Intelligent Software Engineering

Software Testing

Zhilei Ren

Dalian University of Technology

August 15, 2025

bug or feature?

Fuzz Testing

In programming and software development, fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks[1].

[1] https://en.wikipedia.org/wiki/Fuzzing

Differential Testing

Differential testing, also known as differential fuzzing, is a software testing technique that detect bugs, by providing the same input to a series of similar applications (or to different implementations of the same application), and observing differences in their execution. Differential testing complements traditional software testing because it is well-suited to find semantic or logic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Differential testing is also called back-to-back testing[2].

[2] https://en.wikipedia.org/wiki/Differential_testing

Metamorphic Testing

Metamorphic testing (MT) is a property-based software testing technique, which can be an effective approach for addressing the test oracle problem and test case generation problem. The test oracle problem is the difficulty of determining the expected outcomes of selected test cases or to determine whether the actual outputs agree with the expected outcomes[3].

[3] https://en.wikipedia.org/wiki/Metamorphic_testing

Metamorphic Testing

Metamorphic relations (MRs) are necessary properties of the intended functionality of the software, and must involve multiple executions of the software. Consider, for example, a program that implements sin x correct to 100 significant figures; a metamorphic relation for sine functions is $sin(\pi - x) = sin(x)$. Thus, even though the expected value of $sinx_1$ for the source test case $x_1 = 1.234$ correct to the required accuracy is not known, a follow-up test case $x_2 = \pi - 1.234$ can be constructed. We can verify whether the actual outputs produced by the program under test from the source test case and the follow-up test case are consistent with the MR in question. Any inconsistency (after taking rounding errors into consideration) indicates a failure:31 of the program, caused by a fault:31 in the implementation.

Differential Testing

Differential testing, also known as differential fuzzing, is a software testing technique that detect bugs, by providing the same input to a series of similar applications (or to different implementations of the same application), and observing differences in their execution. Differential testing complements traditional software testing because it is well-suited to find semantic or logic bugs that do not exhibit explicit erroneous behaviors like crashes or assertion failures. Differential testing is also called back-to-back testing[4].

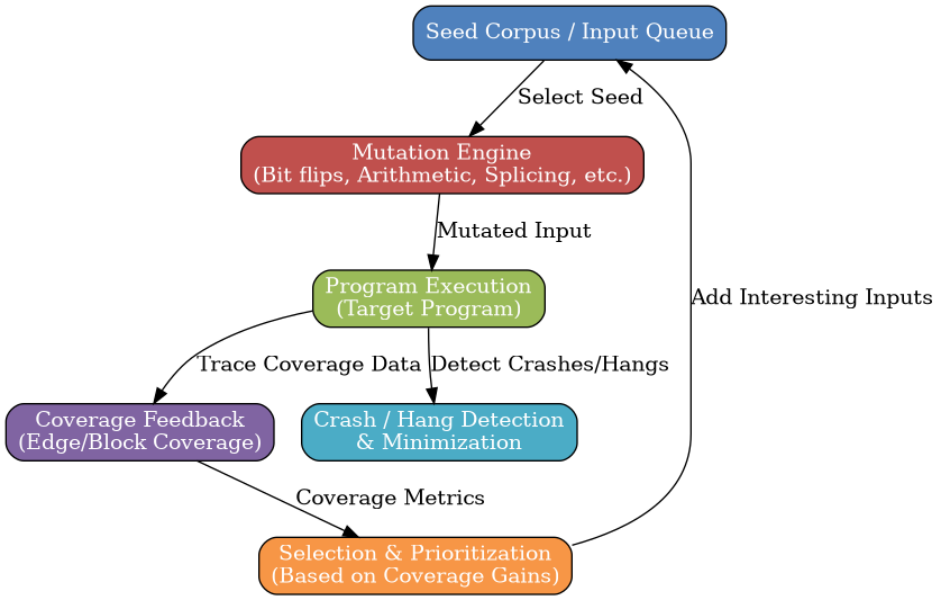
[4] https://en.wikipedia.org/wiki/Differential_testing

EvoSuite

EvoSuite is a tool that automatically generates unit tests for Java software. EvoSuite uses an evolutionary algorithm to generate JUnit tests. EvoSuite can be run from the command line, and it also has plugins to integrate it in Maven, IntelliJ and Eclipse. EvoSuite has been used on more than a hundred open-source software and several industrial systems, finding thousands of potential bugs[5].

[5] https://en.wikipedia.org/wiki/EvoSuite

Evolutionary Fuzz Testing



Main Modules Description

- **Seed Corpus / Input Queue (Blue)**
  - Holds the initial set of seed inputs and manages the queue of inputs waiting to be processed.
- **Mutation Engine (Red)**
  - Generates new test inputs by applying mutation strategies (e.g., bit flips, arithmetic operations, splicing).
- **Program Execution (Green)**
  - Executes the target program using mutated inputs, collects trace coverage data, and monitors for crashes or hangs.
- **Crash / Hang Detection & Minimization (Cyan)**
  - Detects crashes or hangs during execution and minimizes the problematic inputs for debugging.
- **Coverage Feedback (Purple)**
  - Collects data on which parts of the program were exercised by the inputs to guide the fuzzing process.
- **Selection & Prioritization (Orange)**
  - Selects and prioritizes inputs based on coverage metrics to discover new paths or trigger crashes.

Overall Flow

1. **Initialization**
   - Start with a seed corpus or input queue containing initial test inputs.
2. **Mutation**
   - Select seeds from the input queue and apply mutation strategies to generate new inputs.
3. **Execution**
   - Execute the target program with mutated inputs, collect coverage data, and detect crashes or hangs.
4. **Detection and Minimization**
   - Record detected crashes or hangs and minimize the inputs for debugging.
5. **Feedback and Prioritization**
   - Use coverage feedback to evaluate inputs, select and prioritize those that improve coverage.
6. **Iteration**
   - Iterate the process, refining inputs based on coverage feedback to maximize code exploration and crash discovery.