CS520 Exercise 3 Report

- Author: XiaoFan Lu
- GitHub: Link to GitHub Repository

Part 1: Generate, Evaluate, and Refine Specifications

1. Methodology

We utilized **Minimax M2** (`minimax/minimax-m2`) via a custom agent framework to generate formal specifications. The agent was instructed to create executable Python assertions that describe the function's behavior without side effects.

2. LLM Prompt for Specification Generation

The following prompt was used to instruct the agent (from `agent/agent.py`):

```
prompt = f"""
Here is the problem description and function signature:
{prompt_text}

Please generate formal specifications (assertions) for this function.
Ensure the specifications are correct (no self-reference, no side-
effects).
Then, SAVE the specifications to `{spec_file}` using `bash_command`.
"""
```

3. Problem 17: Process Management

Function Signature: `def task_func(process_name: str) -> str`

Generated Assertions (Before Correction): The LLM generated 5 assertions, all of which were **Correct**.

```
# 1. Check return type
assert isinstance(res, str)

# 2. Check return value format
assert res in [f"Process not found. Starting {process_name}.", f"Process
found. Restarting {process_name}."]

# 3. Check process name preservation
assert process_name in res

# 4. Check message structure
assert ("Process" in res and "not found" in res and "Starting" in res) or
("Process" in res and "found" in res and "Restarting" in res)
```

```
# 5. Check message completeness
assert res.endswith(".")
```

Accuracy Rate: 5 / 5 = **100%**

Incorrect Assertions Table: *None. All assertions were logically correct and free of side effects.*

---

4. Problem 15: Command Execution

Function Signature: `def task_func(commands_file_path, output_dir_path) -> list[str]`

Generated Assertions (Before Correction): The LLM generated 6 assertions. 2 were correct, and 4 were incorrect due to side effects or over-specification.

```
 # FORMAL SPECIFICATIONS FOR task_func
# Function signature: def task_func(commands_file_path, output_dir_path) -
> list[str]
#
# Let res = task_func(commands_file_path, output_dir_path)

# ========== REVIEW STATUS: 2 CORRECT, 4 INCORRECT (2/6) ==========

# Check return type - ensures function returns a list as specified in
return annotation
assert isinstance(res, list)
# STATUS: CORRECT ✓

# Check that all returned elements are strings (file paths) - validates
proper output file path format
assert all(isinstance(path, str) for path in res)
# STATUS: CORRECT ✓

# Check file naming convention - ensures output files follow
'command_X_output.txt' pattern where X is index
assert all(path.endswith('_output.txt') and 'command_' in path for path in
res)
# STATUS: INCORRECT ❌
# PROBLEM: Works for full paths but fails when implementations return just
filenames.
# From test_return_value_format_consistency (line 277-289), some solutions
return just filenames.
# CORRECTED VERSION:
# assert all('_output.txt' in path and 'command_' in path for path in res)

# Check that all output files are in the specified output directory -
ensures proper file organization
assert all(output_dir_path in path for path in res)
# STATUS: INCORRECT ❌
# PROBLEM: Assumes all implementations return full paths. Some return just
filenames.
# From test_return_value_format_consistency (line 277-289),
```

```
implementations vary.
# CORRECTED VERSION:
# # This assertion is implementation-specific and should be removed.
# # Instead, check that filenames exist without assuming path format:
# assert all(len(path) > 0 for path in res)

# Check that output directory exists after function execution - validates
requirement to create directory if missing
assert os.path.exists(output_dir_path) and os.path.isdir(output_dir_path)
# STATUS: INCORRECT ❌
# PROBLEM: Uses side effects (file system I/O via os.path.exists).
# Assertions should be pure and not perform I/O operations.
# CORRECTED VERSION:
# # Remove this assertion - it uses side effects (file system access).
# # The existence of the directory is an implementation detail, not a
contract property.

# Check sequential file naming - ensures files are numbered sequentially
from 1
import re
file_numbers = []
for path in res:
    match = re.search(r'command_(\d+)_output\.txt', path)
    if match:
        file_numbers.append(int(match.group(1)))
assert sorted(file_numbers) == list(range(1, len(res) + 1))
# STATUS: INCORRECT ❌
# PROBLEM: Multiple issues:
#   1. Imports module (re) within assertion block - side effect
#   2. Modifies mutable state (file_numbers list) - side effect
#   3. Assumes specific numbering implementation (starts at 1) - over-
specification
# CORRECTED VERSION:
# # This should be expressed without side effects or imports:
# # Pre-compile regex outside assertion context, use pure functions only
# # However, this property is too implementation-specific and should be
simplified:
# assert len(res) >= 0  # Simple property: function returns valid list
length
```

