FailMapper: Detailed Technical Implementation Guide

Abstract

This document provides in-depth technical details of the FailMapper framework, including concrete implementation examples, static analysis processes, context extraction mechanisms, prompt generation strategies, and FA-MCTS bug discovery simulations. It serves as a comprehensive guide for understanding the internal workings of the failure-aware test generation system.

---

1. Multi-Layer Static Analysis Engine

1.1 Three-Tier Parsing Strategy

The static analysis engine employs a robust three-tier fallback mechanism to ensure maximum compatibility with various Java codebases:

Tier 1: Direct AST Parsing

```
def analyze_java_file(file_path: str) -> Dict[str, Any]:
    # Strategy 1: Direct javalang parsing
    try:
        tree = javalang.parse.parse(content)
        return parse_with_javalang(tree, content)
    except Exception as e:
        # Fall back to next tier
```

Tier 2: Preprocessed AST Parsing

```
def preprocess_java_content(content: str) -> str:
    # Remove complex formatting (Java 8+ features)
    content = re.sub(r'String\.format\([^)]*String\.format[^)]*\)',
'String.format(...)', content)

    # Simplify complex generic expressions
    content = re.sub(r'<[^<>]*<[^<>]*>[^<>]*>', '<...>', content)

    # Remove complex lambda expressions
    content = re.sub(r'->\s*\{[^}]*\}', '-> {...}', content)

    # Simplify method references
    content = re.sub(r'::\w+', '::method', content)

    return content
```

Tier 3: Regex Fallback

```
def extract_basic_info_with_regex(content: str, file_path: str) ->
Dict[str, Any]:
    # Extract class information
    class_pattern = r'(?:public\s+)?(?:abstract\s+)?class\s+(\w+)
(?:\s+extends\s+\w+)?(?:\s+implements\s+[\w,\s<>]+)?\s*\{'
    classes = re.findall(class_pattern, content)

    # Extract method signatures
    method_pattern = r'(?:public|private|protected)?\s*(?:static\s+)?
(?:final\s+)?(\w+(?:<[^>]*>)?)\s+(\w+)\s*\([^)]*\)\s*\{'
```

1.2 Data Flow Graph Construction

The data flow analyzer builds comprehensive graphs tracking variable dependencies:

```
def analyze_method_dfg(method: javalang.tree.MethodDeclaration) ->
List[Dict[str, Any]]:
    dfg = []
    variables = set()

    # Track parameter inputs
    for param in method.parameters:
        variables.add(param.name)
        dfg.append({
            "from": "parameter",
            "to": param.name,
            "type": "input",
            "details": f"{convert_type(param.type)} {param.name}"
        })

    # Analyze method body statements
    if method.body:
        analyze_block(method.body, dfg, variables)

    return dfg
```

Example Data Flow Graph Output:

```
{
  "calculateTotal": [
    {
      "from": "parameter",
      "to": "items",
      "type": "input",
      "details": "List<Item> items"
    },
```

```
    {
      "from": ["items"],
      "to": "total",
      "type": "assignment",
      "details": "double total = 0.0"
    },
    {
      "from": ["total", "item"],
      "to": "total",
      "type": "assignment",
      "details": "total += item.getPrice()"
    },
    {
      "from": ["total"],
      "to": "return",
      "type": "return",
      "details": "return total"
    }
  ]
}
```

---

2. Context Extraction and Prompt Generation

2.1 Dependency Resolution Algorithm

The prompt generator implements a sophisticated dependency resolution mechanism:

```
def _resolve_dep_fqns(import_types, package, testable_units,
indirect_deps_for_class):
    """
    Resolves type names to Full Qualified Names (FQNs) with priority:
    1) Already FQN and exists in testable_units
    2) Simple name matches indirect dependencies
    3) Simple name has unique match in testable_units
    """
    resolved = set()
    simple_to_fqns = _build_simple_to_fqn_map(testable_units)

    for t in import_types:
        if not t or t in {"void", "boolean", "int", "long", "float",
"double", "char", "byte", "short"}:
            continue

        # Check if already FQN
        if '.' in t and t in testable_units:
            resolved.add(t)
            continue

        # Resolve simple names
        candidates = simple_to_fqns.get(t, [])
```

```
        if indirect_deps_for_class:
            # Prioritize indirect dependency matches
            matched = [c for c in candidates if c in
indirect_deps_for_class]
            if matched:
                resolved.update(matched)
                continue

        if candidates:
            resolved.update(candidates)

    return list(resolved)
```

