Yeager: An Annotation-based Framework for the Generation of Automated Long Sequence Regression Tests in Python

Casey Doran

Florida Institute of Technology

cdoran2011@my.fit.edu

November 28, 2017

Overview

Play Along at Home

- `github.com/elementc/yeager`
- `github.com/elementc/monica-tests-traditional`
- `github.com/elementc/monica-tests-yeagerized`
- `github.com/elementc/thesis`
- `github.com/monicahq/monica`
- `monica-doran.herokuapp.com`

Why Automate?

- Save time.
- Save money.
- Computers don't get bored.
  - Testing is boring work.
- Doesn't overlook test cases.

How Do You Automate?

- Write functions that exercise the system under test.
- Put these functions in a format that can be consumed by a test runner.
- Call test runner.
- Interpret test runner's output.

Kinds of Test Automation

- Unit- Verify individual code functions work as expected.
- Integration- Verify modules are working together.
- Regression- Verify stuff that worked isn't broken and stuff that's broken before haven't broken again.
- Functional- Validate that system as a whole conforms to requirements (eg, works to a user's eye).
- Many others- see CSE3411 & CSE4415.

Languages

- *Unit frameworks (CUnit, JUnit, CppUnit, etc.) enable the practice, though they're useful for far more than unit testing.
- Testers are usually, unintuitively, **less** trained as programmers.
- Consequently, they prefer "easier" scripting languages like Python or Ruby.
- This discussion will center around Python. All of it can happen in Ruby.

Frameworks

- Include a suite of assertion convenience methods, logging/reporting facilities, and a runner.
- Python: `unittest`, `nose`, `pytest`.
- `unittest` is in the Python Standard Library, we'll use it.

Glass Box Testing

- Test code interacts directly with the program's source.
- Can probe quite deeply.
- Use mock interfaces shims to accomplish testing goals.
- Unit testing and integration testing are automated through Glass Box methods.

Black Box Testing

- Test code interacts with the user (or some other non-transparent) interface into the running program.
- Use external toolkits like Selenium to enable driving user interfaces.
- Usually in a special test environment but otherwise the unmodified software.
- Regression testing and functional testing are automated through Black Box methods.

Selenium

- Programmatic control of web browsers for testing and other automation.
- Driver class allows navigation (get this URL) and document queries (get this node for me to read from or click on or type into).
- Node class allows interaction (click here, type this), and data retrieval (What's the text body? What' site does this form POST to?) and limited Driver-like queries for children.

HTML (summary)

- XML- based documents for the web.
- Tree-structured.
- Nodes have properties, including text, in addition to children.

CSS (summary)

- Language for styling HTML documents.
- Format- `selector: rule;`
- Selectors: strings that identify one, many, or none of the nodes in an HTML document.
- Rules: Specific styling rules to apply to each node matched by preceeding rule.

Monica: A Personal CRM

- Open-Source.
- Life-tracker.
- Friend-keeper.
- Journal.
- In the cloud.

Contacts Book On Steroids

- We've all seen a contact list as an example in a database course.
- Monica extends that to the extreme. Per-contact notebooks, relationship tracking (How many wives did he have?), reminders, etc.
- Screenshots TBD

Page Object Modeling

- Each page on a site corresponds to a Python class.
- Fields or text spots on pages get getters and setters.
- Clickable things get click() functions.
  - If the click should transition to a new page, construct and return that new page's class.
- In class consturctors, assert things that are constant about that page. Username in header? Some error message not in the body? Standard controls are present?

How Web Test Suites Come Together

- Build all the page objects and put them in /pages/.
- Write step-by-step test plan as comments in the body of a function in the runner's format.
- Translate english steps into python code.

Running Tests

- Same as running any other script.
- `python3 test_contacts.py`
- Some frameworks have a multi-script runner.
- `python3 -m unittest`

What Traditional Testing Finds

- Known bugs, whether previously fixed or things that you look to see are working not working.
- Unfinished features. (Write the tests before you write the feature.)
- Clear and obvious program crashes.*
  - *Clear and obvious to the computer. Nonzero return codes.

What Traditional Testing Doesn't Find

- Stuff you didn't think to test for.
- Stuff that isn't an obvious failure.
- Tons of others.

How To Find What Traditional Testing Doesn't Find

- All the things we miss are failures of imagination. If you can imagine a scenario, you can probably write a test for that scenario.
- Computers are really bad at imagining things too, but they're decent at rolling dice.

Examples of The Bugs We Want To Find

- What happens when you put 22 calls on hold on a digital phone system?
- What happens when you leave Atom running for four months while writing a thesis in it?
- What happens when 200k users log on to your system at the start of a workday?
- Other "hard to reproduce" failures.

