

Spec*Bench

Benchmarking Specification Generation with Automated Verification

Baris Bayazit¹ Xujie Si¹

¹Department of Computer Science
University of Toronto

Summer, 2025

Table of Contents

- 1 What is a Specification?
- 2 The Problem
- 3 Previous Work
- 4 Method
- 5 Examples
- 6 Results
- 7 Conclusion



What is a Specification?

- **Specification.** A predicate that formalizes a natural-language description into precise mathematical conditions.
- **Why?** Bridge the gap between informal requirements and formal verification.
- **Key Properties:**
 - *Precision:* Unambiguous mathematical statements
 - *Completeness:* Captures all intended behavior
 - *Consistency:* No contradictory requirements
- **Examples:**
 - `factorial(n)` returns $n!$ for $n \geq 0$
 - `sort(list)` returns a permutation in non-decreasing order
 - `gcd(a,b)` returns the greatest common divisor of a and b
 - RegEx pattern
`/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/`
validates emails.



Problem: Benchmarking LLM Specification Generation

- **The Challenge:** How do we evaluate LLMs' ability to automatically generate specifications from natural language?
- **The Gap:** No sound, LLM-independent, automated benchmark exists
- Specification checking lacks guarantees (soundness, completeness, independence).
- **Spec*Bench solves this!**

An Example of Traditional Verification: Testing

Write me a python function that computes the nth fibonacci number

Here's a clean Python function that computes the **n-th Fibonacci number** using an efficient iterative approach (instead of recursion, which is slower and can hit recursion limits):

python

Copy Edit

```
def fibonacci(n: int) -> int:
    """
    Compute the nth Fibonacci number (0-indexed).
    F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) for n >= 2
    """
    if n < 0:
        raise ValueError("n must be a non-negative integer")
    if n == 0:
        return 0
    elif n == 1:
        return 1

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```



UNIVERSITY OF TORONTO

An Example of Traditional Verification: Testing

```
def fibonacci(n: int) -> int:
    """
    Compute the nth Fibonacci number (0-indexed).
    F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) for n >= 2
    """
    if n < 0:
        raise ValueError("n must be a non-negative integer")
    if n == 0:
        return 0
    elif n == 1:
        return 1
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Python Code

```
class TestFibonacci(unittest.TestCase):
    def test_fibonacci_base_cases(self):
        self.assertEqual(fibonacci(0), 0)
        self.assertEqual(fibonacci(1), 1)

    def test_fibonacci_small_numbers(self):
        self.assertEqual(fibonacci(2), 1)
        self.assertEqual(fibonacci(3), 2)
        self.assertEqual(fibonacci(4), 3)
        self.assertEqual(fibonacci(5), 5)
        self.assertEqual(fibonacci(6), 8)

    def test_fibonacci_large_number(self):
        self.assertEqual(fibonacci(10), 55)
        self.assertEqual(fibonacci(20), 6765)

    def test_fibonacci_negative(self):
        with self.assertRaises(ValueError):
            fibonacci(-1)
```

Unit Testing



An Example of Traditional Verification: The Issue

- Tests pass \implies generated code is correct.
- What if the tests are insufficient?
 - Ensure 100% coverage?
 - The behaviour might be different.
 - Passed by 'chance'?

An Example of Traditional Verification: The Issue

```
def fibonacci(n: int) -> int:
    if n == -1:
        raise ValueError()
    return n * (
        -347*n**7 +
        16877*n**6 -
        307811*n**5 +
        2790095*n**4 -
        13724018*n**3 +
        36920828*n**2 -
        50079144*n +
        28496160
    ) // 4112640
```

Python Code (polynomially fit)

```
class TestFibonacci(unittest.TestCase):
    def test_fibonacci_base_cases(self):
        self.assertEqual(fibonacci(0), 0)
        self.assertEqual(fibonacci(1), 1)

    def test_fibonacci_small_numbers(self):
        self.assertEqual(fibonacci(2), 1)
        self.assertEqual(fibonacci(3), 2)
        self.assertEqual(fibonacci(4), 3)
        self.assertEqual(fibonacci(5), 5)
        self.assertEqual(fibonacci(6), 8)

    def test_fibonacci_large_number(self):
        self.assertEqual(fibonacci(10), 55)
        self.assertEqual(fibonacci(20), 6765)

    def test_fibonacci_negative(self):
        with self.assertRaises(ValueError):
            fibonacci(-1)
```

Unit Testing

The Issue

This passes the unit tests.