2.2 Comprehensive Prompt Structure

The generated prompts follow a structured template that provides comprehensive context:

```
def generate_prompt(class_info, package, dependencies_info,
indirect_dependencies_info):
    prompt = f"""
==============================
JAVA CLASS UNIT TEST GENERATION
==============================

Class: {class_name}
Package: {package}

CRITICAL TESTING REQUIREMENTS:
1. DO NOT use any mocking frameworks (Mockito, EasyMock, PowerMock, etc.)
2. DO NOT use @Mock, @MockBean, @InjectMocks, or any mock-related
annotations
3. Use only real objects and direct instantiation for testing
4. Focus on testing actual behavior with real object interactions

-----------
1. STRUCTURE
-----------
Superclass: {superclass if superclass else 'None'}
Implemented Interfaces: {', '.join(implemented_interfaces) if
implemented_interfaces else 'None'}

Fields:
{json.dumps(fields, indent=4)}

Methods:
{json.dumps(methods, indent=4)}

--------------------
2. DATA FLOW SUMMARY
--------------------
Boundary Conditions:
```

```
{json.dumps(data_flow_summary['boundary_conditions'], indent=4)}

Method Flows:
{json.dumps(data_flow_summary['method_flows'], indent=4)}

------------------------------
4. DEPENDENCY API REFERENCES
------------------------------
(Only use the following APIs for collaborators; do NOT fabricate missing
members.)
{json.dumps(dependency_api_refs, indent=4)}
"""
```

2.3 Complete Prompt Examples

2.3.1 Initial Test Generation Prompt

Here's an actual prompt generated for a `Calculator` class:

```
================================
JAVA CLASS UNIT TEST GENERATION
================================

Class: Calculator
Package: com.example.math

CRITICAL TESTING REQUIREMENTS:
1. DO NOT use any mocking frameworks
2. Use only real objects and direct instantiation
3. Focus on testing actual behavior with real object interactions

-----------
1. STRUCTURE
-----------
Superclass: None
Implemented Interfaces: None

Fields:
[
    {
        "name": "precision",
        "type": "int",
        "modifiers": ["private", "final"],
        "initializer": "2",
        "visibility": "private",
        "is_final": true
    }
]

Methods:
[
```

```
    {
        "name": "divide",
        "return_type": "double",
        "parameters": ["double", "double"],
        "modifiers": ["public"],
        "throws": ["ArithmeticException"],
        "body": "...",
        "is_override": false
    },
    {
        "name": "sqrt",
        "return_type": "double",
        "parameters": ["double"],
        "modifiers": ["public"],
        "throws": [],
        "body": "...",
        "is_override": false
    }
]

--------------------
2. DATA FLOW SUMMARY
--------------------
Boundary Conditions:
[
    "divide: if (divisor == 0.0)",
    "sqrt: if (value < 0.0)"
]

Method Flows:
{
    "divide": [
        "Check: if (divisor == 0.0)",
        "Throw: ArithmeticException",
        "Return: dividend / divisor"
    ],
    "sqrt": [
        "Check: if (value < 0.0)",
        "Throw: IllegalArgumentException",
        "Return: Math.sqrt(value)"
    ]
}

------------------------------
4. DEPENDENCY API REFERENCES
------------------------------
[
    {
        "name": "java.lang.Math",
        "package": "java.lang",
        "type": "class",
        "methods": [
            {
                "name": "sqrt",
```

```
                "parameters": ["double"],
                "return_type": "double"
            }
        ]
    }
]
```