Software Is A State Machine

- Software can be represented as a machine with states, state transitions, inputs, outputs, and a few dozen other tuples.
- This IS the software, just a different way to view it.
- This is already sort of evident.

Testers Write Based On The System's States

- Page Object Model pattern emulates the state model, and includes state transitions.
- Resultant state model is significantly simplified compared to a formal model specification.
- Surprisingly detailed look at how the system is built.

State Models Can Help Us Plan New Tests

- Given a printout of a state model, it's not hard to trace a pen along the model and plan a new test sequence.
- It's also easy to see what parts are tested (covered) and what parts aren't yet tested (still not covered).

Context: What Simplified State Models Don't Capture

- Things you've typed into the program.
- Things the program read from some external source.
- Overheating CPUs, cosmic rays, etc.

Random Walks: Generating New Test Plans Automatically

- Given one of these simplified state models represented as a graph, and an RNG, it's not hard to generate new test plans.
- How many ways can you go from here? Ok, draw a random choice from that many. Go there. Repeat.
- That's the algorithm, by the way.

What Bugs Look Like From A Modeling Perspective

- Places where the model says you should be able to go, and you are unable to go, are bugs.
- They might be bugs in the software.
- While you're learning the software under test, they're probably bugs in the 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. Apparently there are model-capturable bugs in the field of Radiation Oncology. (Therac 25)
- It 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's really hard to get test plans to run as tests.
- It has a lot of overhead to get running.

What Is Yeager?

- Python module.
- Annotate funtions indicating that they cause a state transition.
- Infers a state model.
- Can take a random walk on that model.
- Has debug tools to understand the model you've made.

Yeager's API Fits On A Notecard

- `import yeager`
- `@yeager.state_transition()`
- `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

```
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_correctly(USERNAME, PASSWORD)
```

Repeat

- `github.com/elementc/monica-tests-yeagerized`

Debug Your Models

- Using `enumerate_transitions` function.
- Using `orphaned_states` function & its inverse.

Plan a Test Run

- `yeager.walk()`
- `yeager.walk(50)`
- `yeager.walk(exit_state="state-to-exit-on")`

Run It

- `python3 yeager_test.py`

Let's Test Monica

- We already ahve a pretty robust suite of page models built.
- We're already familiar with the usage of it.

Monica's Intuitive States

- Home page, 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 etc etc.

States Necessitate Transitions

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

Boy, These Look Familiar

- Emulates the Page Object Models.
- States are pages, methods are state transitions (even back to the same state).

Write Some Glue and Go

- For each method in your page object model
- Create a relatively stateless function that calls it.
- Annotate any state transition.

Example Suite's Model

- Need to figure out how to insert pictures on Beamer.

Give It A Run!

- Call walk. That's it. Really.

What It Looks Like When Everything's Good

- No crash.

What It Looks Like When The Model Is Wrong

- Crash on an illogical sequence.

What It Looks Like When The Software Is Wrong

- Crash on a perfectly logical sequence.

What Is High Volume Test Automation?

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

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 (ab)use 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 you hook up the RNG to the SUT.
- In short, how the test drives the System.

Interface

- Black box?
- White box?
- In between? (hitting an HTTP api that's not public but also not unit-ey)

Oracle

- How do you know if you've exposed bad behavior?

Loggers and Diagnostics

- How you know what happened.

Context

- What are you trying to do? Corner a bug? Survey the system? Abuse the system?

Scalability

- Single-threaded? Many-threaded? Hammering the system from thousands of cloud servers?

[Parveen and Tilley(2010)]

Inventors One Through Three

- HP, "evil", 1966
- TI
- Bell

Inventors Four Through Six (And Beyond)

- Microsoft
- Telenova
- Fuzz Tester [Miller et al.(1989)Miller, Fredriksen, and So]

Everybody Thinks It's A Trade Secret

- Why wouldn't they?
- It's massively better at testing than what they read about.

A Call For Academic Consideration

- TBD

Family Tree: Exploring (And Abusing) The Anatomy

- TBD

Long Sequence Regression Testing

- TBD

State Model Testing

- TBD

Exhastive Testing

- TBD

Fuzz Testing

- TBD

Load Testing

- TBD

Testing In Production

- TBD

A/B Testing

- TBD

[Kohavi and Thomke(2017)]

Model-Based LSRT

- TBD

Quick To Implement

- TBD

Good Enough Detail

- TBD

Benefit From Existing Test Code

- TBD

References I

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.

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.

References II

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.