A Look at an Attempted Solution: PBT

```
def fibonacci_correct(n: int) -> int: ...

def fibonacci_generated(n: int) -> int:
    if n == -1:
        raise ValueError()
    return n * (
        -347*n**7 +
        16877*n**6 -
        307811*n**5 +
        2790095*n**4 -
        13724018*n**3 +
        36920828*n**2 -
        50079144*n +
        28496160
    ) // 4112640

@given(st.integers(min_value=0, max_value=30))
def test_fib_matches_spec(n):
    assert fibonacci_correct(n) == fibonacci_generated(n)
```

Encode Properties



```
assert fibonacci_correct(n) == fibonacci_generated(n)
~~~~~
AssertionError
Falsifying example: test_fib_matches_spec(
    n=7,
)
```

Property-Based Testing (PBT)



A Look at an Attempted Solution: PBT

- Previous Work: Verina [5].
- PBT is not sound.
 - Test passes \nRightarrow specifications are equal.
 - You can't possibly check for all answers!

PBT checks only $D' \subset D$:  $S_{\text{gen}} = S_{\text{ref}}$ on D'
Equivalence requires:  $\forall x \in D, S_{\text{gen}}(x) = S_{\text{ref}}(x)$

Samples are *evidence*, not a *guarantee*.



A Look at an Attempted Solution: Proving Equivalence

```
Fixpoint fib (n : nat) : nat :=  
  match n with  
  | 0 => 0  
  | 1 => 1  
  | S (S k as m) => fib m + fib k  
  end.  
  
Fixpoint fib_generated (n : nat) : nat :=  
  (* ... *)  
  
Theorem generated_is_equivalent: forall (n: nat),  
  fib n = fib_generated n.  
Proof.  
  (* Proof of equivalence *)  
Admitted.
```

UNIVERSITY OF TORONTO

A Look at an Attempted Solution: Proving Equivalence

- Previous Work: Clever [3].
- How do we generate the equivalence proof in the ITP?
 - Solution: Ask it to the LLMs!
- What if the LLM is unable to write the proof?
 - Does it mean the specification is incorrect?
 - Or, was the LLM simply unable to write the proof?
 - If you can't write the proof for something, does it mean it's incorrect?

```
Theorem generated_is_equivalent: forall (n: nat),  
  fib n = fib_generated n.  
Proof.  
  (*  
    Now that you have generated your specification,  
    here's what the correct answer was.  
  