2.3.2 FA-MCTS Dynamic Action Prompts

During FA-MCTS execution, different types of dynamic prompts are generated based on actions:

A) Business Logic Bug Detection Prompt:

```
CRITICAL REQUIREMENTS:
1. DO NOT use @Nested annotations or nested test classes - they cause
coverage tracking issues
2. Generate a COMPLETE test class with ALL methods intact - do not omit
any code
3. DO NOT use placeholders like "... existing code ..." or similar
comments
4. Your response MUST contain the ENTIRE test class that can compile
without modifications

STRICT ANTI-MOCKING REQUIREMENTS:
- ABSOLUTELY NO use of any mocking frameworks (Mockito, EasyMock,
PowerMock, etc.)
- ABSOLUTELY NO @Mock, @MockBean, @InjectMocks, or any mock-related
annotations
- Use ONLY real objects and direct instantiation for testing

You are an expert Java test engineer focusing on detecting BUSINESS LOGIC
BUGS.
You need to extend the following test class for Calculator to find a
specific business logic bug.

BUSINESS LOGIC ISSUE DETAILS:
- Method with potential issue: findMax
- Issue type: off_by_one
- Description: Test for potential business logic issue: Loop boundary
condition may cause index out of bounds
- Expected behavior: Method should iterate within array bounds
- Actual behavior: Loop condition uses <= instead of < with array.length
- Specifically test: boundary conditions and array access patterns

Current test coverage: 67.5%

Here is the existing test code:
@Test
public void testFindMaxBasic() {
    int[] arr = {1, 3, 2};
    assertEquals(3, calculator.findMax(arr));
```

```
}

Here is the source code being tested:
public int findMax(int[] arr) {
    if (arr.length == 0) return -1;
    int max = arr[0];
    for (int i = 1; i <= arr.length; i++) {  // BUG HERE
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

ADDITIONAL CONTEXT (dependencies and rules):
Only use APIs explicitly defined in the dependency references section.
```

B) Boundary Condition Testing Prompt:

```
CRITICAL REQUIREMENTS - READ CAREFULLY:
1. DO NOT use @Nested annotations or nested test classes - they cause
coverage tracking issues
2. Generate a COMPLETE test class with ALL methods intact - do not omit
any code
3. WRITE OUT EVERY SINGLE LINE OF CODE - no shortcuts, abbreviations, or
omissions allowed
4. Your response MUST contain the ENTIRE test class that can compile
without modifications

STRICT ANTI-MOCKING REQUIREMENTS:
- Use ONLY real objects and direct instantiation for testing

You are an expert Java test engineer focusing on detecting logical bugs.
You need to extend the following test class for Calculator to find bugs.

Focus specifically on finding logical bugs related to:
1. Boundary conditions
2. Boolean logic errors
3. Off-by-one errors
4. Null handling problems

Current test coverage: 67.5%

Here is the existing test code:
```java
public class CalculatorTest {
    private Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }
```
```

```
    @Test
    public void testFindMaxBasic() {
        int[] arr = {1, 3, 2};
        assertEquals(3, calculator.findMax(arr));
    }
}

Here is the source code being tested:
public int findMax(int[] arr) {
    if (arr.length == 0) return -1;
    int max = arr[0];
    for (int i = 1; i <= arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
Add new test methods to specifically test the boundary condition:
Condition: i <= arr.length
Line: 23

Focus on edge cases around this boundary (e.g., value-1, value, value+1).
```

