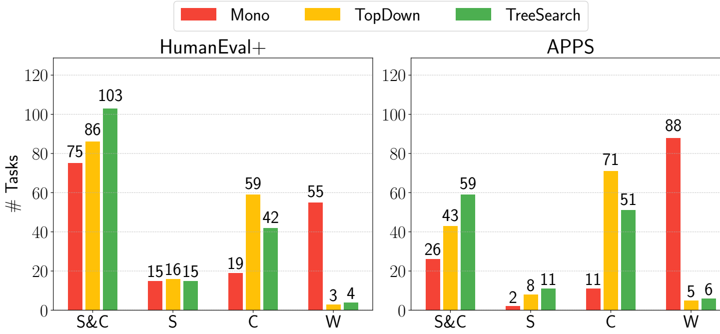

Fig. 9. Comparison of the effectiveness of top-down specification synthesis and tree search in Expecto.

that the generated specifications effectively prevent incorrect behaviors, while still capturing the intended functionality.

In summary, the results demonstrate that Expecto generates more accurate formal specifications from natural language descriptions than both Expecto$^-$ and NL2Postcond. This is mainly because Expecto can effectively decompose complex specification generation tasks into smaller sub-tasks using our top-down approach. This is especially beneficial for complex problems like APPS that require sophisticated reasoning.

4.3 RQ2. Effectiveness of the Top-Down Specification Synthesis with Tree Search

In this section, we evaluate the effectiveness of the two key components of Expecto: the top-down specification synthesis and tree search. To measure the impact of each component, we instantiate two ablated versions of Expecto: (1) Mono: a version that monolithically generates the entire specification in a single step using LLMs. We use few-shot prompting with three examples to guide the generation. (2) TopDown: A version that generates specifications using the top-down approach without tree search. Both versions still use the same validation process as Expecto using syntax and type checking, as well as consistency checking with test cases, allowing up to three repair attempts. We compare these ablated versions with the full version of Expecto that uses both top-down specification synthesis and tree search (TreeSearch).

Fig. 9 shows the effectiveness of our strategy. First, TopDown significantly reduces the number of wrong (**W**) specifications compared to Mono. The number of wrong specifications decreases by 94.5% in HumanEval+ and by 94.3% in APPS after applying top-down specification synthesis. This is mainly because the top-down approach breaks down the complex task of generating a full specification into smaller, manageable sub-tasks. This decomposition allows the LLM to focus on generating specific parts of the specification at each step, leading to more accurate results.

Additionally, the tree search with three samples improves the number of sound-and-complete (**S&C**) specifications compared to TopDown. The number of sound-and-complete specifications increases by 19.8% in HumanEval+ and by 37.2% in APPS after applying tree search. This improvement occurs because the tree search explores multiple refinement paths during specification generation. TopDown follows only a single path determined by the initial predictions of the LLM and stops refining the specification once it encounters an invalid path. This leads to generating a larger number of complete-only (**C**) specifications. In contrast, the tree search explores various possible refinements in parallel and continues refining specifications along different paths even if some paths become invalid. This capability enables it to generate more sound specifications, leading to an overall increase in sound-and-complete specifications.

Table 2 reports the average number of LLM calls for each variant. The results show that the TopDown improves robustness over Mono without substantially increasing the number of LLM

Table 2. Average number of LLM calls per benchmark problem for Mono, TopDown, and TreeSearch.

