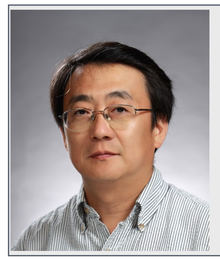
Program Analysis and Software Testing for System Dependability

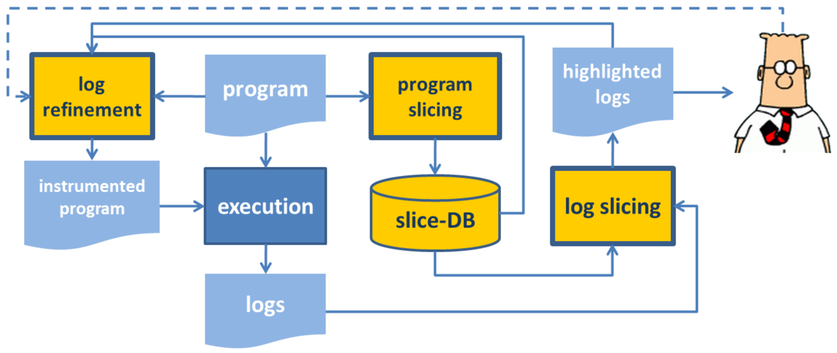
Abstract **Dependability** is usually the most important property of a critical software system. The dimensions of dependability embrace availability, reliability, safety, and security, thus effective and efficient verification approaches are required to guarantee software dependability properly. We intend to improve **how we code, test, and debug software systems**, to help programmers to achieve higher levels of dependability. Especially, we focus on **computer-aided programming, program analysis, regression testing, failure reproducing, and bug detection.**



Jianjun Zhao

Software Theory and Practice Group
Shanghai Jiao Tong University
(http://stap.sjtu.edu.cn)

AutoLog: Facing Log Redundancy and Insufficiency

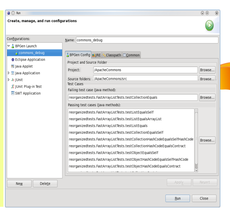


- Logging is the predominant practice when debugging:
  - Easy to add
  - (Usually) no side effects
  - A “program” over the program
- This freedom comes with a cost:
  - Log redundancy: too many irrelevant logs
  - Log insufficiency: critical logs may still be missing
- Log slicing to highlight relevant logs :
  - Identify relevant logs by analyzing program dependencies
- Log refinement to produce sufficient logs:
  - When existing logs are insufficient to cover the root cause
    - Log slicing can provide little help
  - Automatically insert new logging statements

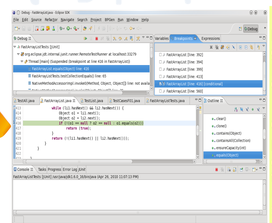
BPGen: An Automated Breakpoint Generator for Debugging

Select the failing and passing test cases.

Press the **RUN** button.



BPGen Process

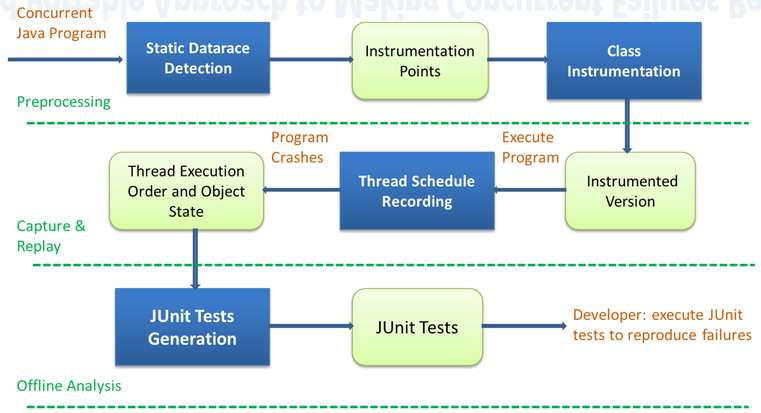


Debug with the generated breakpoints in the JDT debugger.

Optional function: Highlight the difference in execution traces and the dynamic program slice.

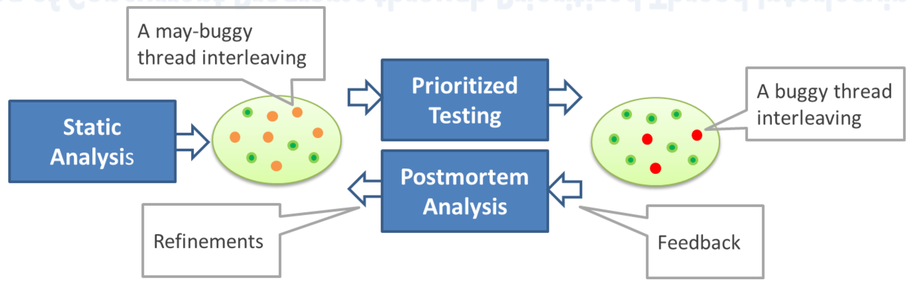
- Debugging and breakpoints:
  - Over 70% debugging developers use breakpoints
- Step 1. Nearest Neighbor Query:
  - Input: failing and passing test cases
  - Output: difference in execution traces
- Step 2. Dynamic Program Slicing:
  - Input: difference in execution traces
  - Output: dynamic program slice
- Step 3. Memory Graph Comparison:
  - Input: dynamic program slice
  - Output: unconditional breakpoints and difference in memory graphs
- Step 4. Breakpoint Condition Generation:
  - Input: difference in memory graphs
  - Output: breakpoint conditions

A Lightweight and Portable Approach to Making Concurrent Failures Reproducible



- Motivation:
  - Bug fixing in concurrent programs is even harder due to non-deterministic execution
- Contributions:
  - A bytecode instrumentation approach to reproducing concurrent failures
  - Lightweight and Portable implementation
- Steps:
  - Run record version program (seeded)
  - When program crashes, collect generated JUnit tests
  - Run JUnit tests on replay version, measure reproduction rate and overhead

Efficient Testing of Concurrent Programs through Prioritized Thread Interleavings



- Essential ideas:
  - Use static analysis to compute may-buggy thread interleavings soundly
  - Use static analysis to group may-buggy thread interleavings and prioritize the testing executions of thread interleavings within one group
  - Use postmortem analysis to detect concurrency bugs, propagate the feedback, re-group and re-prioritize the rest of thread interleavings
- Salient features:
  - Testing one group of may-buggy thread interleavings in a small number of testing executions
  - Testing different groups of the thread interleavings separately, thus different groups can be tested in parallel
  - Eliminating most false positives
  - No false negatives, giving high confidence in the correctness of the target program

NSFC

This material is based upon work supported in part by the National Science Foundation of China (NSFC) under Grants No.60673120 and No. 60970009)

SJTU.EDU.CN 軟件理論與實踐 Software Theory and Practice

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY