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

by

Casey Doran

Bachelor of Science
Software Engineering
Florida Institute of Technology
2015

A thesis
submitted to the School of Computing at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
November 2017

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling in
part the requirements for the degree of
Master of Science in Software Engineering.

"Yeager: An Annotation-Based Framework for the Generation of Automated
Long Sequence Regression Tests in Python",
a thesis by Casey Doran

____

Keith Brian Gallagher, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Thesis Advisor

____

William H. Allen III, Ph.D.
Associate Professor, Harris Institute for Assured Information
Committee Member

____

Anthony Smith, Ph.D.
Assistant Professor, Electrical and Computer Engineering
Outside Member

____

Phil Bernhard, Ph.D.
Associate Professor and Director, School of Computing

Abstract

TITLE:	Yeager: An Annotation-Based Framework for the Generation of Automated Long Sequence Regression Tests in Python
AUTHOR:	Casey Doran
MAJOR ADVISOR:	Keith Brian Gallagher, Ph.D.

This work presents a Python software package, Yeager, designed to enable the generation and execution of high-volume automated long-sequence regression tests. Users apply the package to existing suites of automated regression tests by annotating individual test methods as state changes for the Software Under Test. Given a sufficiently connected state model (as inferred from these annotations), it becomes possible to generate and execute configurable random walks through the SUT's various states instead of simple regression suites as originally written.

Divided into three sections, this thesis provides a concise overview of an exemplar regression test suite in Python for a web application, a guide to the usage of Yeager itself within the context of the aforementioned regression test suite, and an overview of the history, anatomy, and family tree of High Volume Automated Testing while contextualizing the contribution Yeager makes to the practice.

Contents

Acknowledgements

This thesis would not exist if not for the assistance of:

- Dr. Cem Kaner and the Center for Software Testing Education and Research, for taking a chance on an enthusiastic freshman and introducing me to the world of software testing, as well as years of world-class training beyond valuation.
- The wider context-driven testing community, particularly the participants in the Workshops on Teaching Software Testing 11 and 12, my first exposure to the considerable ups and downs of academia.
- Ana Marafuga, Mike DeCabia, Jeff Farr, and Curtis Chambers, and the team at Dycom Industries, the most formative internship I've ever had. They let an intern design their entire corporate test automation strategy, most of which informed this thesis, brave souls all.
- Dr. Richard Ford, whose infectious enthusiasm for others' learning and discovery knows no bounds.
- The Samuels family, particularly Bill Jr. and Rob, and Dave Pickerell, for their helpful contributions through most of my academic tribulations.
- The members of the Ruckus and the Harbor City Hooligans. Soccer clubs typically lack rigor, but on the space coast everybody's a rocket scientist.
- My immediate and adopted family, for loving me to where I am today, including
- kbg, for reminding me to thank them for it.

Dedication

For Joseph Campo, Jr. and Alexander Yanes. Life is short. You'll be forever a lion.

Chapter 1

An Overview of a Regression Test Suite for a Web Application

This thesis proposes a general-purpose Python module for the implementation of high volume automated tests. To properly discuss the nuanced uses of the module, it is first critical to establish a "typical" industrial usage scenario.

To that end, this chapter describes the state of the art in the web test automation field, and walks through the construction of a web test suite for a popular open source relationship management site, Monica, available for use from the website `https://monicahq.org` as well for self-hosting from `https://github.com/monicahq/monica`. Later chapters will discuss implementation of the module for high volume long sequence regression testing as well as the industrial and academic context surrounding the practice of high volume automated testing.

The test suite discussed in this chapter is published in its entirety online at `https://github.com/elementc/monica-tests-traditional`. They are written against the 0.6.5 release of the Monica software, and may be run using Python 3.

1.1 Technologies

There are a considerable number of tools and libraries used in the development and execution of web application tests.[Kaur and Gupta, 2013] Regardless of actual platform, there must be at least a browser driver, a test runner, and probably some set of inspection tools. As later chapters will use a Python library, the following Python-friendly libraries have been selected.

1.1.1 Selenium

The Selenium open source project is a library which permits the programmatic control of a web browser.[Holmes and Kellogg, 2006] This library is ostensibly designed for automated testing purposes[Razak and Fahrurazi, 2011], but it may be used in any case where automated browser interaction is critical, including secretarial desktop automation, the development of testing tools, malicious purposes, and niche industrial purposes.[Kongsli, 2007] It has a number of supported platforms, including Python and a purpose-built IDE.[Bruns et al., 2009] Selenium's purpose-built IDE features capture-playback technology which enables quick recording of test scenarios for later playback, but suffers from drawbacks related to the maintainability of the generated recordings.[Leotta et al., 2013b]

The general programmatic usage cycle is:

1. Instantiate a browser driver, selecting the type of web browser to be driven.

2. Load specific URLs using the driver's `get` method.

3. Query the loaded page using the driver's `find_element` methods.

4. Interact with page components using the element objects and associated methods returned from the above step.

[Artzi et al., 2011]

1.1.2 Python Test Runners

Test runners are executables that load test suites, execute selected subsets, and then report results. There are several different test runners in the Python ecosystem, varying in their usage, provided test libraries, and reporting capabilities. [Nielsen, 2014] Common test runners like pytest and nose live in the Python package archive, and have many users. However, there is a test runner built into the Python standard library named unittest.[Pajankar, 2017]

In the interest of keeping the dependencies of this test suite down (and taking advantage of familiar, high quality documentation), we have selected the unittest library for the runner of this test suite.

1.1.3 Developer Tools and Resources

Test authors need to be able to inspect the application under test from the UI perspective in order to effectively use Selenium. Historically, a web debugger such as Firefox's Firebug tool has been used to fill this requirement, which provides UI inspections, a JavaScript REPL, and other useful functionality.[Nicholus, 2016]

In modern web development, however, the debugger and inspector are built directly into the desktop web browser.[Odell, 2014] These are all roughly equivalent in capability, but for the purposes of this document we'll use terminology consistent with Google Chrome's Inspector toolkit, which can be accessed with the F12 key.

1.2 Architecture

A typical web application test suite is built from three components: a collection of page object models, a set of configuration parameters, and a set of test sequence scripts. In most cases, these will be stored in similarly named directories (/pages/, /config/, and /tests/, which may be stored on top-level depending on suite complexity or runner requirements.).[Leotta et al., 2013a]

1.2.1 Page Objects

To abstract out much of the low level work associated with interacting with the system under test, a typical usage case is to write a Python class for each "page" of the web application.[Kung et al., 2000] In this scenario, each class will have a function related to each of the low level interactions, e.g. setting a field to a value or pulling a string from the page title. In the constructor of the page object, common sanity checks are often run to ensure the system is in a good state.

