11-9-2018

A Model-Based AI-Driven Test Generation System

Dionny Santiago
Florida International University, dsant005@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A MODEL-BASED AI-DRIVEN TEST GENERATION SYSTEM

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dionny Santiago

2018

To: Dean John L. Volakis
College of Engineering and Computing

This thesis, written by Dionny Santiago, and entitled A Model-based AI-driven Test Generation System, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Tariq M. King

Leonardo Bobadilla

Monique Ross

Peter J. Clarke, Major Professor

Date of Defense: November 9, 2018

The thesis of Dionny Santiago is approved.

Dean John L. Volakis
College of Engineering and Computing

Andres G. Gil
Vice President for Research and Economic Development
and Dean of the University Graduate School

Florida International University, 2018

DEDICATION

To my wife Yesenia and my parents for their unending support.

ACKNOWLEDGMENTS

I would like to thank Patrick Alt, Dr. David Adamo, Keith Briggs, Robert Vanderwall, Jason Arbon, Michael Mattera, Brian Muras, John Maliani, Justin Phillips and Philip Daye for their feedback and support on this research. I would also like to thank Rick Anderson and Keith Stobie for their feedback on the publication of a portion of this work in the Pacific Northwest Software Quality Conference Proceedings in 2018. In addition, I wish to thank the thesis committee members, Dr. Tariq M. King, Dr. Leonardo Bobadilla, and Dr. Monique Ross for their support and feedback.

Last but not least, I would also like to especially thank Dr. Peter J. Clarke and Dr. Tariq M. King for their excellent guidance and mentorship.

ABSTRACT OF THE DISSERTATION

A MODEL-BASED AI-DRIVEN TEST GENERATION SYSTEM

by

Dionny Santiago

Florida International University, 2018

Miami, Florida

Professor Peter J. Clarke, Major Professor

Achieving high software quality today involves manual analysis, test planning, documentation of testing strategy and test cases, and development of automated test scripts to support regression testing. This thesis is motivated by the opportunity to bridge the gap between current test automation and true test automation by investigating learning-based solutions to software testing. We present an approach that combines a trainable web component classifier, a test case description language, and a trainable test generation and execution system that can learn to generate new test cases. Training data was collected and hand-labeled across 7 systems, 95 web pages, and 17,360 elements. A total of 250 test flows were also manually hand-crafted for training purposes. Various machine learning algorithms were evaluated. Results showed that Random Forest classifiers performed well on several web component classification problems. In addition, Long Short-Term Memory neural networks were able to model and generate new valid test flows.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

A grand challenge in software engineering is continually and consistently delivering high quality software. Although 50% of software development budgets typically go towards testing, software defects cost the U.S. economy $59.5 billion annually [Tas02]. A study on the IT budget allocation suggests that testing is not as efficient as it should be, and greater efficiency should be an important objective of any quality program [Cap17]. Test automation is one way of achieving higher efficiency.

Although automated test execution provides critical efficiency benefits to the software development process, there are many accompanying problems with the current state of the art. Test scripts are hand-crafted by humans and may break given changes to the underlying system under test, and may not generalize across applications. Consequently, current approaches do not provide for fully automated software testing and substantial manual effort is still required. Also, automated test script oracles are limited and can only detect defects based on the path and assertions that were coded explicitly in the test script.

There is a significant gap between human-present and machine-driven testing. Although there has been extensive research and work into semi-automated testing techniques [AMRS11, RMPM12], these techniques do not: (1) mimic the thought and learning process of human testers; and (2) seamlessly generalize across applications and application domains. Human testers can perceive the state of an application, can act intelligently to attempt to break the software, and can modify their strategy dynamically based on observations. Advances in artificial intelligence (AI) and machine learning (ML) have shown that machines are capable of matching or

surpassing human performance across various problem domains [BDTD+16, Moy16]. Researchers and practitioners are realizing the potential for AI and ML to further improve the capability of machine testing [AIS18, KA18, AH04].

This thesis presents an approach for automatically generating tests using a combination of ML-based techniques for perceiving web page application state and generating abstract test flows. More specifically, we present a method for allowing human testers to express tests as reusable test flows, and for leveraging AI and ML to enable the creation of a trainable test flow generation system. The major contributions of this thesis are as follows: (1) presents a novel approach that models how human testers produce test flows as an application-agnostic abstract sequence problem; (2) defines a language and accompanying grammar that supports the creation of test flows, and presents examples of applying the language to a variety of scenarios; and (3) discusses the implementation and results of a test flow generation prototype.

1.2 Outline of Thesis

Chapter 2 provides background on software testing and artificial intelligence (AI), and proceeds to discuss related works in the intersection of software testing and AI. Chapter 3 puts forth the problem statement, the thesis, and the thesis objectives along with their evaluation criteria. Chapter 4 discusses our AI-driven test generation and execution system, and covers web component classification and test flow generation. Also, Chapter 4 covers our approach to test case generation and execution. In Chapter 5, we evaluate our approach to webpage component classification, test flow generation, and test execution. We also discuss the limitations of the approach. Finally, Chapter 6 concludes the thesis and discusses future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter contains background material to aid understanding of the thesis, and describes related work. The covered background material includes software testing, webpage rendering, artificial intelligence, machine learning algorithms, and the relationship between machine learning and testing. In addition, we cover approaches for AI-driven object recognition and text generation.

2.1 Background

This section contains background material to aid understanding of the thesis, including an overview of software testing, web page rendering, artificial intelligence, machine learning and the application of the aforementioned topics.

2.1.1 Software Testing

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment [Whi00]. Ammann and Offutt [AO17], in regards to software, define testing as "evaluating by observing its execution". Much work has gone into automating software testing in order to reduce repetitive manual labor [ABC+13, VKCF+15, AMRS11]. Automated testing involves creating scripts that are programmed to follow specific test steps. Creating these scripts typically involves creating a model of the application using implementation details from the system under test (SUT). Test scripts refer to the models and exercise a sequence of test steps referred to as *flows* through the SUT. Within the scope of web applications, the test scripts are then executed

against the SUT using tools such as Selenium [Sel18], and the results are recorded and logged. An example of Selenium-based web automation is shown on Figure 2.1.

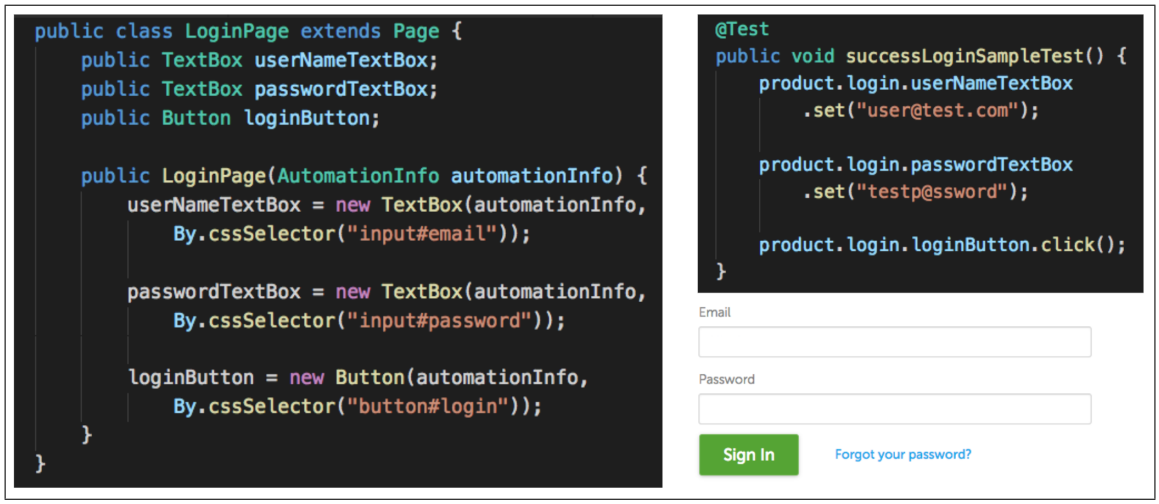

Figure 2.1: Selenium Page Object Model and Test Script

2.1.2 Webpage Rendering

The Document Object Model (DOM) is an object-oriented tree representation of a web page that represents the hierarchical document structure of the page, and serves as a programming interface for web pages [Lui18]. Cascading Style Sheet Object Model (CSSOM) trees extend DOM trees by introducing web element styling information, also in a hierarchical fashion [Gri18]. The CSSOM and DOM trees are combined into a render tree that is used as input to the graphical rendering subsystem of a web browser, where the final layout of each visible element is computed and drawn [Gri18]. The difference between the various tree representations is shown on Figure 2.2.

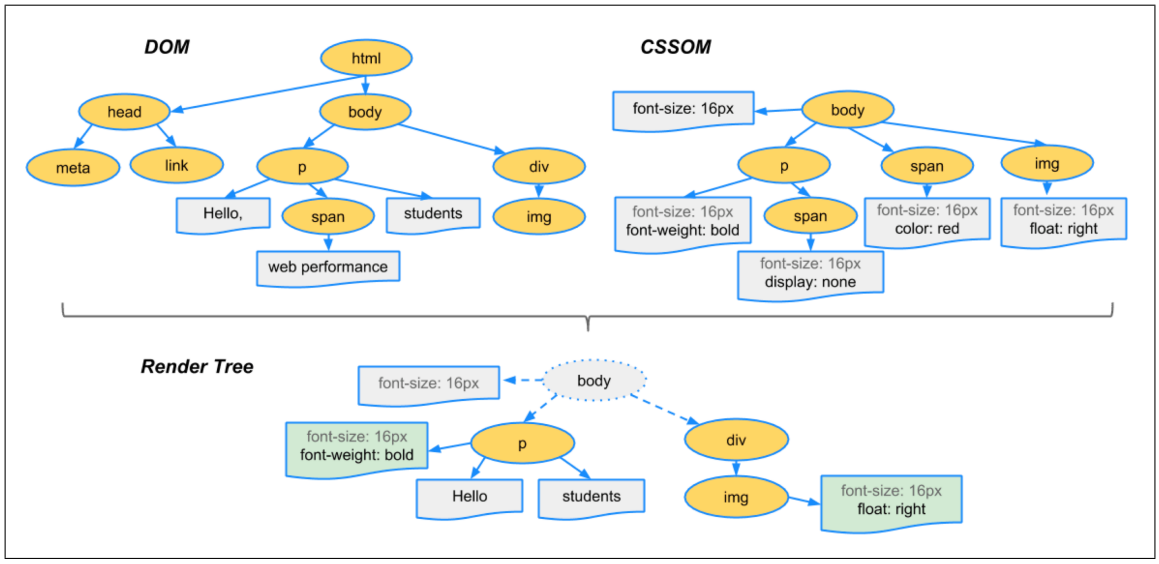

Figure 2.2: Web Rendering Trees

2.1.3 Artificial Intelligence

Artificial intelligence (AI) is a field that attempts to not only understand how humans can perceive, understand, predict, and manipulate their surrounding environment, but also sets out to build intelligent entities that are able to mimic human behavior.

Machine Learning (ML) is a subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. If the performance $\boldsymbol{P}$ of a computer program at completing a task $\boldsymbol{T}$ improves with experience $\boldsymbol{E}$, the program is said to "learn" from experience $\boldsymbol{E}$. Machine learning is the science of getting computers to act without being explicitly programmed [Ng18].

Two major applications of ML are referred to as unsupervised learning and supervised learning. In unsupervised learning, unlabeled data is fed into a training algorithm with the goal of discovering patterns and relationships. Examples include clustering algorithms that attempt to organize data points into groups. Supervised

learning, in contrast, involves human-labeled training data. A supervised learning workflow typically involves multiple iterations of constructing labeled training data, extracting features from the data, choosing an algorithm, training a model, evaluating the model using test or input data, and improving the model. This workflow is shown on Figure 2.3.

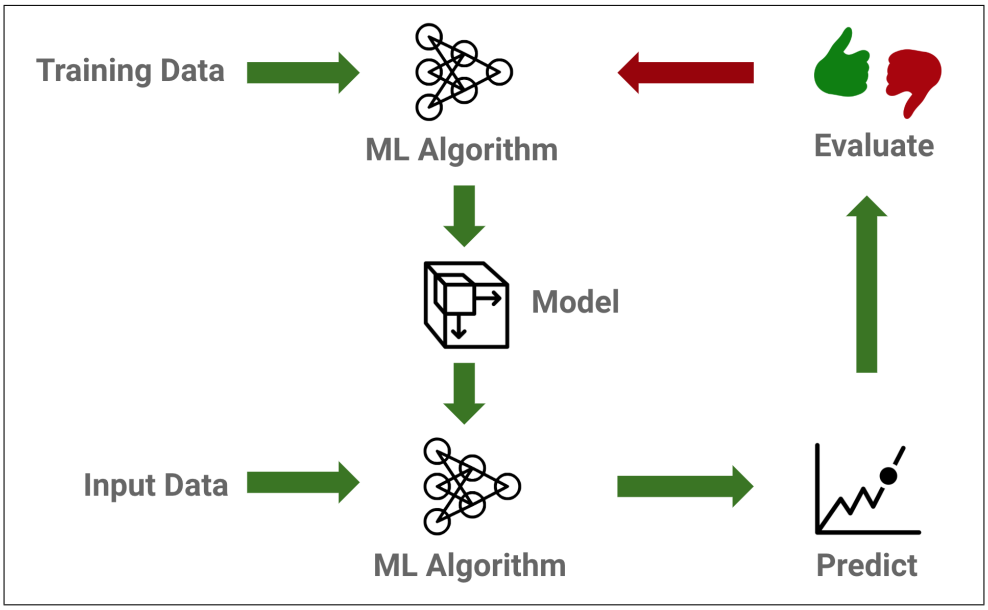

Figure 2.3: Supervised Learning Workflow

Another application of ML is reinforcement learning. This learning approach involves creating algorithms that are capable of learning by using reward systems. Useful actions are positively rewarded, whereas less useful actions may be penalized.