C) Compilation Error Fix Prompt:

```
CRITICAL REQUIREMENTS - READ CAREFULLY:
1. Generate a COMPLETE test class with ALL methods intact - do not omit
any code
2. Your response MUST contain the ENTIRE test class that can compile
without modifications

You are an expert Java test engineer focusing on detecting logical bugs.
You need to extend the following test class for Calculator to find bugs.

Current test coverage: 67.5%

Here is the existing test code:
public class CalculatorTest {
    private Calculator calculator
    // Missing semicolon above

    @Test
    public void testDivide() {
        assertThrows(ArithmeticException.class, () -> {
            calculator.divide(1.0, 0.0)  // Missing semicolon
        }
    }
}

IMPORTANT: The current test code has COMPILATION ERRORS that MUST be
```

```
fixed!

Compilation errors found:
1. ERROR: ';' expected at line 2
   SUGGESTED FIX: Add semicolon after field declaration
2. ERROR: ';' expected at line 7
   SUGGESTED FIX: Add semicolon after method call
3. ERROR: ')' expected at line 8
   SUGGESTED FIX: Close lambda expression parentheses

Your task is to:
1. Fix ALL compilation errors in the test code above
2. Make sure the fixed code is syntactically correct and can compile
3. Preserve all existing test logic while fixing the errors
4. Add any missing imports if needed
5. IMPORTANT: Write out the COMPLETE test class with all fixes applied

Common compilation errors to fix:
- Missing semicolons
- Unclosed brackets or parentheses
- Invalid comment syntax
- Missing imports
- Type mismatches

Remember: You MUST provide the COMPLETE test class, not just the fixes!
```

D) Coverage Improvement Prompt:

```
CRITICAL ANTI-PLACEHOLDER REQUIREMENTS:
I need the ENTIRE test class including ALL original methods, not just the
fixed parts.
Your response must contain:
1. All package declarations
2. All import statements
3. The complete class definition
4. ALL existing test methods, not just the fixed ones
5. All fields and setup methods

ABSOLUTELY FORBIDDEN:
- DO NOT use "// ... existing code ..."
- DO NOT use "// [Previous imports remain exactly the same]"
- DO NOT use ANY placeholders or shortcuts
- You MUST write out every single line of the existing code

================================
JAVA CLASS UNIT TEST GENERATION WITH FEEDBACK
================================

Class: Calculator
Package: com.example.math

Current test code:
```

```
public class CalculatorTest {
    private Calculator calculator;

    @BeforeEach
    void setUp() {
        calculator = new Calculator();
    }

    @Test
    public void testFindMaxBasic() {
        int[] arr = {1, 3, 2};
        assertEquals(3, calculator.findMax(arr));
    }
}

------------------
TEST FEEDBACK
------------------
COVERAGE ANALYSIS:
- Current coverage: 67.5%
- Missed lines: 12, 15, 18 (boundary conditions)
- Uncovered branches: Empty array handling, exception cases

SUGGESTIONS:
1. Add test for empty array input
2. Add test for single element array
3. Add test for array with duplicate maximum values
4. Add test for negative numbers

Based on the above feedback, improve the unit tests for this class. Focus
on:
1. Increasing code coverage
2. Adding tests for uncovered lines and branches
3. Resolving all test errors and failures

Please provide a complete JUnit test class that addresses these issues.
```

---

3. Failure Pattern Detection Engine

3.1 Bug Pattern Detection Categories

The FS_Detector implements 20+ specialized pattern detectors:

Off-by-One Bug Detection

```
def _detect_off_by_one_bugs(self):
    # Pattern 1: Fixed array indices
    array_access_pattern = r'(\w+)\s*\[\s*([^][]+)\s*\]'
    matches = re.finditer(array_access_pattern, self.source_code)
```

```
    for match in matches:
        array_name = match.group(1)
        index_expr = match.group(2)
        line_num = self._get_line_number(match.start())

        # Check for hardcoded indices
        if re.search(r'^\d+$', index_expr):
            context = self._get_context(line_num, 5)
            if f"{array_name}.length" in context:
                self.patterns.append({
                    "type": "off_by_one",
                    "location": line_num,
                    "code": f"{array_name}[{index_expr}]",
                    "risk_level": "medium",
                    "description": f"Hardcoded array index ({index_expr})
near length check"
                })
```