1.2.2 Configuration

It is common for different environments to have different credentials and settings, for instance, a continuous integration server might deploy with one set of passwords while a developer workstation has another, and a user acceptance test server has a third set. To that end, it is critical that such variances are captured correctly, often in a configuration file or by reading correct values from execution environment variables.[Marchetto et al., 2008] This may also be sensitive to certain differences in environments, for instance the need to skip verification of emails sent by the system, or usage of different sets of mock interfaces.

1.2.3 Test Sequences

With page interfaces well defined and a suite of configuration details available, actual test authorship becomes fairly simple, with files conformant to the selected test runner's interface being filled with sequences of fairly easy to understand, high-level steps. Most of the time, a test script will be authored for each user story in the requirements of the system under test.

Scenario-based tests are usually the easiest for novice programmers to write, but advanced methods may include detailed tours of the system, data-flow focused traces of system usage[Liu et al., 2000] or even exhaustive tests of certain feature sets in isolation: test scripts for each of a calendar, a mail client, a presentation tool, and a contacts manager in an office productivity suite. There are even tools for acceptance testing where requirements can be added like in a wiki, which can be modified to generate Selenium code independent of sequences of logical usage.[Wang and Xu, 2009]

1.3 Building the Test Suite

While we submit anecdotally that many would-be test automators are NOT traditionally trained software engineers, resulting in many industrial practices that are somewhat backwards to a software engineering-conscious observer, the composition of a web test automation suite is thoroughly a software engineering exercise. Various documents and projects attempt to reconcile these skillsets with these testers, including many project templates and several useful boilerplates. [Sandström, 2015]

The following domain-specific considerations are relevant to software engineers who would study or experiment with automated tests of a web system. The reader is encouraged to study existing boilerplates or the reference Monica test implementation to supplement these considerations.

1.3.1 Planning a Set of Tests

Testers usually receive two sets of testing goals: verify conformance to some set of design and acceptance criteria, whether through the performance of some set of user story scenarios, touring features from a requirements document, or some other list of "shoulds"; and discover and verify new bugs through careful torture-testing of the system under test. We'll cover how to do the discovery part later on in chapter 2, and the verification process is currently a manual process, but verifying conformance is a straightforward enough task to automate.

In fact, a common way to start writing automated regression tests is to copy the step-by-step instructions for a given scenario into comments in the body of a Python test script. Each step becomes a snippet of code during the authorship process, with steps and checks corresponding to actions and assertions in the code that gets written. Then, as a tester walks through the steps in the application under test, they write the code bit by bit until they've completed walking through an entire scenario. In this way, an automated test is designed and built.[Nguyen, 2001] Later subsections will outline these smaller authorship steps in the context of a web application.

1.3.2 Determining DOM Object Identification Methods

In order to be able to construct a proper page action, it is critical that our test code is able to interact with the right specific parts of the page under test. In Selenium, we use the driver's `find_element_by_*` methods to do so. There are methods for finding page elements by many methods, including HTML ID, HTML name, link text and partial link text, CSS selectors, and several other more unique methods like an XPath string or just the tag's name. All of these methods are convenience wrappers for a base method named find_element that takes a special constant from the `selenium.webdriver.common.by.By` class, such as `By.ID` or `By.CSS_SELECTOR`.

While it may be more immediately readable to write test code using the convenience methods, this does have an effect on maintainability in that if a particular field must change the method it is found by, many function calls will need to be replaced.[Gupta et al., 2003] To prevent such a replacement nightmare, we can use Python tuples and the unzip (splat) operator to combine a method of selection with a string constant as a single "element selector" field which may universally be consumed by a `find_element` function call.

Consider this HTML tag:

```
<input
  type="email"
  class="form-control"
  id="email"
  name="email"
  value=""
>
```

This has a number of useful attributes we could use as a selector, but the best of all is the id field. HTML ids must be unique in an HTML document [Web Hypertext Application Technologies Working Group, 2017b] so selection by ID is extremely resilient. Here's a selector and a call to `find_element` for the id “email”:

```
email_sel = (By.ID, "email")
email_field = driver.find_element(*email_sel)
```

While selecting by the id field is comparatively simple (application authors may wish to give constant ids to parts of their applications they know will be involved in testing)[Web Hypertext Application Technologies Working Group, 2017a], selection by other methods is more complex. Consider this HTML tag:

```
<button
  type="submit"
  class="btn btn-primary"
>
  Login
</button>
```

This button is critical, it must be clicked in order to complete a login! However, it lacks a unique ID. It is tempting to use the `By.LINK_TEXT` selection method since it has a fairly concise and unique body text ("Login"), but this won't work since it's a `<button>` tag and not an `<a>` (anchor, a hyperlink base) tag. The next logical option is to select by class, the class attribute of HTML being whitespace-separated tags which are not guaranteed to be unique. If the software is designed to use classes in a way that relevant tags will have a unique combination of classes, this is an option, but it is not typical. In this case, the button has the `btn` and `btn-primary` classes, an identically styled button would have the same set of classes. This, then, is a candidate for improvement in the system under test, to at least provide a cleaner testing interface in the form of a unique id or class on this button, but in the interim, we can fall back to HTML's built-in selection system, the CSS selector.

A CSS selector is a string conformant to the CSS selection grammar [World Wide Web Consortium CSS Working Group, 2017] which enables detailed selection of DOM element or elements. It can combine an element's tag, id, class, parents, children, or even position. For the purposes of this login button's selection, we require the following features of candidate elements:

- Class `btn`.
- Class `btn-primary`.
- First on the page.

The following CSS selector satisfies these requirements:

`.btn.btn-primary:nth-of-type(1)`.

Here's a Python example:

```
login_sel = (By.CSS_SELECTOR, ".btn.btn-primary:nth-of-type(1)")
login_button = driver.find_element(*login_sel)
```

1.3.3 Scripting Actions

Now that each relevant element of the web page under test has a unique identifier for our use, the next step is to write the code that actually triggers the interactions with them. This is fairly straightforward, we use the page object's driver field to retrieve elements using these identifiers, then take actions on those elements.

