Qubot: Designing a Bot to Perform Autonomous Black Box Testing

Anthony Krivonos
ak4483@columbia.edu
COMS E6156 Topics in Software Engineering

Thanks to advancements in peripheral software developer tools, code is being written faster than ever before. With this comes an unforeseen problem—who's going to be testing all this new code? We hereby present a groundwork for Qubot, an autonomous black box testing library that explores websites under test (WUT) using Q-learning and other machine learning (ML) paradigms. We thoroughly reviewed several papers that propose (1) architectures for automated testing suites and (2) ML techniques for exploratory testing, and then compared these approaches on the bases of *portability*, *feasibility*, and *intelligence*. Next, we performed small-scale experiments to help select the best approaches to incorporate in our hypothetical Qubot library. We finally selected the software architecture from the *Automated Software Testing Framework for Web Applications* [13] paper and the *DRIFT: Deep Reinforcement Learning for Functional Software Testing* [14] paper as the building blocks for Qubot.

I. INTRODUCTION

No matter how advanced our style checkers, bug-fixing plugins, and continuous integration services are becoming, we, as developers, will never truly be barred from writing buggy code. Simple bugs often fall through the cracks of our IDE plugins and continuous integration/continuous delivery (CI/CD) services, causing us hours of excruciating pain as we have yet to realize our tests have failed because of a missing semicolon. Even worse, the time it takes us to write these tests often exceeds the time we spent actually writing code, leading us to feel unrewarded for our efforts and, in the worst case, remorseful for choosing to pursue a career in software engineering in the first place. While we cannot give you back the time you spent fixing *that* bug nor offer you a miraculous software engineering position where you're free from writing tests, there is at least one classic thing we, as developers, can do about tests—automate them.

In this paper, we will be examining several approaches to building an automated test suite called Qubot that performs exploratory black box tests autonomously. Whereas traditional automated test suites require developers to painstakingly write test functions and scripts that mimic user interaction with the software under test (SUT), we will attempt to design Qubot such that there is minimal setup and configuration required for the developer to perform exploratory tests.

A. What's in it for me, the reader?

We will not only examine several papers in both the automated testing and autonomous testing fields to create our hypothetical software suite in this paper. Rather, we will also present common software testing approaches used in the field, such as exploratory and black box testing, and backgrounds in algorithms related to autonomous software testing from a perspective that the reader has basic knowledge of machine learning and software engineering. Background on the different types of software testing will be provided in this section, and background related specifically to each paper will be provided in the paper's respective section. In presenting each paper, we will take an open-ended approach to examining the pros and cons of ideas expressed in papers as if we were actually developing the Qubot framework from scratch. We will attempt to inspire the reader to ask questions like

- How do portable automated testing frameworks interface with their testers?
- How do autonomous white box tests differ from black box tests?
- How do machine learning techniques for autonomous software testing compare?

Before we devise a game plan for the creation of Qubot, we will first examine the various genres of testing mentioned in this paper, as well as the difference between "automated" and "autonomous" testing.

B. What is exploratory testing?

There are many approaches to testing software after it has been developed. Unit tests, in which individual portions of an SUT are tested, and integration tests, in which the collaboration between various units of software are tested, are two of the most common testing techniques, classified as *scripted tests*. In scripted testing, developers code test cases and expected outputs from predefined *acceptance criteria* and user stories. [32] Acceptance criteria are requirements necessary to be fulfilled in order to consider a user story complete. [27] *Acceptance testing*, likewise, is a level of testing used to ensure acceptance criteria are fulfilled for an SUT and that the software may be acceptable for delivery. [30]

In *unscripted testing*, a tester is given virtual autonomy in deciding how to test software, often relying on

the tester's experiences, knowledge, and skills to determine whether or not a release of an SUT is acceptable. In this kind of testing, no test cases nor documentation are prewritten and no preparation is required. [25] Though unscripted testing may sound like a lazy alternative at first, it is, in fact, a practical approach under resource constraints and when acceptance criteria is poorly defined. Another use case for unscripted testing is when a tester does not have direct access to an SUT's source code and is unable to write test scripts within the SUT's codebase. The tester must hence treat the software as a black box and perform *black box (behavioral) testing* on it. In this level of testing, the tester examines a software's performance based on inputs and outputs and acceptance criteria, rather than on the underlying code structure. [31] Black box testing and acceptance testing are thus not mutually exclusive, often being employed simultaneously when acceptance criteria is known but the code structure is not.

A common example of a black box test is an *exploratory test*. In such a test, the functionalities of an SUT are tested in an ad-hoc manner through manual exploration of the SUT, while test cases are determined during, not before, testing. [10] Exploratory tests often expose bugs that test scripts cannot recognize on their own due to the rigidity of scripted testing. These tests are not perfect, however, as testers often find it difficult to reproduce bugs and record testing branches covered during manual exploratory testing. Computers, unlike humans, don't struggle as much with recall, and such issues could be alleviated by outsourcing exploratory testing to a computer algorithm, rather than a QA-testing coworker.

Our motivation for Qubot, which is a play on "Qu(ality Assurance) Bot," is the human *QA Tester* who checks software for bugs and issues prior to deployment and is often responsible for performing exploratory testing. [17] Just like in scripted tests, we hope to automate Qubot through a notably simple test script, but we would also like it to autonomously explore an SUT as if it were a black box, just as real QA Testers would. The difference between "automation" and "autonomy" is crucial to understanding this paper, and so we will be covering this distinction next.

C. "Automated" vs. "Autonomous" Testing

Automated testing is a form of testing in which automated tools and libraries are used to run test cases. [35] Some of the most popular automation tools include Selenium WebDriver, which simulates webpage interactions [29] and Cucumber, a behavior-driven tool that tests software systems using test descriptions written in human language. [5] In most cases, automated testing frameworks are used for scripted testing, such as unit and integration testing.

An *autonomous test* is a form of automated test that uses machine learning techniques such as reinforcement learning to examine software functionality just as a human tester would. [15] These tests are, by definition, unscripted, and are thus useful for acceptance testing software in an exploratory manner. Benefits of autonomous testing suites include a reduction in human error caused by predefined test scripts, fewer hours of developer labor, and the ability to separate the SUT and its testing codebase entirely, for portability. For these reasons, we chose to design Qubot as an autonomous testing library.

D. Design Goals and Evaluation

Dozens of papers published since 2018 on both automated and autonomous testing provide potential building blocks for the automated testing suite we will be designing. We seek to incorporate the most performant and practical solutions from ten examined papers in order for Qubot to perform the following functions:

1. autonomous SUT exploration and
2. textual input generation.

The papers have been split into two categories—automated testing frameworks and techniques for exploratory software testing—with some overlapping literature. Papers that propose architectures for software testing suites include *Automated software testing framework "Stassy"* [20], *Automated Software Testing Framework for Web Applications* [13], and *AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests* [16]. These papers will be examined and compared for the best approach for structuring the Qubot testing software and for constructing the interface through which developers will be interacting with the hypothetical framework.

The next few papers will focus on *white box testing*, a testing approach where the internal code structure is known to the tester, using autonomous methods such as genetic algorithms and ant colony optimization. Whereas we will not be able to incorporate these techniques directly into Qubot for the reason that these are white box tests, exploring these papers will still provide valuable insight into contemporary autonomous testing techniques. The examined papers in this category are *Dorylus: An ant colony based tool for automated test case generation* [3], *NEAT Algorithm for Testsuite generation in Automated Software Testing* [28], *Algorithm or Representation?: An empirical study on how SAPIENZ achieves coverage* [23], and *A Buffered Genetic Algorithm for Automated Branch Coverage in Software Testing* [22].