Benchmark	Mono	TopDown	TreeSearch
HumanEval+	2.59	4.26	19.07
APPS	3.47	5.66	27.29

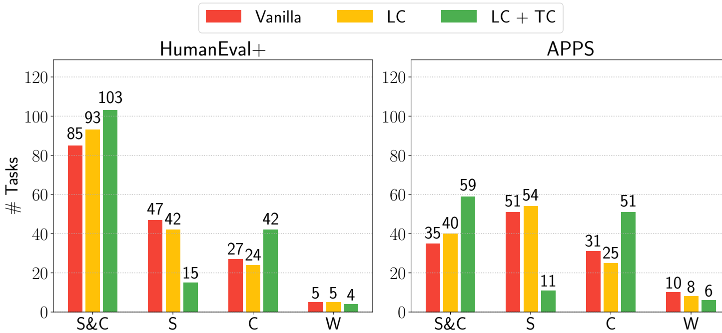

Fig. 10. Comparison of the impact of continuous specification validation.

calls. Although TopDown performs iterative refinement, it requires only about 1–2 more LLM calls than Mono for each benchmark. This is because Mono often introduces syntax, type, and logical errors that trigger subsequent repair attempts. In contrast, the refinement step of TopDown is simpler and less error-prone, leading to fewer repairs.

The increase in the number of LLM calls from TopDown to TreeSearch mainly comes from broader exploration of refinement paths. While TopDown follows a single path determined by the initial predictions of the LLM, TreeSearch generates multiple candidates for each refinement step. This allows Expecto to continue refining specifications along different paths even if some paths become invalid. As a result, TreeSearch not only uses more LLM calls for repairs at each refinement, but also performs more refinements than TopDown.

In summary, both the top-down specification synthesis and the tree search significantly contribute to the performance of Expecto. The top-down approach effectively decomposes complex specification generation tasks. This substantially reduces the likelihood of generating incorrect specifications. The tree search further improves the quality of generated specifications by exploring multiple refinement paths. This results in more sound-and-complete specifications.

4.4 RQ3. Impact of Continuous Specification Validation on Specification Generation

We measure the impact of continuous specification validation in Expecto. As described in § 3.4, Expecto can use user-provided test cases during specification generation. Given a set of test cases, Expecto continuously checks whether the partially generated specification is consistent with them. Even when no test cases are provided, Expecto still checks logical consistency using an SMT solver to ensure that the specification is neither contradictory nor trivially true. To measure the impact of continuous specification validation, we instantiate two ablated versions of Expecto: (1) Vanilla: a version that relies solely on syntax and type checking. (2) LC: a version that performs only logical consistency checking. We compare these ablated versions with the full version of Expecto, LC+TC, which performs all checks, including test-case-based validation. For each setting, we use TreeSearch for specification synthesis. Fig. 10 shows the results of the ablation study.

Notice that Vanilla already outperforms NL2Postcond. It shows a similar number of **S&C** specifications as NL2Postcond for HumanEval+ (85 vs. 86) and a significantly higher number

Let's denote $d(n)$ as the number of divisors of a positive integer n. You are given three integers a, b, and c. Your task is to calculate $\sum_{i=1}^{a} \sum_{j=1}^{b} \sum_{k=1}^{c} d(i \cdot j \cdot k)$ and find the sum modulo 1073741824 (2^{30}). The first line contains three space-separated integers a, b, and c ($1 \leq a, b, c \leq 100$). Print a single integer – the required sum modulo 2^{30}

Example

Input: (2, 2, 2)

Output: 20

(a)

```
def Spec(a: int, b: int, c: int, out: int) {
  (1 <= a <= 100) ∧ (1 <= b <= 100) ∧ (1 <= c <= 100) ∧ (out == (Sum(a, b, c) % 1073741824))
}

def Sum(a: int, b: int, c: int) {
  fold_i(lambda (i_idx: int, x: int, _) =
    (x + fold_i(lambda j_idx: int, y: int, _) =
      (y + fold_i(lambda (k_idx: int, z: int, _) =
        (z + DivisorCount(i_idx * j_idx * k_idx)), 0, [1..c])), 0, [1..b]), 0, [1..a]);
}

def DivisorCount(n: int) { fold(lambda (acc: int, d: int) = (acc +(if (n % d == 0) then 1 else 0)), 0, [1..n]); }
```

(b)

Fig. 11. An example illustrating the impact of logical consistency checking.

of **S&C** specifications for APPS (35 vs. 24). Also, Vanilla generates fewer **W** specifications than NL2Postcond for both benchmarks. This demonstrates the effectiveness of our approach even with minimal validation.