The following snippet from our login page test retrieves an email text field and a password text field by their HTML IDs, as well as a login button by a CSS selector. These elements are then interacted with via the `send_keys()` and `click()` methods. Note that `self.username` and `self.password` are defined in the object constructor from some secure source of testing account credentials.

```
email_sel = (By.ID, "email")
password_sel = (By.ID, "password")
login_btn_sel = (By.CSS_SELECTOR, "button.btn.btn-primary")
def log_in_correctly(self):
  email_field = self.driver.find_element(*self.email_sel)
  password_field = self.driver.find_element(*self.password_sel)
  login_button = self.driver.find_element(*self.login_btn_sel)
  email_field.send_keys(self.username)
  password_field.send_keys(self.password)
  login_button.click()
```

1.3.4 Asserting Validity

Consider what an assertion (the `assert` statement in Python) does: it takes one mandatory argument, a statement that boils down to true or false, and if the statement evaluates to anything but true, the system halts immediately with a message (an optional second argument or a useful default one) as to what assertion failed and in what way. They explicate and then enforce contracts in programming systems and are a staple of high quality programming.

Since contracts are a core part of a test, much of the actual important work of a test suite consists of well-placed assertions about the state of the system under test. While the assert statement is the most obvious and common way to designate an assertion, any snippet of code which does not change the state of the running test and raises an exception if some assumption is not true can fulfill the same purpose- we call these "assert-alikes". For instance, if a Selenium webdriver `find_element` fails to find the element described, it simply raises an exception. There's no need to do something complex like

```
dash = self.driver.find_element(*self.dashboard_sel) or False
assert dash, ''Couldn't find the dashboard.''
```

when, for the purposes of asserting some page element is present, the element finding call itself (`self.driver.find_element(*self.dashboard_sel)`) can stand alone.

Within the page object model, two key places for the insertion of assertions becomes apparent: first, at the initialization of the page object model itself, and second, as necessary during action execution. In the `monicatests-traditional` repository, page object models are derived from a `PageBase` class in the page_base.py file. This includes an overridable function, `self.initial_status`, which is called in the lowest inherited class definition- and can be selectively chained upwards- after object construction is complete. This method is filled with a number of assertions and assertion-alikes for the given page object model at the expected starting state in that page. Methods on the page object model also include assertions as necessary to verify successful operation, including constructing and returning a page model for a new page when a method causes a transition for that new page- therefore triggering the new page's initial_status method all over again.

Chapter 2

Using Yeager to Generate Long Sequence Regression Tests

The test suite assembled in the previous chapter is a great way for a software development team to verify that the core functionality of the system under test is fundamentally operational. When executed, it will test the few well-understood scenarios we have outlined consistently and, assuming enough assertions are present, thoroughly. In fact, the suite requires the entire process from the previous chapter in order to accommodate the addition of new scenarios.

It's a boring, tedious, and repetitious task that can be the entire career of a test engineer. However, as any test automator will know, tasks which are boring, tedious, and repetitious are ripe targets for computer automation, and the task of scenario authorship is no different.

This chapter will outline a method for adapting the existing test suite explored in the previous chapter, using a tool of our own authorship named Yeager, to enable the computer to generate scenarios automatically. Yeager is an MIT- licensed open source Python 3 module, with source available at `https://github.com/elementc/yeager`. It provides a Python annotation and a set of utility functions. Usage of Yeager's state transition annotation allows testers to quickly and easily map an existing suite of test code onto a state machine, in the form of a graph. This graph can then be traversed using the utility functions, thereby generating new test scenarios from the existing code.

The resultant adapted test suite is published online at `https://github.com/elementc/monica-tests-yeagerized` for your convenience.

2.1 Software as a State Machine

Consider the system under test, Monica. As a relationship management web site, it has a few obvious states it can be in: logged out and on the landing page, logged in and on the dashboard, viewing a list of contacts, viewing a list of journal entries, or viewing the settings page. This maps nicely to the page objects we defined in the previous chapter. Actions on those page objects assume a current state (e.g., we're logged in and on the dashboard) and after execution are in a new state which may or may not be the same state (eg, the `Dashboard.click_contacts_button()` method transitions from the dashboard to the contacts list, while the `LoginPage.log_in_incorrectly()` method should result in the system being in the same login page it was before the method was run).

In fact, most modern programs can be looked at as systems composed of a finite set of states (pages, in this case) with some state transitions (links) and a data context (the stuff you've already typed into the system in those states).

Yeager uses this fact to enable automated test sequence generation.

2.1.1 States in Our Example System

Let's consider Monica's pages, which are already built into our test suite, to be states.

We have: the login page (`Login`) and logging in takes us to the `Dashboard` which has tabs for the `Contacts` list and the `Journal` log. There's also a `Settings` page which has subpages for `Import`, `Export`, `Users`, and `Tags`.

The Dashboard and Contacts list both let us `AddAContact`, while the Journal tab lets us `AddAJournalEntry`.

From a given `Contact`, one can `AddASignificantOther`, `AddAChild`, `UpdateJobInformation`, `AddANote`, `AddAnActivity`, `AddAReminder`, `AddAGift`, and `AddADebt`.

For the purpose of our discussions, these pages will constitute the entire set of states in the system under test. Conveniently, each of them is a Python class.

2.1.2 State Transitions as Actions in Our Example System

A graph consists of a set of nodes and a set of edges. If our nodes are the states the system under test may be in, the edges are the actions that may be taken from those states, possibly resulting in a state transition. It is certainly possible for an edge to be a loop connecting the starting state to itself. In the particular case of testing web applications, note that though it's reasonable to author a page object model with each method corresponding to an edge, this is not an assumption that is necessary to make, and would-be Yeager adopters may choose to lump lower-level page object methods into clusters of function calls in new functions and treat those higher-level functions as edges instead. In fact, the example Yeagerized Monica tests do just this, creating a suite of Yeager-friendly functions as snippets of existing test sequences, built from page object function calls.

2.1.3 Capturing Contextual State

Before embarking on the journey of high volume test automation that follows, it is important to consider for a moment the entirety of the software system. More than just software, a system includes the entire context of the software's execution, from the software itself, to the contents of the database, to the number of active system threads, to the ambient temperature of the room the system is running in.

Some of these are impossible to control for in a testing environment: it's unreasonable to unseat your CPU cooler to attempt to replicate a bug related to your cousin's dust-clogged Pentium II for instance. Others are possible to configure with initialization scripts and virtualization to a degree, for instance always having the same base OS image and environment variables, or always having the same amount of RAM and number of CPU cores. Still more can be controlled for using database snapshots and well-documented test data. Regardless, remembering these variables, which are external to the code but internal to the system under test, is critical to the development exercises in this chapter.

