High Volume Automated Testing with Yeager

Casey Doran

Florida Institute of Technology

cdoran2011@my.fit.edu

January 4, 2018

Overview

High Volume Automated Testing

- Anatomy
- History
- Family Tree
- The Case for Yeager

Long Sequence Testing in Yeager

- Software as a State Machine
- Usage
- Yeager In Action

Acknowedgements

This work would not be possible without the support of:

- Cem Kaner, CSTER, and WTST participants
- Curtis Chambers, Jeff Farr, Mike DeCabia at Dycom Industries
- the Ruckus, the Harbor City Hooligans, the Samuels family
- Rob Atilho and Ryan Bomalaski, and many more on campus
- kbg, Richard Ford, actual and adopted family

Relevant URLs

- `github.com/elementc/yeager`
  - The library in question.
- `github.com/elementc/thesis`
  - The thesis and all related materials, including these slides (`conference-presentation.{tex, pdf}`)
  - See (`presentation.{tex, pdf}`) and (`main.{tex, pdf}`) for more detail, including a primer on modern web testing.

This Work Emerges From Academia

A disclaimer: I have never worked in “the real world” on advanced testing. This work presents a tool I think is useful in some testing contexts, and a framing for already-existing techniques towards the academic body of knowledge. A key theme of my thesis is that many technqiues in the family described are lost to history because nobody in industry wrote it down. Omissions are omissions of documentation, not judgement- I welcome relevant interruptions and anecdotes, and hope the conversation continues outside of these walls.

What Is High Volume Test Automation (HiVAT)?

Tests that algorithmically generate, execute, and evaluate the results of arbitrarily many test actions on a system, in such volume as to:[Kaner, 2013]

1. Exceed the volume a reasonable testing staff could do manually.
2. Expose behaviors of the system not normally exposed during traditional testing techniques.
3. Simulate use and abuse of the system more realistically and dynamically than would be attainable through traditional techniques.
4. Generate test scenarios that are not outside the realm of possibility or even probability due to the high-availability nature of modern software systems.

Generators

- How test cases are generated
- How the system is driven
- An engineering consideration

Interface

- Black box or white box
- Shades of grey, maybe hitting a private REST service instead of the UI directly
- A consideration of engineering and testing goals

[Hoffman, 2013]

Oracle

- How to programmatically determine correctness of generated tests
- Comparison of some sort
  - To assertions in previously written code
  - To expectations from a formal Finite State Machine
  - To a previous version of the system
  - To a competitor's system
  - To systemic expectations, like not crashing
  - Room for research here
- A consideration of engineering and testing goals

Loggers and Diagnostics

- Keeping track of test trace
- Keeping track of system health during test
- Possibly characterizing system degradation
- A consideration of testing goals

Context

- Testing objectives regardless of engineering
  - Surveying the system for new bugs
  - Determining system resillience through abuse
  - Cornering hard-to-replicate bugs in suspect modules
  - Characterizing system resource consuption over time

Scalability

- How volume in these tests is generated
  - A single, long-running thread
  - A cluster of many threads
  - A swarm of many cheap cloud servers [Parveen and Tilley, 2010]
  - A virtualization service testing a breadth of configurations
- A consideration of the testing context and engineering constraints

Purported Inventors

- HP's "evil"
  - Oldest in my literature review from 1966
- TI
- Bell
- AT&T
- Microsoft
- Telenova
- Rohm
- FAA contractors
- Automotive industry
- Miller et al. [1989] with the Fuzz Tester
  - First from academia, 1989 technical report and 1990 article.

Industrial Inventors Are Reticent To Publish

- HiVAT is perceived as a competititve advantage
- Disclosing these practices would expose testers to risk of termination or legal retaliation
- Swept away as part of efforts to minimize maintenance-related tasks

LSRT: Long Sequence Regression Testing

- Accomplished by modifying exisiting test suites
- Set tests to run continuously
- Remove cleanup between test runs

State Model Testing

- Build a detailed Finite State machine
- Algorithmically exercise the machine to generate testable theorems about the system

[Lee and Yannakakis, 1996]

Exhaustive Testing

- Lower level
- Test every single possible parameter value to a function
- Needs another implementation for an oracle
- Gets prohibitively slow for multiple parameters
- Analysis, using slices for instance [Gallagher and Lyle, 1991], can prove parameter independence and eliminate the need to test combinations of parameters

A Tale Of Two Exhaustive Tests

Hoffman [2003]

- Suspected a trig function of bugs
- Used another implementation
- Fed both functions every number in the range of a 32 bit float
- Found two errors in a few minutes

Dawson [2014]

- Suspected a trig function of bugs
- Used another implementation
- Fed both functions every number in the range of a 32 bit float
- Found one error 826k times in about 90 seconds

Fuzz Testing

- Miller's tool generates streams of random bytes and feeds them as input to UNIX command line utilities. [Miller et al., 1990]
- A test fails if the program crashes.
- Fuzz testing has grown into a diverse family of subtechniques, popular among security researchers.