Accuracy Rate: 2 / 6 = **33.33%**

Incorrect Assertions Table:

Incorrect Assertion (Original)	Issue Explanation	Corrected Version

Incorrect Assertion (Original)	Issue Explanation	Corrected Version
`assert all(path.endswith('_output.txt') and 'command_' in path for path in res)`	**Over-specification**: Assumes specific file extension/naming that is an implementation detail, not a contract requirement. Fails if full paths are returned.	`assert all('_output.txt' in path and 'command_' in path for path in res)`
`assert all(output_dir_path in path for path in res)`	**Over-specification**: Assumes full absolute paths are returned, which may not be true for all valid implementations.	`assert all(len(path) > 0 for path in res)`
`assert os.path.exists(output_dir_path) and os.path.isdir(output_dir_path)`	**Side Effect**: Checks file system state (I/O) which is forbidden in pure assertions.	*Removed entirely* (Implementation detail)
`import re; ...; assert sorted(file_numbers) == list(range(1, len(res) + 1))`	**Side Effect & Complexity**: Imports modules, mutates lists, and performs complex logic that mirrors implementation rather than specifying behavior.	`assert len(res) >= 0`

Part 2: Use Specifications to Guide Test Improvement

1. LLM Prompt for Test Generation

The corrected specifications were fed back into the LLM to generate test cases using the following prompt (from `agent/agent.py`):

```
prompt_test = f"""
Read the formal specifications from `{spec_file}`.
Read the existing test file `{test_file}` to understand the context.
Generate spec-guided test cases and APPEND them to `{test_file}`.
Use `bash_command` to append the code.
"""
```

2. Spec-Guided Tests

The following tests were generated and appended to the test suites:

Problem 15 (`test_BigCodeBench_15.py`):

- `test_spec_return_type_is_list`
- `test_spec_all_elements_are_strings`
- `test_spec_file_naming_convention`

- `test_spec_non_empty_paths`
- `test_spec_list_length_non_negative`
- `test_spec_non_empty_list_elements_contain_output_txt`
- `test_spec_combined_properties_single_command`
- `test_spec_combined_properties_multiple_commands`
- `test_spec_empty_input_consistency`
- `test_spec_path_format_variations`

Problem 17 (`test_BigCodeBench_17.py`):

- `test_spec_return_type_string`
- `test_spec_return_value_format_process_not_found`
- `test_spec_return_value_format_process_found`
- `test_spec_process_name_preservation_not_found`
- `test_spec_process_name_preservation_found`
- `test_spec_message_structure_not_found`
- `test_spec_message_structure_found`
- `test_spec_message_completeness_ends_with_period`
- `test_spec_comprehensive_all_assertions`
- `test_spec_exact_message_patterns_only`
- `test_spec_alternate_pattern_rejection`

3. Coverage Analysis

The formal specifications **did not improve code coverage** for either problem.

Problem	Metric	Before Spec	After Spec	Change
Problem 17	Statement Coverage	56%	56%	0%
	Branch Coverage	67%	67%	0%
	Total Coverage	**58%**	**58%**	**0%**
Problem 15	Statement Coverage	84%	84%	0%
	Branch Coverage	83%	83%	0%
	Total Coverage	**84%**	**84%**	**0%**

4. Case-Specific Insight & Conclusion

Why did coverage not improve?

1. **Problem 17 (Process Management)**:
    - **Insight**: The generated specifications (checking return strings) were redundant with existing tests. The existing tests already thoroughly checked the return messages ("Process found...", etc.). The missing coverage in this problem likely relates to exception handling (e.g., `psutil.NoSuchProcess`, `AccessDenied`) or specific process states (zombies) which are hard to trigger via simple input/output assertions without extensive mocking, which the spec-guided tests did not introduce.

2. **Problem 15 (Command Execution)**:
    - **Insight**: Similarly, the specifications focused on the *structure* of the output (list of strings, naming convention). The existing tests already covered the "happy path" and several error cases. The missing coverage involves deep edge cases in subprocess execution or file I/O failures that simple return-value assertions cannot easily drive. The spec-guided tests effectively re-verified what was already tested (that the function returns a list of files).

Overall Conclusion: Formal specification generation with **Minimax M2** achieved mixed accuracy (100% on simple string tasks, 33% on complex system tasks). Crucially, it **failed to improve test coverage**. This is primarily because:

1. **Complexity of System State**: The problems involve subprocesses and file I/O, which are difficult to verify with pure, side-effect-free assertions.
2. **Surface-Level Specifications**: The generated assertions focused on return types and formatting (syntax) rather than deep behavioral correctness (semantics), leading to tests that merely restated existing checks.