2.1.4 Taking a Walk on the Graph

Imagine standing on a giant picture of the program under test's state graph, at the starting point. You are able to walk along the lines in the directions they are drawn to new points, but you can't walk backwards against their direction of travel. It's a strange looking environment for sure, many of the nodes you might stand on have leaving edges that just loop back to where they started. Existing test scripts are like following directions along this map, "from A, go to B, turn right at C, stop at D" and so on. These pre-planned scripts are an effective way to make sure you visit the whole map at least once.

But, imagine that you had a lot of time to kill and had already done all of your maps for the day. An interesting way to spend your time might be to go wandering: wherever you are, pick one of the paths before you at random and walk that way. Keep flipping coins or rolling n-sided dice and walking and you might eventually trigger a secret passage in the labyrinth you've been wandering. Well, that or crash the program under test. How exciting!

Contrived thought experiments aside, the notion of wandering around a program is a useful one for testing. First, it simulates human usage a little more realistically than many test scenarios (how many users actually start from a freshly booted computer, load up and login to the dashboard, create one record, search for that record, delete that record, then log out?). Second, such a process can be of any arbitrary length which, while also contributing to a more realistic usage simulation[1], permits test managers to use as much of the technique as they want to- wandering the program under test for a few hours

[1] Consider that the author's instances of the Atom text editor and the GNOME desktop environment have been open on their laptop since August, while the typical Atom CI instance takes 30 minutes to build the software and run all tests.[Atom Open Source Project, 2017]

on their laptop during a conference call or over a month on a virtual machine hosted in a cloud somewhere.

2.2 Yeager State Transition Annotations

The meat of Yeager testing is accomplished through the annotation of Python test methods. An *annotation*, also known as a *decorator* or a *function decorator* is a special Python function which is executed at the time of another function's definition, receives the function being defined as well as any other required items as parameters, and can optionally wrap the function being defined in a special modifier. Yeager is implemented as a special Python annotation and a set of utility functions which register after definition and can then call plain old Python functions.

The annotation, `yeager.state_transition` and some utility functions (`yeager.add_state_to_blacklist`, `yeager.orphaned_states`, `yeager.reachable_states`, and `yeager.walk`) are described in this section.

A note on Python convention: there are two kinds of parameters a function may take. The first, *positional arguments*, are passed like this: `print("some argument")` or `math.pow(3,2)`. The second, *keyword arguments*, sometimes shortened to *kwargs*, are passed like this: `timedelta(hours=3, minutes=7, seconds=20)`. Order does not matter with kwargs. Positional arguments and kwargs that aren't explicitly defined in a function's signature can both be captured for use by a function. Positional arguments are captured into a Python `list` by defining a (potentially semi-) final argument prefixed with a single asterisk, keyword arguments into a Python dictionary (`dict`) with a final argument prefixed with two asterisks, like this:

```
def function(arg1="Default val", *args_var, **kwargs_var)
```

2.2.1 State Identifiers

Anything that can be a Python `dict` key can serve as a state identifier. For simplicity's sake we use strings in this document, but as long as Python will allow it, so will Yeager. Enterprising Yeager hackers may use the actual Python page object model class, for instance.

The example implementation of a Yeager test uses unique strings for state identifiers. There's no strong reason for this, it's just illustrative.

2.2.2 Basic State Transition Annotations

The fastest way to get started with using Yeager is to define a function for each of the state transitions you wish to use in the test. These will probably be short snippets from the traditional-style test sequences. Then, for each of these functions, you should use the `yeager.state_transition` annotation to mark the transition of that function. Here's an example using some of our Monica test code from the previous chapter:

```
from pages.login import LoginPage
from pages.dashboard import DashboardPage
from yeager import state_transition


@state_transition(None, "login-page")
def open(driver, **kwargs):
```

```
    driver = webdriver.Chrome()
    driver.get("https://app.monicahq.com/")


@state_transition("login-page", "dashboard-page")
def log_in(driver, **kwargs):
    login = LoginPage(driver)
    login.log_in_correctly()


@state_transition("dashboard-page", "login-page")
def log_out(driver, **kwargs):
    dashboard = DashboardPage(driver)
    dashboard.log_out()
```

Note that we use the Python `None` constant as a reference to the uninitialized system. Yeager treats `None` as a special node in the implied state model our annotations provide: it's assumed to be the entry point.

2.2.3 Verifying Connectedness

Yeager provides a utility function to check for states which cannot be reached from a given state, probably due to misconfigured annotations. The function, `yeager.orphaned_states`, takes one optional argument (the starting state, it defaults to `None`), and returns a `list` of all states that Yeager knows about but doesn't know how to get to. The inverse set, the known states, is also provided as a utility function with the same optional argument, as `yeager.reachable_states`. Though the orphaned states function is useful for debugging, it can be used in other automated ways, for instance as a test coverage check or a way to automate `walk` calls with each of the "orphaned" states actually being new entry points. They're useful in a number of different contexts for enterprising testers.

2.3 Yeager Test Harnesses

A suite of Yeager-annotated Python functions, while neat, is neither immediately useful (it's still just a chunk of naked functions), nor particularly intrusive (annotating a function with a Yeager state transition only adds a print statement before the function executes). Analysis of the state transition graph can be done manually for sure (`from yeager import nodes, edges`), but Yeager also provides a set of utility functions to actually exercise the system under test.

2.3.1 Test Setup and Entry Point

It's up to testers to generate Python scripts that start up and execute a Yeager test, but the process is very easy.

The first step is to cause the Python interpreter to parse all of the relevant Yeager annotations. In simple test scripts, it's enough to simply write the test code and annotations at the top of the file, but in large test suites, it may be necessary to import those Python files at the top of the Yeager test script instead. Critically, Yeager annotation metadata exists as long as the Python interpreter instance does, so it doesn't matter what modules or other structure applies to the code the Yeager annotations are spread around in. If it has been parsed, Yeager knows about it.[2]

[2]Unless modifications have been made to the `nodes` or `edges` data structures.

To actually start taking a walk on the state model, simply call Yeager's `walk` function.