LC highlights the impact of logical consistency checking. LC produces 9.4% more **S&C** specifications than Vanilla for HumanEval+ and 14.3% more for APPS. Fig. 11 shows a contradictory specification generated by Vanilla. The specification is defined by using `fold_i`, whose first parameter is the 0-based index of the list. Thus, initially, `i_idx = j_idx = k_idx = 0` and `DivisorCount(i_idx*j_idx*k_idx)` evaluates to `DivisorCount(0)`. However, `DivisorCount(n)` is defined using the range `[1..n]`. Thus, `DivisorCount(0)` is undefined because the range `[1..0]` is contradictory. LC can prune such cases through logical consistency checking.

While test cases are not strictly required by Expecto, their availability helps improve robustness by preventing trivial errors. In our experiments, LC+TC generates 10.8% more **S&C** specifications than LC for HumanEval+ and 47.5% more for APPS. From the user's perspective, this provides a practical mechanism to enhance the robustness of specification generation by providing additional test cases. For critical systems that demand high assurance, users can further strengthen the robustness of generated specifications by providing a larger set of test cases.

In summary, Expecto shows reasonably high performance even without test cases. Moreover, a few test cases significantly improve the quality of generated specifications. Note that we provide only three test cases for each problem, which is a reasonable amount for users to write manually. This indicates that Expecto can be effectively used in practice with minimal user effort.

4.5 RQ4. Effectiveness of Expecto for Bug Detection in Real-World Software

We evaluate the effectiveness of Expecto for bug detection on the Defects4J benchmark, compared to NL2Postcond. Among the various types of bugs in Defects4J, we selected assertion violation bugs as they represent inconsistencies between the intended specification and actual implementation. We only consider bugs that are reproducible in Java 8 or later because NL2Postcond depends on those Java versions. In total, we collected 501 methods involved in 336 bugs.

Table 3. Performance on the Defects4J benchmark

Benchmark	Result	Expecto	Base	Simple
Defects4J	**S&C**	42	6	14
	S	24	53	95
	C	401	224	220
	W	34	218	172

Following the prior work [9], we measure the quality of specifications of the collected methods using the provided unit tests. To generate a specification for a method, we provided its Javadoc comment, source code, and method signatures written in our DSL as input prompts to Expecto. For NL2Postcond, we used the same prompt as in their artifact, which includes the Javadoc comment, source code, and relevant method signatures.

Unlike NL2Postcond, which generates assertions in Java, the current implementation of Expecto does not support using Java objects and methods in specifications. Therefore, we execute the unit tests provided in Defects4J and serialize the input and output state of the target method. Then we evaluate the correctness of the generated specifications based on the serialized input-output pairs.

We evaluate the performance using a similar criterion as before, but with a focus on bug detection capability. Unlike APPS and HumanEval+, Defects4J explicitly provides both bug-triggering and correct unit tests. Thus, we consider a specification *sound* if it detects the bug using at least one of the bug-triggering unit tests. We use the same definitions for the other categories: a specification is considered *complete* if it accepts all passing unit tests, and *wrong* otherwise.

As shown in Table 3, Expecto generates 42 **S&C** specifications, significantly outperforming Base (6) and Simple (14). Both Base and Simple generate a large number of **W** specifications (218 and 172, respectively). They often generate incorrect assertions because the target method implements complex logic involving object states and side effects. Moreover, many of the specifications generated by NL2Postcond fail to compile due to using undefined variables, attributes, or methods. In contrast, Expecto generates only 34 **W** specifications as the validation process prevents such simple compile errors and the top-down synthesis effectively handles complex logic. For **S** specifications, both variants of NL2Postcond generate a larger number of specifications (53 and 95) than Expecto (24). This is because most specifications generated by NL2Postcond are logically incorrect, causing assertion errors in most correct cases.

