NISS

Performance-Oriented Regression Testing of Fielded Software

Alan Karr

March 3, 2005

Antecedent: ITR Project on Lightweight Instrumentation

- Driving question: Can useful data be collected about fielded software?
- Why do it?
  - Learn about users, usage, performance
  - Richer data—cannot be duplicated in a lab
- Why is it challenging?
  - Unobserved heterogeneity
  - Privacy
  - Performance implications of heavy instrumentation

Count Data

- Full instrumentation consists of associating a counter with each
  - Procedure
  - Statement
  - …
- Save counts
  - Every *n* seconds
  - At end of execution
- There may also be covariate information

Fundamental Questions

- Is there as much information in some (small) subset of the counts (LWI) as in the full set of counts (HWI)?
- If so,
  - Is the effect real?
  - Is there associated science?
- Can LWI be designed without HWI?
  - Validation problem

Prototype Study: Protocol

- 23 versions of JABA, a Java byte code analyzer
  - 22 buggy versions, some with multiple errors
  - 1 gold standard version
- 707 test cases targeting different APIs
- Run each test case on each version
- Fully instrumented
  - Statements
  - Methods (caller x callee)
  - Throws and catches (exception handling)

The Data

- For each version and test case,
  - ~12K statement counts
  - ~1200 call counts (callee)
  - ~100 throw/catch counts
  - Binary success/failure response
    - Success = got same answer as gold standard
    - Failure = crashed or ran and got wrong answer
- Percent of failing test cases ranged from 0 to 52%

The Results

- GOOD
  - Random forests identified small (2-7) sets of good statement counts
    - Call counts added nothing
    - Throw/catch counts were generally useless
  - Multiplicity was not an issue
- BAD
  - For some versions, the same was true for random forests applied to randomly selected 5% of the statements
  - Good predictors for single-error versions did not work well for multiple-error versions containing the same error
- UGLY
  - Only limited scientific verification

The Skoll DCQA Process: Around the World, Around the Clock Distributed Testing

- Set of clients available for testing
  - Each has characteristics such as OS, …
- Server that responds to client requests: sends
  - Instance of software with certain configuration
  - Testing task(s)
- Client performs tests and returns results to server
- Reference: www.cs.umd.edu/projects/skoll

Applications of Skoll

- Settings
  - Regression testing
  - Testing of new functionality
- Typical goal
  - Diagnose faults, often by identifying problem configurations
- Challenges
  - High-dimensional configuration space
  - Constraints on configurations, possibly dependent on client characteristics
  - Limited data

The Performance Setting

- Software system with fielded instances *i* characterized by
  - Configuration
  - Observable characteristics (example: OS)
  - Unobservable characteristics (example: CPU speed, memory, other processes)
- Performance measure *P*
  - Assume higher is better

Example Scenarios

- Performance-oriented regression testing
  - Does new functionality interfere with performance?
- Optimization
  - Given observables, what configuration optimizes performance?
    - Averaged over unobservables
    - Worst case unobservables
- Identifying underperforming configurations and observables
  - For what (C,O) pairs is performance unacceptable?
    - Averaged over unobservables
    - Worst case unobservables

A Mathematical Formulation

- Configuration $C_i \in C =$ configuration space (which is big, involves constraints, ...)
- Observables $O_i \in O =$ space of observables (example: OS)
- Unobservables $\theta_i \in \Theta =$ generic space of unobservables (example: baseline speed or load, memory)

C, O, Θ are all high-dimensional

Basic Statistical Model

$$P_i = f(C_i, O_i, \theta_i) + \varepsilon_i,$$

where

- $\theta_i \in \Theta =$ unobservables for instance i, treated as a random variable
- $\varepsilon_i =$ measurement error

Approach

- Space-filling design in C
- Bayesian techniques for random effects models

Challenges

- Selection of performance measures
- Calibration
- Model validation
- Adaptive versions

Scenario 2

Average case performance: Solve

$$C^*(O_0) = \arg\max_C \int f(C, O_0, \theta) p(\theta) d\theta$$

Worst case performance: Solve

$$C^*(O_0) = \arg\max_C \min_\theta f(C, O_0, \theta).$$

Then substitute $\hat{f}$ for f.

Scenario 3

Average performance:

$$B^* = \left\{(C, O) : \int f(C, O, \theta) p(\theta) d\theta \leq \alpha\right\}$$

Worst case performance:

$$B^* = \left\{(C, O) : \min_{\theta} f(C, O, \theta) \leq \alpha\right\}$$

Then substitute $\hat{f}$ for f.

Collaborator

- Adam Porter, CS, UMaryland

Reference

- A. F. Karr and A. A. Porter (2005). Distributed performance testing using statistical modeling. Submitted to ICSE 2005 Workshop on Advances in Model-Based Software Testing (A-MOST).