2.3.2 Walk Options and Execution

The function `yeager.walk` takes some special arguments that determine how a test will eventually come to an end. If a tester just wants to go walking until they tell it to stop, they can call it with no args and it will run until the test is killed with a `SIGINT`. If a tester wants to just take a fixed number of steps, they can pass that as a naked integer or kwarg named count to `walk` and it will run for that many steps and then return. If a tester doesn't care about how long a test runs, and only wants to run until the program gets to some particular state, using the kwarg `exit_state` with the desired end state will cause the `walk` call to return as soon as Yeager wanders to that state. And, finally, if a tester wishes to start the walking from a state other than the default starting state of `None`, they may do so by supplying the kwarg `start_state` with the desired starting state.

All of these different preferences are just plans- the intended way to wander the graph. If an exception is raised in the underlying code, the exception is thrown all the way to the caller. Yeager doesn't try to continue walking since the system under test may be in a corrupt state. It's not possible to resume the existing `walk`, but it is possible to call `walk` again from scratch.

2.3.3 Application Context

All other kwargs that are passed to the `yeager.walk` call will be passed to the transition functions by Yeager, so a driver kwarg could be used to provide a webdriver to a web application's test suite, or a `dict` named context could store contextual information about the system under test.

Changes to mutable objects are preserved for the rest of execution, so it becomes possible to memoize things like data entered into the system or previously-captured search results. All kwargs unrecognized by the `walk` call are passed to all of the transition functions that Yeager steps through, so it is in the Yeager test style to have all test functions use the `**kwargs` catch-all argument in case a new context argument is added in the future of the test suite's development.

2.3.4 Logging in Yeager

Yeager doesn't make any particular assumptions about the logging toolkit that you use. It uses standard output to print its own data, though future revisions might use the standard Python logging interface. For Long Sequence Regression Testing, it is very important to log with vigor, as a failure is often the result of many consecutive steps instead of one instant.

2.3.5 Controlling the Path: Blacklists

While it may seem counterintuitive after going to the effort to define them, it is possible to mark a state as one to not visit during a `walk`. This is useful, say, in cases where testers might want a run configuration that avoids certain known-buggy regions of the system under test, or try a Yeager test but know parts of their Yeager-specific code is still incomplete. It's accomplished by using the `yeager.add_state_to_blacklist` function. Any state identifier, when passed as an argument, will not be visited by a Yeager `walk`.

2.3.6 Controlling the Path: Weights

Humans using software don't truly do actions equally randomly. A user of the Atom text editor, for instance, probably spends more time typing and saving than they do moving tabs around, opening consoles, running compile/lint commands, changing themes or settings, and so on. To enable better simulation of these more-probable actions, Yeager supports the notion of weighting edges.

An edge may be weighted by using a standalone function (`yeager.set_edge_weight`) or by using the weight kwarg with the state transition annotation (`@state_transition("st-a", "st-b", weight=10)`). Notionally, an unweighted edge has a weight of 1. This edge gets one entry into the pool of candidates for selection by the `walk` algorithm. An edge with a weight of 5 gets five entries into the pool. A final edge with a weight of 2 gets two entries into the pool. From the combined pool (with eight entries), one is chosen as a random draw and executed.

Chapter 3

High Volume Automated Testing and Long Sequence Regression Testing in Context

Yeager is just one implementation of a technique called long sequence regression testing in a family of advanced testing techniques called High Volume Automated Testing, which we refer to sometimes as “High Volume Test Automation” and “HiVAT”. This chapter aims to capture the wider context that Yeager exists in despite the relative novelty to academia that the field of high volume test automation provides: first through a detailed anatomy of what qualifies as such a test, second through a discussion of the craft’s history as we are able to research, third through an overview of the family tree, and finally through a defense of the usefulness and reasonability of Yeager’s particular branch.

3.1 Anatomy of a High Volume Automated Test

High Volume Automated Tests are software tests that, rather than exercising a particular set of preplanned scripts as a verification of a specific requirement's compliance, algorithmically generate, execute, evaluate, and potentially summarize the results of arbitrarily many test actions on a system under test, in such volume as to attain some or all of the following goals:

1. Exceed the volume of a reasonable testing staff to do manually.
2. Expose behaviors of the system not normally exposed during traditional testing techniques, e.g. through loading and stressing the system in a manner akin to when a large numbers of users may interact with it.
3. Simulate (ab)use of the system more realistically and dynamically than would be attainable through traditional techniques.
4. Generate test scenarios that, while difficult to imagine by a tester, are not outside the realm of possibility or even probability due to the high-availability nature of modern software systems.

While this is an extremely broad definition, this is due to the fact that these goals can be met through the arrangement of a large number of different test design factors. We propose six degrees of freedom, many driven by technology choice, that combine exponentially to create a diverse and comprehensive family tree of test techniques.

3.1.1 Generation: What Actions Are Taken

The first degree of HiVAT test design freedom we propose is that of the test's generation. This is essentially an engineering question: how will the application under test be driven? In the case of yeager, tests are generated through the random selection and arrangement of pre-defined test code snippets according to a state model. Other HiVAT techniques may instead replay recorded user interactions, or send a random stream of input to the system, or drive the system through any other method.

3.1.2 Interface: Black Box vs. White Box

A prerequisite to the above degree of test design freedom, and one that itself drives a degree, is that of the testing interface. Tests, based on the tester's ability or intent to interact with the source code or build artifacts of the system under test, may be treated as white/glass box, or black box tests. The tradeoffs associated with testing decision are well-studied, and both have valid uses.

Despite the techniques' names' implication, this is actually not a black-and-white issue: Meaningful tests can be achieved by interacting with the internals of some classes of program while still treating these interactions as black-box, for instance by interacting directly with the http APIs associated with a web application under test without dealing with the weight of the web application's user interface, potentially simplifying or accelerating the test's design and execution.[Hoffman, 2013]

3.1.3 Oracle: Determining Correct Behavior

Purely manual testing verifies compliance through tester observation and judgement. That is, while a manual tester is working through their test script, they are looking for known indications of noncompliance, for instance bad application behavior, strange output, or system crashes. The high volume and speed of testing present in any automated test, let alone a high volume automated test, precludes human observation and therefore necessitates a computer "oracle" to verify the application under test is behaving correctly.

In the specific case of automated unit tests and related non-high-volume test techniques, testers write checkable assertions about the state of the system at various points during the test's script, which if found to be untrue are reported and treated as a test's failure. High Volume techniques may incorporate different oracles than just these assertions (though, notably, yeager doesn't). For instance, a HiVAT technique might compare the output of a system to the output generated by a previous version of the system, or a competitor's system, or a (simplistic) alternative implementation built by the tester. Or, a technique might entirely ignore the state of the system under test and might instead focus on observable metrics like system CPU load, or memory usage, or response time, or disk write patterns. The problem summarized as "How do we know if the program is behaving correctly?", also known as the "oracle problem", has many context-driven solutions, and drives this proposed degree of HiVAT design freedom.