2.1.4 Machine Learning Algorithms

This work focuses on using and comparing the performance of decision tree learning, random forests, the k-Nearest Neighbor algorithm, Support Vector Machines (SVM), and artificial neural networks (ANN) for various webpage classification and automated test generation tasks. Decision tree learning uses a hierarchy of con-

ditional statements as a predictive model for making choices and predictions on a specific data point using observable information about the data point [Sci18]. Various algorithms exist for automatically inferring decision trees from a training dataset [Qui96, PS13].

The k-Nearest Neighbor (KNN) algorithm is a technique that does not require training; instead, it looks at all data points at prediction time and arrives to a decision based on measuring distance from a new data point to training data points [Pet09]. Support vector machines are learning models that may be used for classification and regression analysis problems [Wan05]. Artificial neural networks are computing systems vaguely inspired by biological neural networks, and are based on a collection of connected units called artificial neurons [Sch97]. ANN units are usually grouped into a hierarchy of layers, where each layer's output serves as input into the subsequent layer. Figure 2.4 shows an example neural network.

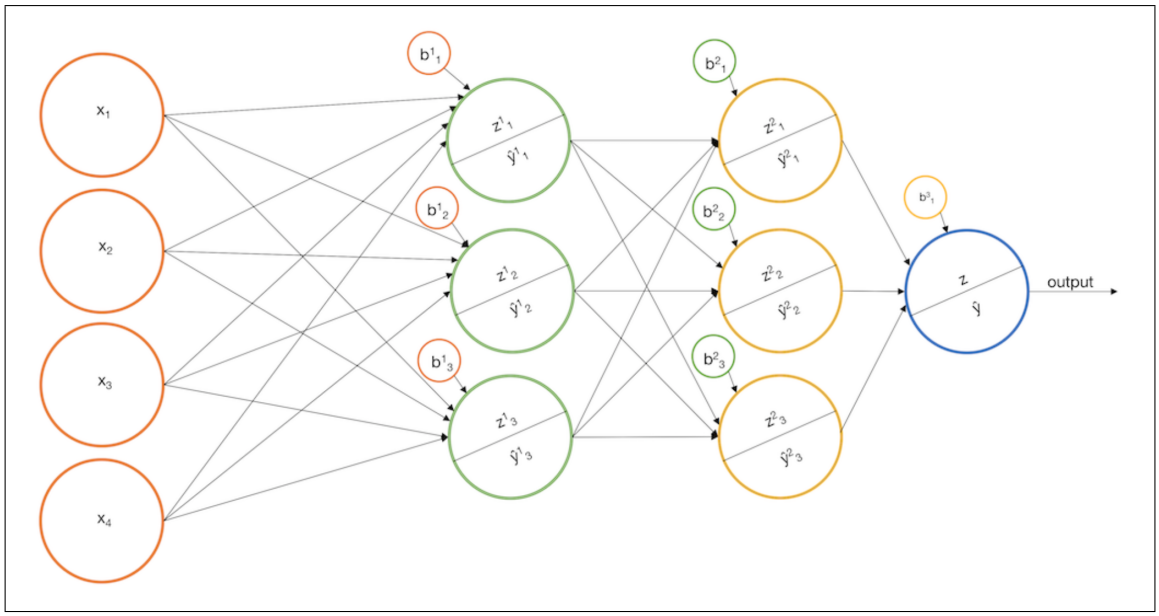

Figure 2.4: Simple Neural Network

Long Short-Term Memory (LSTM) recurrent neural networks (RNN) are a special type of neural network that are able to generate sequential data with long-range structure [Gra13]. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. This is illustrated on Figure 2.5.

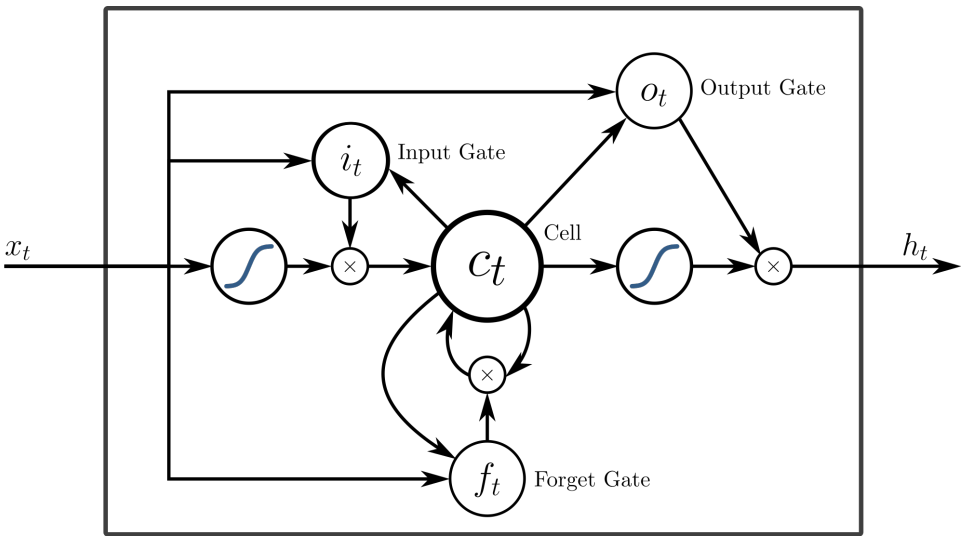

Figure 2.5: Long Short-Term Memory Block

A Bayesian network, also known as Bayesian belief network, is a directed acyclic graph in which each node is associated with a conditional probability distribution [Mar14]. Conditional probability is the likelihood of an event happening, given that it has some relationship to one or more other events [Gut13]. For example, the probability of finding a parking space is dependent on the location being visited, the day of the week, and time of day. Formally, a Bayesian network model is defined by: (1) a set of nodes where each node corresponds to a random variable, which may be discrete or continuous; (2) a set of directed edges that connect pairs of nodes, where an edge from node X to node Y indicates that Y is dependent on X; and

(3) a set of annotations in the form of conditional probability tables (CPTs), where each CPT represents a conditional probability distribution $P\left(X_i|Parents(X_i)\right)$ that quantifies the effect of the parents on node X_i.

Overfitting is a common problem in ML, and refers to the issue of having a predictive model that does very well on training data, but poorly on unseen new data. Various techniques exist for controlling overfitting, such as random forests and gradient boosted trees [LW$^+$02, De'07].

2.1.5 Relationship Between ML and Testing

There is a direct mapping from the software testing problem to a machine learning solution. The testing problem involves applying a test input to an application or function, then comparing the output to an expected result. This is precisely what machine learning does. A set of inputs (or features) is supplied to a training algorithm. In supervised learning, the correct answer is also supplied to the training algorithm with each set of inputs. The job of the machine learning system is to iteratively (slightly) reconfigure the internal computation units, each time getting closer to providing the correct answers based on the provided input sets. Therefore, all of the existing and ongoing research and development that has gone into building these ML systems are providing a direct benefit towards further automating the software testing problem.

Self-driving cars and the technology that powers them can be mapped to the problem of software testing. Just as humans can teach learning algorithms how to drive a car, we can envision building a system capable of learning from a user's testing journey. Similarly, NLP and natural language generation research may map to test case generation. Lastly, game theory and reinforcement learning may map

well to the problem of discovering a system and hunting for bugs. For example, reinforcement learning may be used to reward an intelligent system for uncovering a system crash or an exception.

Researchers and practitioners realize the potential for AI and ML to be leveraged to help bridge the gap between the testing capabilities of humans and those of machines [KA18, AIS18]. Existing work on applications of ML to software testing has explored applying supervised ML to the testing problem [Arb17]. There is also an abundance of emerging research on new AI and ML techniques and algorithms. It is imperative that the testing community make a concerted effort to keep up with AI research and look for ways to innovate within the testing field.

2.1.6 AI-Driven Object Recognition

There exist several problems and areas of research surrounding object recognition within images. Image classification involves visually inspecting an image and deciding whether the image belongs to a particular class. For example, given an image of a cat as input, an image classifier may return the label "cat" as an output.

In some cases, classifying an image with a single label is not sufficient. Suppose the input image contains both a cat and a dog. While assigning either label would be correct, we may instead want to produce two output labels for the single input image. Research efforts have been focused on ML algorithms that can produce multiple outputs. Lastly, in some cases, we are not just interested in generating classes for a given input image. Depending on the problem at hand, we may want to build a system capable of detecting the location of the object within an image. Figure 2.6 shows an example of object recognition problems.

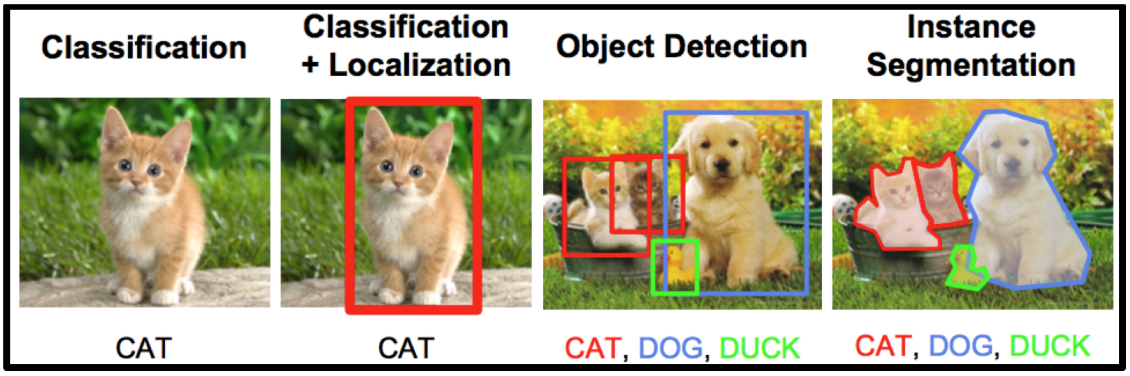

Figure 2.6: Object Recognition Problems

A common approach for encoding an image to be used as input into a neural network is to treat each pixel as a feature as shown on Figure 2.7.

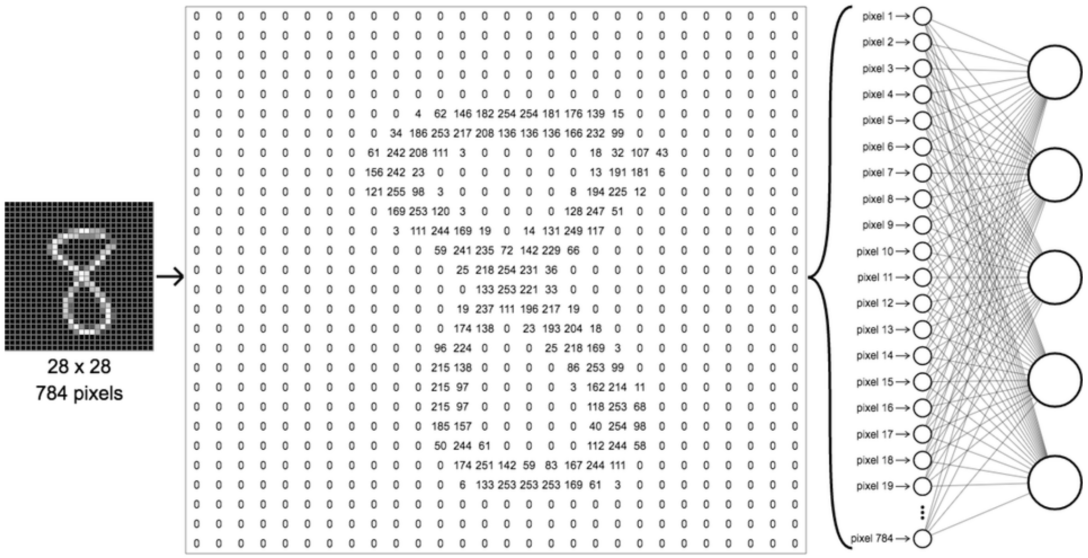

Figure 2.7: Encoding Image into Neural Network

2.1.7 AI-Driven Text Generation

Much research exists on generating sequential information using ML-based techniques. In the field of natural language processing (NLP), several approaches to

text generation have been studied [Gra13, Bro18]. A conventional approach is to treat text generation as a sequence-to-sequence problem. A sequence of text is fed into a trainable algorithm that in turn outputs a sequence of text. The training goal is that the concatenation of both sequences results in a plausible sentence. Since machines do not understand words, a common practice is to map words to integer values, resulting in what we may refer to as a word encoding. This means that both the input and output of the sequence-to-sequence problem are a sequence of integers. Word encodings are challenging to create and maintain as there are many possible words that may appear, especially when different languages are used.

An alternative approach is to create a character encoding, where each character (for example, each character from A-Z) is mapped to an integer. Although this simplifies the text-to-machine mapping process, it creates additional complexity from a training standpoint. There are many valid (and invalid) arrangements of character sequences that can first form words and eventually form sentences. In contrast, there are much fewer unique arrangements of complete words. ML-based text generation techniques that leverage character-based encoding typically require more training to reach stability and feasibility-of-use over word-based encoding approaches.

An example of an ML approach to sentence generation using word encodings is shown on Figure 2.8. Research has shown that specialized ANNs are capable of being trained to handle sequence-to-sequence problems (such as text generation). Long Short-Term Memory (LSTM) recurrent neural networks (RNNs) are a particular type of neural network capable of generating sequential data with long-range structure [Gra13]. For the interested reader, Jason Brownlee [Bro18] provides a great step-by-step tutorial that leverages tools such as Keras and TensorFlow to build LSTMs capable of generating plausible sentences for a problem domain based on training from classical texts [Bro18].

