How to Not Fail SE 465

Winter 2017

Contents

1 Faults, errors, and failures (L02)

- **Validation** is ensuring that the code does the right thing.
- **Verification** is ensuring that the code conforms to the specs.
- **Faults** are static defects present in the software.
- An **error** is bad internal state caused by some fault.
- **Failures** are bad external behavior.

1.1 RIP fault model (L02)

- **Reachability**: The fault must be reachable.
- **Infection**: The program state must be wrong after reaching the fault.
- **Propagation**: The infected state must propagate to output.

1.2 Dealing with faults (L02)

- **Avoidance**: Use Rust, don't allow buffer overflows, better system design.
- **Detection**: Testing.
- **Tolerance**: Let system keep working even when there are faults.

2 Static/dynamic testing (L04)

2.1 Static testing

Find fault directly by analyzing the program. Examples: Type checking, dead code analysis, null check, bounds check, code review, formal verification.

2.2 Dynamic testing

Find failures by running the program with inputs and comparing them to expected outputs.

- Black-box: Don't look at code.
- White-box: Look at code.

2.3 Controlling/observing

- **Observability**: How easy it is to observer the system's behavior.
- **Controllability**: How easy it is to provide the system with inputs.

3 Test cases (L04)

- A **test case value** is an input.
- An **expected result** is the expected output.
- **Prefix values** are inputs to prepare software for the test cases.
- **Postfix values** are inputs to tidy up software after test cases.
  - Verification values: Show results of test cases.
  - Exit commands: Terminate or return to initial state.
- A **test case** is (test case values, expected results, prefix values, postfix values).
- A **test set** is a set of test cases.
- A **test requirement** is a property of the program that a test case can satisfy.
  - Examples:
    * This branch is followed
    * This method is called
  - Can be infeasible if there is dead code.
- A **criterion** generates test requirements. e.g.
  - All branches are followed
  - All methods are called
- The **coverage level** of a test set is the fraction of test requirements satisfied by it.

4 Exploratory testing (L05)

- It's good for realism, prioritization of important bugs, and evaluating risks.
- Process:
  - Start with a **charter**. e.g. "Explore the product elements"
  - Decide which area of the software to test.
  - Design a test.
  - Execute a test and log bugs.
  - Repeat.
- Outputs: Set of bug reports, test notes, artifacts (input/output pairs)

5 Control flow graph (L06-L08)

- **Control flow graph**: Graph of program execution.
  - Nodes are sequential statements
  - Edges mean "the program can follow this path during execution"
- **Basic blocks** can be combined into a single node.
- **Coverage criterion**: Function from CFG to set of test requirements.
  - e.g. "statement coverage" is "given G, generate all nodes in G"
- **Test path**: The path in the CFG that the program follows given a particular **test input**.

- Test set T **satisfies** coverage criterion C on graph G iff there exists some test path in path(T) that satisfies each test requirement generated by C.
- **Nondeterministic** programs or inputs could result in multiple test paths for a single input.
- **Statement coverage**: TR contains a requirement to visit n for each node $n \in reach_G(N_0)$.
- **Branch coverage**: TR contains each reachable path of length at most 1 in G.
- **Complete path coverage**: TR contains all paths in G. Impossible for graphs with loops.
- For real programs, 80% coverage is usually good enough.

6 Finite state machine (L09)

- Higher level graph to describe your program state.
  - Nodes are software states
  - Edges are transitions between them
  - Edges may be guarded by preconditions and post-conditions
- Test requirements and coverage are analogous to those for CFGs
- **Simple round trip coverage**: TR has at least one round-trip path for every reachable node in G in a round-trip path.
- **Complete round-trip coverage** TR has all round-trip paths for every reachable node in G.
- To obtain an FSM:
  - CFGs can be considered really bad FSMs
  - Look at the software structure or specs.
  - Model state using state variables and prune using domain knowledge.
  - iComment, Daikon

7 Syntax-based testing (L10)

- Grammars can be input-space (for test inputs) or grammar-space (for mutations).
- Grammars can be regexes or context free grammars.
- Obtaining invalid strings
  - Mutate the grammar by adding, removing, permuting terminals and nonterminals
  - Or just misderive rules the same way