3.1.4 Loggers and Diagnostics: What Happened?

Diagnostics are a unique case in that, while they can act as an oracle themselves, they overlap with the fundamental question (and degree of HiVAT test design freedom) of how the tester is to determine what circumstances fed into a detected fault.

There's a broad answer, in logging, but what kind of data gets logged and what form these logs take falls to the discretion of the tester. Some tests might need detailed accountings of allocated memory bytes for every millisecond recorded, while others might get by with just a summary list: "function X got called 5623 times while function Z got called 2383 times". It might be the case that a log serves as a future test plan- e.g. a test tool consumes its own logs to replay a previously failed test, or a tool might just dump executable test code itself.

3.1.5 Testing Context: Cornering/Surveying/Abusing

No test exists in a vacuum, even spacecraft tests for NASA. Every test sets out to prove to some stakeholder(s) that some requirement(s) are met to their satisfaction.

HiVAT techniques are incredibly versatile in that they can be used as tools to show compliance with requirements that were very difficult or expensive to verify under traditional techniques. Their adaptation can help the tester to accomplish goals like ferreting out reliable reproduction cases for intermittent "Heisen"bugs, or generating leads on the breadth of bugs that might have been introduced in a new major revision, or tick the box on seemingly absurd-to-prove requirements like "the system shall not exceed a response time of 300 milliseconds more than twenty-five times in thirteen million requests over the course of a one-hour duration".

These testing goals, driven by the testing context, define a degree of HiVAT design freedom that exists at a higher level than the engineering and technology questions previously described.

3.1.6 Scalability: Parallelized vs. Sequential

The notion of "high volume" does not necessitate the "high density" that is implied by the high availability of modern supercomputing, particularly inexpensive cloud clusters or other massively parallel systems. That said, many systems under test are already relatively well-suited for this kind of testing, especially web API systems, and a compelling case can be made for the adoption of cloud testing techniques.[Parveen and Tilley, 2010] Other requirements might explicitly preclude the usage of massively parallel testing techniques, for instance testing long-term use of a single-workstation native application.

This is perhaps the most limited of these six degrees of HiVAT test design freedom in that, while the decision is already made by the requirements of above design decisions, there are still (many) cases where both can apply, and it becomes a design decision unto itself.

3.2 On the History of High Volume Automated Testing

It's hard to discuss the history of this family of techniques due to their origins within the field, especially as the techniques are rarely a component of a product and instead are essentially company-internal development practices- practices which companies defend jealously so as to preserve their perceived competitive advantage.

There's so little real recorded scholarship in this field, possibly driven by the fact there aren't many advanced software testing degrees offered at major universities, that occurrences of rediscovery are seemingly commonplace. For instance, Dawson [2014] reported an interesting high volume test in which a new CPU iteration and associated software was suspected to have a bug in floating point math functions, tested when the author decided to enumerate all of the errors in the range of float (four billion or so) by iterating over each value and comparing the output of the functions under test to a known-good implementation. The author reports detecting 864,026,625 inconsistencies in the span of about ninety seconds on a computer running Microsoft Windows, which would later be used to compose the blog post reporting the test. This was a revolutionary test report among the testing community, gaining massive traction on HackerNews and Reddit, and as one commenter pointed out, "15 years ago this probably would not have been terribly practical. 25 years ago even thinking of it would have been absurd."

This is, of course, ignoring Hoffman [2003]'s report of a test in which a new CPU iteration and associated software was suspected to have a bug in floating point math functions, tested when the author decided to enumerate all of the errors in the range of float (four billion or so) by iterating over each value and comparing the output of the functions under test to a known-good implementation. The author reports detecting 2 inconsistencies in the span of a few minutes on a parallel system about twenty times faster than a single-CPU machine of the day, which would later be used to analyze and correct images transmitted by the Hubble Space Telescope prior to OV-105 (Space Shuttle Endeavour)'s 1993 lens replacement mission, STS-61.

An absurdity, according to Reddit, but a published absurdity, which should remind us that all of this has happened before, and will happen again.

3.2.1 HiVAT Has Been Invented Six Times

In fact, despite the reality that the first HiVAT techniques we could find in academic literature were invented one "dark and stormy night" [Miller et al., 1989], many anecdotal HiVAT-linkable testing regimes were in use within the software engineering industry as early as 1966 with Hewlett-Packard's unfortunately-yet-aptly named "Evil" program whose story is now lost among the millions of sufficiently dramatic modern consumer complaints about the quality of their printer drivers. In fact, Kaner [2013] reports several instances of HiVAT-ey testing policies, including the 1987 implementation of "ideas implemented years ago at other telephone companies", long-sequence regression testing of a word-processing application in 1984 or 1985, and a 1991 company with a popular commercial product designed to generate many fake phone-calls to enable load testing of call center systems.

In the face of Kaner's collection of anecdotal reports of HiVAT techniques used at such companies as Texas Instruments, Microsoft, AT&T, Rolm, and other telephone companies, as well as usage in verifying FAA systems and systems at automotive manufacturers, it becomes clear that HiVAT has a rich and diverse history in industry, with many lessons to be learned from past testers. But why, then, does the academic world think the concept was invented one dark and stormy night in 1988?

3.2.2 Every Industrial Inventor Thinks it's a Trade Secret

It's the very nature of the software engineering industry that companies aim to ship products as quickly as possible and with as few bugs as they can tolerate. In fact, many managers within the industry view maintenance-related tasks (such as testing) to be a problem in need of fixing, something which is desirable to minimize, even though it is, in reality, a solution to the problems posed by external requirements, normal development faults, and human processes.[Glass, 2002] It's not a crazy thought, though, as any manager would be severely concerned by the notion that roughly 80 percent of their company's development effort goes into maintenance-related tasks, and certainly seeking to cut costs in that department.[Pigoski, 1996]

Imagine a situation in which an enterprising tester has discovered a way to run thousands or millions of tests on a product in the span that a small team could previously do maybe a few dozen. A test manager whose first thought was to write an academic report explicating what exactly the process does and how it works so that others could apply these fascinating techniques in their own development practices would be only fired on a good day, and most likely litigated into oblivion by any rational company.

