Regresion Testing and Writing Test Oracles

Idaho State University | Computer Science

Isaac Griffith

CS 4422 and CS 5599
Department of Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outcomes

At the end of Today's Lecture you will be able to:

- Understand the basics of regression testing
- Understand and describe the issues facing regression testing and potential mitigation approaches
- Understand the oracle problem and approaches to handling it

ROAR

Regression Testing for Evolving Software

Regression Testing

- **Regression Testing** - process of re-testing software that has been modified
  - Constitutes the vast majority of testing effort
  - Large systems = Large Regression Test (RT) sets
- Key Issue: Minor changes to one part of a system can cause problems in other distant parts of the system
  - RT aims to identify instances of this issue
- To work RT must be automate
  - Thus, it is typically incorporated into CI/CD

Idaho State University
Computer Science

Automating RT

Typically done using commercially available tools

- JUnit (Java)
- HTMLUnit (Web)
- Selenium (Web)
- SUnit (Smalltalk)
- PHPUnit (PHP)
- CUnit (C)
- TestNG (Java)
- NUnit (.NET)
- CPPUnit (C++)

Additionally:

- Capture/replay systems all for UI RT
- Version control systems allow for the versioning of tests along side the systems they test.

Operationalizing RT

- In addition to tolls, we need scaffolding to make RT a reality
- Scripting engines are used to:
  - Manage the process
  - Obtain inputs
  - Execute the system
  - Marshall output
  - Compare actual vs. expected
  - Generate test reports

Goldilocks Problem

- Test engineers face the Goldilocks Problem:
  - Which tests are to be included in an RT set?
- Including every test becomes too large to manage
  - Can't be run as often as system changes, but should be run every night
  - This disrupts the dev process since tests don't run in a reasonable amount of time
  - Becomes a computational resources/cost problem
- Set too small will not cover functionality sufficiently
  - Passes many faults onto the user
- How do we find the set that is just the right size?

RT Set Size

- Common Approach from Industry
  - For every reported fault (from users) add a test that will detect this fault.
  - Is this reasonable? Why? Why Not?
    - Users do not constantly see the same problems
    - Supports traceability
- Better Approach
  - Utilize coverage criteria

The Software Changed

If one or more RT fails then:

- Determine if the change to the software or the test is faulty

If no RT fails then:

- Review the RT set for the new version

Types of Change

- **Corrective** - a defect is corrected
- **Perfective** - some quality aspect of the software is improved without changing its behavior
- **Adaptive** - the software is changed to work in a different environment
- **Preventitive** - the software is changed to avoid future problems without changing its behavior

- **All types require Regression Testing!!**
  - And this means each change implies a review of the RT set!

Evolving the RT Set

- Adding a small number of tests to the set is simple
  - But over time, can grow to be unwieldy
- Removing tests also creates problems
  - Faults will occur where tests were removed
- Changing external system interfaces can cause an RT set to fail

Thus, we not only need automated generation and execution of tests but also automated maintenance of tests.

Selecting Tests

- When evolving a test set we can utilize the same criteria that was used to create the set.
- As the set grows the time to execute grows
  - We then need a method to identify an appropriate RT subset to execute
- This is known as the **Regression Test Subset Selection Problem**

Selection Techniques

Several Selection Techniques have been proposed

- Linear equations
- Symbolic execution
- Path analysis
- Data flow analysis
- System dependence graphs
- Program dependence graphs
- Modification analysis
- Firewall definition
- Cluster identification
- Slicing
- Graph walk
- Modified entity analysis

Selection Technique Criteria

- Selection techniques must be evaluated against several criteria
  - **Inclusive** - degree to which the technique is “modification-revealing”
    - Unsafe techniques have inclusiveness less than 100%
  - **Precise** - the extent to which the technique omits regression tests that are not modification-revealing
  - **Efficient** - the degree to which the technique determines the appropriate subset of the RT set that is less computationally intensive than simply executing the omitted tests
  - **General** - the degree to which the technique applies to a wide variety of practical situations

Writing Effective Test Oracles

Test Oracles

- **Test Oracle** - encoded expected results
- Test oracles are key to automated testing
- Test oracles, in order to exhibit effective revealability, must strike a balance between
  - Checking too much (unnecessary cost)
  - Checking too little (not revealing failures)

What Should be Checked?