7.1 Fuzzing (L11)

- Everything is bad and please write to your MPs.
- **Mutation-based**: Change existing test cases. Either randomly flip bits, or parse and mutate.
- **Generation-based**: Increasingly sophisticated generation. Example for a C compiler:
  1. Random bitstring
  2. Random ASCII chars
  3. Sequence of words, separators, and whitespace
  4. Syntactically correct programs
  5. Type-correct programs
  6. Statically conforming programs
  7. Dynamically conforming programs
  8. Model-conforming programs

8 Test coverage in reality (L13)

- Open source: 20-95% statement coverage
- Industry: 80% statement coverage
- JUnit: 85% statement coverage. 13000 lines of system code, 15000 lines of test code
  - 65% coverage on deprecated code
  - 93% coverage on non-deprecated code

8.1 Reasons for not testing

- Code is too simple (getters, setters, empty methods)
- Dead by design (code that should never actually be run)
- Hard to execute code (like OOM handlers)

9 Mutation testing (L14)

- Change program so it's wrong and see if tests fail.
- **Ground string**: A program. ("a string belonging to a programming language's grammar")
- **Mutation operator**: A way to change a program. ("specifies syntactic variations of a string")
- **Mutant**: A changed program. ("the result of applying a mutation operator to a ground string")
- A test case **kills** a mutant if the test case distinguishes between the mutant and the original.
- **Mutation score**: % of mutants killed given a fixed set of mutants
- **Mutation testing**: Keep adding tests until the mutation scores reaches some target
- Uninteresting mutants include those which are:
  - Stillborn: Can't compile or immediate crash
  - Trivial: Killed by almost any test case
  - Equivalent: Same as original program

9.1 Strong and weak (L15)

- Strong mutation: The fault must propagate to output (RIP)
- Weak mutation: The fault need only infect state (RI)
- Strong killing: The mutation is killed by a mismatch in output
- Weak killing: The mutation is killed by any internal error state

- **Strong mutation coverage** (SMC): All mutants are strongly killed by some test in TR.
- **Weak mutation coverage** (WMC): All mutants are weakly killed by some test in TR.

9.2 Mutation testing algorithm

- Create mutants
- Eliminate known-equivalent mutants
- While not enough mutants killed:
  - Generate test cases
  - Run test cases on program
  - Run test cases on mutants
  - Filter out bogus test cases (ones which kill no mutants)
- Is program output on test cases correct?
  - Yes: Wünderbar!
  - No: Fix program, start from beginning

9.3 Integration mutation testing

- Change param values in caller
- Change choice of callee
- Change callee inputs and outputs
- In Object Oriented:
  - Modify object of field accesses / method calls

9.4 Coverage vs. mutation testing

- Out of 300 real bugs:
  - 73% were found by mutation testing
  - 50% were found by branch coverage
  - 40% were found by statement coverage

10 Test suite engineering (L18)

- Many small tests are better than one big test.
- Name tests well.
- Make it easy to add new tests.
- Focus on unit tests, but eventually have integration tests.
- Write tests as you code, or even before.
- Don't make flaky tests.
- Avoid having tests look into internal state.

11 Selenium (L19)

- It automates browser tests
- IDE allows for record/replay
- WebDriver is an API for browsers
- **Page objects** are layers of abstraction between the test and the WebDriver.

12 Bug finding (L21)

- Look for contradictions and deviances.
- **MUST-beliefs**: Things you know for sure
- **MAY-beliefs**: Things that are merely likely
- Confidence = Support of both / Support of one (L22)

13 Self-checking tests (L25)

- Verifying:
  - State: Field accesses and value comparison.
  - Behaviour: Calls made.

13.1 Types of test objects (L26)

- Dummies: Don't do anything except exist
- Fakes: Real behaviour. Example: In-memory database.
- Stubs: Canned answers.
- Mocks: Canned answers, and checking for appropriate calls.
- Spies: Wraps the real object. Just records interactions.