3.2.3 A Call for Academic Consideration

It is therefore not surprising that history became legend, legend became myth, and some testing techniques that should not have been forgotten passed out of all knowledge. There are certainly many valuable lessons still to be learned from these historical testers, and it's unlikely that this perverse incentive which drives the secreting away of advanced maintenance practices will go away in the foreseeable future.

The task, then, falls to academia to find and preserve knowledge of these practices and probably many more in related niches of software engineering. The author is not aware of any professional software engineering historians, but it seems this is a rich area of further research. Perhaps, even, a career.

3.3 The High Volume Test Automation Family Tree

Despite the relative dearth of well-documented high volume automated testing prior to 1989, a large number of techniques have grown and spread since Miller's introduction of Fuzz Testing. In this section, we introduce a few well-known ones and attempt to reconcile them within some of our six degrees of HiVAT test design freedom.

3.3.1 Long Sequence Regression Testing

One of the simplest methods for getting started with High Volume Automated Testing is that of Long Sequence Regression Testing, wherein testers modify an existing suite of regression, integration, or functional tests such that no "cleanup" is performed and state is preserved between individual test case execution, then modifying the test runner to randomly call these test cases until the system crashes. Such modifications should only take a few hours at most assuming a sufficiently stable suite, and offer a relatively straightforward way to exploit HiVAT's ability to expose obscure bugs through simulated (ab)use of the system under test. The notion, here, is to generate scenarios that testers wouldn't think to test for, but which might happen and therefore are important to verify before users get their hands on it. The kinds of bugs that this technique finds manifest as a test case that ran fine the first time, ran fine the twentieth time, but crashes the system the instant the `put_call_on_hold` method gets called for the twenty-first time.

In the case of LSRT, the oracle is the set of assertions built into the test suite, the generator is the modified test runner, and the testing context is surveying the system for bugs. This can be either a black box or a white box test depending on the type of suite to be modified.

Yeager shares a lot in common with this method since it, too, depends on existing test code assertions, acts as a replacement test runner, and is good for surveying the system for bugs. However, Yeager is advantaged versus this method since it is aware of the system's state and can consequently have more fine-grained and powerful execution than traditional LSRT implementations which treat the system as a single-state machine.

3.3.2 State Model Testing

Speaking of states, state models, also referred to as finite state machines, are useful abstractions to describe the behavior of complex systems not just in the realm of software engineering, but also in circuit design, communication protocols, and many other electrical and computer engineering tasks. One application of these models is known in the practice of electrical engineering as conformance testing, in which a model is treated as a specification of a system, and input sets are generated systematically and algorithmically, then used to verify conformance to the specification by monitoring expected outputs of the system under test.[Lee and Yannakakis, 1996]

The oracle, in this case, is the model and the theorems of system operation that can be generated algorithmically from it. The generator is an algorithm that consumes the model and generates input sets from it, the system under test is treated as a black box, and the testing context is attempting to survey the system under test to find behaviors of noncompliance. To be clear, this is an incredibly powerful testing procedure for this testing context, because the model is an absolute specification from which every single valid input can be theoretically derived. It is, however, prohibitively difficult to provide a fully specified model for many modern software systems, and generating one is an engineering task that vastly exceeds traditional testing tasks in terms of time and resources consumed for comparatively little gain in terms of meaningful software validation.

Yeager incorporates parts of this technique insofar as users build a state transition model of the system under test which is explored systematically (via a weighted random walk algorithm), but it's a limited implementation of a FSM as defined by Lee and Yannakakis. Yeager state transition models have no use for the Input and Output sets from their definition, and consequently disregards the λ set as well, which is comprised of the FSM's output functions. Yeager test code, however, is not necessarily ignorant of these three sets, as output checks are inherently provided by assertions built into the tester's provided state transition methods, and a global state context (for memoizing the program's Inputs and Outputs so far, for instance) can be provided by the kwargs-storing-and-forwarding feature built into Yeager.

3.3.3 Exhaustive Testing

Deferring to Dawson's and Hoffman's examples above, exhaustive testing is a method of HiVAT in which a specific program function is identified for testing, and all supposedly valid inputs are fed to it so as to identify any specific values or ranges of values for which the implementation is incorrect. Usually, a known-good implementation is used as the oracle in these kinds of test.

Regardless of oracle, this method is usually only for identifying specific bugs in specific low-level functions, as each additional degree of input freedom (parameter) increases the testing space exponentially. Yes, a test of a function which takes one 32-bit floating point parameter is eminently exhaustible, but one that takes a set of five such parameters and requires 1.4615016e+48 unique test cases could not be completed in our lifetimes on a computer. The application of program slicing [Gallagher and Lyle, 1991] or other forms of analysis to determine whether any of these parameters are independent (reducing the increase in search space from an exponential one to an additive one) is not unreasonable and certainly a consideration of the typical "back of the napkin" calculations that might lead a tester to adopting this method. However, after a certain point this becomes a formal verification problem and not a testing problem.

3.3.4 Fuzz and Other Monkey-Based Testing

Miller's Fuzz tester was a program that produced a random string of characters as input to UNIX command line utilities, purportedly inspired by a noisy dial-up phone line one dark and stormy night, monitoring the programs receiving the input for crashes or abnormal exit conditions. It's not unreasonable to imagine a random stream of input crashing a program eventually, and many of us fear the eventuality of a million monkeys banging on our program's metaphorical typewriters. This is especially popular among security researchers, who find the toolkit useful for identifying program vulnerabilities, though many bugs detected through other methods also may have security implications.

Though the command line is rapidly leaving the realm of important interfaces to test relative to web and GUI interfaces, Fuzzing and its ilk have a place in the HiVAT family tree, especially as the technique inspires new tools for these new interfaces.

3.3.5 Load or Performance Testing

Load testing, sometimes referred to as performance testing is most at home in the context of web API testing. To verify an API conforms, testers will manually compose some queries that test the positive path, normal usage and behavior, and then maybe some queries that should be invalid or blocked or discarded so as to verify these safeguards. The test then executes each of these queries, verifies the response is as expected, and terminates. Load testing is the practice of executing arbitrarily many of these queries though the use of some sort of parallelization tool, gathering response time and other performance metrics along the way, so as to verify that the system is able to handle the load of many users simultaneously.

One such open-source tool, Locust[Heyman et al., 2011], is designed to scale to simulate millions of users and has been used extensively in industry by such companies as EA/DICE who claim the practice is “a mandatory part of the development of any large scale HTTP service built at DICE at this point.” This technique is not great for ferreting out new functional bugs so much as it is at finding bugs related to massive loads. It also serves as a way to verify that system performance scales in a way consistent with operational expectations, such as server managers and bandwidth purchasers. It is inherently a black box technique.