Fig. 12 shows an example of a bug-detectable specification generated by Expecto. The bug occurs because of integer overflow during the multiplication process when a large value of `n` is given as input. The precondition and exceptions are written in the Javadoc comment (green box, yellow box). Our specification accurately captures the precondition with the definition of the binomial coefficient function within the range that satisfies the precondition (blue box). In contrast, the assertion generated by Base naively calculates the binomial coefficient without considering integer overflow during multiplication and checks the precondition insufficiently (it misses the constraint that n is less than or equal to 66). Simple's assertion just checks that `returnValue` is greater than or equal to 1, which is insufficient for bug detection.

Overall, Expecto generated 3.0–7.0× more **S&C** specifications than NL2Postcond. This demonstrates that Expecto can generate specifications not only for competitive programming problems but also for real-world software with complex logic. We expect that Expecto's performance can be further improved by introducing more effective preprocessing methods, such as smart serialization through static analysis (e.g., including only fields that are read or written by the target method).

```
/**
 * Returns an exact representation of the Binomial Coefficient, "n choose k",
 * the number of k-element subsets that can be selected from an n-element set.
 * * Preconditions:
 * - 0 <= k <= n  (otherwise IllegalArgumentException is thrown)
 * - The result is small enough to fit into a long. The largest value of n for which all coefficients
 * are < Long.MAX_VALUE is 66. If the computed value exceeds Long.MAX_VALUE an ArithmeticException is thrown.
 * ...
 */
public static long binomialCoefficient(final int n, final int k) {
  if (n < k) throw new IllegalArgumentException("must have n >= k for binomial coefficient (n,k)");
  if (n < 0) throw new IllegalArgumentException("must have n >= 0 for binomial coefficient (n,k)");
  if ((n == k) || (k == 0)) return 1;
  if ((k == 1) || (k == n - 1)) return n;
  long result = Math.round(binomialCoefficientDouble(n, k));
  if (result == Long.MAX_VALUE) throw new ArithmeticException("result too large to represent in a long integer");
  return result;
}
```

(a)

```
def Spec(param: record[k: int, n: int], entry_self: option[nonetype],
     exit_self: option[nonetype], ret: option[int]) -> bool {
  var n, k = param.n, param.k;
  var valid_input: bool = n >= 0 ∧ 0 <= k ∧ k <= n ∧ n <= 66;
  valid_input ==> (is_some(ret) ∧ unwrap(ret) == BinomialCoefficient(n, k) ∧ unwrap(ret) >= 0) ∧
  ¬valid_input ==> is_none(ret)
}

def BinomialCoefficient(n: int, k: int) -> int { Factorial(n) / (Factorial(k) * Factorial(n - k)) }

def Factorial(x: int) -> int { if x == 0 then 1 else product([1 .. x]) }
```

(b)

```
// Specification generated by the Base strategy of NL2Postcond
assert (0 <= k && k <= n) && ( (k == 0 || k == n) ? (returnValue == 1) : (k == 1 || k == n - 1) ?
      (returnValue == n) : returnValue == factorial(n) / (factorial(k) * factorial(n - k)) );

// Specification generated by the Simple strategy of NL2Postcond
assert returnValue >= 1;
```

(c)

Fig. 12. A buggy method from Defects4J (Math-92) (a), a bug-detectable specification generated by Expecto (b), and an insufficient specification generated by NL2Postcond (c).

5 Related Work

Autoformalization. There has been growing interest in converting natural language descriptions into formal specifications, a process often referred to as *autoformalization*. Initially developed in the context of mathematics [11, 29, 34], researchers have since explored this technique across various domains including mobile AI agents [16, 27], temporal logic [7], and code generation [9]. However, most of the existing techniques heavily rely on the reasoning capabilities of LLMs using prompt engineering or retrieval-augmented generation (RAG). In contrast, Expecto introduces a systematic approach that gradually refines specifications through a top-down synthesis process. We demonstrated that this structured approach significantly improves the accuracy and robustness of the generated specifications compared to direct generation methods.