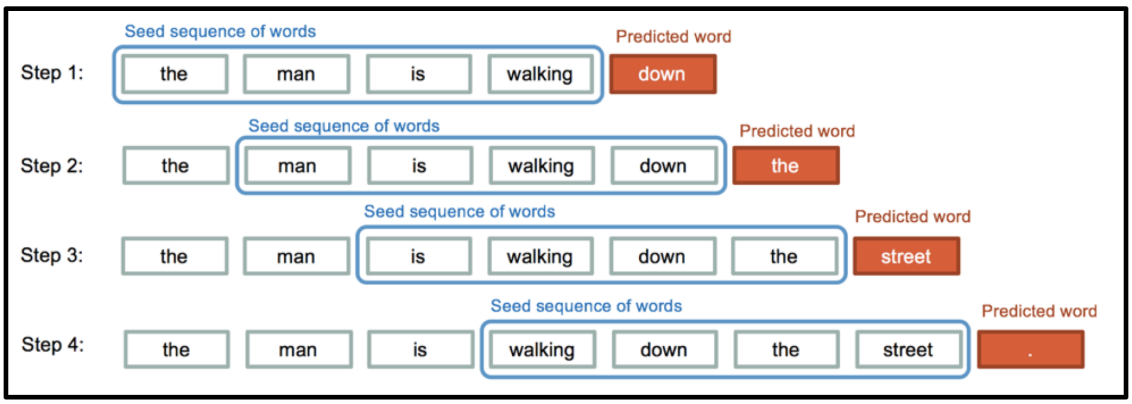

Figure 2.8: Sentence Generation Example

2.2 Related Work

Memon et al. [MBN03] present an approach to automatically reverse engineer a graphical user interface (GUI) for testing. Memon [Mem04] also demonstrates a technique for automated GUI regression testing using AI planning. Memon represents GUI test cases as pairs of initial and expected goal states along with sequences of events. Vos et al. [VKCF+15] present Testar, a tool for test automation at the user interface level. Li and Offutt [LO17] present findings for building test oracles when using model-based testing. Our work on abstract test flow sequencing uses ideas from these works for incorporating generic oracles.

Onan [Ona16] evaluates webpage classification accuracy across different feature selections, ensemble learning methods, and base learners (Naive Bayes, K-Nearest Neighbor, C4.5 Algorithm and FURIA). Gogar et al. [GHS16] describe an approach for webpage information extraction. The authors present prior research and emphasize the limited generalization power of extraction using predefined website templates. The authors introduce a mechanism to encode webpage nodes into a spatial bag of words model that captures positional information. The study presents promising results classifying product images, prices, and names from shopping ap-

plications. The work done by Onan and Gogar et al. is important for recognizing web pages, elements, and components for the purposes of testing.

Graves [Gra13] shows how to leverage LSTMs in order to generate complex sequences with long-range structures. Graves shows how this can be done by predicting one data point at a time. The approach is demonstrated in the context of generating text and online handwriting. LSTMs are also capable of learning simple context-free and context-sensitive languages, as shown by Gers and Schmidhuber [GS01]. Brownlee [Bro18] shows how to create a character-by-character generative model for text using LSTMs, and demonstrates the generation of text sequences from classical texts. Our work extends Brownlee's work by using word encodings instead of character encodings to support a test flow generation language.

Arbon [Arb17] presents an approach to AI-driven software testing that includes learning systems capable of recognizing application state, applying test inputs, and verifying behavior. Arbon focuses on mobile applications and applies ML techniques including decision tree learning, random forests, and ANNs. Arbon et al. [ANT+18] also presents an Abstract Intent Test (AIT) language that allows one to manually define test cases using domain concepts. The AIT language has elements that are similar to the language presented in our work. An important distinction is that our language is designed to allow ML-based test flow generation, but does not specify intent.

CHAPTER 3

PROBLEM STATEMENT

This chapter introduces the problem, the problem statement, the thesis, and the objectives along with their evaluation criteria.

3.1 Problem

By solving technical challenges such as web browser automation and by providing simple programming interfaces, tools such as Selenium [Sel18] have facilitated writing test scripts that are capable of automatically executing complex test cases against a web application. Automated software testing involves writing both page object code [Fow13] and scripts that are web application specific. Due to implementation coupling, test scripts do not generalize across applications, and may easily break given changes to the underlying application.

The cost due to manual effort of testing software remains high as a result of gaps in existing test automation practices. ML-based test generation can provide a higher degree of test automation that is more general and representative of human testing than current approaches. This research examines three questions:

1. How can ML be used to raise the level of abstraction for representing webpage components?
2. How can ML be used to generate test flows for a web application?
3. How can the ML approaches for web component classification and test flow generation from (1.) and (2.) be combined to automatically generate and execute tests for a web application?

3.2 Objectives and Evaluation Criteria

Primary Objective – Investigate the feasibility of using machine learning approaches such as decision trees and neural networks to automatically generate tests in a way that is independent of any specific web application.

Evaluation Criteria: The developed approach should be capable of generalizing from previously trained models, and should generate executable test cases against an unseen web application.

The following sub-objectives each are required towards reaching the primary objective:

Sub-Objective 1 – Develop a trainable classifier which perceives application state and allows for the extraction of web page components in a way that is agnostic to a web application.

Evaluation Criteria: The classifier will be evaluated against multiple pages from several disparate web applications using accuracy, precision, and recall.

Sub-Objective 2 – (1) Develop a language that enables the generalized description of web application test flows; and (2) Develop an ML classifier for generating test flows.

Evaluation Criteria: Given a representative set of hand crafted test cases for a web application: (1) The language should be able to express each test case as an abstract test flow; and (2) the ML classifier should be able to generate a comparable set of abstract test flows for a previously unseen web application.

Sub-Objective 3 – Develop a test execution engine that uses ML-generated test flows and seeded test inputs to produce executable tests for a previously unseen web application.

Evaluation Criteria: The tests executed by the ML-based system should be able to discover a comparable number of defects compared to a human testing approach.

CHAPTER 4

AI-DRIVEN TEST GENERATION APPROACH

In this chapter we present the overall approach to solving the problem of reducing the gap between manual testing and automated testing. We then describe each of the processes used in the overall approach including: web component classification, test flow sequencing, the test flow specification language, test flow generation, and test case generation and execution.

4.1 Overall Approach

Our overall approach is two-pronged, targeting three major sub-problems: (1) webpage component classification; (2) test flow generation; and (3) test case generation. We define a test flow as a sequence of events that can represent the semantics of a test case. A test flow may contain action and observation steps. An action step describes a particular interaction that must be executed against the System Under Test (SUT). An observation step describes perception or expectational information; for example, we may perceive the existence of a particular field on a form, and we may have a particular expectation based on previously performed steps, such as an expectation on the existence of an error message.

The first part of our approach focuses on teaching a computer system to understand various different components on a given webpage. This problem can be framed as a supervised learning classification problem. Our approach involves creating a tool that functions as a web browser plugin, allowing a human user to supply labeled information about any webpage component on any webpage. This labeled data may be used to iteratively improve underlying machine learning models.

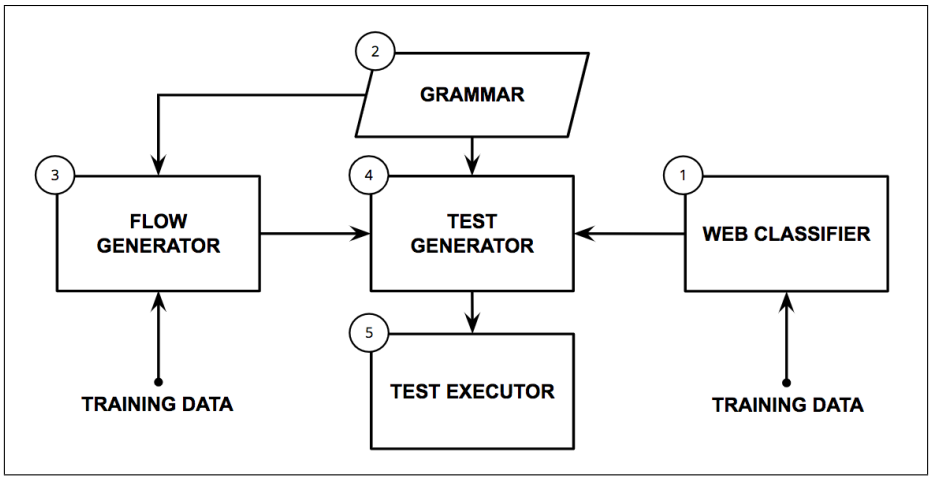

Figure 4.1: AI-Driven Test Generation Approach

The second part of our approach focuses on teaching a computer system the ability to understand test flows, process them, and also to generate new test flows. In order to teach a system to understand valuable test flows, our approach also involves creating a web browser plugin that is capable of observing a human user as they carry out their testing. By combining the information from our webpage component classification system, along with the test steps being executed by a human user, our approach can generalize and generate test flows. Given a mechanism to generate test flows, we also present a machine learning algorithm capable of learning and generalizing test flows. Given the ability to generate and learn test flows, we also include a test generation component that is capable of generating test cases by mapping generated test flows to the SUT.

Our approach for this process is shown in Figure 4.1. We use a trainable web classification system (1) to recognize web components using common abstractions applicable across SUTs. A language and grammar (2) is used to allow the definition of test flows. A trainable flow generation system (3) is used to produce abstract test flows. A test generation system (4) maps an abstract test flow to an executable concrete flow. Lastly, a test execution (5) system performs actions against an SUT.

4.2 Web Component Classification

We frame the problem of understanding webpage components as a supervised learning classification problem. Our approach is illustrated on Figure 4.2. The feature extraction step (A1) focuses on scraping information from a given webpage. This step produces a Computed Render Tree (CRT) representation of a given webpage. A standard web-based render tree includes information such as: (1) hierarchy; (2) element types; (3) element attributes; and (4) style sheet information. We extend this basic definition by collecting the render tree only once the web browser has finalized rendering, and by calculating additional information such as: (1) element positions; and (2) element sizes.

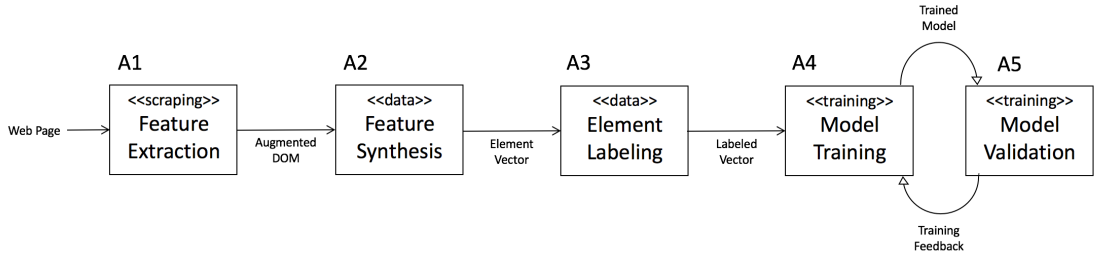

Figure 4.2: Webpage Component Classification Approach

Using the CRT representation, the next step is to perform a feature synthesis step (A2). This step performs an element-wise pass through all of the elements in the CRT, synthesizing several features for each element. Although the synthesis is done at the local level for each element, the full global context (information on the entire set of elements) is taken into account in order to synthesize most features. The total synthesized feature set is described in Table 4.1.

Given the synthesized feature set for each element on the webpage, the next step of our approach is to hand label all of the elements with various different classes. Examples of meaningful classes include: (1) page title; (2) label candidates - text

Feature	Description
HTML Tag	The tag for the given element.
Parent HTML Tag	The tag for the given element's parent.
"For" Attribute	The existence of a value for the HTML "For" attribute.
Num. Children	The number of HTML nodes that are children of the given element's node.
Num. Siblings	The number of HTML nodes that are siblings of the given element's node.
Depth	The depth of the given element's node within the ADOM tree.
Horizontal Percent	The relative horizontal position (in percentage) of the given element.
Vertical Percent	The relative vertical position (in percentage) of the given element.
Font Size	The relative (normalized against the full set of elements) font size of the given element.
Font Weight	The relative (normalized against the full set of elements) font weight of the given element.
Is Text	Describes whether a given element is a text node.
Nearest Color	The closest color computed using CIEDE2000 algorithm.
Nearest Background Color	The closest background color computed using CIEDE2000 algorithm.
Distance from Input	The relative (normalized against the full set of elements) distance to the closest input widget from the given element.
Text	The actual text associated with the given element.

Table 4.1: Computed Render Tree Feature Synthesis

elements which may be associated with input widgets on the page; (3) required label candidates - label candidates that suggest a field to be required; and (4) error messages. Once all of the data has been labeled, we can now make use of several different machine learning classifiers, comparing their performance on our dataset, and ultimately deciding on the best one for the given problem. An example of a simple webpage form along with a possible set of component classifications is illustrated on Figure 4.3.

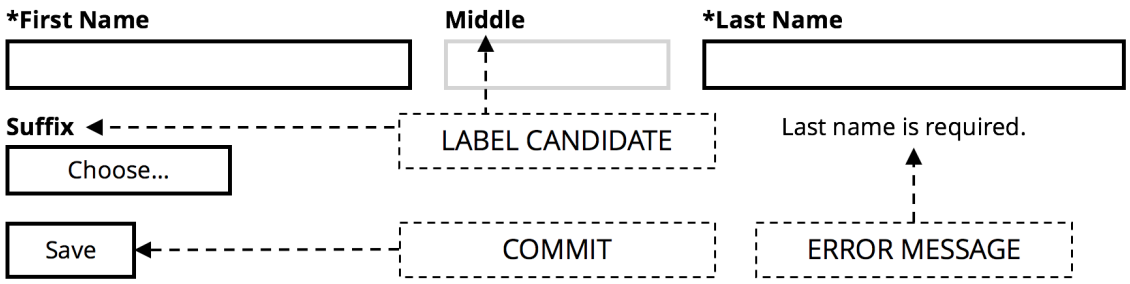

Figure 4.3: Sample Web Form with Classifications

Our preliminary work involved creating a dataset containing a total of 17,231 labeled examples. Our approach focuses primarily on training both feed-forward neural networks, and random forest classifiers in order to classify webpage components. For each given classification subtask, the better of the two is ultimately chosen. Several different approaches are used in order to validate the classification and generalization power of our trained machine learning models. Primarily, a percentage split approach is used: 60% of the dataset is used for training, while 40% is used for testing. In addition, cross-validation is used as a secondary technique. For both approaches, accuracy, precision, recall, and F1 scores are used to evaluate the model's performance. Steps A4 and A5, shown in Figure 4.2, focus on tweaking our dataset and tuning hyperparameters to reduce bias and variance.