3.3.6 Testing in Production (Safely!)

A test can be thought of as a scientific experiment in that the test's goals can be thought of as a hypothesis, for instance, “this new iteration of the software doesn't break when you do this thing”, and the test seeks to prove or disprove the hypothesis, usually by trying to do the thing. There's even control groups when you think of things like the previous version of the software or the requirements document, or a set of competing bugfixes.

In other fields, psychology for instance, humans are the determining factor in experiments. A recent trend uses unwitting human software users as the ultimate oracle in tests of things that are harder to verify by computer. These are less integrated into the classical testing phase and more related to deployments in many cases. A Microsoft engineer on the Bing team once described to Florida Tech's school of computing seminar course their practice of staged rollouts, first by serving all users responses from the older version of the software but duplicating a portion of live queries to check the new build for crashes and to compare output of the old and new version, then by slowly transition incoming queries over to the new build and verifying user and program behavior doesn't change much, and then finally rolling the new build out to all users. At any step along the way, failure can trigger the immediate rejection of the modified candidate and reversion to the previous software iteration.

In this case, actual live user queries are the test generator, their observed behavior compared to typical behavior on the previous iteration is the oracle, and the context is verification of seamless and successful deployment of the new build. Testing In Production, previously a joke about insufficient test practices, is a valid and valuable testing technique through the lens of High Volume "Automated" Testing.

3.3.7 A/B Testing

The notion of Testing In Production has applications in other fields than just software engineering. Marketers have adapted the technique to hone their advertising campaigns, and a culture of live experimentation has prevailed in many fields. The process is simple: two or more competing yet functionally equivalent versions of, say, an advertising email, are created, and released to a subset of the audience, maybe one percent each of the whole mailing list. Test operators watch user interaction via link clicks, determine which version is performing better, then proceed to continue to distribute the winning message to the whole mailing list.

The process can be adapted to a wide number of scenarios: site homepage layouts, app welcome screens, search advertisements (surprise, Microsoft does this with Bing), or even which stories get presented on a news site's frontpage. It's a well-accepted practice, with companies performing millions of such experiments each year. [Kohavi and Thomke, 2017]

3.4 The Case for Model-Based LSRT

Compared to these established techniques, Yeager's usage is unique in that it straddles two exiting techniques, state model testing and long sequence regression testing. As discussed above, these two techniques are powerful ones when testers are hoping to survey the system for bugs. Particularly, they find bugs by simulating usage of the system that, in theory, should be valid, but, in practice, aren't.

There's a tradeoff between the two techniques, in that LSRT is quick to implement and leverages an existing test suite, but treats the system as a single-state machine thereby limiting the granularity and novelty of the test's execution. Contrarily, SMT necessitates the huge investment of the creation of a detailed system model, a model which might not even be feasible to implement, and has no ability to leverage an existing test automation investment. However, SMT's detailed accounting of the system's specification enables the application of systematic methods for test generation and provides fine-grained verification of the system's implementation.

Yeager is meant to be a way to bridge the gap between the two techniques. It can leverage existing test code, to accelerate implementation. It generates a relatively detailed model of the system under test enabling the application advanced model traversal algorithms (even though the current implementation only provides a weighted random walk). Yeager tests respect the system's structure and more accurately simulate human usage by enabling the traversal of the system in smaller, modular chunks as opposed to the same few large, repeated cases.

Yeager represents a powerful, easy-to-adopt form of HiVAT in some scenarios, enabling discovery of otherwise-elusive bugs and better testing practices across contexts.

Bibliography

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A framework for automated testing of javascript web applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 571–580. IEEE, 2011.

Atom Open Source Project. Circleci atom builds, 2017. URL `https://circleci.com/gh/atom/atom`.

Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. Web application tests with selenium. *IEEE software*, 26(5), 2009.

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.

Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

Suhit Gupta, Gail Kaiser, David Neistadt, and Peter Grimm. Dom-based content extraction of html documents. In *Proceedings of the 12th international conference on World Wide Web*, pages 207–214. ACM, 2003.

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

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.

Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Agile Conference, 2006*, pages 6–pp. IEEE, 2006.

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

Harpreet Kaur and Gagan Gupta. Comparative study of automated testing tools: Selenium, quick test professional and testcomplete. *International Journal of Engineering Research and Applications*, 3(5):1739–43, 2013.

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

Vidar Kongsli. Security testing with selenium. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 862–863. ACM, 2007.

David Chenho Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 111–120. IEEE, 2000.

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

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 108–113. IEEE, 2013a.

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 272–281. IEEE, 2013b.

Chien-Hung Liu, David Chenho Kung, and Pei Hsia. Object-based data flow testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 7–16. IEEE, 2000.

Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130. IEEE, 2008.

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.

Hung Q Nguyen. *Testing applications on the Web: Test planning for Internet-based systems*. John Wiley & Sons, 2001.

Ray Nicholus. Understanding the web api and vanilla javascript. In *Beyond jQuery*, pages 19–29. Springer, 2016.

Finn Årup Nielsen. Python programmingtesting. 2014.

Den Odell. Browser developer tools. In *Pro JavaScript Development*, pages 423–437. Springer, 2014.

Ashwin Pajankar. *Python Unit Test Automation: Practical Techniques for Python Developers and Testers*. Apress, 2017.

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.

Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

Rosnisa Abdull Razak and Fairul Rizal Fahrurazi. Agile testing with selenium. In *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, pages 217–219. IEEE, 2011.

Martin Sandström. marteinn/selenium-python-boilerplate: A boilerplate for running selenium tests with python. https://github.com/marteinn/Selenium-Python-Boilerplate, September 2015. (Accessed on 07/05/2017).

Xinchun Wang and Peijie Xu. Build an auto testing framework based on selenium and fitnesse. In *Information Technology and Computer Science, 2009. ITCS 2009. International Conference on*, volume 2, pages 436–439. IEEE, 2009.

Web Hypertext Application Technologies Working Group. Document object model standard. https://dom.spec.whatwg.org/, June 2017a. (Accessed on 07/04/2017).

Web Hypertext Application Technologies Working Group. Html standard. https://html.spec.whatwg.org/, September 2017b. (Accessed on 09/14/2017).

World Wide Web Consortium CSS Working Group. Selectors level 4. https://drafts.csswg.org/selectors/, August 2017. (Accessed on 09/14/2017).