After determining the structure and developer interface of the library, several different approaches to autonomous exploratory testing will finally be explored through the *Fastbot: A Multi-Agent Model-Based Test Generation System* [4], *DRIFT: Deep Reinforcement Learning for Functional Software Testing* [14], and *Humanoid: A Deep Learning-based Approach to Automated*

Black-box Android App Testing [21] papers. A graphical comparison of each paper is depicted in Figure 1.

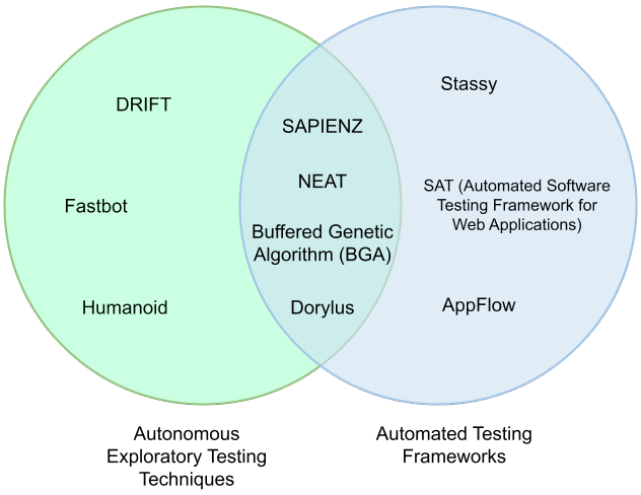

FIG. 1: Venn Diagram Comparing Papers Examined for Qubot

After each paper is examined, they will be put head to head in a set of minor experiments. Qubot will finally be evaluated on the following criteria through these literary comparisons and experiments:

- **Portability**: Can Qubot work on any SUT without changing the codebase, and can it work across SUT versions?
- **Intelligence**: Can Qubot learn to generate inputs on its own and to navigate its environment?
- **Feasibility**: Is Qubot efficient or practical enough to be used as the sole acceptance tester for even a small portion of a codebase under test?

II. ARCHITECTING A PORTABLE TESTING LIBRARY

As previously mentioned, our first step in designing Qubot is to choose a software architecture suitable for autonomous software testing with a simple interface for testers. Whereas the papers in this section do not specifically mention black box testing, the proposed architectures may be useful for a variety of different testing goals.

A. Stassy: A Highly Portable Automated Testing Framework

The first paper in our study, Stassy, presents a Java framework that is built atop Selenium WebDriver to expedite test creation by both technical and non-technical product owners. This structure, as the authors describe, promotes code reuse, ease of maintenance, and portability, and its organization is described package-wise. [20] The *Custom Exceptions* package contains skeleton code for QA testers to define their own errors for logging purposes. The *Main* package contains classes that call XML files, known as *TestNG* files, that define the test configuration, such as the test cases to run and their run order. The *Object Repository* package contains stub classes that access and manipulate DOM (Data Object Model) elements in the browser. The *Reusable Functions* package contains reusable test functions where developers can define their own methods, and the *Utilities* package contains arbitrary configuration files used for setting up a Stassy test. Finally, the *Test Cases* package contains the main test functions and scripts defined in the TestNG file.

After the TestNG file and test cases in the *Test Cases* package are coded out by a developer or a tester, the test cases are called through this XML file. The functions from the *Reusable Functions* package, which contain Selenium WebDriver methods that perform actions in the browser, are then accessed for use in the test cases. If an error occurs during testing, it is then captured as a *Custom Exception*.

The authors describe saving 1.3 hours of testing a Google Web application when using the Stassy framework, in contrast to using Selenium WebDriver on its own, as shown in Figure 2. They do not mention whether this reduction in hours of labor is a direct comparison of the test suites' runtimes or of the time spent configuring each testing scenario.

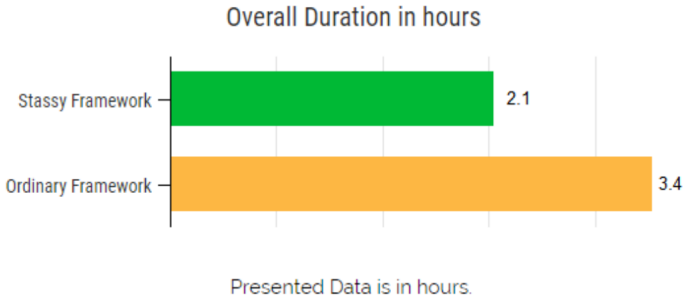

FIG. 2: Visual Comparing Stassy vs. Selenium WebDriver [20]

Despite laying out the structure of the framework, the authors unfortunately lack information on how to use the Stassy framework from a tester's perspective and miss key details about the purposes and functions of each package—keeping their explanations to a bare minimum. Still, the use of the TestNG configuration file to simplify test case definition and to encourage both code reuse and portability is an important takeaway from this paper.

B. Software Automated Testing Framework (SAT) for Web Applications

The Software Automated Testing (SAT) framework proposed in *Automated Software Testing Framework for Web Applications* is another extension of Selenium WebDriver that seeks to enhance collaboration between developers and testers and make software testing easier. [13] The SAT framework, which focuses mainly on scripted website testing, aims to automate the process of generating test scripts altogether.

Unlike in the Stassy framework, the authors of SAT break down the steps taken in testing a web application. SAT first crawls the website under test (WUT) and detects all HTML elements that can be interacted with. Then, in the preprocessing step, SAT deterministically selects values to be used for each HTML interaction. This algorithm is able to determine HTML interactions based on input type, such as `ClickButton` for a `<button>` element, saving the developer from having to manually select the appropriate keyword from each HTML tag. Next, the Test Case Steps (TCS) sheet, similar to the TestNG file, is automatically generated and, finally, the tester is able to edit and use the TCS to automatically run tests on the WUT.

Generation of the TCS sheet can either be partially automated or fully automated. In the former method, the tester lists the test case steps manually within the sheet, whereas these test cases are generated automatically via crawling of the WUT in the automated method. After the test cases in the TCS sheet are available to SAT, the suite will then automatically generate the test script.

Whereas SAT is not an autonomous solution, it *is* still an exploratory one. With its ability to automatically generate test scripts, SAT was shown to have saved approximately 75% of the total time and effort involved in the automation process using traditional methods (such as testing using the Cucumber framework) and 21% as compared to using Selenium WebDriver on its own, as well as only requiring 92 lines of code in the case study within the paper as opposed to 968 lines in a traditional method.

SAT is a highly portable and feasible framework for testing software at scale, but it does not fully alleviate the tester's need from having to predefine HTML interactions, at least in the partially automated TCS generation method. Moreover, the web crawling feature is highly oversimplified, and may induce days of development work when actually developing SAT. As the source code for SAT has not been provided with the article, the test cases run during the case study are ambiguous, and could potentially be trivial to the point where they test simple things such as the ability to enter characters into a text box. All this being said, SAT aligns with the vision where Qubot requires minimal configuration from the tester.

C. A Synthesizable ML Approach to Automated Testing with AppFlow

AppFlow is another automated testing suite like SAT that dynamically generates full tests, but it differs in that it relies on machine learning techniques to classify UI widget text and contexts within Android applications (*apps* for short). [16] Moreover, unlike Stassy and SAT, it does not build up on Selenium WebDriver, but rather is a standalone testing suite for Android.