The next question to explore is: ***How can image classification and object detection be leveraged to aid in software testing?*** To generate and execute test cases against a web application, the current state of the art involves constructing page objects that are coupled to the implementation details of a specific web application. Page objects often include references to document object model (DOM) and stylesheet information for a specific SUT. It is beneficial to shift towards an approach that leverages ML to raise the level of abstraction by which generated test cases interact with webpage components. Rather than interacting with elements us-

ing information that is SUT-specific, the goal is to be able to leverage ML to identify web components based on models that have been trained across various SUTs.

Having the ability to perceive web application state and recognize objects using ML-based approaches moves us one step closer to being able to write test cases devoid of any SUT-specific implementation details. Armed with enough example training data, we can leverage existing AI research and techniques for object recognition to raise the level of abstraction by which our automated test cases refer to objects. There are several benefits to this approach, including the ability for test scripts to self-heal if the SUT changes. For instance, while a change in the way the SUT renders a shopping cart button would likely break test scripts that leverage traditional DOM-based element selection strategies, a sufficiently trained ML model may be able to locate the new shopping cart button, allowing a test script that leverages ML-based element selection to continue execution. Also, by raising the level of abstraction, it becomes possible to reuse test cases across different SUTs. With enough training data, an ML model could be trained to recognize various components of a web application state. An example of webpage decomposition and object recognition is shown on Figure 4.4.

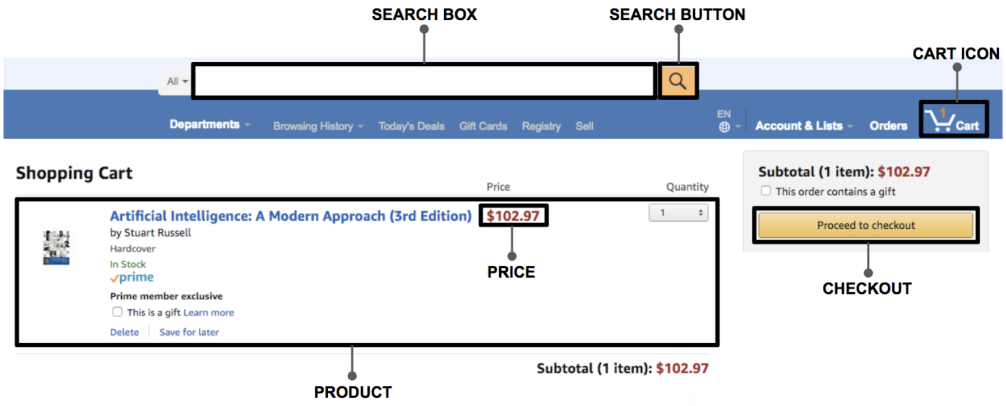

Figure 4.4: Webpage Decomposition and Object Recognition

Figure 4.5 shows another example of an actual ML-based classification system recognizing various components (Page title, widget labels, and error messages) on an arbitrary web page.

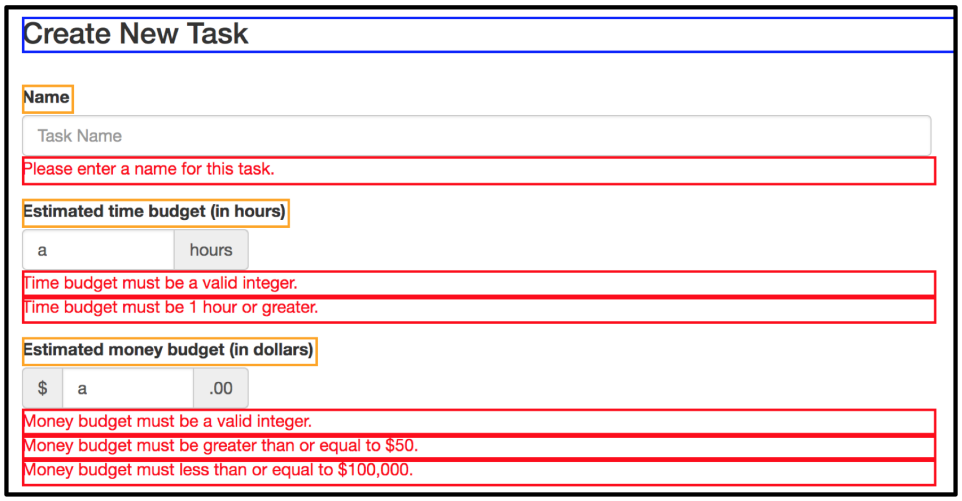

Figure 4.5: Webpage Component Recognition

4.3 Test Flow Sequence Problem

The next step of our approach is to model the test flow process as a sequence problem. Figure 4.6 shows the workflow for developing and validating a test flow generator using neural networks. The phases of the workflow are as follows:

1. *Model Test Flow Process As Sequence Problem.* Before modeling the test flow process, we must first define the term for our purposes. A *test flow* is a sequence of events that can be performed against a web application, and that can be used to represent a test. Given this, we model the human test flow execution process as a sequence problem. Framing test flows as such enables us to apply various ML techniques for generation purposes.

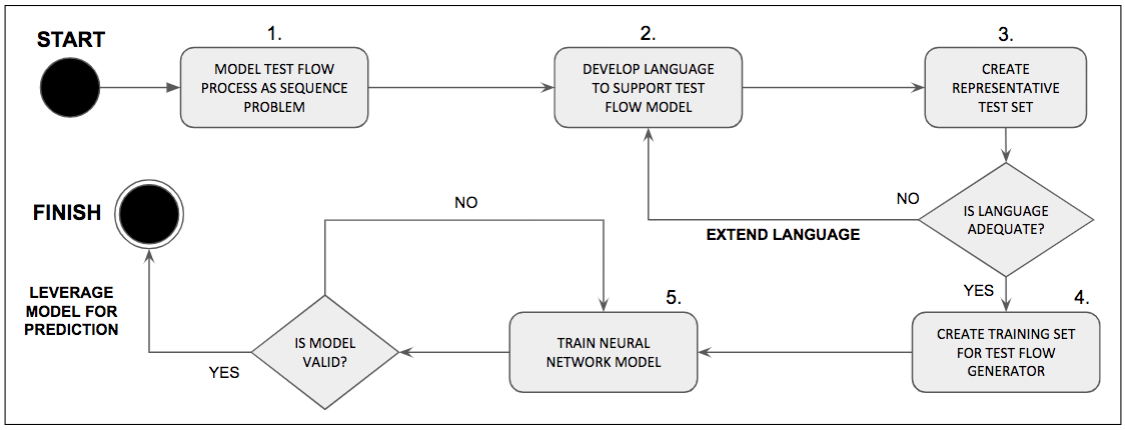

Figure 4.6: Workflow for Developing and Validating a Test Flow Generator using Neural Networks

2. *Develop Language to Support Test Flow Model.* To enable the generalized description of web application test flows, we develop a language and accompanying grammar, *Test Flow Specification Language*. The language must be expressive enough to cover meaningful tests, yet constrained enough to reduce the data complexity of ML-based techniques to test flow generation.

3. *Create Representative Test Set.* A set of tests that span several web pages belonging to disparate domains is hand-crafted. To measure the adequacy of the language, we test the ability of the language to express test flows that cover the representative test set. Test flows are hand-crafted with the purpose of covering tests in a manner that is reusable across SUTs. This is an iterative process that involves extending the language, and possibly expanding the test set.

4. *Create Training Set For Test Flow Generator.* A set of test flows that represent meaningful tests against web applications is hand-crafted as training data.

5. *Train Neural Network Model.* An ML-based system capable of learning sequential information is developed and trained to generate valid test flows that belong to the Test Flow Specification Language. The ML system is trained until the model is able to produce valid test flows at a rate above a desired threshold. The generated test flows are validated for correct grammar using a language parser.

Beyond representing a sequence of actions executable on an SUT, a test flow is also associated with a set of expected observations. Next, we describe an approach to mimicking and modeling the behaviors of human testers when executing test flows. Specifically, we focus on the human ability to *perceive*, *act*, and *observe*. The steps that a human tester follows while executing test flows may be modeled as shown on Figure 4.3.

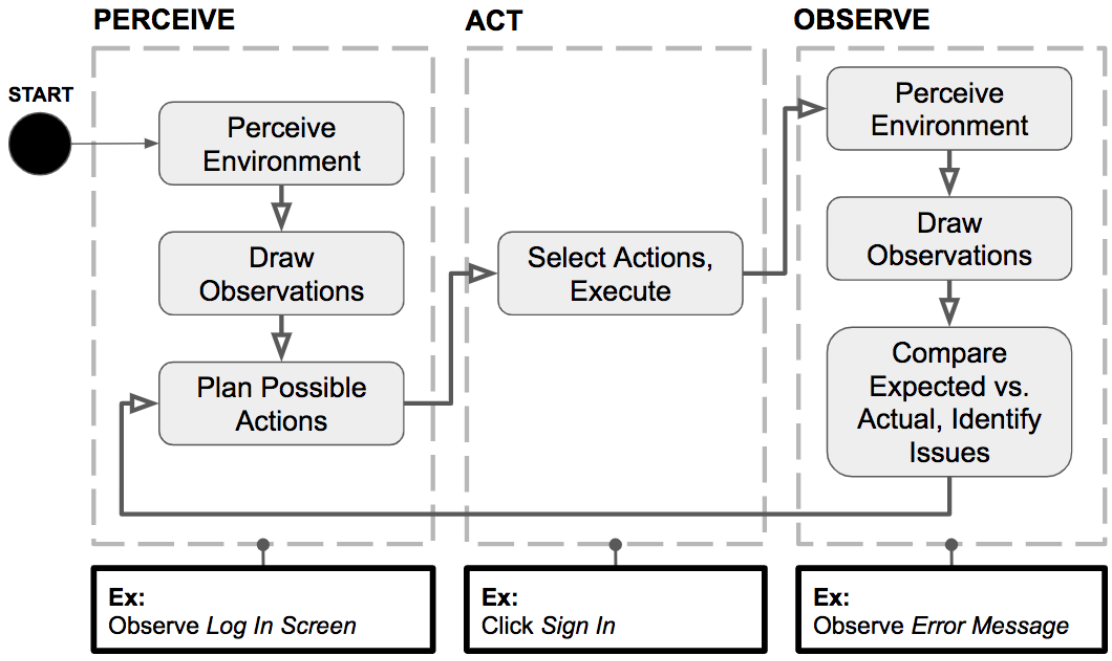

Figure 4.3: Event Sequence Model for Test Flows

We present the test flow execution process as a sequence of steps where each step belongs to one of three categories: *Perceive*, *Act*, *Observe*. Steps belonging to the *Perceive* category focus on establishing preconditions for a test flow, and involve drawing observations from the environment, and action planning before selecting an appropriate set of actions. Steps belonging to the *Act* category focus on selecting and executing appropriate actions based on the previously constructed plan. Finally, steps belonging to the *Observe* category focus on comparing expected vs. actual SUT behavior, and deciding on correctness. We support the continuity property of the testing process by allowing the results of the *Observe* steps to initiate a new *Perceive* stage, thus forming a cycle.

Defining the test flow process using this framework is the first step towards the goal of being able to represent the process in a machine-understandable format. Next, we define a language that may be used to express test flows that fit into our framework. The language must promote the use of webpage component abstractions that utilize web classifiers in place of SUT-specific information. This supports the generality of the approach.