Boundary Condition Bug Detection

```
def _detect_boundary_condition_bugs(self):
    boundary_check_pattern = r'if\s*\(\s*([^)]+?)\s*([<>=!]+)\s*
([^)]+?)\s*\)'
    matches = re.finditer(boundary_check_pattern, self.source_code)

    for match in matches:
        left = match.group(1).strip()
        operator = match.group(2)
        right = match.group(3).strip()
        line_num = self._get_line_number(match.start())

        # Find checks against 0 or 1
        if right in ["0", "1"] or left in ["0", "1"]:
            context = self._get_context(line_num, 3)
            if "[" in context and "]" in context:
                self.patterns.append({
                    "type": "boundary_condition",
                    "location": line_num,
                    "code": f"{left} {operator} {right}",
                    "risk_level": "high",
                    "description": f"Boundary check against {right} near
array access"
                })
```

3.2 Example Detected Patterns

For a sample code snippet:

```java
public int findMax(int[] arr) {
    if (arr.length == 0) return -1;

    int max = arr[0];
    for (int i = 1; i <= arr.length; i++) {  // BUG: should be < not <=
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

The detector would identify:

```
[
    {
        "type": "off_by_one",
        "location": 5,
        "code": "i <= arr.length",
        "risk_level": "high",
        "description": "Potential off-by-one in loop condition using <=
with length"
    },
    {
        "type": "boundary_condition",
        "location": 2,
        "code": "arr.length == 0",
        "risk_level": "high",
        "description": "Boundary check against 0 near array access"
    }
]
```

---

4. FA-MCTS Bug Discovery Simulation

4.1 Enhanced Node Structure with Failure Rewards

```python
class FA_MCTSNode:
    def __init__(self, state, parent=None, action=None):
        # Traditional MCTS components
        self.wins = 0.0
        self.visits = 0

        # Failure-specific rewards
        self.logic_bug_rewards = 0.0          # Rewards for finding
logical bugs
        self.failure_coverage_rewards = 0.0   # Rewards for covering
failure patterns
```

```
        self.high_risk_pattern_rewards = 0.0  # Rewards for high-risk
pattern coverage

        # Bug tracking
        self.bugs_found = 0
        self.bug_types_found = set()
        self.covered_patterns = set()
```

4.2 Enhanced UCB Score Calculation

```
def ucb_score(child):
    # Base UCB1 score
    exploitation = child.wins / child.visits if child.visits > 0 else 0.0
    exploration = exploration_weight * (2 * (self.visits / child.visits)
** 0.5) if child.visits > 0 else float('inf')

    # Failure-specific rewards with decay
    logic_bonus = 0.0
    if child.visits > 0:
        # Bug discovery rewards
        logic_bug_term = child.logic_bug_rewards / child.visits

        # Pattern coverage rewards
        logic_coverage_term = child.failure_coverage_rewards /
child.visits

        # High-risk pattern rewards
        high_risk_term = child.high_risk_pattern_rewards / child.visits

        # Novelty bonus for new discoveries
        novelty_bonus = 0.2 if child.is_novel else 0.0

        # Visits decay to encourage exploration
        visits_decay = 1.0 / (1.0 + 0.1 * child.visits)

        # Strategy diversity bonus
        diversity_bonus = 0.15 if different_strategy_type else 0.0

        # Combined logic bonus
        logic_bonus = f_weight * (
            (logic_bug_term + logic_coverage_term + high_risk_term +
novelty_bonus) *
            visits_decay
        ) + diversity_bonus

    return exploitation + exploration + logic_bonus
```

4.3 Action Generation Strategy

The FA-MCTS generates actions based on detected failure patterns:

```
def generate_possible_actions(self, test_prompt, source_code,
uncovered_data=None,
                              f_model=None, failures=None,
strategy_selector=None):
    possible_actions = []

    # 1. Business logic actions from detected issues
    if hasattr(self.state, 'business_logic_issues'):
        for issue in self.state.business_logic_issues:
            action = {
                "type": "business_logic_test",
                "issue_type": issue.get('type', 'unknown'),
                "method": issue.get('method', ''),
                "description": f"Test for potential business logic issue:
{issue.get('description', '')}",
                "confidence": issue.get('confidence', 0),
                "business_logic": True
            }
            possible_actions.append(action)