AppFlow incorporates an extension of the Gherkin [6] language, which is used to specify testing scenarios in the Cucumber platform. These scenarios, called flows, are used to script actions performed in an SUT and the resulting state changes arising from these actions. An example flow called "add to shopping cart" is shown in Figure 3.

```
Scenario: add to shopping cart [stay at cart]
  Given screen is detail
  And cart_filled is false
  When click @addtocart
  And click @cart
  And not see @empty_cart_msg
  Then screen is cart
  And set cart_filled to true
```

FIG. 3: "Add to shopping cart" in AppFlow Language [16]

Testing with AppFlow is a top-down approach. Testers seeking to prepare tests for an entire category of apps (i.e. "shopping apps") first create a test library using AppFlow's language and then use AppFlow utilities to capture a dataset of screens and widgets prior to labeling them manually. After this data has been labeled, AppFlow uses it to train ML classifiers that distinguish screens and widgets of this specific category.

To test a new app within a pretrained category, testers must use AppFlow's GUI utility to discover classification errors by the ML models and to override these errors, and then testers may extend the test flows written for an entire category to the specific app they are testing. Then, testers must run the AppFlow testing software to acquire results from the scripted tests. After initial tests, testers can later re-run these tests on newer versions of the SUT to locate regressions, or software bugs that were thought to have been fixed in previous versions.

The AppFlow framework was put through a series of studies of which three are important. In the first study, 40 shopping apps and 20 news apps from the Play Store were smoke tested using reusable AppFlow tests, whereby 55.2% of tests could be reused for the shopping apps and 53.0% of tests could be reused for the news apps. In the context of Qubot, these results implied considerable testing portability. The second study involved exploring two significantly different versions of a news app to examine AppFlow's feasibility. In this study, AppFlow correctly

recognized canonical widgets across both apps, displaying AppFlow's feature of version-independence, and completed several basic test cases across different UIs. The last test concerns a user study of fifteen testers on creating tests for a shopping app, which was estimated to take 5 hours, 29 minutes using traditional testing software. In contrast, it was estimated to only take approximately 30 minutes with AppFlow.

Among all automated testing frameworks covered thus far, AppFlow inherently provides the greatest amount of portability. That is, it allows for test reuse not only across app versions, but across apps themselves. That being said, it requires considerably more effort from the tester to configure tests, specifically labeling each UI element in a dataset and writing test cases in the AppFlow language. SAT, on the other hand, requires that the developer provides a URL of the WUT and tweaks the TCS sheet in case of errors. Development of a platform like AppFlow—which includes machine learning models, a UI-based labeling system, a programming language parser, a test scripts generator, and more—would require extraordinarily more hours of development work than the web crawler used in SAT. The differences between the three examined automated test suites will be examined in greater detail in the Experiments section.

III. EXPLORING AUTONOMOUS WHITE BOX TESTING STRATEGIES FOR QUBOT

Now that we have examined the architectures and testing procedures of multiple automated testing suites, the next step in the design of Qubot is to give it the ability to think.

The papers in this section are explicitly white box tests, in which the source code is known to the tester. Since the overall approaches from these papers cannot directly be incorporated into Qubot, we will be examining them for techniques that can be transferred to black box tests.

A. Dorylus: Using an Ant Colony Algorithm to Generate Inputs

The authors of Dorylus [3] present a tool that generates test suites using Ant Colony Optimization for Continuous Domains (ACO_R). [8] In regular ant colony optimization, a set of agents called *artificial ants* are tasked with finding the best path along a weighted graph. These ants record their paths with smelly *pheromone* trails, chemical paths that let other ants know which paths taken have resulted in finding "food" (or, in our case, a terminal state in the graph). The pheromone paths are stored in a hash table so that further iterations of the searching algorithm can be accomplished in less time and convergence to a locally optimal solution is avoided. On continuous domains, two approaches to discretizing the pheromone distribution involve slicing a continuous domain into buckets and simulating the pheromone distribution on a discrete domain via a Gaussian distribution. [12]

Dorylus uses branch distance defined by Korel in [18], and uncovered branches become primary targets for the ants in the algorithm. Each target undergoes analysis which identifies parts of the input such the mutation of this input affects the future branches traversed.

To test the Ant Colony Technique, authors put it head to head against EvoSuite in two separate trials. EvoSuite is a state-of-the-art test generation tool written for Java programs that uses an ever-evolving search-based approach. [9] Both Dorylus and EvoSuite were given 2 minutes to achieve as much branch coverage as possible for a corpus consisting of 12 programs in the first trial and a corpus consisting of 936 programs in the second trial. In the first trial, EvoSuite covered an average of 95.1% of branches, whereas Dorylus covered 97.7%. It was hypothesized that Dorylus excelled for programs with nested conditional statements, whereas EvoSuite was quicker on simple programs. In the second trial, Dorylus merely outperformed EvoSuite on 29 programs, and Dorylus's mean branch coverage of 31.29% lagged greatly behind EvoSuite's 54.56%.

As a result of Dorylus's poor performance in the second trial, the authors suggested introducing more entropy into the branch search function in order to increase branch visit percentage. The key takeaway from this paper, despite its proposal of a white box testing approach, is that considerable testing must be done in order to achieve the right combination of exploration and *exploitation*, which is the preference of already-visited branches in order traverse deeper within these branches.

B. Generating Test Suites with the NEAT Genetic Algorithm

The NeuroEvolution of Augmenting Topologies (NEAT) algorithm is an evolutionary algorithm proposed in the 2002 paper *Evolving neural networks through augmenting topologies*. [33] The NEAT algorithm describes methods for altering the weights at each node of a neural network and the overall structures of networks in an attempt to balance diversity of a solution network and its adaptation to the learning environment.

An important contribution of the NEAT paper is the proposed encoding of neural network architectures. Each network architecture has a genotype, or the genetic constitution of an organism from a biological perspective [1], and a phenotype, or the organism's observable characteristics. [2] The set of nodes within a genotype indicates whether or not the nodes are inputs, hidden nodes, or outputs, and the set of connections within a genotype indicate contain metadata about its input and output, weight, whether it is enabled or disabled, and its *innovation number*, which is used to select nodes (genes) across

neural networks that can be crossed over with one another. Ultimately, NEAT uses these encodings to perform either one of two mutations—adding a connection between unconnected nodes or adding a node in the middle of a connection—and, as we will see, these are useful operations for generating test suites.

The *NEAT Algorithm for Testsuite generation in Automated Software Testing* paper uses a multi-step process in generating test suites autonomously, doing so once again in a *white box* approach, where the source code is provided. [28] First, the tester is required to create a naïve test suite such as a simple unit test for a multi-branch function `triangleTester`, which checks to see if three points, x, y, and z, form a triangle. This complex triangle classifier program, taken directly from the NEAT paper, is shown in Figure 4.

```
#  The Triangle Classifier Program

# x : side 1
# y : side 2
# z : side 3

type_of_triangle = 'Not Triangle'
is_triangle = False

def TriangleTester(x, y, z):
        if (((x+y==z) or (y+z==x) or (z+x==y)) or (x<=0 or y<=0 or z<=0)) :
                type_of_triangle = 'Not Triangle'
                is_triangle = False
        else :
                if ((x*x==(y*y+z*z)) or (y*y==(x*x+z*z)) or (z*z==(y*y+x*x))) :
                        type_of_triangle = 'RightTriangle'
                        is_triangle = True
                else :
                        if (x!=y and y!=z and z!=x) :
                                type_of_triangle = 'ScaleneTriangle'
                                is_triangle = True
                        else if ((x==y and y!=z) or (z==y and x!=z) or (x==z and y!=z))
                                type_of_triangle = 'IsoscelesTriangle'
                                is_triangle = True
                        else if (x==y==z) :
                                type_of_triangle = 'EquilateralTriangle'
                                is_triangle = True

return is_triangle, type_of_trianlge  # pragma: no cover
```

FIG. 4: Triangle Classifier Program from the NEAT Paper [28]

Next, a seemingly arbitrary neural network with an undetermined number of hidden layers must be initialized by the tester. Once parameters such as mutation rate and the maximum number of continuous generations with no improvements in fitness are configured by the tester, the tester must define a *fitness function* and a *fitness threshold*. When the output of the fitness function on the created network exceeds this threshold, the specific network is said to be at its "fittest." In other words, the network with maximum coverage is selected.

