

Exercise 3: Specification-Guided Test Improvement

-Aryan Ghosh

[GitHub Repository](#)

Table of Contents

PART 1: GENERATE, EVALUATE, AND REFINE SPECIFICATIONS	2
1. HUMAN EVAL/160	2
2. HUMAN EVAL/139	6
PART 2: SPECIFICATIONS TO GUIDE TEST IMPROVEMENT	9
1. HUMAN EVAL/160	9
2. HUMAN EVAL/139	11

Part 1: Generate, Evaluate, and Refine Specifications

LLM Used: GPT5

1. HumanEval/160

- **Function/Method Signature**

```
def do_algebra(operator: list[str], operand: list[int]) -> int
```

- **Natural Language Problem Description**

Given two lists operator and operand. The first list has basic algebra operations, and the second list is a list of integers. Use the two given lists to build the algebraic expression and return the evaluation of this expression.

Example:

```
operator=['+', '*', '-']
array=[2, 3, 4, 5]
result = 2 + 3 * 4 - 5 => result = 9
```

The length of the operator list is equal to the length of the operand list minus one. Operand is a list of non-negative integers. Operator list has at least one operator, and the operand list has at least two operands.

- **LLM Prompt**

Write formal specifications for the following Python function, `do_algebra`. Your specifications should be written as Python `assert` statements, expressing the relationship between input and output using pure arithmetic and boolean logic.

- Do not call or reference `do_algebra` inside the assertion (no self-reference).
- Do not use methods that cause side effects, such as modifying data structures, performing I/O (print, read, write), randomness, or timing functions.
- Use `res` to denote the expected output value.

Problem description:

Given two lists, operator and operand. The first list has basic algebra operations, and the second list is a list of integers. Use the two lists to build the algebraic expression and return the evaluation of this expression.

Example:

```
operator = ['+', '*', '-']
operand = [2, 3, 4, 5]
result = 2 + 3 * 4 - 5 => result = 9
```

The length of the operator list is equal to the length of the operand list minus one. Operator list has at least one operator, and the operand list has at least two operands.

Method signature:

```
def do_algebra(operator: list[str], operand: list[int]) -> int
```

Please write about 5 specification-style assertions in Python. Each assertion should use only pure arithmetic and boolean logic, relating `operator` and `operand` to `res` (the expected output).

- **LLM-Generated Assertions**

```
# Specification 1
```

```
assert len(operator) == len(operand) - 1
```

```
# Specification 2
```

```
assert len(operator) >= 1 and len(operand) >= 2
```

```
# Specification 3
```

```
assert res == (
```

```
    operand[0]
    + sum(
        (operator[i] == '+') * operand[i+1]
        - (operator[i] == '-') * operand[i+1]
        for i in range(len(operator)))
    )
    + sum(
        (operator[i] == '*') * (operand[i] * operand[i+1] - operand[i] - operand[i+1])
        for i in range(len(operator))))
)
```

```
# Specification 4
```

```
assert all(op in ['+', '-', '*', '/'] for op in operator)
```

```
# Specification 5
```

```
assert (all(op == '+' for op in operator)) == (res == sum(operand))
```

```
# Specification 6
```

```
assert (all(op == '*' for op in operator)) == (
    res == (operand[0] * operand[1] * (1 if len(operand) == 2 else
        __import__("functools").reduce(lambda a,b: a*b, operand[2:], 1))))
```

Assertion Accuracy

Specification #	Status	Notes
1	Correct	Validates structural constraint between lists
2	Correct	Validates structural constraint between lists
3	Incorrect	Does not accurately model operator precedence and evaluation order
4	Correct	Validates operator domain
5	Correct	Correctly captures all-addition case
6	Incorrect	Uses forbidden import and function calls

- **Accuracy Rate**

4/6 = 66.7%

- **Incorrect Specifications - Analysis and Corrections**

1. **Original Assertion**

```
assert res == (operand[0] + sum((operator[i] == '+') * operand[i+1] - (operator[i] == '-') * operand[i+1] for i in range(len(operator))) + sum((operator[i] == '*') * (operand[i] * operand[i+1] - operand[i] - operand[i+1]) for i in range(len(operator))))
```

- **Analysis**

This assertion attempts to encode operation logic but does not accurately represent left-to-right evaluation with proper operator precedence. The multiplication clause incorrectly tries to “undo” previously added values, which fails for actual algebraic evaluation rules.

- **Manual Correction**

```
assert (operator == ['+', '*', '-'] and operand == [2, 3, 4, 5]) implies (res == 9)
```

Use concrete example-based assertions rather than attempting to generalize the evaluation logic.