    # 2. Boundary condition actions
    if strategy_id == "boundary_testing" and f_model:
        boundary_conditions = f_model.boundary_conditions
        if boundary_conditions:
            selected_conditions = random.sample(
                boundary_conditions,
                min(2, len(boundary_conditions))
            )

            for condition in selected_conditions:
                action = {
                    "type": "boundary_test",
                    "condition": condition.get("condition", ""),
                    "line": condition.get("line", 0),
                    "strategy": strategy_id,
                    "description": f"Test boundary condition:
{condition_str[:40]}..."
                }
                possible_actions.append(action)

    # 3. Uncovered line targeting
    if uncovered_lines:
        selected_lines = random.sample(uncovered_lines, min(5,
len(uncovered_lines)))
        for line_info in selected_lines:
            line_action = {
                "type": "target_line",
                "line": line_info.get("line", 0),
                "content": line_info.get("content", "").strip(),
                "description": f"Target uncovered line {line_num}:
{content[:40]}..."
            }
```

```
                possible_actions.append(line_action)

        return possible_actions
```

4.4 Bug Discovery Simulation Example

Consider discovering a bug in the `findMax` method:

MCTS Iteration 1:

- Initial state: Empty test class
- Action: Generate basic test method
- Result: Test passes, low reward

MCTS Iteration 5:

- State: Basic tests exist
- Action: Generate boundary test for empty array
- Result: Test passes, medium reward for boundary coverage

MCTS Iteration 12:

- State: Several tests exist
- Action: Generate test targeting loop condition (detected off-by-one pattern)
- Test generated:

```
@Test
public void testFindMaxArrayBoundary() {
    int[] arr = {1, 2, 3};
    assertThrows(IndexOutOfBoundsException.class, () -> {
        calculator.findMax(arr);
    });
}
```

- Result: Test fails with IndexOutOfBoundsException, HIGH REWARD for bug discovery

MCTS Iteration 15:

- State: Bug found, high node reward
- Action: Generate additional edge case tests based on bug pattern
- Result: More tests targeting similar off-by-one issues

Final Outcome:

- Bug discovered and verified
- Additional related tests generated
- High confidence bug report produced

4.5 Strategy Selection Adaptation

```
class TestStrategySelector:
    def select_strategy(self, current_state, available_actions):
        # Calculate strategy effectiveness
        for strategy in self.strategies:
            if strategy.times_used >= 5:
                success_rate = strategy.calculate_success_rate()
                strategy.adjust_weight(success_rate)

        # Select based on weighted probabilities
        weights = [s.weight for s in self.strategies]
        selected_strategy = random.choices(self.strategies,
weights=weights)[0]

        # Record usage
        selected_strategy.record_usage()

        return selected_strategy
```

---

5. Bug Verification and Classification

5.1 LLM-Powered Bug Analysis

```
class BugVerifier:
    def verify_potential_bug(self, source_code, test_code, failure_info):
        verification_prompt = f"""
        CONTEXT:
        Source Code:
        {source_code}

        Test Code:
        {test_code}

        Failure Information:
        {failure_info}

        TASK: Analyze if this represents a real bug in the source code.

        CRITERIA:
        1. Is the failure due to a logical error in the source code?
        2. What type of bug is this (off-by-one, null pointer, boundary
condition, etc.)?
        3. What is the severity (low, medium, high, critical)?
        4. Provide a concise explanation of the root cause.