Finally, the aforementioned network and parameters are inputted into the NEAT algorithm, which selects the fittest network. A test suite T is fed into the neural network and a newly generated test-suite N is spit out of the NEAT function. The goals of this algorithm are to obtain the maximum branch coverage and to generate the resulting test suite as quickly as possible.

Results from an experiment involving `triangleTester` concluded that this method obtained the highest possible branch coverage when run for 800 iterations. This result beats out EvoSuite, both by number of test cases required to achieve coverage (5 vs. 18) and branch coverage percentage (96% vs. 80%), having taken only 21 more seconds than the other high performer.

Although the NEAT algorithm was shown to achieve very high branch coverage via the pseudo-exploratory method of test case mutation, the NEAT algorithm does not take the burden of test case generation completely out of developers' hands due to the requirement of a naïve test suite at the start. Moreover, parameters such as the number of hidden layers in the neural network and the fitness threshold may be difficult to optimize for the average tester who does not have ample background in machine learning, making it not very feasible in the context of Qubot. Nonetheless, the use of genetic algorithms to modify test cases on the fly may be used to have Qubot regenerate the test cases it comes up with while exploring an SUT.

C. SAPIENZ: Exploring a Software Environment with a Genetic Algorithm

SAPIENZ is another software tool that uses genetic algorithms to generate test suites for Android apps. [23] It employs the *Elitist Non-dominated Sorting Genetic Algorithm* (NSGA-II) [7], a *multi-objective* optimization algorithm that produces a *Pareto Curve* with *elitist* individuals (not to be confused with pure mathematicians). A multi-objective problem is one that aims to achieve the best tradeoff between several qualities such as maximizing statement coverage and fault detection while minimizing test length, and the most dominant combinations of these qualities form the Pareto Curve. An elitist individual is a test suite that has the best traits from the parent individuals from whom it mutated. SAPIENZ replaces longer generated test sequences with equally performant test sequences to maintain Pareto optimality.

SAPIENZ frames a test suite as a multi-tiered individual. Several chromosomes (test cases) are made up of multiple genes (test events), which contain random combinations of atomic genes, which are test events that cannot be further decomposed, and *motif genes*, which are generalized usage patterns within the SUT. The motif genes are used to study common usage patterns during app exploration, which the authors unfortunately do not go into great depth about.

In a study comparing the NSGA-II algorithm plus the incorporation of motif genes, which were selected based off an unmentioned method, the branch coverage mean across eight Android apps was 47.87%, which is only slightly higher than the combination of random search and motif genes (46.95%) and NSGA-II as a standalone

algorithm (44.07%). The main takeaway from this experiment is that the multi-objective NSGA-II algorithm outperforms other algorithms with the combination of the motif genes, and, in the context of Qubot, this implies that there could be a significant advantage in pinpointing user interactions that are one-offs and those that are repetitive or pattern-introducing. Unfortunately, the small quantity of SUTs in the study is a great threat to validity of the SAPIENZ algorithm and the substantially lesser branch coverage percentage of this algorithm versus that proposed in the NEAT paper provides little inspiration to use SAPIENZ over NEAT.

D. A Buffered Genetic Algorithm for Test Input Generation

Like NEAT and SAPIENZ, the proposed Buffered Genetic Algorithm (BGA) attempts to maximize coverage, specifically of branches traversed, using genetic algorithms. [22] Another white box testing technique, BGA generates a set of candidate test inputs that are best suited to solving the problem, referred to as the *population*, and iteratively combines these solutions to find the optimum candidate. The *Path Prefix Strategy (PPS)* was used to traverse a directed graph that represents the control flow within an SUT. [26] This strategy involves using past testing sequences to guide the selection of future testing sequences.

Like in SAPIENZ, a random initial set of chromosomes (test cases) are evaluated for branch distance (a fitness measure), a subset of them are selected for crossover and mutation, and this sequence repeats. The genetic algorithm used contains a buffer space in which target data could be stored to indicate the genetic algorithm not to regenerate tests for a given branch if the branch has already been visited.

Several sample programs, including four triangle classification programs, were tested with 10 different population sizes, with a crossover probability of 0.8 and a mutation probability of 0.01. It drastically outperformed a standalone genetic algorithm that was used as a benchmark, and even 99.98% branch coverage was attained on a triangle classifier.

IV. INCORPORATING AN AUTONOMOUS BLACK BOX TESTING TECHNIQUE

The next three papers diverge from test suite generation using ML methods and focus explicitly on SUT exploration and black box testing.

A. A Q-Learning Approach to Exploration with Fastbot

The Fastbot model-based test generation technique (MBT) uses purely exploratory methods to achieve coverage.[4] First, a dynamic acyclic graph (DAG) is created either manually or through app scraping, where software states are graph nodes and actions are edges of the graph. In order to balance exploration and exploitation, the authors assign an *n-Step UCB (Upper Confidence Bounds) Value* to each node in the DAG.

Given a number of steps n, a discount factor γ which describes how important future rewards are to a learning agent [38], a parent visit count V_p, a current node visit count V_c, a current node visit priority P_c, and a configurable constant C, the n-Step UCB of the current node is as follows.

$$UCB_c = \frac{\sum(\gamma^i \sum P_{ci})}{V_c} + C\sqrt{\frac{\ln V_p}{V_c}}. \tag{4.1}$$

The DAG is then augmented into a *Monte Carlo Search Tree*, or *MTree* for short, in which each node stores the actions that can be taken in order to navigate to each child node from the current node. When a new state is discovered, the tree is backpropogated in order to update the number of new remaning actions, or activities, from the parent. An example MTree from the paper, where each color represents a different activity, is shown in Figure 5.

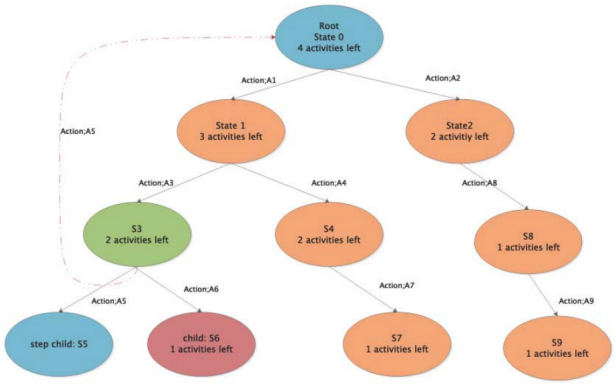

FIG. 5: Example MTree Used by Fastbot

Fastbot uses a supervised ML technique called Q-learning to traverse the DAG/MTree pair. The *Q-function*, which is the function that influences an agent's navigation, calculates the quality of a state/action combination within an environment, and this quality is determined by the reward an agent expects to receive by performing an action at a given state.