2. Original Assertion

```
assert (all(op == '*' for op in operator)) == (res == (operand[0] * operand[1] * (1 if  
len(operand) == 2 else __import__("functools").reduce(lambda a,b: a*b, operand[2:],  
1))))
```

- **Analysis**

Uses `__import__()` and `functools.reduce()`, which are function calls and imports explicitly forbidden in the specification guidelines.

- **Manual Correction**

```
assert (all(op == '*' for op in operator) and len(operand) == 2) implies (res == operand[0]  
* operand[1])
```

Limit to simple cases without imports or function calls.

2. HumanEval/139

- **Function/Method Signature**

```
def special_factorial(n: int) -> int
```

- **Natural Language Problem Description**

The Brazilian factorial is defined as: $\text{brazilian_factorial}(n) = n! * (n-1)! * (n-2)! * \dots * 1!$
where $n > 0$

For example:

$\text{special_factorial}(4) = 288$

The function receives an integer as input and should return the special factorial of this integer.

- **LLM Prompt**

Write formal specifications for the following Python function, `special_factorial`. Your specifications should be written as Python assert statements, expressing the relationship between input and output using pure arithmetic and boolean logic only.

Do NOT call or reference `special_factorial` inside the assertion (no self-reference).

Do NOT use methods that cause side effects, such as modifying data structures, performing I/O (print, read, write), randomness, or timing functions.

Use `res` to denote the expected output value.

Problem description:

The Brazilian factorial is defined as:

$\text{brazilian_factorial}(n) = n! * (n-1)! * (n-2)! * \dots * 1!$

where $n > 0$

For example:

$\text{special_factorial}(4) = 288$

The function will receive an integer as input and should return the special factorial of this integer.

Method signature:

```
def special_factorial(n: int) -> int
```

Please write about 5 specification-style assertions in Python. Each assertion should use only pure arithmetic and Boolean logic, relating the parameter `n` and expected output `res`.

- **LLM-Generated Assertions**

Specification 1

```
assert n > 0 # n must be a positive integer
```

Specification 2

```
assert res > 0 # res must be positive
```

Specification 3

```
assert res == ( # res equals the product of all factorials from 1 to n
```

```
    1 *
    (1 if n < 1 else 1) * # 1! = 1
    (1 if n < 2 else 2) * # 2! = 2
    (1 if n < 3 else 6) * # 3! = 6
    (1 if n < 4 else 24) * # 4! = 24
    (1 if n < 5 else 120) # truncated for brevity)
```

Specification 4

```
assert (n == 1) == (res == 1) # If n == 1, result must be 1
```

Specification 5

```
assert (n == 2) == (res == 2) # If n == 2, result must be 1! * 2! = 2
```

Specification 6

```
assert (n > 1) implies (res > 1) # res grows strictly as n increases (monotonic)
```

Assertion Accuracy

Specification #	Status	Notes
1	Correct	Validates that input is within accepted domain
2	Correct	Ensures output is positive for valid inputs
3	Correct	Truncated, valid for small n, needs expansion for large n
4	Correct	Edge case n==1
5	Correct	Edge case n==2
6	Correct	Ensures monotonic growth as n increases

- **Accuracy Rate**

$6/6 = 100\%$

- **Incorrect Specifications - Analysis and Corrections**

All generated assertions were correct; no incorrect specifications required manual correction.

Part 2: Specifications to Guide Test Improvement

1. HumanEval/160

- **Prompt**

Using the following formal specification assertions for the function `do_algebra`, please write Pytest-style test cases in Python that verify each logical property. Each test should be clearly labeled as "spec-guided." Do not include tests that just duplicate the provided sample examples. Assume `do_algebra` is imported from `do_algebra.py`.

Specifications:

```
# Length constraint
assert len(operator) == len(operand) - 1

# Minimum size requirements
assert len(operator) >= 1 and len(operand) >= 2

# Valid operator domain
assert all(op in ['+', '-', '*', '/'] for op in operator)

# All-addition property
assert (all(op == '+' for op in operator)) implies (res == sum(operand))

# Concrete example validation
assert (operator == ['+', '*', '-'] and operand == [2, 3, 4, 5]) implies (res == 9)

# Simple multiplication case
assert (all(op == '*' for op in operator) and len(operand) == 2) implies (res == operand[0]
* operand[1])
```

Instructions:

- Generate test functions that directly reflect these assertions as Pytest test cases.
- Use `assert do_algebra(...) == ...` in your tests, with clearly labeled function names.
- Only use standard Python and Pytest syntax.
- Label each test as "spec-guided" in the function name or a comment.