4.4 Test Flow Specification Language

In this section, we specify a language which can be used to represent the process outlined in Section 4.3. The language specification is defined as follows:

Test Flow Language Grammar Specification

⟨*flow*⟩ ::= ⟨*observation-list*⟩ ⟨*component-action-list*⟩

⟨*observation-list*⟩

⟨*observation-list*⟩ ::= ⟨*observation*⟩ ' ' ⟨*observation-list*⟩ | ⟨*observation*⟩

⟨*observation*⟩ ::= '`Observe`' ⟨*qualifier-list?*⟩ ⟨*component*⟩ |

'`NotObserve`' ⟨*qualifier-list?*⟩ ⟨*component*⟩ |

'`Or(`' ⟨*boolean-observations*⟩ '`)`' |

'`Observe`' ⟨*qualifier-list?*⟩ '`In Collection`' |

'`NotObserve`' ⟨*qualifier-list?*⟩ '`In Collection`' |

'`Observe`' ⟨*capture*⟩|⟨*not-capture*⟩ '`In Collection`' |

'`NotObserve`' ⟨*capture*⟩|⟨*not-capture*⟩ '`In Collection`' |

'`Observe`' ⟨*qualifier-list?*⟩ ⟨*component*⟩ ⟨*capture*⟩

⟨*boolean-observations*⟩ ::= ⟨*observation*⟩ '`,`' ⟨*boolean-observations*⟩ |

⟨*observation*⟩

⟨*qualifier-list*⟩ ::= ⟨*qualifier*⟩ ' ' ⟨*qualifier-list*⟩ | ⟨*qualifier*⟩ | ''

⟨*qualifier*⟩ ::= '`Required`' | '`Disabled`' | ⟨*learned-qualifier*⟩

⟨*component*⟩ ::= ⟨*element-class*⟩ ⟨*ident*⟩ | ⟨*ident*⟩

⟨*component-action-list*⟩ ::= ⟨*component-action*⟩ ' ' ⟨*component-action-list*⟩ |

⟨*component-action*⟩

⟨*component-action*⟩ ::= '`Try`' ⟨*equivalence-class*⟩ ⟨*component*⟩ |

'`Focus`' ⟨*capture*⟩ '`In Collection`' |

'`Click`' ⟨*component*⟩ |

'`Try`' ⟨*equivalence-class*⟩ ⟨*component*⟩ ⟨*capture*⟩ |

'`Try`' ⟨*capture*⟩|⟨*not-capture*⟩ ⟨*component*⟩

⟨*equivalence-class*⟩ ::= '`VALID`' | '`BLANK`' | '`WHITESPACE`' | ⟨*learned-eq-class*⟩

⟨*element-class*⟩ ::= '`Textbox`' | '`Dropdown`' | '`ErrorMessage`' |

'`Commit`' | '`Cancel`' | ⟨*learned-el-class*⟩

⟨*capture*⟩ ::= '$' ⟨*ident*⟩

⟨*not-capture*⟩ ::= '!' ⟨*ident*⟩

⟨*learned-qualifier*⟩ ::= ⟨*ident*⟩

⟨*learned-eq-class*⟩ ::= ⟨*ident*⟩

⟨*learned-el-class*⟩ ::= ⟨*ident*⟩

⟨*ident*⟩ ::= _?[A-Z][_A-Z0-9]*

The main building blocks of the language are *components*, *actions*, and *observations*. Components represent elements on a web page. Observations represent information about components that can be perceived from a given web page. Finally, actions are interactions that may be performed on components. By interleaving observations and actions, the language allows for the specification of test flows. The language supports the use of learned abstract objects, instead of using specific input values and observed text values that may only be pertinent to a single SUT or only pertinent to a specific domain of software applications. Example test flows constructed using the language are shown in Figure 4.4.

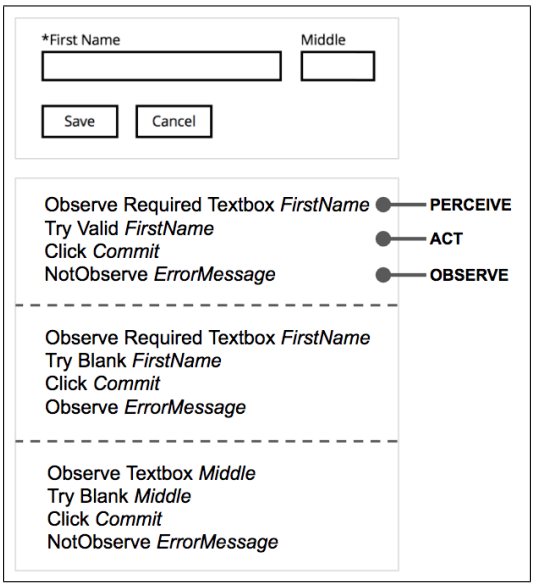

Figure 4.4: Example Test Flows

The example test flows use basic language features to express simple tests focused on testing the required field behavior of the shown form. The language supports building test flows that cover a variety of additional tests. In this section, we will discuss a few of the notable language features.

4.4.1 Boolean Operators

In some cases, it is useful to introduce flexible observations into a test flow in order to promote re-usability. Some minute differences in system behavior across disparate SUTs can be settled using a simple boolean operator. For example, when filling out a form using missing or invalid values, most systems will produce appropriate error messages upon form submission. Alternatively, other system designs may prevent the form submission in the first place, usually by disabling the form submission widget. When constructing a test flow for such an example, a boolean operator enables the definition of a single test flow that accepts both system behaviors. The *Sign In* example test flow shown in Table 4.2 displays the use of boolean operators.

4.4.2 Variables and Captures

In many cases, it is important to recall values that were input into the SUT in order to determine the expected behavior following a subsequent step. For example, when searching an employee directory, knowing to look for the original search string within the search results is necessary to verify correctness. To address this, the language provides for the definition of variables that hold captured values. The *Register* example test flow shown in Table 4.2 displays the use of *variables* and *captures* to keep track of passwords being entered into the system.

Screen	Websites	Selected Test Cases	Example Reusable Test Flow
Sign In	Bing LinkedIn Amazon Gmail Indeed	Attempt to sign in using blank username/password Attempt to sign in using long username/password Attempt to sign in using username/password containing special characters Attempt to sign in using username/password containing cross-site scripting attack Sign in using valid username and password	Observe Screen *Sign In* Try Blank *User Name* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)
Register	Bing LinkedIn Amazon Gmail Indeed	Attempt to register using valid data and matching passwords Attempt to register using mismatching confirmation password Attempt to register using blank/whitespace email/username Attempt to register using blank/whitespace password	Observe Screen *Register* Observe Required Textbox *Password* Observe Required Textbox *Confirm Password* Try Valid *Password* $PASS Try !PASS *Confirm Password* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)
Search	Google Yahoo Bing Indeed Amazon	Search for a product or item, and verify at least one result exists Search using a cross-site scripting attack string, and verify no results exist	Observe Screen *Search* Observe Textbox *Search Textbox* Try Valid *Search Textbox* $SEARCH Enter *Commit* Observe $SEARCH In *Collection*
Shopping Cart	Amazon Walmart Best Buy Ebay Etsy	Add product to shopping cart, ensure added Delete specific product from shopping cart, ensure deleted	Observe Screen *Product Page* Observe *Product Name* $PRODUCT Click *Add To Cart* Navigate *Shopping Cart* Observe $SEARCH In *Collection*
Generic Forms	Contact Information Form Task Manager	Attempt to submit a form with an empty/whitespace required field Attempt to submit a form with a cross-site scripting attack on each field	Observe Required Text box *Generic* Try Blank *Generic* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)

Table 4.2: Representative Test Set and Example Test Flows

4.4.3 Collections and Focus

Collections commonly appear across many web applications. On search websites, the search results are collections. When viewing a list of products, each item is part of a collection. When viewing a shopping cart on an e-commerce application, all of the products are typically represented as a collection. As such, it is important to be able to construct test flows that support interacting with specific items within a collection, as well as support the ability to draw observations about specific items within a collection. The *Shopping Cart* example test flow shown in Table 4.2 displays the use of *collections* for ensuring that a collection of added products exists on a shopping cart.

4.5 Test Flow Generation

Much research exists on generating sequential information using ML-based techniques [Gra13, Bro18]. A common approach for text generation is to feed a text sequence into a trainable algorithm that in turn outputs a text sequence. The training goal is that the concatenation of both sequences results in a plausible sentence. Examples include teaching a computer system to generate plausible sentences based on training from classical texts [Bro18].

A common practice in NLP is to map words to integers, thereby supporting machine readable sequences for input and output, resulting in a *word encoding*. Our approach leverages word encodings and focuses on generating text using a vocabulary that is limited by the language presented in Section 4.4. It is important that the generated text adhere to the language as this allows us to convert from the generated text to an executable test flow. The result is an ML-based trainable test flow generation system.

To create a trainable sequence generation system, we leverage a specialized form of RNNs. LSTM units are used in order to learn sequential information. LSTM networks have been used for several NLP tasks, and have been shown to be able to generate new sequential data that is plausibly correct for a given problem domain, given enough training data. LSTM networks have also been shown to be able to generate strings belonging to context-free and context-sensitive languages [Gra13]. In order to use test flows as training data, a sliding window data augmentation technique is used in order to expand the training data.

We frame the test flow learning problem as a *supervised learning* problem by taking each individual test flow from the training set and expanding it one word at a time, creating a new training example with a bigger sentence at each iteration.

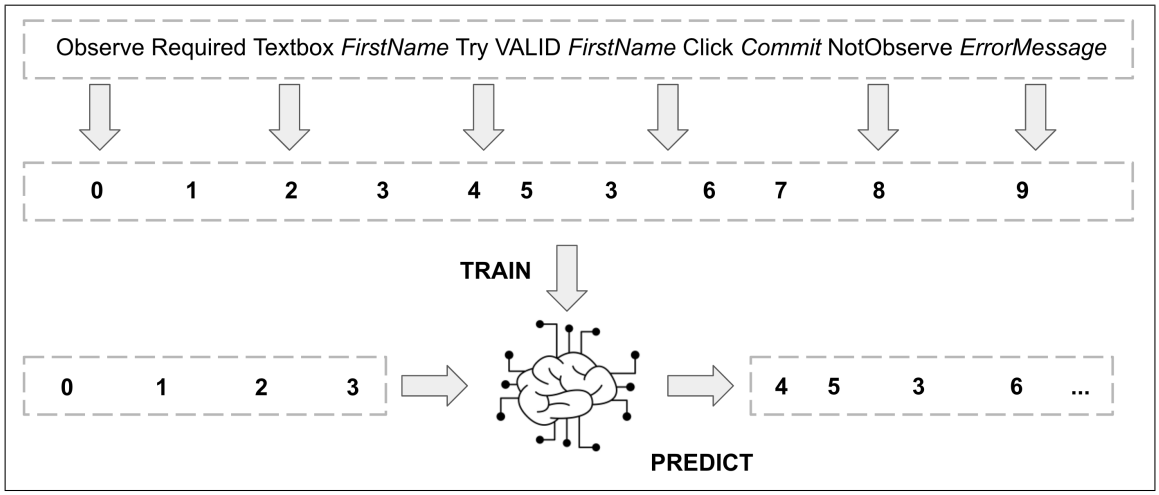

Figure 4.5: LSTM Sequence Training

For each generated training example, the next step of the sentence is the correctly assigned label. An example of how the training data is created is illustrated in Figure 4.5. Once an LSTM network has been trained, a typical prediction task involves querying the trained model for the next step of a sequence, given an initial set of elements. Our overall approach combines this capability of LSTMs with the test flow language specification in order to enable the learning, generalization, and generation of meaningful test flows. An example of how LSTM training data is mapped is shown in Figure 4.5. Our word encoding, augmentation, and vectorization algorithms are detailed below.

Algorithm 1 Create word encoding from a set of test flows

```
1:  procedure CreateEncoding(F)
2:      wordToIndex = {}
3:      indexToWord = {}
4:      index = 0
5:      for n = 1 to |F| do
6:          for i = 1 to |F[n].steps| do
7:              word = F[n].steps[i].lower().strip()
8:              if word ∉ wordToIndex then
9:                  index = index + 1
10:                 wordToIndexs[word] = index
11:                 indexToWord[index] = word
12:     return wordToIndex, indexToWord
```

Algorithm 1 shows our procedure for creating a word encoding from a set of test flows. Lines 2-4 initialize the mapping from a word to an index, and a mapping from an index to a word. Line 5 loops over each input test flow, and line 6 loops over each step within the current test flow. Line 7 retrieves the next word in the test flow. Lines 8 checks if an index does not already exist for the current word. If the word has not been processed yet, an index will not exist for it, and lines 9-11 will assign an index to the word and update both mappings. Finally, the algorithm returns the word and index mappings.

Algorithm 2 Generate expanded vectorized training set

```
1: procedure GenerateTrainingSet(F)
2:     wordToIndex, indexToWord = CreateEncoding(F)
3:     stepLists = []
4:     nextStep = []
5:     X = np.zeros(|F|, MAXLEN, |indexToWord|))
6:     Y = np.zeros(|F|, |indexToWord|)
7:     for n = 1 to |F| do                          ▷ Encode each test flow step.
8:         for i = 1 to |F[n].steps| do
9:             F[n].steps[i] = F[n].steps[i].lower().strip()
10:            F[n].steps[i] = wordToIndex[F[n].steps[i]]
11:    for n = 1 to |F| do ▷ Expand training data using sliding subflow windows.
12:        for i = 1 to |F[n].steps| do
13:            stepLists.append(F[n].steps[: i + 1])
14:            nextStep.append(F[n].steps[i + 1])
15:    for n = 1 to |stepLists| do                  ▷ Vectorize training data.
16:        for i = 1 to |stepLists[n]| do
17:            X[n, i, stepLists[n][i]] = 1
18:        Y[n, nextStep[n]] = 1
19:    return X, Y
```

Algorithm 2 shows our procedure for creating a training data set given a set of test flows. Line 2 invokes algorithm 1 to create a word encoding. Line 3 initializes the list that will hold the expanded set of training data points. Line 4 initializes a list that will hold the correct next word for each training data point. Lines 5-6 initialize the vectors necessary to properly represent the training data as input to an LSTM neural network. Lines 7-10 loop through each word in each test flow, converting each word into the appropriate encoded value. The algorithm frames the flow learning problem as a supervised learning problem by taking each individual test flow from the training set and expanding it one word at a time, creating a new training example with a bigger sentence at each iteration. For each generated training example, the next step of the sentence is the correct assigned label. Lines 11-14 expand the training data by processing each test flow and growing each test

flow one word at a time, storing the currently grown test flow and the correct next word as a new training data point. An example of how this works is shown in Table 4.5. Lines 15-17 vectorize the expanded training data points and represent them as matrices, where each element represents the existence of a specific word on a specific position within a specific training data point sentence. Line 18 vectorizes the correctly assigned next word for each training data point. Finally, Line 19 returns both the training data and correct answer vectors.