Load/Performance Testing

- API tests put into a massive thread pool
- The “accepted” way to verify many users won’t crash a system
- Popular tool in this family: Locust [Heyman et al., 2011]

Testing In Production

- A practice at Microsoft [Andrews, 2012]
- Candidate builds of Bing fed actual user input
- Output compared to current build
- Enables automated, staged deployments

A/B Testing

- Marketing practice
- Release candidate revisions to a subset of users and monitor for desireable behavior
- Promote the most effective revision to general availability
- Email marketing, site homepages, search engine ads, news stories

[Kohavi and Thomke, 2017]

Model-Based LSRT

- Benefits of LSRT by building on existing test automation investment, and exposing behavior under arbitrarily long test sequences
- Benefits of FSM modeling by thoroghly exploring the system, as well as providing valuable insight into the construction of the system

Quick To Implement

- Tests can be built as quickly as the tester can write Python.
- Tests benefit from good engineering practices elsewhere in the testing effort.
- Tests can focus on areas of the system under inspection, an incomplete model is still valuable unlike in FSMs.

Selective Detail

- Testers can hammer small details like keystrokes into a textbox or focus only on big-picture program flow.
- Testers make as many or few assertions as they wish.
- Testers can control the flow of their walks depending on the testing context.

Bugs That Traditional Testing Finds

- Known bugs, whether previously fixed or bugs that are defended against
- Unfinished features
  - As in Test Driven Development
- Clear and obvious program faults
  - Obvious to the computer
  - Crashes, for instance
  - Nonzero return codes

What Traditional Testing Does Not Find

- Faults the tester did not think to test for
- Faults that are not obvious
- Faults the tester deems improbable

How To Find What Traditional Testing Does Not Find

- All the bugs missed are failures of imagination.
  - If a scenario can be imagined, a test can be written for it.
- Computers are really bad at imagining, too, but are passable at rolling dice.

Examples of The Bugs We Want to Find

- Digital phone system that crashes when the 22nd line is put on hold
- Flakey text editor that has been running for months on a grad student's laptop
- System that buckles when 200k users log on at the start of a workday
- Other "hard to reproduce" failures

Software Is A Finite State Machine

- Software representable as a machine with states, state transitions, inputs, outputs, and other tuples
- FSMs exactly describes the software's behavior
- Technique is popular in Electrical Engineering and for testing protocols

Testers Write Based On The System's States

- Page Object Model testing pattern emulates the system's underlying state model, and includes state transitions.
- Implied state model is significantly simplified compared to a formal FSM specification.
- POM provides a detailed look at how the system is built.

State Models Can Help Us Plan New Tests

- Given a printout of a state diagram, one can trace a pen along the model and plan a new test sequence.
- What parts of the System Under Test are tested and what parts are not yet tested becomes obvious.

Context: What Simplified State Models Don't Capture

- Input typed into the program
- Data the program read from some external source
- Overheating CPUs, full disks, cosmic rays, etc.

Simplified State Models Can Be Represented As Directed Multigraphs

- System states are vertexes, or nodes.
- Test functions are edges, connecting an in-node to an out-node.
- Each edge connects one in-node to one out-node, however
  - a given function might work as a transition to an out-node from multiple compatible in-nodes.
  - This behavior is a byproduct of convenicence features in the software under test, like having a logout button on every page.
  - For brevity's sake, treat a list of in-nodes on an edge's definition as a separate edge definition for each listed in-node.

Random Walks: Generating New Test Plans Automatically

Given one of these simplified state models represented as a graph, and a source of random numbers, automatically generating test plans is straightforward.

- For a given node, the current state, from the set of nodes
- Gather all of the edges, the transition functions, which have that state as their from-node
- Select and execute one of the gathered functions
- The selected function's to-node becomes the new current state
- Repeat until some planned condition is met or execution of a selected function is not possible

What Bugs Look Like From A Modeling Perspective

- Bugs manifest as nodes which the model says should be reachable, but execution cannot successfully reach.
- Such occurrences might be bugs in the software.
- Such occurrences might be bugs in the tester's model.

Prior Art: Model Based Testing

- Jonathan Jacky, in Radiation Oncology, of the University of Washington, made an excellent Python model-based tester called PyModel.
- PyModel consumes a handcrafted model.
- It can emit a test plan that covers the whole model.
- It can emit a test plan that takes a random, should-be valid walk of the software under test.

Weaknesses in PyModel

- It requires a handcrafted model in a finicky domain-specific language.
  - Not Plain Old Python.
- It is difficult to connect to test execution.
- It requires a lot of time to get running.

What Is Yeager?