- Test designers must decide correct behavior before each test is run
- In general this includes the output state of everything that is produced by the software under test
  - Outputs to the screen, files, databases, messages, and signals
- For unit testing we also include
  - explicit return statements
  - parameters modified during execution
  - non-local variables that are modified

Test Oracle Strategies

- **Test Oracle Strategy** - a rule or set of rules that specify which program states to check
- Test Oracle Strategy features:
  - **Precision** - refers to how much the output state should be checked.
    - More State Checked = Higher Precision
  - **Frequency** - refers to when and how often the output state should be checked

Guidelines

- **It is important to check some outputs**
  - **Null oracle strategy** - checking only whether the software produces a runtime exception or crashes
    - Only 25% - 56% of failures result in a crash, thus 44% - 75% of tests are wasted
- **It is important to check the right outputs**
  - **Poor assertions** check outputs unlikely to be affected by the test
  - **Excellent assertions** check outputs directly affect by tests

Guidelines

- **It is not necessary to check a lot of outputs**
  - low precision is okay
- **It is not necessary to check the output state multiple times**
  - low frequency is okay
  - Most failures are revealed if the final state is checked only once.

Remember This

- Each test should have a goal
  - Good testers pay attention to why the test exists
  - Good testers write assertions that check what the test is trying to check and nothing else
- In practice this means
  - **Unit Testing** - We check the method return values and returned parameter values
  - **System Testing** - We check the directly visible output

Determining Correct Values

- **Challenges:**
  - How do we know what the correct behavior or output should be?
  - What if we don't know the correct answer? (a.k.a **The Oracle Problem**)
- **Approaches:**
  - Specification-based direct verification of outputs
  - Redundant computations
  - Consistency checks
  - Metamorphic testing (discussed in upcoming lecture)

Direct Verification

- Human evaluation is effective but costly
- Automated evaluation via direct verification is then one of the best methods
- Direct verification requires a verification algorithm, but not simply another algorithm for the same problem
- Unfortunately writing effective verifiers tends to be hard
  - Requires a specification which is difficult to write
- Additionally, often impossible
- Rarely used in industry

Idaho State University
Computer Science

Redundant Computations

- Useful alternative to direct verification
- Uses an alternative algorithm or approach to the same problem and compares results
- Formal Definition:
  - Suppose that the program under test is labeled `P`, and `P(t)` is the output of `P` on test `t`. A specification `S` of `P` also specifies an output `S(t)`, and we usually demand that `S(t) = P(t)`.
  - Suppose that `S` is, itself, executable, thereby allowing us to automate the checking process. If `S` itself contains one or more faults, a common occurrence, `S(t)` may very well be incorrect.
  - If `P(t)` is incorrect in exactly the same way, the failure of `P` goes undetected. If `P` fails in some way that is different from `S` on some test `t`, then the discrepancy will be investigated with at least the possibility that the faults in both `S` and `P` will be discovered
- A potential problem is when `P` and `S` have faults that result in incorrect

Redundant Computations

- Testing one implementation against another is an effective and practical technique
- Often used in industry for Regression Testing
- Look for alternative algorithms/approaches for the same problem
  - But beware the **Common Failure Problem**
    - Where two independent approaches may both generate the same incorrect output because of separate flaws.

Consistency Checks

- Alternative to direct verification or redundant computations
- From RIPR model we know:
  - For an error to be detected the infection must propagate
  - This is the problem of external checks
- Consistency checks use internal checks and thus only require the RI
- Internal checks raise the probability of detecting faulty behavior
  - Requires using techniques such as Program by Contract to create relations to evaluate object state based on:
    - Program Invariants
    - Method Pre/Post Conditions

Are there any questions?