Input Sentence	Prediction
Observe	Required
Observe Required	Textbox
Observe Required Textbox	FirstName
Observe Required Textbox FirstName	Try
Observe Required Textbox FirstName Try	Valid
Observe Required Textbox FirstName Try Valid	FirstName
Observe Required Textbox FirstName Try Valid FirstName	Click
Observe Required Textbox FirstName Try Valid FirstName Click	Commit
Observe Required Textbox FirstName Try Valid FirstName Click Commit	NotObserve
Observe Required Textbox FirstName Try Valid FirstName Click Commit NotObserve	ErrorMessage

Table 4.3: Test Flow Training Data Expansion

Given the training data, our approach focuses primarily on training a specialized form of recurrent neural networks. Long short-term memory (LSTM) units are used in order to learn sequential information. LSTM networks have been used for several natural language processing tasks, and have been shown to be able to generate new sequential data that is plausibly correct for a given problem domain, given enough training data. Once an LSTM network has been trained, a typical prediction task is ask the trained model for the next step of a sequence, given an initial set of elements. Our approach combines LSTMs with an abstracted flow language specification in order to enable the learning of meaningful test flows, in order to be able to generalize

across examples, and in order to generate new test flows. Our work involved creating a dataset containing a total of 5534 labeled examples, which were expanded from 922 test flows. Steps B4 and B5 of our approach, shown in Figure 4.6, focus on tweaking our dataset and tuning hyperparameters to reduce bias and variance.

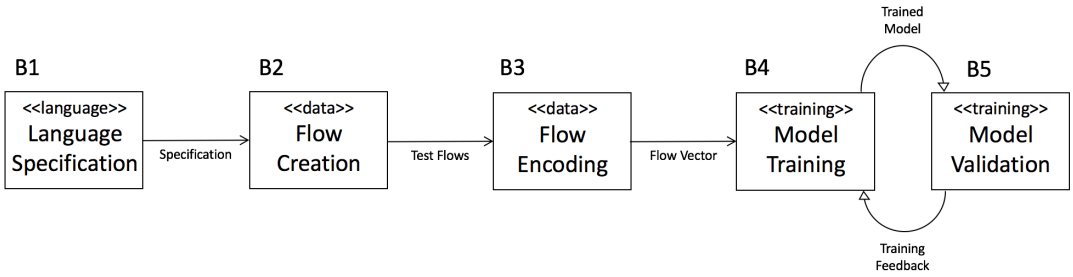

Figure 4.6: Test Flow Learning Approach

4.6 Test Case Generation and Execution

Once we have reached the point where we can both learn to classify web components, and learn meaningful test flows against learned components, we can use our trained models to generate executable test cases against a SUT. Our test generation and execution system consists of a collection of agents that work in cooperation with the test flow generation and web classification systems. A test coordinator is used to orchestrate, distribute, and load-balance queued generated test flows. A knowledge base is included to keep track of and remember information on which test cases have been completed. The system allows the number of agents to be scaled, and all communication and orchestration happens over a message bus channel. All major system components are shown in Figure 4.7.

Each test agent implements the necessary systems to orchestrate the various required interactions. Given a new SUT, we will first generate a list of observations

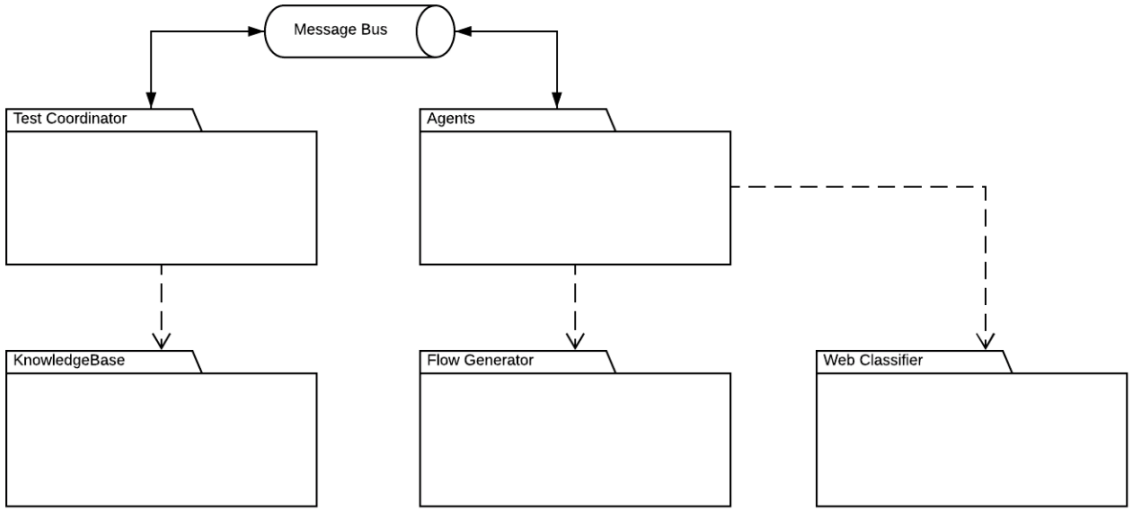

Figure 4.7: Test Case Generation and Execution Packages

using our web component classification system. We will then encode test flow fragments, and use our trained LSTM networks to expand upon possible next steps of previously trained test flows. Since test flows are generalized using our test flow specification language, we can map a generalized test flow into a concrete test flow that is executable for a given SUT. All of the major packages and classes that are necessary to carry this process out are shown in Figure 4.8.

Our agent-based design involves the following major packages within an agent:

1. *Grammar*. Contains all necessary classes and implementation for processing test flows using a specialized grammar. Contains the *TestFlow* class, an implementation of a test flow. A *test flow* is a sequence of events that can be performed against a web application, and that can be used to represent a test. In addition, contains the *SequenceParser* class which utilizes the Python Lark framework and a predefined grammar to parse test flow sequences generated by the test flow generation system.

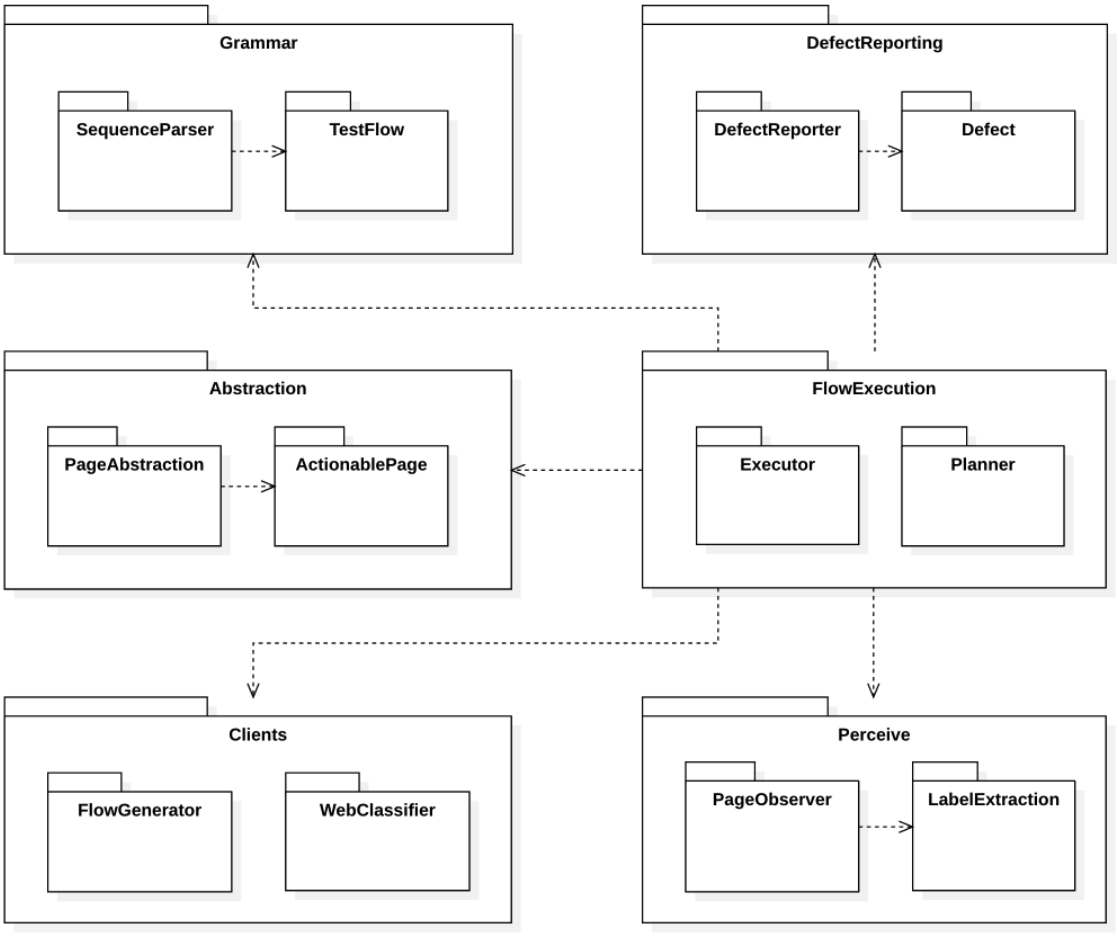

Figure 4.8: Test Agent Packages

2. *Defect Reporting.* Responsible for collecting and reporting defect information. Contains the *Defect* class, an implementation that contains information about problems found in the SUT during execution. The *DefectReporter* class is responsible for triaging defects and preventing duplicate defects from being reported.

3. *Abstraction.* Responsible for reducing state space explosion when automatically crawling web applications. Contains the *ActionablePage* class which represents abstract forms of all encountered SUT pages. Abstract pages contain

information about actionable widgets and important static widgets. Static widget importance is determined by the web classifier system. The *Page-Abstraction* class is responsible for carrying out the abstraction process that converts a concrete page to an abstract page.

4. *Clients.* Contains all necessary client wrappers to communicate with the AI-based classifiers. The *WebClassifier* client allows communication with the web classification system, and the *FlowGenerator* client allows communication with the test flow generation system.

5. *Perceive.* Responsible for perceiving application state. The *PageObserver* class extracts information from the current page being visited in the SUT. The *LabelExtraction* class implements heuristics for extracting widget labels.

6. *Flow Execution.* Responsible for automatically executing test flows against a concrete SUT. The *Planner* class processes test flows recommended by the *FlowGenerator* client and filters out any flows that cannot be executed based on the current execution context. The *Executor* class is responsible for carrying out concrete execution actions.

A snippet of the implementation for the main control loop for a test agent is shown in Figure 4.9. On line 2, the current computed render tree is captured from the web driver. A page analysis that processes the computed tree in order to extract widget classifications is executed on line 3. The abstract state (a simplified version of the computed tree) is created on Line 4. A set of observations derived from the abstract state and the page analysis is constructed on Line 5. On lines 9-16, a test flow is generated, parsed, and planned for each observation. On lines 18-19, each planned test flow is published onto an external and shared planned flow queue. On lines 21-23, a planned test flow is popped off the external queue and executed.

```
1  while True:
2      computed_tree = web_driver.get_computed_render_tree(context)
3      page_analysis = web_classifier.run_analysis(context, computed_tree)
4      abstract_state = page_abstraction.process(context, computed_tree)
5      observations = observer.perceive(abstract_state, page_analysis)
6
7      test_flow_queue = []
8
9      for observation in observations:
10         generated_flow = flow_gen.generated_flow(context, str(observation))
11         if generated_flow is not False:
12             parsed_flow = sequence_parser.parse(generated_flow)
13             if parsed_flow:
14                 planned_flows = flow_planner.plan(context, abstract_state, page_analysis, parsed_flow)
15                 if planned_flows is not False:
16                     test_flow_queue.extend(planned_flows)
17
18     for planned_flow in test_flow_queue:
19         flow_publisher.publish(planned_flow)
20
21     if len(test_execution_queue) > 0:
22         planned_flow_to_execute = test_execution_queue.pop(0)
23         flow_executor.execute(context, abstract_state, web_driver, planned_flow_to_execute)
```

Figure 4.9: Agent Control Loop

A snippet of the implementation provided by the *FlowExecution.Planner* system is shown in Figure 4.10. The algorithm is invoked with an abstract state, the page analysis, and the generated test flow as input. The purpose of the algorithm is to map the abstract generated flow to a concrete executable flow. It is possible for a single abstract flow to generate multiple concrete flows. Line 2 initializes a list that will hold any flows planned by the algorithm. Line 3 initializes a list that holds all of the planned steps. Line 5 loops through every action called for by the generated test flow. Lines 6-10 handle the case of the abstract flow requiring text input onto a widget. Line 7 attempts to find the widget using the label described by the generated test flow, and Line 10 appends both the action and the concrete widget onto the planned steps (explanation continues on next page).

```
1  def plan(self, context, abstract_state, page_analysis, flow):
2      planned_flows = []
3      plan_steps = []
4
5      for action in flow.act.actions:
6          if action.action == 'TRY':
7              widget = abstract_state.find_widget_with_label(action.component, 'SET')
8              if not widget:
9                  return False
10             plan_steps.append([(action, widget)])
11
12         elif action.action == 'CLICK':
13             if action.element_class not in page_analysis['analysis']:
14                 return False
15             candidate_widgets = page_analysis['analysis'][element_class]
16             possible_steps = []
17             for candidate_key in candidate_widgets:
18                 actual_widget = abstract_state.widget_map[candidate_key]
19                 if 'CLICK' not in actual_widget['actions']:
20                     continue
21                 possible_steps.append((action, actual_widget))
22             plan_steps.append(possible_steps)
23
24     cartesian_product = itertools.product(*plan_steps)
25
26     for sequence in cartesian_product:
27         planned_flows.append(PlannedTestFlow(abstract_state, flow, sequence))
28
29     return planned_flows
```