To enable intelligent navigation of the DAG/MTree pair, a Q-learning function was incorporated with a reward function **R** such that visiting newly discovered states accrues a positive reward and visiting states that

have already been seen issues a negative reward value. Given a *learning rate* α, which influences the extent to which newly acquired information outweighs prior information, a current (*state*, *action*) pair (s, a), and a future pair (s', a'), the Q-function update rule used by Fastbot is as follows. [34]

$$Q(s,a) \mathrel{+}= (1-\alpha)Q(s,a) + \alpha(\mathbf{R} + \gamma Q(s',a')) \quad (4.2)$$

Compared to Humanoid, a deep learning-based automated black box testing suite that will be discussed later in this section, Fastbot achieved 10.27% faster code coverage per hour of test. That being said, in a comparison between a non-Q-learning approach composed of just MTrees and UCB values, a hardcoded algorithm that traverses the MTree converged substantially faster in terms of activity visits, as compared to the Q-learning solution. In fact, the Q-learning solution was the least performant in the benchmark run by the authors of the paper, raising the big question of whether fully autonomous exploration of software is feasible. The authors, unfortunately, failed to give any insights as to why non-ML algorithms, as a whole, performed better than the Q-learning approach. However, the simplicity of this approach invites further investigation into using Q-learning for exploration.

B. Another Q-Learning Approach to Exploration with DRIFT

DRIFT, which stands for Deep Reinforcement Learning for Functional Software Testing, is a more refined Q-learning approach to autonomous software testing of the Windows Operatnig system. [14] On top of Q-learning, the paper incorporates *off-policy* learning, which is the process of splitting the exploration policy and the policy being learned, and *Batch-RL* which is a subset of off-policy learning where an agent uses a (often pregenerated) fixed corpus during training. [11] Moreover, *graph neural networks*, or GNNs, were used during training. A graph neural network is a neural network that takes in graphs as inputs, rather than discrete input vectors, and the dependence between graphs is captured via message-passing between the nodes in the graphs. [39]

The goals of the authors were to test the System Settings App by having an agent navigate to the notifications panel and by having the agent navigate to the menu where it can add a bluetooth device. Additionally, they sought to train the agent to add a page to Favorites in the Edge browser. Like in the Fastbot paper, the authors used a *UITree* with between 2 and 842 total states and various user actions to be taken from these states. The method the authors used was made pretty simple thanks to the UI Automation Tool available on the Windows platform. The tool records user interactions in a JSON format shown on the right side. Each interaction is connected to other interactions, outputting what the authors call a UITree. With the UI tree taken directly from the automation tool, the authors were able to make this setup fully portable, a desired goal for Qubot.

To speed up the training process, the authors ran the UI Automation Tool to pre-generate the test scenarios, and then cached these scenarios a corpus to be used for both training and testing.

Instead of manually learning a Q-function similar to that in Fastbot, the authors opted to estimate it via a GNN. The authors unfortunately omitted details as to why they chose to generate a Q-function this way.

The authors also glossed over their committing of what some other authors had called the "Deadly Triad," which is the combination of estimating the Q-function, approximating the distribution of action-state pairs, and using off-policy learning. This was shown in the *Deep Reinforcement Learning and the Deadly Triad* paper to induce unbounded and inconsistent results. [36]

The newly created UITrees were converted to *one-hot-encoded* vectors using their action and a unique index. One-hot encoding is the process of converting categorical data into an array of 0s and 1s, with each index corresponding to whether or not it a data point fits a given category. These vectors were then used to train the GNN.

The authors chose to split their experimentation into two algorithms at first, which were the DRIFT-Greedy algorithm and the DRIFT-Sampler algorithms. The DRIFT-Greedy algorithm, which maximizes expected reward, seeks the shortest path to a desired termination state. The DRIFT-Sampler algorithm, on the other hand, selects actions using a random policy and is used to explore the various paths that can be taken during testing.

After experimentation on the previously mentioned test flows, DRIFT-Sampler performed surprisingly well. It achieved about 400 as many rewards at *low temperatures*, where actions were biased toward maximum reward, as a random agent. That being said, it did see significantly fewer states at low temperatures. At *higher temperatures*, where actions were biased toward creating a uniform distribution, it was able to see more states but accrue fewer rewards.

DRIFT-Greedy performed very well, according to the authors, as it took a mere 2 steps to accomplish any of the aforementioned tasks 100% of the time.

A perfect task success rate indicates nothing about total application coverage, so it would be futile to directly compare DRIFT-Greedy with the Q-learning agent from Fastbot. That being said, if the goal of Qubot is to perform exploratory testing with the goals of finding bugs and accomplishing basic tasks within a small portion of a codebase, there might still be hope for using Q-learning for black box testing. We will dive deeper into exploratory testing with Q-learning in the Experiments Section.

C. Using Deep Learning for Automated Black Box Testing

Despite its promising nature, Q-learning is not the only choice of ML algorithm that can be applied in Qubot. Humanoid is an approach to test generation powered by a model trained on actual human interactions with Android applications. [21] The main component of Humanoid is a *Deep Neural Network (DNN)* [37], or a neural network with a great amount of hidden/intermediary layers and additional functionality, that predicts which interactions are more likely to be performed by human users. The other part is a random search agent that is biased towards unexplored states based on the probabilities outputted by the DNN. The set of human interactions understood by this model involved short touches, long touches, directional swipes, and text inputs. To capture generalized interaction patterns, interactions were stored as *UI context* traces. At time i,

$$\text{context}_i = (s_{i-1}, a_{i-1}), (s_{i-2}, a_{i-2}), (s_{i-3}, a_{i-3}), \quad (4.3)$$

where s indicates a state and a indicates an action.

The input to the DNN is the combination of the current state/action tuple (s_i, a_i) and context_i.

The DNN used a dataset of 304,976 human interactions, which resulted in 12,278 interaction flows belonging to a total of 10,477 tested apps. 100 of these apps were used for testing and the other 10,377 for training. For each state in the testing set, the interaction model was used to predict the probabilities for each of the popular subsequent actions and then sort these actions from most to least likely. These probabilities were then compared against real actions taken by humans under the same circumstances, and it was found that Humanoid usually prioritized such actions in the top 10% of all available actions.

Humanoid was also compared to six other Android generators—Monkey, PUMA, Stoat, DroidMate, DroidBot, and SAPIENZ—based on line coverage. Humanoid achieved a mean line coverage of 43.3%, which was the highest among all other models. Given that the Humanoid article was written a year prior to SAPIENZ, whose authors claimed SAPIENZ to be faster than Humanoid, we assume that SAPIENZ has overtaken this algorithm in the context of line coverage, which was said to be 47.87%. Another major discrepancy between the DNN approach of Humanoid and the genetic algorithm approach of SAPIENZ is the total training time. Whereas it took 4.320 hours to perform the entire experiment presented in the latter article, it took about 66 hours for the authors of Humanoid to train the model that achieves less coverage. Although code coverage is not a priority for Qubot, the substantially longer training time renders the Humanoid approach to exploratory testing highly unfeasible to the average tester. Moreover, the extensive dataset required to achieve such results in Humanoid—as compared to the more compact autogenerated training data in Fastbot and DRIFT—makes it impractical in a real-world setting wherein the pretrained Humanoid model is faced with a niche SUT that it does not understand yet how to navigate.

V. EXPERIMENTS

In this section, we will explore in greater depth the proposed automated testing architectures and testing algorithms presented in the various aforementioned papers. In doing so, we will decide on a promising design for the Qubot framework that will effectively achieve the goal of autonomous black box testing.

A. Portability and Feasibility: Deciding on a Robust Automated Testing Architecture

1. Portability Test

The first experiment is a qualitative study, using domain knowledge, to determine the portability and feasibility of the Stassy [20], SAT [13], and AppFlow architectures. The chosen criterion for *portability* is the estimated number of minutes it takes for testers to test an individual SUT with each respective framework. Since the goal of black box acceptance testing is to ensure a user is able to perform critical functions in an SUT without failures, the speed of the test becomes an important factor, as any leftover time can be used to solve critical bugs in the software.