14 Code review (L28)

14.1 Things to do

- Positive tone
- Explain why
- Don't be Dr. No
- Specific and actionable advice
  - Linked to places in the code
- Acknowledge effort, say things that are good
- Don't ask for unnecessary changes
- Provide shorter chunks for review
- Think beyond the superficial
- Make sure the code works
  - And has tests
  - That have been run
- Ask for clarifications on areas of confusion

14.2 Things to check

- Formatting and naming
- Don't repeat yourself
- Code should fail fast
- Avoid magic numbers
- One purpose per variable

15 Bug reports (L29)

15.1 Things to do

- Explain bug's consequences
- Explain steps to reproduce
  - Specific steps
  - Minimal test case
- Respond to follow-up
- Correct tone

16 Testing tools

- You can verify your code using
  - Manual testing
  - Automated testing suite, manually generated
  - Automatically-generated suites
  - Static analysis tools

16.1 FindBugs (L32)

- Static analysis of JVM bytecode
- Finds patterns, such as
  - off-by-one errors
  - null pointer derefs
  - ignored read() return values
  - some ignored return values
  - uninitialized reads in constructor

16.2 Facebook Infer (L33)

- Static analysis tool
- Also enforces generic program properties
- Open source and runs on industrial codebaces
- Works on C, Objective C, C++, Java
- Written in OCaml
- Infer Eradicate
  - References treated as non-null by default
  - @Nullable allows null
- Does inter-procedural analysis
- Leak detection
  - Memory leaks
  - Resource leaks (like files)
- Taint analysis
  - Data from untrusted source should not go to trusted sink
  - ... unless it passes through a sanitizer

16.3 Dynamic analysis tools (L35)

- Memory errors
  - Memory errors with Memcheck
  - Race conditions with Helgrind
- Memcheck
  - Illegal reads and writes
  - Reads of uninitialized memory
  - Bad frees: `free(n); free(n); // no`
  - Memory moves: `memcpy` must not overlap
  - Bad arg values: `malloc(-3)`
  - Memory leaks (threw away all pointers to allocated memory)
  - Performance penalty: 10x-50x
- Address Sanitizer
  - Alternative to memcheck
  - 2x slowdown
  - Use-after-free
  - Use-after-return/after-scope
  - Double-free, invalid free
  - Memory leaks
- Design decisions
  - Valgrind may return false positives
  - Address Sanitizer does NOT return false positives
    - terminates upon any error
- Implementation techniques
  - Valgrind: Virtual CPU and check relevant instructions
  - Address Sanitizer: Rewrite every memory access to check it, keep metadata about valid memory

17 Index of tools

- **Cobertura**: Measures coverage
- **iComment**, **Daikon**: Automatically generates FSMs (L09)
- **Fuzzinator**, **american fuzzy loop**: Fuzzers (L11)
- **Chaos Monkey**: Saboteur. Randomly takes servers down (L11)
- **PIT**: Java mutation testing.
- **Coverity**: Versatile bug finder.
- **PMD**: Static property checking using XPath. (L30-L31)
- **FindBugs**: Static analysis of Java bytecode. Harder rules. (L32)
- **Korat**: Generates all possible Java objects in a category, for use in testing.
- **FB Infer**: Facebook open-source Coverity, kinda. (L33)
  - Eradicate: Detect null pointers
  - Analyzer: Resource and memory leaks
- **ESC/Java2**: Reads JML specs and verifies them statically. (L34)
- **Valgrind**: Dynamic analysis. (L35)
  - Memcheck: Default tool. Memory errors.
  - Helgrind: Detects race conditions.
- **AddressSanitizer**: Like Valgrind Memcheck but faster.
- **Randoop**: Automatically generate unit tests

18 XPath examples

- `book[@style]`: All `<book>` elements with a `style` attribute, in the current context
- `book/@style`: All style attributes of `<book>` elements, in the current context
- `book[/bookstore/@specialty=@style]`: All `<book>` elements whose `style` is equal to the `specialty` of the root bookstore, in the current context
- `author[1]`: The first `<author>` in the current context
- `author[degree][award]`: All `<author>` elements with a `<degree>` and `<award>`
- `author[degree and award]`: Same
- `degree[@from != "Harvard"]`: Degrees not from Harvard
- `book[last()-1]`: Second last book
- `book[price > 35]/title`: Titles of `<book>` elements with a `<price>` element greater than 35