Our work is complementary to prior research on automatic proof generation for code. These techniques typically aim to generate formal specifications and proofs for existing code snippets [21, 22, 28, 30]. Recently, Sun et al. [26] proposed Clover, which generates code, specifications, and proofs simultaneously from natural language. Clover adopts a monolithic specification generation strategy, whereas Expecto contributes a complementary top-down, modular approach to specification synthesis with continuous specification validation. SpecGen [21] addresses a different problem: generating specifications for existing code, where the reference code can be used for validation. In contrast, Expecto focuses on generating accurate formal specifications from natural language intent. We believe that our approach can be combined with these proof generation techniques to establish a complete pipeline from natural language intent to verified code.

Neuro-Symbolic Program Synthesis. Recent years have seen significant advancements in neuro-symbolic program synthesis that combine LLMs with symbolic methods to generate programs. Ye et al. [32] uses an RNN model to predict AST nodes from natural language descriptions and uses program analysis to prune unpromising partial programs. Li et al. [19] use LLM outputs to learn probabilistic context-free grammars (PCFGs) and guide the synthesis process. Cai et al. [3] propose a neuro-symbolic program refinement framework that transforms high-level formal specifications into executable code. They combine LLMs with automated theorem provers, where the model generates program refinement rules and the prover verifies their correctness. These neuro-symbolic program synthesis techniques are effective for generating programs in domain-specific languages [2, 6]. Expecto is also based on a neuro-symbolic synthesis framework that combines an LLM with an automated theorem prover. However, Expecto focuses on synthesizing formal specifications from natural language intent while their work focuses on synthesizing executable code from formal specifications.

Structured Reasoning with LLMs. Expecto is inspired by recent advances in structured reasoning techniques for LLMs. Least-to-Most prompting decomposes complex problems into simpler subproblems, solving them sequentially [33]. Tree of Thoughts (ToT) explores multiple reasoning paths in a tree structure, allowing language models to deliberate and backtrack when needed [31]. These methods improve model performance by decomposing complex tasks into manageable units and systematically exploring the solution space. When combined with search algorithms (e.g., MCTS), they effectively solve challenging problems in domains such as mathematics and competitive programming. We adapt these structured reasoning techniques to extract formal specifications from natural language intent. To incorporate these ideas into the specification synthesis problem, we introduce a new concept: *informal definitions*. Each informal definition provides a natural language description and a typed signature, but lacks a formal body. This provides a structured way to define auxiliary definitions using natural language, which enables both top-down modular synthesis and continuous validation of partial specifications.

6 Conclusion

We have presented Expecto, a neuro-symbolic approach to extracting formal specifications from natural language descriptions. The key idea is to perform a top-down and modular specification extraction at the function level and continuously validate the correctness of intermediate specifications. This approach breaks down the complex task of specification extraction into smaller and more manageable sub-tasks. This modular approach significantly reduces the reasoning burden on LLMs, enabling them to generate more accurate and reliable specifications. We evaluated Expecto on a wide range of benchmarks and demonstrated its effectiveness in generating formal specifications from natural language descriptions. The results show that Expecto consistently outperforms existing methods, highlighting the benefits of our approach.

Data-Availability Statement

Expecto is available on GitHub (https://github.com/prosyslab/expecto-artifact) and Zenodo (https://doi.org/10.5281/zenodo.19450217) [15]. These resources include detailed instructions and scripts needed to reproduce the experiments described in this paper. They also contain the preprocessed datasets used for those experiments.

Acknowledgements

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00338454, RS-2021-NR060080). This work was partly supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02305705, Development of AI agents to improve the accessibility of mobile app services for the elderly).

References

[1] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. https://arxiv.org/abs/2108.07732

[2] Celeste Barnaby, Qiaochu Chen, Chenglong Wang, and Isil Dillig. 2024. PhotoScout: Synthesis-Powered Multi-Modal Image Search. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI)*.

[3] Yufan Cai, Zhe Hou, David Sanán, Xiaokun Luan, Yun Lin, Jun Sun, and Jin Song Dong. 2025. Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. *Proceedings of the ACM on Programming Languages (PACMPL)* 9, POPL (2025), 2057–2089.

[4] Jialun Cao, Yaojie Lu, Meiziniu Li, Haoyang Ma, Haokun Li, Mengda He, Cheng Wen, Le Sun, Hongyu Zhang, Shengchao Qin, Shing-Chi Cheung, and Cong Tian. 2025. From Informal to Formal – Incorporating and Evaluating LLMs on Natural Language Requirements to Verifiable Formal Proofs. In *Association for Computational Linguistics (ACL)*.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. https://arxiv.org/abs/2107.03374

[6] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web Question Answering with Neurosymbolic Program Synthesis. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.

[7] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. 2023. Nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models. In *International Conference on Computer-Aided Verification (CAV)*.

[8] Delimarsky Den and Lam John. 2025. Spec-Kit. GitHub. https://github.com/github/spec-kit

[9] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

[10] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.

[11] Ruikang Hu, Shaoyu Lin, Yeliang Xiu, and Yongmei Liu. 2025. LTRAG: Enhancing Autoformalization and Self-refinement for Logical Reasoning with Thought-Guided RAG. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*. Association for Computational Linguistics, 2483–2493. https://aclanthology.org/2025.findings-acl.126/

[12] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

[13] Kiro. 2025. Kiro. https://kiro.dev/

[14] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs Reason About Program Semantics? A Comprehensive Evaluation of LLMs on Formal Specification Inference. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, Vol. 1. 21991–22014.

[15] Dongjae Lee and Kihong Heo. 2026. Expecto: Extracting Formal Specifications from Natural Language Description for Trustworthy Oracles. doi:10.5281/zenodo.19450217

[16] Jungjae Lee, Dongjae Lee, Chihun Choi, Youngmin Im, Jaeyoung Wi, Kihong Heo, Sangeun Oh, Sunjae Lee, and Insik Shin. 2025. Safeguarding Mobile GUI Agent via Logic-based Action Verification. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*.
[17] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*.
[18] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[19] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *International Conference on Computer-Aided Verification (CAV)*.
[20] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.
[21] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *International Conference on Software Engineering (ICSE)*.
[22] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Zhou Yuanyuan. 2025. Laurel: Unblocking Automated Verification with Large Language Models. *Proceedings of the ACM on Programming Languages (PACMPL)* 9, OOPSLA (2025), 1519–1545.
[23] OpenAI. 2025. Gpt-4.1-Mini. https://openai.com
[24] OpenAI. 2025. Text-Embedding-3-Large. https://openai.com
[25] Stack Overflow. 2025. 2025 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2025/
[26] Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *First International Symposium on AI Verification (SAIV)*.
[27] Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2026. AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. In *International Conference on Software Engineering (ICSE)*.
[28] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *International Conference on Computer-Aided Verification (CAV)*.
[29] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[30] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages (PACMPL)* 9, OOPSLA2 (2025), 396:3454–396:3482.
[31] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[32] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Empirical Methods in Natural Language Processing (EMNLP)*.
[33] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *International Conference on Learning Representations (ICLR)*.
[34] Jin Peng Zhou, Charles Staats, Wenda Li, Christian Szegedy, Kilian Q. Weinberger, and Yuhuai Wu. 2024. Don't Trust: Verify - Grounding LLM Quantitative Reasoning with Autoformalization. In *International Conference on Learning Representations (ICLR)*.

Received 2025-11-14; accepted 2026-04-03