FailMapper: Technical Architecture Documentation

Abstract

FailMapper is a Failure-Aware Monte Carlo Tree Search (FA-MCTS) framework for automated unit test generation in Java projects. Unlike traditional coverage-based approaches, FailMapper incorporates failure model extraction and bug pattern detection to generate tests specifically targeting logical bugs and edge cases. This document provides a comprehensive technical analysis of the framework's architecture, algorithms, and implementation details.

1. Introduction

1.1 Problem Statement

Traditional automated test generation tools focus primarily on achieving high code coverage but often fail to detect logical bugs—subtle defects in program logic that may not manifest through simple coverage metrics. FailMapper addresses this limitation by incorporating failure-aware heuristics into the test generation process.

1.2 Key Contributions

- **Failure Model Extraction**: Automated extraction of boundary conditions, logical operations, and control flow patterns from Java source code
- **Failure Pattern Detection**: Detection of 20+ categories of common logical bug patterns based on empirical research
- **FA-MCTS Algorithm**: Enhanced Monte Carlo Tree Search with failure-aware reward functions
- **Multi-Provider LLM Integration**: Support for multiple AI providers (Claude, GPT, DeepSeek) for robust test generation

2. System Architecture

2.1 Three-Stage Pipeline Architecture

FailMapper follows a modular three-stage pipeline:

```
Input: Java Project → [Static Analysis] → [Prompt Generation] → [FA-MCTS
Test Generation] → Output: Test Cases + Bug Reports
```

Stage 1: Static Analysis Engine

- **File Analyzer** (`file_analyzer.py`): Multi-layer Java parsing with fallback mechanisms
- **Data Flow Analyzer** (`data_flow_analyzer.py`): Constructs data flow graphs
- **Dependency Analyzers** (`dependency_analyzer.py`, `indirect_dependency_analyzer.py`): Direct and indirect dependency analysis

- **Boundary Exception Analyzer** (`boundary_exception_analyzer.py`): Exception handling pattern detection

Stage 2: Prompt Generation System

- **Extractor** (`extractor.py`): Failure model extraction from source code
- **Prompt Generator** (`prompt_generator.py`): Converts analysis results into structured prompts

Stage 3: FA-MCTS Test Generation

- **FA-MCTS Engine** (`fa_mcts.py`): Failure-aware Monte Carlo Tree Search
- **Strategy Selector** (`test_generation_strategies.py`): Intelligent strategy selection
- **Bug Verifier** (`bug_verifier.py`): LLM-powered bug verification

2.2 Core Components Detail

2.2.1 Static Analysis Engine (`static_analysis.py`)

The static analysis engine employs a robust multi-layer parsing approach:

```
def analyze_project(project_path: str) -> dict:
    # Multi-file analysis with error handling
    # Combines DFG, dependency, and indirect dependency analysis
    # Returns comprehensive project model
```

Key Features:

- Graceful fallback from AST-based parsing to regex-based analysis
- Unicode surrogate character cleaning for robust JSON serialization
- Comprehensive error reporting and success rate tracking

2.2.2 Failure Model Extractor (`extractor.py`)

The Extractor class builds a comprehensive logical model:

```
class Extractor:
    def __init__(self, source_code, class_name, package_name):
        # Core logic model components
        self.boundary_conditions = []
        self.operations = []
        self.control_flow_paths = []
        self.data_dependencies = []
        self.decision_points = []
```

Extracted Features:

- **Boundary Conditions**: Numeric comparisons, array bounds, null checks

- **Operations**: Logical operators, arithmetic operations, string operations
- **Control Flow**: Branch conditions, loop structures, exception handling
- **Data Dependencies**: Variable relationships, method parameter flows

2.2.3 Failure Scenario Detector (`failure_scenarios.py`)

The FS_Detector implements pattern detection for 20+ bug categories:

```
class FS_Detector:
    def __init__(self, source_code, class_name, package_name,
f_model=None):
        # Pattern detection results
        self.detectors = [
            self._detect_operator_precedence_bugs,
            self._detect_off_by_one_bugs,
            self._detect_boundary_condition_bugs,
            self._detect_null_handling_bugs,
            # ... 16+ additional detectors
        ]
```

Bug Pattern Categories:

1. **Boundary Condition Bugs**: Off-by-one errors, missing boundary checks
2. **Logical Operator Bugs**: Wrong operators, precedence issues
3. **Null Handling Bugs**: Missing null checks, dereferencing issues
4. **String Comparison Bugs**: Using == instead of equals()
5. **Resource Management Bugs**: Leaks, improper cleanup
6. **Concurrency Issues**: Race conditions, improper synchronization
7. **Integer Overflow Bugs**: Arithmetic overflow conditions
8. **Array/String Index Bugs**: Bounds checking issues

3. Failure-Aware Monte Carlo Tree Search (FA-MCTS)

3.1 Algorithm Overview

The FA-MCTS algorithm enhances traditional MCTS with failure-specific components:

```
class FA_MCTS(EnhancedMCTSTestGenerator):
    def __init__(self, ..., f_weight=2.0, bugs_threshold=1000):
        # Failure-aware parameters
        self.f_weight = f_weight  # Weight for failure-related rewards
        self.bugs_threshold = bugs_threshold
        self.verified_bug_methods = []
```

3.2 Enhanced Node Structure

FA-MCTS nodes maintain failure-specific metrics:

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

        # Failure-specific rewards
        self.logic_bug_rewards = 0.0
        self.failure_coverage_rewards = 0.0
        self.high_risk_pattern_rewards = 0.0

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

3.3 Reward Function Enhancement

The reward function incorporates multiple failure-aware components:

```
Total_Reward = Base_Coverage_Reward +
               f_weight × (Bug_Reward +
                           Pattern_Coverage_Reward +
                           Risk_Level_Reward)
```

Reward Components:

- **Base Coverage Reward**: Traditional line/branch coverage
- **Logic Bug Reward**: Points for detecting verified bugs
- **Pattern Coverage Reward**: Points for covering high-risk patterns
- **Risk Level Reward**: Weighted by pattern risk assessment

3.4 Strategy Selection System

The TestStrategySelector dynamically chooses testing strategies:

```python
class TestStrategySelector:
    def select_strategy(self, current_state, available_actions):
        # Multi-factor strategy selection
        # Considers: bug patterns, code complexity, previous success
        return selected_strategy
```

Strategy Categories:

1. **Boundary Value Testing**: Targets boundary conditions
2. **Exception Path Testing**: Forces exception scenarios
3. **Logic Branch Testing**: Explores complex logical conditions

4. **Data Flow Testing**: Tests data transformation paths
5. **State Mutation Testing**: Tests state consistency

4. Implementation Details

4.1 Multi-Provider LLM Integration

FailMapper supports multiple LLM providers through a unified interface:

```
# Provider selection based on availability and configuration
api_response = call_anthropic_api(prompt)  # Primary
api_response = call_gpt_api(prompt)        # Alternative
api_response = call_deepseek_api(prompt)   # Alternative
```

4.2 Test Execution and Validation

The framework integrates with Java build tools for test execution:

```
def run_tests_with_jacoco(project_dir, test_file_path, project_type):
    # Supports both Maven and Gradle
    # Executes tests with JaCoCo coverage
    # Returns coverage metrics and compilation errors
```

4.3 Bug Verification Pipeline

The BugVerifier class provides LLM-powered bug analysis:

```
class BugVerifier:
    def verify_potential_bug(self, source_code, test_code, failure_info):
        # Uses LLM to analyze test failures
        # Classifies bugs by type and severity
        # Returns structured bug report
```

5. Configuration and Extensibility

5.1 Configuration Parameters

Key configuration parameters for fine-tuning:

- `max_iterations`: Maximum MCTS iterations (default: 27)
- `f_weight`: Failure reward weight (default: 2.0)
- `bugs_threshold`: Bug discovery threshold (default: 1000)
- `target_coverage`: Coverage target (default: 100.0%)
- `verify_mode`: Bug verification timing (immediate/batch/none)

5.2 Extensibility Points

The framework provides several extension mechanisms:

1. **Custom Bug Patterns**: Add new detectors to FS_Detector
2. **Custom Strategies**: Implement new LogicTestStrategy subclasses
3. **Custom Reward Functions**: Modify FA_MCTSNode reward calculation
4. **Custom LLM Providers**: Add new API integration functions

6. Performance Characteristics

6.1 Scalability Analysis

- **File Processing**: Handles large codebases with graceful degradation
- **Memory Usage**: Bounded by MCTS tree size and analysis caching
- **Time Complexity**: O(iterations × actions × LLM_latency)

6.2 Optimization Strategies

1. **Parsing Fallbacks**: Multi-layer parsing reduces failure rates
2. **Strategy Caching**: Previously successful strategies are prioritized
3. **Incremental Analysis**: Reuses analysis results across runs
4. **Parallel Processing**: Supports batch processing of multiple classes

7. Empirical Evaluation Framework

7.1 Metrics Collection

The framework tracks comprehensive metrics:

```
self.metrics = {
    "classes_processed": 0,
    "bugs_found": 0,
    "avg_coverage": 0.0,
    "execution_time": 0,
    "bug_patterns": defaultdict(int)
}
```

7.2 Output Formats

Generated outputs include:

- **JUnit Test Files**: Executable test cases
- **Bug Reports**: Structured defect analysis
- **Coverage Reports**: JaCoCo-generated coverage metrics
- **Analysis Summaries**: JSON-formatted results

8. Limitations and Future Work

8.1 Current Limitations

- **Language Support**: Currently Java-only
- **LLM Dependency**: Requires API access for optimal functionality
- **Build Tool Support**: Maven fully supported, Gradle experimental

8.2 Future Research Directions

1. **Multi-Language Support**: Extension to C++, Python, JavaScript
2. **Advanced Bug Classification**: ML-based bug type prediction
3. **Integration Testing**: Support for multi-class test scenarios
4. **Performance Optimization**: Reduce LLM API dependencies

9. Conclusion

FailMapper represents a significant advancement in automated test generation by incorporating failure-aware heuristics into the MCTS framework. The combination of comprehensive static analysis, intelligent pattern detection, and enhanced search algorithms enables the discovery of logical bugs that traditional coverage-based tools miss. The modular architecture and extensive configuration options make it adaptable to various Java project types and testing requirements.

The framework demonstrates the potential of AI-assisted testing tools to move beyond simple coverage metrics toward more intelligent, bug-focused test generation strategies. As software systems continue to grow in complexity, such failure-aware approaches will become increasingly important for ensuring software reliability and quality.

This technical documentation is based on the FailMapper framework implementation and the associated academic research paper "FailMapper: Automated Generation of Unit Tests Guided by Failure Scenarios".