FailMapper: Failure-Aware Monte Carlo Bug Detection Architecture

Installation

Prerequisites

- Python 3.7 or higher
- Java 8 or higher (for running tests)
- Maven or Gradle (for Java project build)
- Junit5
- Jacoco 0.8.8

Install Dependencies

Option 1: Full Installation (Recommended)

```
pip install -r requirements.txt
```

Option 2: Minimal Installation

```
pip install -r requirements-minimal.txt
```

Option 3: Manual Installation

```
pip install javalang numpy pandas requests anthropic typing-extensions
```

Overview

FailMapper is an advanced automated unit test generation framework designed specifically for Java projects. It combines static code analysis, failure model extraction, and an enhanced Monte Carlo Tree Search (MCTS) algorithm to intelligently generate unit tests that are optimized for bug detection, particularly logical bugs.

Unlike traditional test generation tools that focus primarily on code coverage, FailMapper takes a failure-aware approach that specifically targets potential bugs and edge cases through intelligent analysis of code structure, data flow, and logical patterns.

Key Features

Failure-Aware Test Generation

- Intelligent bug detection targeting logical errors, boundary conditions, and edge cases
- Enhanced MCTS algorithm with failure-aware reward functions
- Multi-dimensional test generation strategies

Comprehensive Static Analysis

- Multi-layer Java code parsing with fallback mechanisms
- Data flow graph construction and analysis
- Dependency analysis (both direct and indirect)
- Boundary condition and exception handling pattern detection

Anti-Mock Philosophy

- Enforces testing with real objects instead of mocking frameworks
- Ensures authentic integration testing
- Reveals real-world bugs that mocks might hide

AI-Powered Bug Verification

- Integrates with multiple LLM providers (Claude, GPT, DeepSeek, etc.)
- Intelligent bug verification and classification
- Automated defect reporting with severity analysis

Build Tool Support

- Maven and Gradle project support
- Automatic project type detection
- JaCoCo integration for coverage tracking

Architecture

FailMapper follows a three-stage pipeline architecture:

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

Stage 1: Static Analysis Engine

- Extracts project structure, dependencies, and data flow information
- Builds comprehensive analysis models for informed test generation

Stage 2: Intelligent Prompt Generation

- Converts static analysis results into structured prompts
- Resolves dependencies and injects relevant API information
- Creates comprehensive testing context

Stage 3: Failure-Aware MCTS Test Generation

- Uses enhanced MCTS with failure-aware capabilities
- Employs intelligent strategy selection based on code patterns
- Provides real-time bug verification and classification

Installation

Prerequisites

- Python 3.8 or higher
- Java 8 or higher (for target projects)
- Maven or Gradle (depending on your project)

Setup

1. Clone the repository:

```
git clone <repository-url>
cd FailMapper
```

2. Install Python dependencies:

```
pip install -r requirements.txt
```

3. Configure your LLM provider (set API keys as environment variables):

```
export ANTHROPIC_API_KEY="your-claude-api-key"
# or other supported providers
```

Usage

Quick Start

FailMapper provides a simple entry point script (`run.py`) that orchestrates the entire pipeline:

```
python run.py <project_path> --output_dir <output_directory> --class_name
<ClassName> --package <package.name>
```

Parameters

- `project_path`: Path to your Java project root directory
- `--output_dir`: Directory where analysis results and test cases will be saved
- `--class_name`: Name of the specific class to test
- `--package`: Full package name of the class

- `--project_type`: Project type (`maven` or `gradle`). Auto-detected if not specified

Example

```
python run.py /path/to/my-java-project --output_dir ./results --class_name
Calculator --package com.example.math
```

Advanced Usage

For more control over the testing process, you can use the `failmapper.py` directly:

```
python static_analysis.py /path/to/{project_name} --output_dir {output}
python prompt_generator.py json_path
/path/to/{output}/{project_name}/{project_name}_combined_analysis.json --
output_dir {prompt_output}
python failmapper.py --project /path/to/{project_name} --prompt
/path/to/{prompt_output} --class Calculator --package com.example.math
```

Advanced Parameters

- `--max-iterations`: Maximum MCTS iterations (default: 27)
- `--target-coverage`: Target coverage percentage (default: 100.0)
- `--f-weight`: Weight for failure-related rewards (default: 2.0)
- `--bugs-threshold`: Number of potential bugs to find before terminating search (default: 1000)
- `--batch`: Process all classes in the project
- `--verbose`: Enable verbose logging

Batch Processing

To test all classes in a project:

```
python failmapper.py --project /path/to/project --prompt /path/to/prompts
--batch --output results.json
```

Output

FailMapper generates several types of output:

1. **Static Analysis Results**: JSON files containing project structure, dependencies, and data flow information
2. **Generated Test Cases**: JUnit test files targeting specific bugs and edge cases
3. **Bug Reports**: Detailed reports of discovered bugs with severity classification
4. **Coverage Reports**: Code coverage metrics and analysis
5. **Execution Metrics**: LLM usage statistics and performance metrics

Configuration

API Key Configuration

FailMapper supports multiple AI providers for test generation. You need to configure at least one API key to use the system.