    Prove that your answer is correct by proving  
    this equivalence theorem.  
  *)  
Admitted.
```

Proof Generation

LLMs were only able to write %0.62 of the proofs correctly [3].

A Look at an Attempted Solution: LLM as a Judge

- Key Idea: Ask the same LLM (or a different one) whether the answer is correct. If not, ask for a counterexample.

Was your answer correct? Evaluate. Be honest.

Yes — my answer was correct. Let me carefully evaluate it, step by step:

- The LLM said correct, so it must be true...



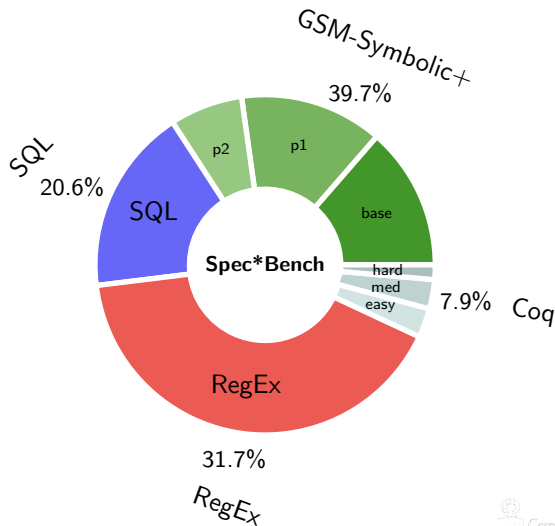
Spec*Bench introduces **the first benchmark and protocol for spec generation with verifiable and automated checking.**

Method	Sound?	LLM-indep. Verification?	Successful in Practice?
LLM as a Judge	×	×	?
PBT	×	✓	✓
LLM + ITP	✓	×	×
Spec*Bench	✓	✓	✓

Table: Trade-offs among existing specification generation benchmarks.



Composition of Questions



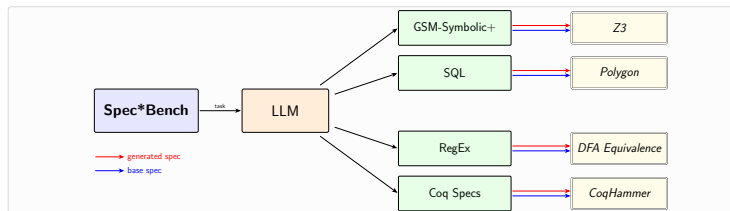


Figure: Overview of Spec*Bench methodology. Each natural language task is given to an LLM, which produces a candidate specification. The generated specification is compared against the base specification using domain-specific verifiers.



- **Input.** A natural-language task picked from one of the domains and a hidden base specification S_{ref} .
- **Generation.** The LLM proposes a candidate specification S_{gen} .
- **Verification.** Check $S_{\text{gen}} \equiv S_{\text{ref}}$ with the respective solver.
- **Verdict.** *Verified*, *Refuted* (counterexample), or *Unknown* (timeout/failure).



Example: GSM-Symbolic+

Grade School Math (GSM) is a benchmark that evaluates AI models on simple, grade-school math questions.

Example

A fog bank rolls in from the ocean to cover a city. It takes 10 minutes to cover every 100 miles of the city. If the city is 1000 miles across from the oceanfront to the opposite inland edge, how many minutes will it take for the fog bank to cover the whole city?

Example: GSM-Symbolic+

Previous research [2] has found that generating variants of questions by replacing numbers has resulted in a drop of accuracy.

- '10 minutes to cover' \Rightarrow Generate questions: '5 minutes to cover', '2 minutes to cover', ...
- '100 miles of the city' \Rightarrow Generate questions: '20 miles of the city', '30 miles of the city', ...
- And so on...



Example: GSM-Symbolic+

GSM-Symbolic+ enhances this further by templating all variables, and sending the solution to an SMT solver.

- '10 minutes to cover' \Rightarrow : '{time} minutes to cover'
- '100 miles of the city' \Rightarrow : '{area} miles of the city'

A fog bank rolls in from the ocean to cover a city. It takes {time} minutes to cover every {area} miles of the city. If the city is {total} miles across from the oceanfront to the opposite inland edge, how many minutes will it take for the fog bank to cover the whole city?

$$\text{Minutes} = \frac{\text{total}}{\text{area}} \times \text{time}.$$

(You multiply the number of *area*-mile segments in the city, *total/area*, by the minutes per segment, *time*.)



Example: GSM-Symbolic+

The LLM's answer and the Base answer is provided to the SMT solver, which checks whether these two expressions are equal.

$$\text{Minutes} = \frac{\text{total}}{\text{area}} \times \text{time.}$$

LLM Answer

$$\frac{\text{total} \times \text{time}}{\text{area}}$$

Base Answer (hidden from the LLM)



Example: SQL

Polygon is a Symbolic SQL Reasoning engine that is able to prove properties about a subset of bounded-SQL in a sound and complete way, using an SMT solver [6].

Example

- Query: Are these two SQLs equivalent under all possible tables for this schema?
 - `SELECT * FROM my_table;`
 - `SELECT DISTINCT * FROM my_table;`
- Polygon Response: Not equal. Consider this my_table: [3, 3, 4].



Example: SQL

Example question from a LeetCode SQL Problem ¹, which Spec*Bench SQL is partly based on:

Domain: SQL

Solver: Polygon (equivalence/refutation)

Task. Largest number that appears exactly once.

Base (S_{ref}). `SELECT MAX(num) AS num
FROM (SELECT num FROM MyNumbers GROUP BY num HAVING
COUNT(*)=1) t;`

LLM (S_{gen}). `SELECT MAX(num)
FROM MyNumbers GROUP BY num HAVING COUNT(*)=1;`

Counterexample. num: {1, 2, 2, 5}

Outputs. S_{ref} : 5, S_{gen} : 1, 5

Verdict. *Refuted.*

¹leetcode.com/problems/biggest-single-number



Example: RegEx

Previous work (pre-LLM era) has been done on benchmarking regex synthesis of ML models [4]. Our benchmark extends the previous work, converting the RegEx trees to Python.

Example

- a list of two string. the first string consist of 2+ 'n' and final string consist of 1-4 lowercase.
- Hidden answer:
`concat(repeatatleast(<n>,2),repeatrange(<low>,1,4))`
- Model Generates: `r"(?:n2,)(?:low1,4)"`, which we translate back to the base answer.
- Then, against the base RegEx answer, we simply check for DFA equivalence using the previous work [4].

Example: Coq Specifications

Coq (now renamed to Rocq) is an Interactive Theorem Prover (ITP) that allows users to write formal specifications and machine-checkable proofs.