Figure 4.10: Test Flow Execution Planner

For readability, Figure 4.11 is a copy of the previous code snippet. Lines 12-22 handle the case of the abstract flow requiring clicking on a widget. Multiple candidate widgets may be found when the previously collected page analysis is processed for matching widget labels, resulting in a list of possible actions being appended as a single step to the overall planned steps. Line 24 handles this possibility by constructing the Cartesian product of all possible planned flows across all possible multiple match subsets. Lines 26-27 convert each flow in the Cartesian product into a planned test flow. Finally, Line 29 returns the planned flows.

```
1  def plan(self, context, abstract_state, page_analysis, flow):
2      planned_flows = []
3      plan_steps = []
4
5      for action in flow.act.actions:
6          if action.action == 'TRY':
7              widget = abstract_state.find_widget_with_label(action.component, 'SET')
8              if not widget:
9                  return False
10             plan_steps.append([(action, widget)])
11
12         elif action.action == 'CLICK':
13             if action.element_class not in page_analysis['analysis']:
14                 return False
15             candidate_widgets = page_analysis['analysis'][element_class]
16             possible_steps = []
17             for candidate_key in candidate_widgets:
18                 actual_widget = abstract_state.widget_map[candidate_key]
19                 if 'CLICK' not in actual_widget['actions']:
20                     continue
21                 possible_steps.append((action, actual_widget))
22             plan_steps.append(possible_steps)
23
24     cartesian_product = itertools.product(*plan_steps)
25
26     for sequence in cartesian_product:
27         planned_flows.append(PlannedTestFlow(abstract_state, flow, sequence))
28
29     return planned_flows
```

Figure 4.11: Test Flow Execution Planner (Copy)

A snippet of the implementation provided by the *FlowExecution.Executor* system is shown in Figure 4.12. This algorithm is responsible for executing a planned test flow. Lines 1-11 loop over all necessary actions that are called for by the planned flow. Lines 4-7 handle the case where a given widget must be set to a value that falls within an equivalence class specified by the planned flow. Line 5 retrieves a concrete input for a given equivalence from a system that provides seeded inputs. Lines 9-11 handle the case where a widget must be clicked. Lines 13-16 handle retrieving the latest computed render tree, page analysis, abstract state, and observations. Line 17 creates a list of actual observation hash codes. Lines 19-26 loop through each expected observation and ensure that each expected positive observation is present within the actual observation hashes, and that each expected negative observation is not present within the actual observation hashes. On line 26, a defect is raised if an expectation is not met.

```
1   for step in planned_flow.bound_actions:
2       action = step[0]
3       widget = step[1]
4       if action.action == 'TRY':
5           value = self.aide.get_concrete_inputs(action.component, action.equivalence_class)
6           success = web_driver.perform_action(context, widget["selector"], "SET", value)
7           if not success: return False
8
9       elif action.action == 'CLICK':
10          success = web_driver.perform_action(context, widget["selector"], "CLICK", None)
11          if not success: return False
12
13  computed_tree = web_driver.get_computed_render_tree(context)
14  page_analysis = web_classifier.run_analysis(context, computed_tree)
15  abstract_state = page_abstraction.process(context, computed_tree)
16  observations = observer.perceive(abstract_state, page_analysis)
17  actual_observations_hashed = [hash(obs) for obs in observations]
18
19  for i in range(len(planned_flow.observe.observations)):
20      expected_observation = flow.observe.observations[i]
21      expected_hash = hash(expected_observation)
22      success = expected_observation.is_positive and
23                expected_hash in actual_observations_hashed
24      success = success or (expected_observation.is_negative and
25                            expected_hash not in actual_observations_hashed)
26      if not success: self.defect_reporter.add_defect(planned_flow, i)
27
28  return True
```

Figure 4.12: Test Flow Executor

4.7 Summary

In this chapter we presented our approach to an AI-driven test generation approach to reduce the gap between manual testing and automated testing. Our approach includes using training data to train the web classifier and the flow generator, whose output are then fed into the test generator. The test generator uses a test specification language and the input from the web classifier and flow generator to create test cases that are then executed by the test executor.

CHAPTER 5

EVALUATION

In this chapter, we describe a prototype implementation and experimentation results for the various different parts of an AI-driven test generation approach described in Section 4.1, including webpage component classification, test flow generation, and test generation and execution. Also, we discuss the results and identify limitations with the approach.

5.1 Webpage Component Classification

In Section 4.2, we discussed the application of image-based classification to the problem of perceiving web application state. Aside from collecting and labeling images, as part of our research we collected and labeled structural information from web pages. To leverage structural information from a web page, we start by collecting a render tree. A render tree contains information on the structure and styling of nodes on the Document Object Model (DOM). A Computed Render Tree (CRT) representation of the webpage is then constructed. In contrast to a standard render tree, a CRT extends browser render trees by collecting the render tree only once the web browser has finalized rendering, and by calculating additional information such as: (1) element positions; and (2) element sizes.

Using the CRT representation, a feature synthesis step is then performed. An element-wise pass is done through all of the elements in the CRT, synthesizing several features for each element. Although the synthesis is done at the local level for each element, the full global context (information on the entire set of elements) is used to compute several features. The feature synthesis results in the generation of training data that can be annotated for the purpose of training ML models.

In order to evaluate the web classification approach, CRTs were collected and hand-labeled for 7 different SUTs, comprised of 95 web pages and 17,360 web elements. Among the labeled data were 122 elements labeled as page titles, 384 elements labeled as widget labels, 91 labeled as error messages, and 16,762 noise data points. For each of the three component classes analyzed, an experiment was done to compare the performance of the following ML classifiers: (1) random forests, (2) J48 decision trees; (3) k-nearest neighbor; (4) support vector machines; and (5) Bayesian networks.

The evaluations were done using percentage split and cross-validation. Performance was measured using accuracy, precision, recall, and F1 score [GG05]. For the problem of classifying widget labels, the results show that the random forest algorithm performed best with an F1-Score of 96.3%. The detailed results are shown in Table 5.1.

	Label Candidates				Error Messages			
Classifier	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Random Forest (100 Estimators)	**99.83%**	**98.9%**	**93.8%**	**96.3%**	**99.99%**	**100.0%**	**98.9%**	**99.4%**
J48 Decision Tree	99.64%	95.0%	88.8%	91.8%	99.94%	91.8%	98.9%	95.2%
K-Nearest Neighbor (K=3)	99.61%	92.7%	89.6%	91.1%	99.98%	97.8%	98.9%	98.4%
SVM	99.29%	99.2%	68.8%	81.2%	99.97%	95.7%	98.9%	97.3%
Bayesian Network	99.13%	78.2%	84.1%	81.1%	99.88%	83.2%	97.8%	89.9%

	Page Titles			
Classifier	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Random Forest (100 Estimators)	99.52%	67.3%	62.3%	64.7%
J48 Decision Tree	99.56%	79.5%	50.8%	62.0%
K-Nearest Neighbor (K=3)	**99.64%**	**82.8%**	**63.1%**	**71.6%**
SVM	99.41%	85.7%	19.7%	32.0%
Bayesian Network	97.32%	17.0%	72.1%	27.5%

Table 5.1: Webpage Component Classification Results

Examples of sampled decision trees from the generated random forests are shown in Figures 5.1 to 5.3. Each figure displays a randomly chosen sampled decision tree generated by the random forest algorithm. Each node on the tree has a true or false outcome for a specified condition against one of the features extracted from the web

component. Following each tree from the root to a leaf will produce a classification for the input web component. The random trees may vary greatly in complexity and size.

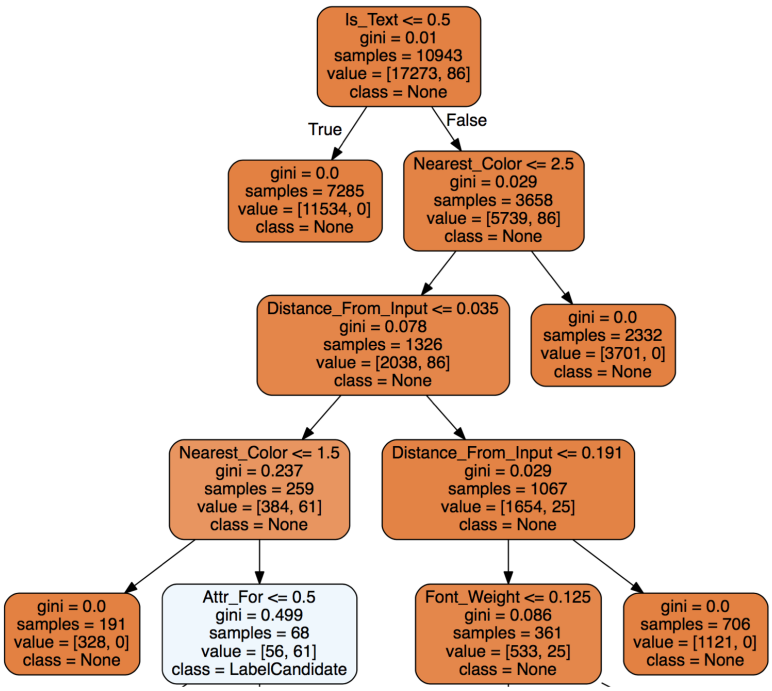

Figure 5.1: Random Forest Sampled Decision Tree (First Sample)

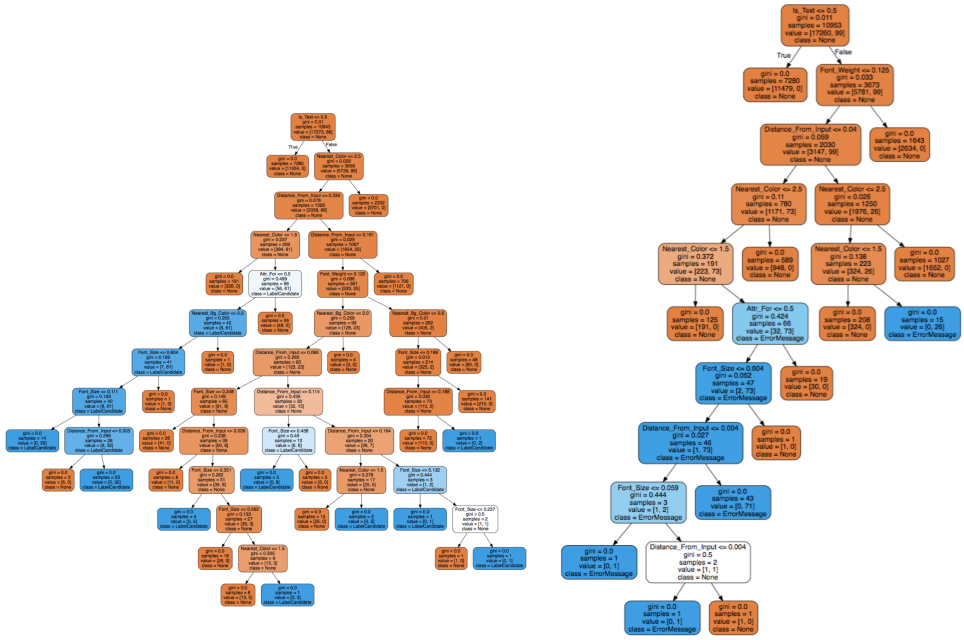

Figure 5.2: Random Forest Sampled Decision Tree (Second Sample)

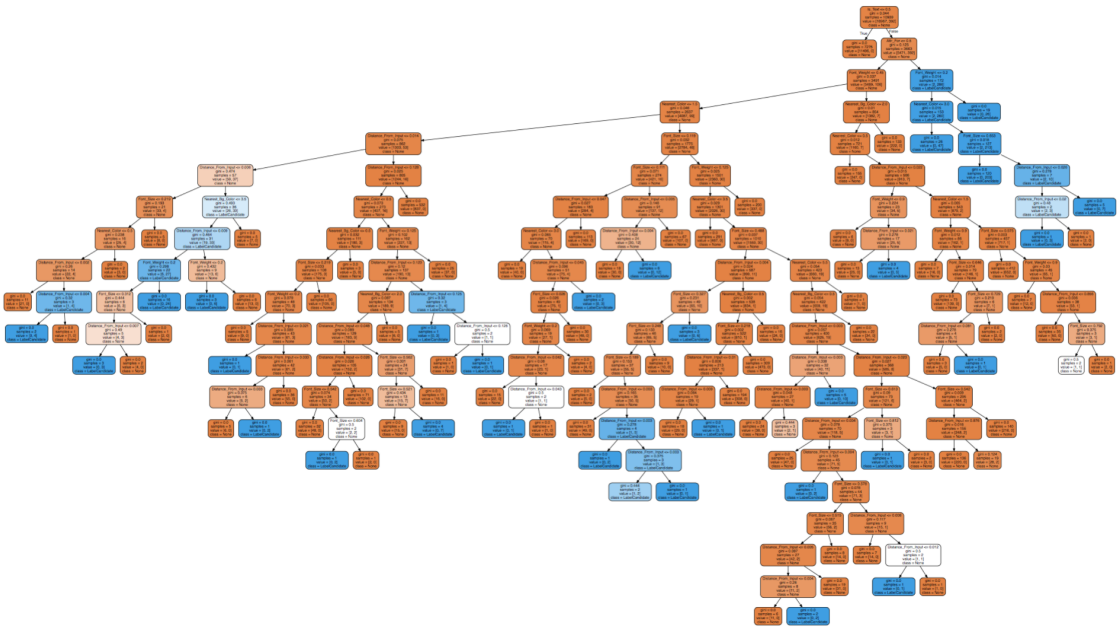

Figure 5.3: Random Forest Sampled Decision Tree (Third Sample)

5.2 Test Flow Generation

In order to validate both the designed language, as well as the test flow generation, we constructed a prototype. We created an Extended Backus-Naur Form (EBNF) [MR] language parser using the Lark library available in Python [Lar18]. We created a total of 250 test flows utilizing all of the features of the language, and successfully parsed the test flows using the custom Lark parser.