We first examine the number of minutes to set up the Stassy framework for a given test. Since a Java class must be manually defined for each widget and page in an SUT by hand, we estimate this to take 120 minutes. Additionally, writing test cases for each of these classes will likely take the same amount of time on average. Generating reports of the results, however, should only take about 1 minute. This estimate will decrease as more test cases are added for the SUT, however, as all pages and widgets will not have to be re-developed as Java classes. The evaluation of Stassy is shown in Figure 6.

Setup Step	*# of minutes*
Developer defines all pages in SUT	60
Developer defines all widgets in SUT	60
Developer writes test cases	120
Stassy generates reports after tests conclude	1
Total	242

FIG. 6: Stassy Test Setup Estimation

Next, we will look at the time estimate for the SAT architecture. This architecture proposes partially and fully automated test case generation thanks to webscraping, and so we should see the testing time decrease drastically. The steps necessary to set up the test are estimated using Figure 2 from [13], and we will assume the

developer chooses the fully-automated procedure for TCS sheet generation.

The developer must first define the URL SAT will webcrawl, which should take about 1 minute. Crawling a few small webpages should take approximately 10 minutes, per domain experience. Then, we give SAT 5 minutes to generate the TCS sheet and the tester 35 minutes to examine and reconfigure the TCS sheet to their preference. This evaluation is depicted in Figure 7.

Setup Step	*# of minutes*
Developer defines URL to test in SUT	1
SAT crawls the webpage at the given URL	10
TCS sheet is automatically generated	5
Developer examines TCS sheet for errors	30
Developer runs test cases from the TCS sheet	5
Total	51

FIG. 7: SAT Test Setup Estimation

The Stassy setup steps are roughly in the same magnitude as those for SAT. Firstly, the tester must take screenshots of the SUT before AppFlow's ML models label widgets and screens from these images, taking approximately 15 minutes in total. Then, it takes 80 minutes for the tester to relabel any errors in classification and then to write test cases using the AppFlow language extension of Gherkin. Finally, AppFlow should take roughly 5 minutes to run all tests and to generate reports. See Figure 8 for a rundown of each step.

Setup Step	*# of minutes*
Tester takes screenshots of SUT	5
AppFlow labels widgets and screens	10
Tester manually relabels classification errors	20
Tester writes test cases in AppFlow language	60
AppFlow runs tests	5
Total	96

FIG. 8: AppFlow Test Setup Estimation

Although the above approximations were made solely with prior knowledge of the author of this paper, there is still a substantial difference in testing duration between Stassy and the other two frameworks, indicating that the latter two are more *feasible* for testers to interact with. A graphical representation of these findings is shown in Figure 9.

2. Feasibility Test

In designing Qubot, we seek to minimize the hours spent creating the testing framework so to have more time for actually running tests on software. The Stassy, SAT, and AppFlow papers were hence reviewed in detail to project approximately how many code files an individual would have to write to replicate the software structure mentioned by the authors. Detailed explanations for each projection can be found in this paper's GitHub repository. [19]

Stassy and SAT, unfortunately, did not include source code and so the descriptions of each framework were loosely converted into a code file count. The file count estimation for Stassy is shown in Figure 10 and for SAT is shown in Figure 11.

As mentioned previously, the authors of AppFlow included source code for the framework in the paper, and the number of code files for this framework were taken directly off GitHub. [16] The file count estimation for AppFlow is shown in Figure 12.

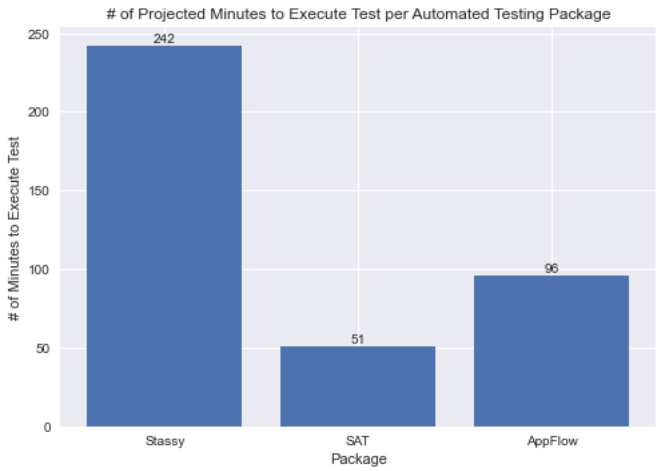

FIG. 9: Number of Estimated Minutes to Set Up Each Testing Suite

File/Package Name	*# of code files*
`custom_exception_package`	1
`main_package`	1
`object_repository_package`	114
`reusable_functions_package`	1
`utilities_package`	1
`reports_package`	1
`library_compressed_logic`	1
Total	119

FIG. 10: Stassy Projected File Count

File/Package Name	*# of code files*
`selenium_connectivity_package`	1
`json_data_recording_package`	1
`tcs_generator`	1
`tcs_runner`	1
`utilities_package`	1
Total	5

FIG. 11: SAT Projected File Count

Stassy, which requires a DOM-equivalent Java class to be created for every single HTML element, as well as AppFlow, which is feature-rich (i.e. with a UI interface and several machine learning models), both require drastically more code to be written prior to use, compared to SAT. These differences are graphically captured in Figure 13.

B. Intelligence: Making Qubot Think for Itself

Now that we have chosen an architecture for Qubot, the other big task is to choose a method for autonomous and unscripted exploratory testing. Since NEAT and SAPIENZ proposes scripted approaches to test generation, our choices for experimentation are Fastbot, DRIFT, and Humanoid. We chose DRIFT as the basis for our final intelligence experiment because of the memorable performance of the DRIFT-Greedy algorithm.

We designed a simple framework containing a web-scraper and a Q-learning agent to perform simple black box testing. The goal of the agent was to navigate from the root of `https://upmed-starmen.web.app/` to the "Log in as a Healthcare Provider" button on `https://upmed-starmen.web.app/signin`, which requires the agent to navigate to the next page.

In order to perform website navigation, the Selenium WebDriver library was installed in Python. Next, custom `UITree` and `UITreeNode` classes were created, similar to those mentioned in the DRIFT paper [14]. The shape of a `UITree` can be seen in Figure 14.

File/Package Name	*# of code files*
`src_package`	98
Total	98

FIG. 12: AppFlow Projected File Count

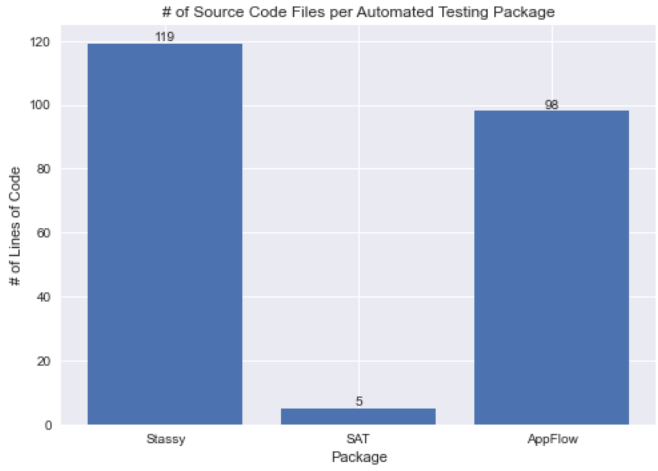

FIG. 13: Approximate Number of Code Files per Testing Suite

The anchor (`<a>`) and button (`<button>`) tags were attributed the `UIAction.LEFT_CLICK` actions and all other tags (i.e. `<div>`) were attributed the `UIAction.NAVIGATE` actions in the tree. This means that the bot performs a left click when it reaches a button, for instance, but it will simply continue exploring the website if it encounters a generic content container such as a `<p>` tag.