Supported Providers

- **ANTHROPIC** (Claude) - Primary provider, required
- **OpenAI** (GPT) - Optional, can be used as alternative
- **DeepSeek** - Optional, can be used as alternative

NOTE: We provide OpenAI and DeepSeek usage,currently, you need to uncomment it in code. We will provide a switch for user friendly.

```
# run.py
# OpenAI API key (optional)
if args.openai_api_key or os.environ.get('OPENAI_API_KEY'):
    openai_key = get_api_key(args, "OpenAI", "OPENAI_API_KEY",
"openai_api_key")
    os.environ['OPENAI_API_KEY'] = openai_key
    print("✓ OpenAI API key configured successfully")

# DeepSeek API key (optional)
if args.deepseek_api_key or os.environ.get('DEEPSEEK_API_KEY'):
    deepseek_key = get_api_key(args, "DeepSeek", "DEEPSEEK_API_KEY",
"deepseek_api_key")
    os.environ['DEEPSEEK_API_KEY'] = deepseek_key
    print("✓ DeepSeek API key configured successfully")


parser.add_argument('--openai_api_key', help='OpenAI API key (can also be
set via OPENAI_API_KEY environment variable)')
parser.add_argument('--deepseek_api_key', help='DeepSeek API key (can also
be set via DEEPSEEK_API_KEY environment variable)')
```

```
api_response = call_gpt_api(prompt)
api_response = call_deepseek_api(prompt)
```

Configuration Methods

1. Command Line Arguments

```
python run.py /path/to/project --output_dir ./output --class_name MyClass
--package com.example --anthropic_api_key YOUR_API_KEY
```

2. Environment Variables

```
export ANTHROPIC_API_KEY="your_anthropic_api_key_here"

python run.py /path/to/project --output_dir ./output --class_name MyClass
--package com.example
```

3. Interactive Input

If no API key is provided via arguments or environment variables, the system will prompt you to enter them:

```
python run.py /path/to/project --output_dir ./output --class_name MyClass
--package com.example
# You will be prompted: "Please enter your ANTHROPIC API key:"
```

Usage Examples

1. **Basic usage with ANTHROPIC key:**

```
python run.py /path/to/project --output_dir ./output --class_name
Calculator --package com.example.math --anthropic_api_key sk-ant-api03-...
```

2. **Using environment variables:**

```
export ANTHROPIC_API_KEY="sk-ant-api03-..."
python run.py /path/to/project --output_dir ./output --class_name
Calculator --package com.example.math
```

Project Type Detection

FailMapper automatically detects whether your project uses Maven or Gradle by looking for:

- `pom.xml` (Maven)
- `build.gradle` or `build.gradle.kts` (Gradle)

You can override the detection by specifying `--project-type`, recommend for testing Maven projects, Gradle adaption is under test.

Key Components

- **Static Analysis Engine**: Multi-layer Java parsing and analysis
- **Failure Scenario Detector**: Identifies potential bug patterns
- **Enhanced MCTS**: Failure-aware Monte Carlo Tree Search

- **Bug Verifier**: LLM-powered bug verification and classification
- **Test Strategy Selector**: Intelligent test generation strategy selection

Technical Details

Supported Java Features

- Java 8-21 language features
- Generic types and lambda expressions
- Complex inheritance hierarchies
- Exception handling patterns

Bug Detection Categories

- Boundary condition errors (off-by-one, missing checks)
- Conditional logic errors (wrong operators, precedence issues)
- State management errors (concurrent access, inconsistent state)
- Arithmetic operation errors (overflow, divide-by-zero)
- Data processing errors (null pointer, type casting)

Limitations

- Currently supports Java projects only
- Requires LLM API access for optimal functionality

Contributing

We welcome contributions! Please see our contributing guidelines for details on how to submit improvements and bug fixes.

Support

For questions, issues, or feature requests, please create an issue or contact the development team.