- Python version 3 module
- Annotate funtions indicating that they cause a state transition
- Infers a state model
- Can take a random walk on that model
  - Can terminate random walks under selectable conditions
- Has debug tools to understand the inferred model

Yeager's API Fits On A Notecard

- `import yeager`
- `@yeager.state_transition(from, to)`
- `yeager.walk()`
- Tweak: `yeager.add_state_to_blacklist()`, `yeager.add_transition_to_blacklist()`, `yeager.remove_state_from_blacklist()`, `yeager.remove_transition_from_blacklist()`, and `yeager.set_edge_weight()`
- Debug: `yeager.enumerate_transitions()`, `yeager.reachable_states()`, `yeager.orphaned_states()`

Write a Function

```python
def login(driver):
    from pages.login import LoginPage
    lp = LoginPage(driver)
    lp.log_in_correctly(USERNAME, PASSWORD)
```

Annotate the State Transition

```
@yeager.state_transition("login", "dashboard")
def login(driver):
    from pages.login import LoginPage
    lp = LoginPage(driver)
    lp.log_in(USERNAME, PASSWORD)
```

Debug Yeager Models

- Using `enumerate_transitions` function as show in `enumerate_transitions.py`
- Using `orphaned_states` & `reachable_states` functions as shown in `orphaned_states.py` & `reachable_states.py`

Plan And Execute A Test Run

- `yeager.walk()`
- `yeager.walk(50)`
- `yeager.walk(exit_state="state-to-exit-on")`
- In development: after some visitation goal

Test Monica With Yeager

- Have a robust suite of Page Object Models
- Intuitive and meaningful system
- Public service

Intuitive States of Monica

- login page
- dashboard
- contacts list
- looking at a contact
- editing a contact
- logging a phone call or meeting with a contact
- writing in the journal
- etc.

States Necessitate Transitions

- Filling in the login form transitions from the login page to the dashboard
- Clicking a contact in the contacts list transitions to the viewing-a-contact state
- etc.

Use Existing Page Object Models As A Guide

- Emulates the Page Object Models' structure
- States are pages
- Methods are state transitions
  - Some transitions can be loopbacks

Write Some Glue and Go

For each method in the page object models:

- create a relatively stateless function that calls it.
- annotate any state transition that function triggers.

A Note on “Relative Statelessness”

- This will vary from tester to tester according to their gumption.
- It is reasonable for a test function to require a shared webdriver so page objects can be used.
- It might be reasonable for a test function to require a list of all the Contact names put into the system so far.
- It is unreasonable for a test function to require a memoizing key-value store with hundreds or thousands of entries.
- All extra arguments passed to `walk` are forwarded to test functions.
- Mutable arguments can be modified and these modifications persist across execution.

Example Suite's Model

It is straightforward to use the Yeager graph inference with graph visualization software. A routine is provided to allow users to visualize with the `graph_tool` module, which can further export to graphviz natively.

```
python3 visualize_graph.py
```

Take a Walk

- Execution begins with a call to `yeager.walk()`
- A demo: `python3 yeager_test.py`

What It Looks Like The Test Is Going Well

- No crash
- No assertions being tripped
- Software appears to be being executed

What It Looks Like When The Model Is Wrong

- Crash on an illogical sequence
- Example:
  - Click “Create Contact”
  - Click “Add this Contact”
  - Expected: On Contact pages
  - Actual: On Add Contact Page with an error message about needing to input a name
- A suite can generate this fault: `yeager_bad_model_test.py`

What It Looks Like When The Software Is Wrong

- Crash on a perfectly logical sequence
- Example:
  - Open a contact
  - Click "Add Reminder"
  - Fill in a date
  - Fill in a title
  - Check the "Remind me about this just once" box
  - Click the save button
  - Expected: On the contact's page, with a new reminder
  - Actual: On a 500 internal server error page
- `https://github.com/monicahq/monica/issues/326`

References I

Mike Andrews. System, heal thyself. Talk presented at the Florida Institute of Technology Computer Science Department weekly seminar, Melbourne, Florida, on September 28th., 2012.

Bruce Dawson. There are only four billion floats, so test them all!, 2014. URL https://randomascii.wordpress.com/2014/01/27/theres-only-four-billion-floatsso-test-them-all/.

Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE transactions on software engineering*, 17(8):751–761, 1991.

J Heyman, J Hamrén, C Byström, and H Heyman. Locust: An open source load testing tool., 2011. URL http://locust.io.

References II

Dan Hoffman. Key tradeoffs in high volume test automation. Talk presented at the 12th Annual Workshop on Teaching Software Testing, Melbourne, Florida, 2013. URL `http://wtst.org/wp-content/uploads/2013/01/DanHoffmanwtst2013.pdf`.

Douglas Hoffman. Exhausting your test options. *STQE Magazine*, pages 10–11, July/August 2003.

Cem Kaner. An overview of high volume automated testing, 2013. URL `http://kaner.com/?p=278`.

Ron Kohavi and Stefan Thomke. The surprising power of online experiments, 2017. URL `https://hbr.org/2017/09/the-surprising-power-of-online-experiments`.

David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

References III

Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of operating system utilities. Technical Report 830, University of Wisconsin–Madison, 1989.

Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

Tauhida Parveen and Scott Tilley. When to migrate software testing to the cloud? In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 424–427. IEEE, 2010.