        RESPONSE FORMAT:
        {{
            "is_real_bug": true/false,
            "bug_type": "category",
            "severity": "level",
```

```
            "explanation": "detailed explanation",
            "suggested_fix": "brief fix description"
        }}
        """

        response = call_anthropic_api(verification_prompt)
        return self._parse_verification_response(response)
```

5.2 Example Bug Verification

Input:

- Source: `findMax` method with off-by-one error
- Test: Boundary test causing IndexOutOfBoundsException
- Failure: Array index 3 out of bounds for length 3

LLM Analysis:

```
{
    "is_real_bug": true,
    "bug_type": "off_by_one",
    "severity": "high",
    "explanation": "The loop condition 'i <= arr.length' allows accessing
index equal to array length, causing IndexOutOfBoundsException. Should use
'i < arr.length'.",
    "suggested_fix": "Change loop condition from 'i <= arr.length' to 'i <
arr.length'"
}
```

---

6. Performance Optimization Techniques

6.1 Caching and Memoization

```
class AnalysisCache:
    def __init__(self):
        self.dfg_cache = {}
        self.pattern_cache = {}
        self.dependency_cache = {}

    def get_cached_dfg(self, file_path, last_modified):
        cache_key = f"{file_path}:{last_modified}"
        return self.dfg_cache.get(cache_key)

    def cache_dfg(self, file_path, last_modified, dfg_data):
        cache_key = f"{file_path}:{last_modified}"
        self.dfg_cache[cache_key] = dfg_data
```

6.2 Adaptive Exploration

```
class AdaptiveMCTS:
    def adjust_exploration_weight(self, iteration, total_iterations):
        # Decrease exploration weight as search progresses
        progress = iteration / total_iterations
        base_weight = 1.0

        if progress < 0.3:
            # High exploration in early phases
            return base_weight * 1.5
        elif progress < 0.7:
            # Balanced exploration/exploitation
            return base_weight
        else:
            # Focus on exploitation in final phase
            return base_weight * 0.6
```

---

7. Integration and Workflow

7.1 Complete Workflow Example

```
# 1. Static Analysis
python static_analysis.py /path/to/project --output_dir ./analysis

# 2. Prompt Generation
python prompt_generator.py ./analysis/project_combined_analysis.json --
output_dir ./prompts

# 3. FA-MCTS Test Generation
python failmapper.py --project /path/to/project --prompt ./prompts --class
Calculator --package com.example.math --f-weight 2.0 --max-iterations 30
```

7.2 Output Analysis

Generated Test Example:

```
@Test
@DisplayName("Test boundary condition with empty array")
public void testFindMaxEmptyArray() {
    int[] emptyArray = {};
    int result = calculator.findMax(emptyArray);
    assertEquals(-1, result, "findMax should return -1 for empty array");
}

@Test
@DisplayName("Test off-by-one bug in loop boundary")
```

```java
public void testFindMaxBoundaryError() {
    int[] testArray = {5, 2, 8, 1};

    // This test will expose the off-by-one error
    assertThrows(IndexOutOfBoundsException.class, () -> {
        calculator.findMax(testArray);
    }, "findMax should not access array out of bounds");
}
```

Bug Report:

```
{
    "class_name": "Calculator",
    "bugs_found": [
        {
            "method": "findMax",
            "bug_type": "off_by_one",
            "severity": "high",
            "line": 23,
            "description": "Loop condition allows accessing array index
equal to length",
            "test_that_found_it": "testFindMaxBoundaryError",
            "suggested_fix": "Change 'i <= arr.length' to 'i <
arr.length'"
        }
    ],
    "coverage": 95.7,
    "total_tests_generated": 12,
    "bugs_verified": 1
}
```

---

8. Conclusion

The FailMapper framework represents a sophisticated approach to automated test generation that goes beyond traditional coverage metrics to specifically target logical bugs. Through its multi-layered static analysis, intelligent pattern detection, failure-aware MCTS algorithm, and LLM-powered verification, it demonstrates the potential for AI-assisted testing tools to significantly improve software quality assurance.

The detailed implementation examples and simulations provided in this document illustrate how the various components work together to create an effective bug discovery system. The framework's modular architecture and extensive configuration options make it adaptable to various Java project types and testing requirements, while its failure-aware approach ensures that the most critical bugs are discovered and addressed.

This detailed technical documentation provides comprehensive insights into the FailMapper framework's implementation, serving as both a reference guide for developers and a foundation for future research in failure-aware test generation.