Course Introduction

17-654/17-765

Analysis of Software Artifacts

Jonathan Aldrich

Why is Building Quality Software Hard?

- Compare to other engineering disciplines
  - Often done; sometimes valid, sometimes not
- For other disciplines we do pretty well
  - Well-understood quality assurance techniques
  - Failures happen, but they are arguably rare
  - Engineers can measure and predict quality
- For software, we aren't doing well
  - How many cars get recalled for a patch once a month?
  - Failure is a daily or weekly occurrence
  - We have relatively poor techniques for measuring, predicting, and assuring quality

Quality in other Engineering Disciplines

- Traditional engineering disciplines
  - Electrical, mechanical, civil
  - Governed by mathematics of ***continuous*** systems
- Some quality strategies
  - Divide and conquer
    - Break a big problem into parts
      - Physical location: floor, room…
      - Conceptual system: frame, shell, wiring, plumbing…
    - Solve those parts separately
  - Overengineer
    - Build two so if one fails the other will work
    - Build twice as strong to allow for failure
  - Statistical analysis of quality
    - Relies on continuous domain
  - These work because the different parts of the system are independent
    - Never completely true, but true enough in practice

Software Quality

- Software Engineering
  - Built on ***discrete*** mathematics
- Old quality strategies fail!
  - Divide and conquer
    - Butterfly effect: small bugs mushroom into big problems
  - Overengineering
    - Build two, and both will fail simultaneously
  - Statistical quality analysis
    - Most software has few meaningful statistical properties
- Underlying problems: lack of module independence
  - Partly due to discrete nature of software
  - Partly because we don't know how to decompose very well
  - Partly because the software world is more complex

Assuring Software Quality with Analysis

- How then can we assure software?
- Fight fire with fire
  - Software is discrete
  - Unlike physical engineering disciplines, can prove properties of software!
    - Can eliminate possibility of failure, something we can't do in any other field
    - NB: the hardware may still fail, but traditional engineering can handle that well

Analysis is revolutionizing software quality today in market leaders

How Microsoft got Religion

- Original process: manual review
  - Too many paths to consider as system grew
- Next process: add massive testing
  - Tests take weeks to run
  - Inefficient detection of common patterns
    - Non-local, intermittent, uncommon path bugs
  - Was treading water in Longhorn release
- Current process: add static analysis
  - Weeks of global analysis
  - Local analysis on every check-in
  - Lightweight specifications
  - Huge impact
    - 7000+ bugs filed in June 2005
    - Check-in gate eliminates large classes of errors from main codebase
- Take-home
  - Different forms of analysis are complimentary
  - Supplement testing and reviews with static analysis

Problem-Specific Focus

- Impractical to prove total correctness
  - We'll discuss principles, but the practice doesn't scale
  - Even harder for machines than for humans
- Instead, static analysis focuses on particular problems
  - Amenable to mechanical proof
    - Simple reasoning that must be consistent across program
  - Difficult to assure through testing, inspection
    - Non-local – hard to see during inspection
    - Intermittent – unlikely to be caught by tests
- Examples
  - Memory and resource errors
    - Buffer overruns, null dereference, memory leaks
  - Race conditions
    - Interference among threads
  - Protocol errors
    - Getting ordering wrong
  - Exceptional conditions
    - Divide by zero, overflow, application exceptions

A Broad View of Analysis

- *The systematic examination of an artifact to determine its properties*
- Includes testing, reviews, model checking as well as static code analysis
- Properties
  - Functional: code correctness
  - Quality attributes: evolvability, security, reliability, performance

Analysis Success Stories

- *Static analysis, an important focus of this course, is revolutionizing software development today*
- Windows regression tests take weeks to run. Analysis helps Microsoft choose which tests to run before a critical release.
- Stanford researchers found hundreds of possible crashing bugs in Linux
  - Tool commercialized by Coverity
- Windows analysis tool group: June 2005
  - Filed 7000 bugs
  - Added 10,000 specifications to code
  - Analyze for security, pointer errors on every check-in
  - Tools available in Visual Studio 2005

A "Simple" Analysis: the Halting Problem

- Given a program P, will P halt?