Test Cases Generated

- **Coverage Comparison (Statement & Branch)**

Coverage Metric	Before Spec-Guided Tests	After Spec-Guided Tests
Statement Coverage	85%	92%
Branch Coverage	81%	90%

Name	Stmts	Miss	Cover
<hr/>			
do_algebra.py	13	1	92%
test_do_algebra.py	38	0	100%
<hr/>			
TOTAL	51	1	98%

Name	Stmts	Miss	Branch	BrPart	Cover
<hr/>					
do_algebra.py	13	1	8	1	90%
<hr/>					
TOTAL	13	1	8	1	90%

- **Failing Tests**

Original suite failure:

test_examples failed for assert do_algebra(‘*’, ‘**’, 2) == 1 (function returned 18). This failure was present before spec-guided tests were added

Spec-guided domain test failure

test_operator_domain_spec_guided raised ValueError for ‘/’

The implementation allows {‘+’, ‘-’, ‘*’, ‘//’, ‘**’} only, though the spec originally listed ‘/’. The test correctly revealed the operator domain mismatch.

- **Insights**

The inclusion of specification-guided tests led to a clear, measurable improvement in both statement and branch coverage for the do_algebra function, with more code paths, input validations, and edge cases exercised than before. Coverage increased substantially, yet a few residual missed lines remain related primarily to untested exception handling and strict domain control.

The process uncovered subtle mismatches between the specification’s assumed operator set and the actual implementation (for example, the use of ‘/’ in the spec-guided domain test, which is unsupported in the code).

The added tests both validated the function more thoroughly and highlighted areas where original test expectations and code logic may need revision, guiding further suite improvement and deeper alignment between specifications and implementation.

2. HumanEval/139

- **Prompt**

Using the following formal specification assertions for the function `special_factorial`, please write Pytest-style test cases in Python that verify each logical property. Each test should be clearly labeled as “spec-guided.” Do not include tests that just duplicate any sample examples. Assume `special_factorial` is imported from `special_factorial.py`.

Specifications:

```
# n must be a positive integer
assert n > 0

# res must be positive
assert res > 0

# res equals the product of all factorials from 1 to n (for small n, check concrete values)
assert (n == 1) == (res == 1)
assert (n == 2) == (res == 2)
assert (n == 3) == (res == 12)
assert (n == 4) == (res == 288)

# res grows strictly as n increases (monotonic)
assert (n > 1) implies (res > 1)
```

Instructions:

- Generate test functions that directly reflect these assertions as Pytest test cases.
- Use `assert special_factorial(...)` in your tests, with clearly labeled function names.
- Only use standard Python and Pytest syntax.
- Label each test as “spec-guided” in the function name or include a comment.

Test Cases Generated

Concrete value checks are included in spec 3. Specs 4 and 5 combined in 3

- **Coverage Comparison (Statement & Branch)**

Coverage Metric	Before Spec-Guided Tests	After Spec-Guided Tests
Statement Coverage	89%	89%
Branch Coverage	85%	85%

Name	Stmts	Miss	Cover
<hr/>			
special_factorial.py	9	1	89%
test_special_factorial.py	36	0	100%
<hr/>			
TOTAL	45	1	98%

Name	Stmts	Miss	Branch	BrPart	Cover
<hr/>					
special_factorial.py	9	1	4	1	85%
<hr/>					
TOTAL	9	1	4	1	85%

- **Failing Tests**

All tests passed. No new test failures were observed with the addition of the spec-guided tests.

- **Insights**

Adding specification-guided tests provided explicit verification for theoretical properties such as input positivity, output positivity, small-n calculations, and monotonicity. However, the existing test suite had already exercised the logical branches and statements present in the code, so statement and branch coverage percentages remained unchanged. The remaining uncovered lines likely correspond to error handling for invalid inputs (e.g., $n < 1$), which were not tested as per specification and valid input constraints.

- **Case-Specific Explanation**

Coverage did not increase after adding the specification-guided tests for special factorial because the original benchmark suite already exercised all major logical properties listed in the formal specification. For example:

- Input positivity ($n > 0$) was ensured by only passing valid n values in all original test cases.
- Result positivity ($res > 0$) was inherently validated by asserts on specific output values for several n.
- Concrete result checks for small n ($special_factorial(1) == 1$, $special_factorial(2) == 2$, $special_factorial(3) == 12$, etc.) were already tested in original test functions covering these exact inputs.
- Monotonicity (res increases as n increases) was verified by sequentially testing for increasing n and checking results.
- No invalid inputs, such as $n < 1$, were used, so no additional error handling branches were exercised by new tests.