```
1 Definition two_times (n: nat) := n + n.  
2  
3 Theorem one_times_two_is_two: two_times 1 = 2.  
4 Proof.  
5   unfold two_times.  
6   reflexivity.  
7 Qed.  
8
```

Proof

Main 1 Shelved 0 Given up 0

Goal 1

(1 / 1)

$1 + 1 = 2$

Messages

This is a simple example - Rocq definitions can get complex really quick (i.e., you can encode a LeetCode problem as a definition)!

Example: Rocq Specifications

```
(*  
signature: "Definition generated_spec (impl: bool -> bool -> bool) (b1 b2: bool) : Prop :="   
description: Implementation impl must return the XOR of the two boolean values.  
examples:  
| - input: true, false  
|   output: true  
| - input: true, true  
|   output: false  
*)
```

Natural Language Input

```
Definition problem_spec (impl: bool -> bool -> bool) (b1 b2: bool) : Prop :=  
exists r,  
| impl b1 b2 = r /\  
| r = if b1 then negb b2 else b2.
```

Hidden Rocq Answer



Example: Rocq Specifications

Write Coq specification pls

signature: "Definition generated_spec (impl: bool -> bool -> bool)
(b1 b2: bool) : Prop :="

description: Implementation impl must return the XOR of the two
boolean values.

examples:

- input: true, false
output: true
- input: true, true
output: false



Here's a Coq specification that matches your given signature, description, and examples:

coq

Copy Edit

```
Definition generated_spec (impl: bool -> bool -> bool) (b1 b2: bool) : Prop :=  
  impl b1 b2 = xorb b1 b2.
```

LLM Generates



Computer Science

UNIVERSITY OF TORONTO

Example: Rocq Specifications

- Problem: How do we prove equivalence between the generated specification? There are multiple ways to write the same code.