Now that a `UITree` for the WUT can be created, the next step is to create an environment in which a reinforcement learning (RL) agent can navigate. A custom OpenAI Gym [24] environment was created for traversal of the `UITree`. The parameters used for this test can be found in the source code for this paper. [19]

Within 60 training episodes, the Q-learning agent began to rapidly accrue rewards by visiting the aforementioned button in the second page, as shown in Figure 15.

The same agent was tasked with finding the image of black sand visible at the `https://upmed-starmen.web.app/signin` URL during the testing portion of this experiment. After a mere 3 episodes of training, the Q-learning agent once again began to accrue rewards, as shown in 16. The increasing rate at which rewards were being accrued implied that the agent successfully learned to navigate to this image, and was able to do so more rapidly the more training episodes underwent.

```
NAVIGATE: <html id="" class="">
    NAVIGATE: <head id="" class="">
        NAVIGATE: <meta id="" class="">
        NAVIGATE: <link id="" class="">
    LEFT_CLICK: <a id="" class="text-left pt-2 pb-2">
...
```

FIG. 14: A Sample UITree Printed to Console

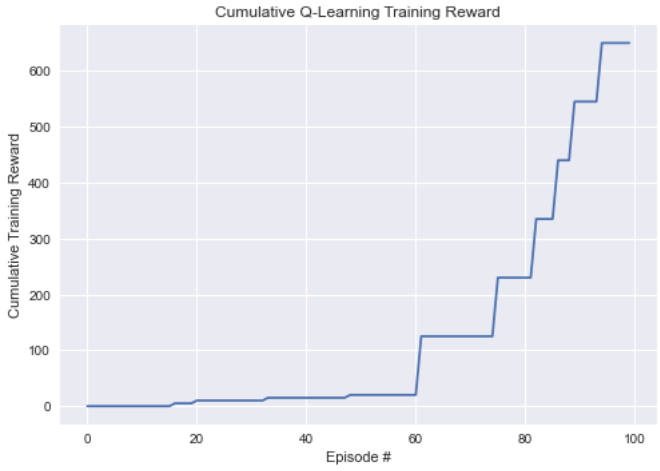

FIG. 15: Cumulative Training Reward of Q-Learning Agent over 100 Episodes

VI. RESULTS

A. Chosen Architecture for Qubot

The combination of shortest duration to set up a test and the least amount of code files to write makes **SAT** the most suitable automated software testing architecture to use for our hypothetical Qubot framework.

B. Chosen Autonomous Testing Technique for Qubot

The total lines of code for obtaining the `UITree` and then both training and testing the Q-learning agent is a mere 10 lines, and can be run on a wide range of other websites. Websites that direct the agent to endless other websites may cause the agent to timeout, however. Despite this issue, the agent effectively achieves the goal of finding the "Log in as a Healthcare Provider" button during training and the landmark image during testing, indicating that Q-learning inspired by **DRIFT** is, indeed, an effective approach to autonomous acceptance testing small portions of software.

The chosen architecture and autonomous testing technique for Qubot are shown in Figure 17.

C. Threats to Validity

Threats to validity of the experiments in this paper include the estimation of test setup time and code files from domain experience, as well as the absence of negative rewards in the Q-learning test. It was discovered that the agent performs poorly if it is penalized for reaching an HTML element with no child elements, and so it received no punishments in the intelligence experiment.

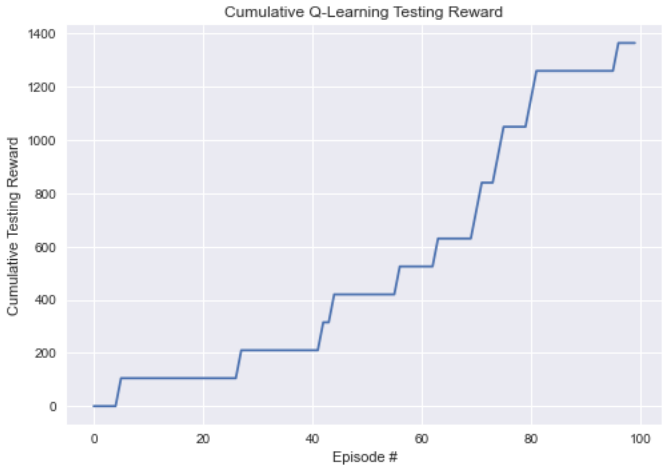

FIG. 16: Cumulative Testing Reward of Q-Learning Agent over 100 Episodes

VII. CONCLUSION

A. Key Takeaways

In surveying ten papers on the journey to create Qubot, we have learned a considerable amount of new approaches and techniques in contemporary software testing. Firstly, SAT's [13] feature in which developers can either partially or fully automate test case generation implies that manual and autonomous tests do not have to be mutually exclusive. AppFlow's [16] use of machine learning to classify screens and widgets prior to autonomously testing these UI elements was shown to be successful for black box tests of apps, and can have great use as a plugin for continuous integration on GitHub (i.e. such that apps can be regression-tested whenever a developer attempts to push to the `main` branch). The uses of ant colony optimization in Dorylus [3] and genetic algorithms in NEAT [28], SAPIENZ [23], and BGA [22] in generating test inputs and cases to bolster coverage provide inspiration for manipulating ML classifiers used in black box testing algorithms to improve parameters that affect exploration of SUTs. Lastly, we have seen, firsthand, the ability of Q-learning agents to navigate software environments with little configuration, indicating that, perhaps one day, they may become intangible tools for QA testing in teams that do not have the resources to hire human testers.

B. Future Work

In the future, we first hope to incorporate the chosen architecture for Qubot into a standalone testing suite for Python. This suite will enable developers to run exploratory tests on their websites within 10 lines of code, just as in the Experiments Section. Additionally, we hope

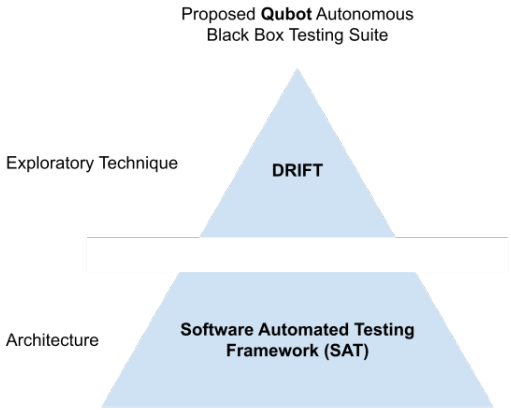

FIG. 17: Proposed Qubot Framework

to extend Qubot for use in mobile applications and eventually make it platform-agnostic. This will require interfacing with mobile interaction libraries used by some of the proposed papers such as AppFlow [16] and SAPIENZ [23]. We finally hope to extend Qubot to the white box testing use case, alongside black box testing. Giving developers the ability to run this testing framework in 10 lines of code on an SUT, and have Qubot generate test cases while performing exploratory tests may make it an attractive all-in-one testing library for use in the field. With this in mind, it may not be too far-fetched to imagine a world where developers will never have to find their own bugs again.