Quick Undecidability Proof

- Theorem: There does not exist a program Q that can decide for all programs P, whether P terminates.
- Proof: By contradiction.
  - Assume there exists a program Q(x) that returns true if x terminates, false if it does not.
  - Consider the program “R = if Q(R) then loop.”
  - If R terminates, then Q returns true and R loops (does not terminate).
  - If R does not terminate, then Q returns false and R terminates.
  - Thus we have a contradiction, and termination must be undecidable

Analysis isn’t Perfect

- Impossible to decide almost *any* program property without solving the halting problem
- Example: divide-by-zero bug finder
  - Is there a bug in this program?
  - Assume f() is defined elsewhere, but does not affect x

```
int x = 0;
f();
int y = 10/x;
```

Analysis as an Approximation

- Analysis must approximate in practice
  - May report errors where there are really none
    - False positives
  - May not report errors that really exist
    - False negatives
  - All analysis tools have either false negatives or false positives
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A *sound* tool has no false negatives
    - Never misses an error in a category that it checks
- The halting problem affects human analysis, too
  - Otherwise, we could solve the halting problem by building a computer big enough to simulate the human brain
  - So human analysis has to approximate as well

Analysis Tradeoffs

- Point in lifecycle
  - Finding errors early is cheap
  - Many analysis techniques require code
- Automated vs. manual
  - Automated: cheap, exhausive, can provide guarantees
  - Manual: can check a richer array of properties
- Incremental vs. global
  - Incremental analysis scales better, is more precise
  - Often requires programmer annotations
  - Important criterion: immediate benefit for annotation effort
- Soundness vs. completeness
  - Soundness: finds all errors of a particular class
    - Safe: no false negatives
    - Goal: provide assurance
  - Completeness: accepts all valid programs
    - Precise: no false positives
    - Goal: find bugs

Course Goals

- Understanding
  - Where different analyses are appropriate
  - Tradeoffs between analysis techniques
  - Theory sufficient to evaluate new analyses
- Experience
  - Writing simple analyses
  - Applying analysis to software artifacts

Course Outline

- Theory
  - Semantics and representations of code
  - Reasoning about correctness
  - Abstraction and soundness
- Automated Analysis
  - Dataflow analysis and tools
  - Bug-finding, concurrency assurance, protocol checking
  - Model Checking designs and code
- Analysis across the Software Lifecycle
  - Defect prediction
  - Code reviews
  - Testing techniques and coverage metrics
  - Regression testing, test prioritization and generation
  - Reengineering
- Quality Attributes
  - Security, Performance, Reliability

Homeworks and Projects

- Prove small programs correct with Hoare logic
- Check program correctness with ESC/Java
- Check protocols and concurrent code using a model checker
- Design a dataflow analysis, prove it sound, and implement in in an analysis framework
- Participate in a code review
- Analyze the throughput and reliability of a software system
- Read and discuss 10-15 key papers from the analysis literature
- Run a commercial or research analysis tool on source code and report on the experience

Evaluation

- Assignments (~60%)
  - Basic understanding of analysis techniques
  - Engineering tradeoffs
- Final Project (~15%)
  - Evaluate analysis tools on studio or other project
    - Written reports and in-class presentations
  - Write and apply custom analyses
- Final exam (~15%)
  - Theory and engineering
- Class participation and readings (~10%)
  - Reading questions on papers from the literature
  - Discussion, presentations

Policies

- Time Management
  - Keep track of time spent on each assignment
- Late Work
  - 5 free late days; use whenever you like
  - No other late work except under extraordinary circumstances
- Collaboration Policy
  - You may discuss the lectures and assignments with others, and help each other with technical problems
  - Your work must be your own. You may not look at other solutions before doing your own. If you discuss an assignment with others, throw away your notes and work from the beginning yourself.
  - You must cite sources if you use or paraphrase any material
  - If you have any questions, ask the instructor or TAs

Introductions

- Instructor
  - Jonathan Aldrich
    aldrich+ at cs.cmu.edu
- TAs
  - Thomas LaToza
    latoza at gmail.com
  - Gabriel Zenarosa
    gzen+ at cs.cmu.edu

Ph.D. Projects

- Possible topics
  - Literature survey
    - Study techniques, put into framework, identify open problems
  - Comparative evaluation
    - Your experience with multiple techniques or tools
  - Development of a new analysis technique
  - Application of an analysis technique to a new problem domain
- Requirements
  - Written report
    - Length depends on nature of project
  - Research emphasis
  - Class presentation
- Details to be arranged with instructor