```
Theorem spec_equivalence :  
  forall impl, (forall b1 b2, problem_spec impl b1 b2) <=>  
    | | | | | | | (forall b1 b2, generated_spec impl b1 b2).  
Proof.  
  | (* How do we prove this? *)  
Admitted.
```

Equivalence Theorem

Problem

How do we even write the proof? As previously explained, LLMs are unable to write this equivalence proof.

Example: Rocq Specifications

- CoqHammer to the rescue!
- CoqHammer delegates Rocq proofs to SMT solvers, later constructing the proof automatically in Rocq [1].

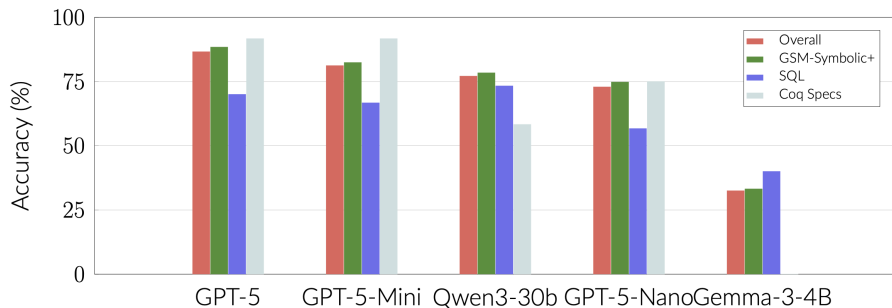
```
Theorem spec_equivalence :  
  forall impl, (forall b1 b2, problem_spec impl b1 b2) <=>  
  | | | | | (forall b1 b2, generated_spec impl b1 b2).  
Proof.  
| | hammer.  
Qed.
```

Automatic Proof by `hammer`.

Limitations

CoqHammer does not work for very complex specifications, and we have to be mindful of the specification difficulty, and provide a reasonable timeout.

Results



Spec*Bench performance across models and domains for a subset of the benchmark, with no refinement rounds and a 4 second timeout. RegEx was omitted for this experiment.



Analysis

Description | Accepted X | Editorial | Solutions | Submissions

← All Submissions

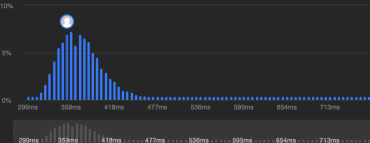
Accepted 15 / 15 testcases passed
Itc24816 submitted at Aug 17, 2025 12:51

Editorial Solution

Runtime

357 ms | Beats 72.90%

Analyze Complexity



Code | MySQL

```
SELECT email AS Email
FROM Person
GROUP BY email
HAVING COUNT(*) > 1;
```

More challenges

1350. Students With Invalid Departments

2837. Total Traveled Distance

3126. Server Utilization Time

Write your notes here

Code

MySQL Auto

```
1 SELECT email AS Email
2 FROM Person
3 GROUP BY email
4 HAVING COUNT(*) > 1;
```

Saved Ln 4, Col 21

Testcase Test Result

5	null
4	null

Output

Email

null

Expected

Email

Contribute a testcase

A Leetcode counterexample



Computer Science
UNIVERSITY OF TORONTO

- **SQL evaluation gap.** Most SQL failures are *still accepted* by LeetCode. Counterexamples are valid and fail LeetCode's own "Run Test." ⇒ **Tests are insufficient to evaluate specifications.**
- **Numbers vs. variables.** Accuracy drops from numeric variations are known [2], but GSM-Symbolic+ reveals a *further decline* when templating with variables.
- **Scaling for Coq.** Smaller models underperform at writing Coq specifications.



Conclusion

- **The Problem:** Existing specification generation benchmarks lack soundness, LLM-independence, or practical success.
- **Our Solution:** Spec*Bench provides the **first sound, automated, and LLM-independent benchmark** for specification generation across diverse domains.
- **Impact:** Spec*Bench establishes a new standard for evaluating AI-generated specifications, advancing the field toward **autoformalization**.



Spec*Bench

Benchmarking Specification Generation
with Automated Verification



github.com/bbayazit16/specbench

Baris Bayazit

baris@cs.toronto.edu

Xujie Si

six@cs.toronto.edu

Thank You!



Computer Science
UNIVERSITY OF TORONTO