VIII. REFERENCES

[1] Britannica, The Editors of Encyclopaedia. 05 Sept 2016. *Genotype.* https://www.britannica.com/science/genotype

[2] Britannica, The Editors of Encyclopaedia. 05 Sept 2016. *Phenotype.* https://www.britannica.com/science/phenotype

[3] Bruce, D, Menendez, H.D, Clark, D. August 2019. *Dorylus: An ant colony based tool for automated test case generation.* 2018 IEEE Symposium Series on Computational Intelligence (SSCI). London, UK. University College. https://ieeexplore.ieee.org/document/8628668

[4] Cai, T, Zhang, Z, Yang, P. October 2020. *Fastbot: A Multi-Agent Model-Based Test Generation System.* AST '20: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test (2020). Beijing Bytedance Network Technology Co., Ltd. https://dl.acm.org/doi/pdf/10.1145/3387903.3389308

[5] *Cucumber.* 2019. SmartBear Software. https://cucumber.io/

[6] Cucumber. 2021. *Gherkin.* SmartBear Software. https://cucumber.io/docs/gherkin/

[7] Deb, K. et. al. 07 August 2002. *A fast and elitist multiobjective genetic algorithm: NSGA-II.* IEEE Transactions on Evolutionary Computation (Volume: 6, Issue: 2). https://ieeexplore.ieee.org/document/996017

[8] Dorigo, M, Stützle, T. June 2004. *Ant Colony Optimization.* The MIT Press. Cambridge, MA. https://mitpress.mit.edu/books/ant-colony-optimization

[9] *EvoSuite.* 06 October 2020. New 1.1.0 release. https://www.evosuite.org/

[10] Fowler, M. 18 November 2019. *Exploratory Testing.* https://martinfowler.com/bliki/ExploratoryTesting.html

[11] Fujimoto, S, Meger, D, Precup, D. 10 August 2019. *Off-Policy Deep Reinforcement Learning without Exploration.* International Conference on Machine Learning (2019). arxiv.org. https://arxiv.org/abs/1812.02900

[12] Guo, P, Zhu, L. May 2012. *Ant colony optimization for continuous domains.* 8th International Conference on Natural Computation (2012). Chongqing, China. https://ieeexplore.ieee.org/document/6234538

[13] Hanna, M, Aboutabl, A.E, Mostafa, M.M. 2018. *Automated Software Testing Framework for Web Applications.* International Journal of Applied Engineering Research (2018). Research India Publications. http://www.ripublication.com/ijaer18/ijaerv13n11_141.pdf

[14] Harries et. al. 16 July 2020. *DRIFT: Deep Reinforcement Learning for Functional Software Testing.* 33rd Deep Reinforcement Learning Workshop (NeurIPS 2019). Vancouver, Canada. Microsoft. https://www.microsoft.com/en-us/research/publication/drift-deep-reinforcement-learning-for-functional-software-testing/

[15] Hexaware. 2021. *End-to-end Autonomous Software Testing.* Hexaware Technologies Limited. https://hexaware.com/services/digital-assurance/autonomous-software-testing/

[16] Hu, G, Zhu, L, Yang, J. 04 November 2018. *AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests.* ESEC/FSE '18. Lake Buena Vista, FL, USA. Columbia University. https://www.cs.columbia.edu/~junfeng/papers/appflow.pdf

[17] JobHero. 21 February 2021 (Accessed). *QA Tester Job Description.* https://www.jobhero.com/job-description/examples/engineering/quality-assurance-tester

[18] Korel, B. August 1990. *Automated software test data generation.* IEEE Transactions on Software Engineering (Volume: 16, Issue: 8). https://ieeexplore.ieee.org/document/57624

[19] Krivonos, Anthony (Author). 13 February 2021. *Qubot*, `experiments.ipynb`, GitHub repository. https://github.com/anthonykrivonos/qubot/blob/main/experiments.ipynb

[20] Lefterov, D, and Svetoslav, E. June 2019. *Automated Software Testing Framework "Stassy".* International Scientific Journal "Industry 4.0". Plovdiv University, Bulgaria. https://stumejournals.com/journals/i4/2019/4/171.full.pdf

[21] Li, Y. et. al. 09 January 2019. *Humanoid: A Deep Learning-based Approach to Automated Black-box Android App Testing.* arxiv.org Computing Research Repository (CoRR). https://arxiv.org/pdf/1901.02633.pdf

[22] Manikumar, T, and Kumar, J.S. 01 February 2019. *A Buffered Genetic Algorithm for Automated Branch Coverage in Software Testing.* Journal of Information Science and Engineering". Taipei, Taiwan. https://jise.iis.sinica.edu.tw/JISESearch/pages/View/PaperView.jsf?keyId=167_2220

[23] Moreno I.A, Galeotti, J.P, Garbervetsky, D. 15 July 2020. *Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage.* 2020 IEEE/ACM 1st International Conference on Automation of Software Test (AST). https://dl.acm.org/doi/10.1145/3387903.3389307

[24] *OpenAI.* 18 December 2020. Gym 0.18.0. https://gym.openai.com/

[25] Patel, P. 06 August 2019. *Difference between Scripted and Unscripted Testing.* GeeksforGeeks. Noida, Uttar Pradesh, India. https://www.geeksforgeeks.org/difference-between-scripted-and-unscripted-testing/

[26] Prather, R.E., Myers, J.P. July 1987. *The Path Prefix Software Testing Strategy*. IEEE Transactions on Software Engineering (Volume: SE-13, Issue: 7). https://ieeexplore.ieee.org/document/1702287

[27] ProductPlan. 21 February 2021 (Accessed). *Acceptance Criteria*. https://www.productplan.com/glossary/acceptance-criteria/

[28] Raj, H.L, P, Chandrasekaran K. 18 November 2018. *NEAT Algorithm for Testsuite generation in Automated Software Testing*. 2018 IEEE Symposium Series on Computational Intelligence (SSCI) Bangalore, India. https://ieeexplore.ieee.org/document/8628668

[29] *Selenium WebDriver*. 20 February 2021. The Selenium Browser Automation Project. https://www.selenium.dev/documentation/en/webdriver/

[30] Software Testing Fundamentals. 13 September 2020. *Acceptance Testing*. https://softwaretestingfundamentals.com/acceptance-testing

[31] Software Testing Fundamentals. 17 September 2020. *Black Box Testing*. https://softwaretestingfundamentals.com/black-box-testing

[32] Software Testing Fundamentals. 13 September 2020. *Unit Testing*. https://softwaretestingfundamentals.com/unit-testing/

[33] Stanley, K, O, and Miikkulainen, R. 2002. *Evolving Neural Networks through Augmenting Topologies*. Evolutionary Computation (Volume: 10, Issue: 2). The University of Texas at Austin. http://nn.cs.utexas.edu/?stanley:ec02

[34] Sutton, R.S., Barto, A.G. 04 January 2005. *Reinforcement Learning: An Introduction*. The MIT Press. Cambridge, Massachusetts. http://incompleteideas.net/sutton/book/ebook/the-book.html

[35] Testim. 06 August 2019. *What Is Test Automation? A Simple, Clear Introduction*. https://www.testim.io/blog/what-is-test-automation/

[36] Van Hasselt, H. et. al. 06 December 2018. *Deep Reinforcement Learning and the Deadly Triad*. arxiv.org Preprint. https://arxiv.org/abs/1812.02648

[37] Wang, H, Raj, B. 03 March 2017. *On the Origin of Deep Learning*. arxiv.org Preprint. https://arxiv.org/abs/1702.07800

[38] Weitzman, M.L. March 2001. *Gamma Discounting*. The American Economic Review (Volume: 91, Issue: 1). https://scholar.harvard.edu/files/weitzman/files/gamma_discounting.pdf

[39] Zhou, J. et. al. 10 July 2019. *Graph Neural Networks: A Review of Methods and Applications*. arxiv.org Preprint. https://arxiv.org/abs/1812.08434