The Lark grammar is defined as follows:

```
start: observation_list SPACE component_action_list
        SPACE observation_list  -> test_flow


observation_list: observation SPACE observation_list
                 -> observation_list_sublist
                | observation -> observation_list_single


observation: "OBSERVE" SPACE qualifier_list? SPACE?
                component -> observe
            | "NOTOBSERVE" SPACE qualifier_list? SPACE?
                component -> not_observe
            | "OBSERVE" SPACE capture SPACE "IN" SPACE
                "COLLECTION" -> observe_in_collection
            | "NOTOBSERVE" SPACE capture SPACE "IN" SPACE
                "COLLECTION" -> not_observe_in_collection
            | "OBSERVE" SPACE qualifier_list? SPACE? component
                SPACE capture -> observe_capture
            | "OR(" SPACE conditional_observations SPACE ")"
                -> conditional_observation_list
```

```
conditional_observations: observation SPACE "," SPACE
                conditional_observations
                  -> conditional_list_sublist
              | observation -> conditional_list_single

qualifier_list: qualifier SPACE qualifier_list
                  -> qualifier_list_sublist
                | qualifier -> qualifier_list_single

qualifier: "REQUIRED" -> required
         | "DISABLED" -> disabled
         | "SCREEN" -> screen
         | learned_qualifier

component_action_list: component_action SPACE
                         component_action_list_single
                         -> component_action_list_sublist
                     | component_action
                         -> component_action_list_single

component_action: "TRY" SPACE equivalence_class SPACE
                         component -> try_
                | "TRY" SPACE equivalence_class SPACE
                         component SPACE capture -> try_capture
                | "TRY" SPACE (capture|not_capture) SPACE
                         component -> try_captured
                | "CLICK" SPACE component -> click
                | "ENTER" SPACE component -> enter
```

```
                | "NAVIGATE" SPACE component -> navigate
                | "FOCUS" SPACE capture SPACE "IN" SPACE
                        "COLLECTION" -> focus_in_collection

component: element_class SPACE TOKEN -> component_1
         | element_class -> component_2
         | TOKEN -> component_3

equivalence_class: "VALID" -> valid
                 | "INVALID" -> invalid
                 | "BLANK" -> blank
                 | "WHITESPACE" -> whitespace
                 | "INVALID_LONG" -> invalid_long
                 | "INVALID_SPECIAL_CHARACTERS"
                    -> invalid_special_characters
                 | "INVALID" -> invalid
                 | "INVALID_XSR" -> invalid_xsr
                 | learned_eq_class

element_class: "TEXTBOX" -> textbox
             | "DROPDOWN" -> dropdown
             | "ERRORMESSAGE" -> error_message
             | "COMMIT" -> commit
             | "CANCEL" -> cancel
             | learned_el_class

capture: "$" TOKEN -> capture
```

```
not_capture: "!" TOKEN -> not_capture

learned_qualifier: "LEARNED_QUAL_" TOKEN

learned_eq_class: "LEARNED_EQCLASS_" TOKEN

learned_el_class: "LEARNED_ELCLASS_" TOKEN

TOKEN: /_?[A-Z][_A-Z0-9]*/

%import common.WS -> SPACE

%import common.ESCAPED_STRING -> _STRING
```

Each of the 250 crafted test flows captured real tests that could feasibly be performed by a human tester against a web application. Table 5.2 presents a subset of the tests. Using the sliding window data augmentation technique, we created a dataset containing a total of 2610 labeled examples for the purposes of using ML for test flow generation. We used the Keras [Cho17] framework to build the neural networks for the prototype.

The performance of the LSTM was measured using the built-in categorical cross-entropy loss function available in the Keras neural network programming framework [Cho17]. Training set accuracy was based on whether or not a predicted test flow was a valid test flow sub-sequence in the training data. In addition, the output predictions were only considered valid if the resulting test flow was able to be parsed by the custom Lark parser. Different network topologies, optimization algorithms, and regularization technique settings were compared [KB14, SHK+14, TH12]. The ability of the test flow learning system to extract patterns and generalize was also

Screen	Websites	Selected Test Cases	Example Reusable Test Flow
Sign In	Bing LinkedIn Amazon Gmail Indeed	Attempt to sign in using blank username/password Attempt to sign in using long username/password Attempt to sign in using username/password containing special characters Attempt to sign in using username/password containing cross-site scripting attack Sign in using valid username and password	Observe Screen *Sign In* Try Blank *User Name* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)
Register	Bing LinkedIn Amazon Gmail Indeed	Attempt to register using valid data and matching passwords Attempt to register using mismatching confirmation password Attempt to register using blank/whitespace email/username Attempt to register using blank/whitespace password	Observe Screen *Register* Observe Required Textbox *Password* Observe Required Textbox *Confirm Password* Try Valid *Password* $PASS Try !PASS *Confirm Password* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)
Search	Google Yahoo Bing Indeed Amazon	Search for a product or item, and verify at least one result exists Search using a cross-site scripting attack string, and verify no results exist	Observe Screen *Search* Observe Textbox *Search Textbox* Try Valid *Search Textbox* $SEARCH Enter *Commit* Observe $SEARCH In *Collection*
Shopping Cart	Amazon Walmart Best Buy Ebay Etsy	Add product to shopping cart, ensure added Delete specific product from shopping cart, ensure deleted	Observe Screen *Product Page* Observe *Product Name* $PRODUCT Click *Add To Cart* Navigate *Shopping Cart* Observe $SEARCH In *Collection*
Generic Forms	Contact Information Form Task Manager	Attempt to submit a form with an empty/whitespace required field Attempt to submit a form with a cross-site scripting attack on each field	Observe Required Text box *Generic* Try Blank *Generic* Click *Commit* OR(Observe *ErrorMessage* , Observe Disabled *Commit*)

Table 5.2: Selected Test Cases

tested. As one example, the experiment was concerned with investigating the ability of the learning system to learn the generalized concept of required field validation.

The research and results show that it is possible to model human test flow execution behavior as a continuous sequence problem that involves perceiving web application state, acting upon the webpage, and observing resulting state. We demonstrate how to restrict such testing behavior within the confines of a well-defined language, and we present an ML-based approach for learning to recall and generate strings that belong to the language.

To test the language, 5 categories of disparate web application screen types were selected. For each of the categories, 5 different examples were selected across distinctive web sites. For each of the examples, a set of tests was hand-crafted. The ability of the language to be able to express each of the tests was then determined.

The language was iteratively expanded to accommodate more functionality and expressiveness. Table 5.2 contains a subset of the tests that were selected for this exercise. Each group of tests in the table also contains an example test flow.

Our results suggest that LSTMs can model and generate test flows. One of the challenges observed when training LSTMs was instability in the accuracy of the trained model. Significant fluctuations in accuracy were observed during training. In addition, instability was also observed in the ability of the LSTM to learn patterns and generalize across training examples. Results suggest that the system is able to extract patterns from the training data and is able to form generalizations. For example, results showed that the following pattern was extracted from the training data: after observing a required field, attempting to leave it blank and submitting a form should raise an error message. Table 5.2 shows a detailed breakdown of the performance of each configuration that was evaluated when training the neural networks.

Each trained LSTM model was used for generating a total of 2925 test flows. When evaluating each LSTM configuration, accuracy was measured by dividing the number of generated test flows that were valid strings in our language by the total number of generated test flows. Experiments included varying the number of training iterations, called *epochs*, the number of LSTM layers, and the number of LSTM units per layer. The Adam and RMSProp optimizers were also evaluated, as well as dropout regularization. Based on the results, the highest accuracy (**93.85%**) was achieved when training a single LSTM layer network for 500 epochs, using the RMSProp optimization algorithm. Equivalent accuracy was observed when using multiple stacked LSTM layers. Generally, increasing the number of layers and units improved accuracy; however, training time was also observed to increase. Stacking LSTM layers appears to be a viable option for increasing accuracy as more training

data is added. While dropout regularization generally reduced training set accuracy, it did not greatly affect accuracy when training for longer durations or when using larger networks.

Epochs	Layers	Units	Dropout	Optimizer	Accuracy	Epochs	Layers	Units	Dropout	Optimizer	Accuracy
50	1	32	No	Adam	63.62%	**500**	**1**	**256**	**No**	**RMSProp**	**93.85%**
50	1	32	Yes	Adam	39.86%	**500**	**1**	**256**	**Yes**	**RMSProp**	**93.85%**
50	1	64	No	RMSProp	84.17%	50	2	64+64	No	RMSProp	88.17%
50	1	64	Yes	RMSProp	82.63%	50	2	64+64	Yes	RMSProp	89.03%
50	1	64	No	Adam	81.57%	50	2	64+64	No	Adam	73.06%
50	1	64	Yes	Adam	84.92%	50	2	64+64	Yes	Adam	81.06%
50	1	128	No	RMSProp	88.55%	50	2	128+128	No	Adam	86.67%
50	1	128	Yes	RMSProp	82.63%	50	2	128+128	Yes	Adam	86.67%
50	1	128	No	Adam	88.68%	50	2	128+128	No	RMSProp	89.98%
50	1	128	Yes	Adam	85.64%	50	2	128+128	Yes	RMSProp	85.57%
50	1	256	No	RMSProp	59.59%	50	3	128*3	No	RMSProp	91.49%
50	1	256	Yes	RMSProp	87.15%	50	3	128*3	Yes	RMSProp	86.50%
50	1	256	No	Adam	91.52%	**500**	**3**	**128*3**	**No**	**Adam**	**93.74%**
50	1	256	Yes	Adam	86.19%	**500**	**3**	**128*3**	**Yes**	**Adam**	**93.85%**

Table 5.3: Flow Learning LSTM Results

5.3 Test Generation and Execution

The test generation system was evaluated using the Spring Pet Clinic application [Pet13], shown in Figure 5.4. After training the AI classifiers, the test flow execution system was able to generate 700 unique test cases for the Spring Pet Clinic application, and as a consequence our approach was able to find 12 bugs in the application in under 10 minutes using 5 test agents. In comparison, a human tester was able to find the same number of bugs in 30 minutes, and found a total of 20 bugs in 45 minutes. Most of the bugs that were found include lack of field validation across many fields in the Spring Pet Clinic application. For example, many fields accepted cross-site scripting values, and invalid values in general. In addition, a few bugs were found with required field functionality.

As part of our experiments, we also varied the number of agents to evaluate the scalability of the approach. We found that with a single agent it took about 45

minutes to find as many bugs as 5 agents were able to find in 10 minutes. With 50 agents, it took about 8 minutes to find the same number of target bugs. During the 50 agent experiment, we identified that the response times of the underlying machine learning services (running a single service per machine learning system) significantly degraded, which may explain the diminishing return on defect detection efficiency. Figure 5.5 showcases the distributed nature of the multi-agent system.

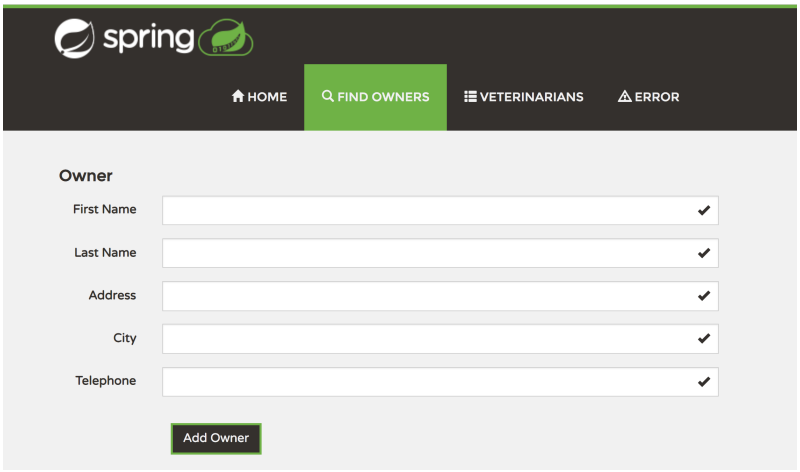

Figure 5.4: Sprint Pet Clinic Website

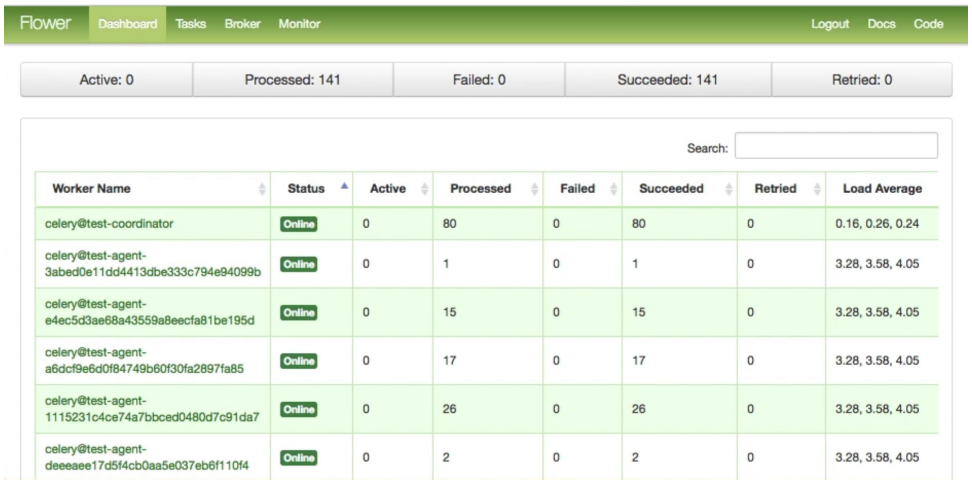

Figure 5.5: Distributed Generated Test Execution

5.4 Discussion

The primary objective of the thesis is to investigate the feasibility of using machine learning approaches such as decision trees and neural networks to automatically generate tests in a way that is independent of any specific web application. The evaluation criteria is that the developed approach should be capable of generalizing from previously trained models, and should generate executable test cases against an unseen web application. As discussed in Sections 5.1 and 5.2, various machine learning systems were trained on a number of different web sites and on a number of different manually hand-crafted test flows. In Section 5.3 we discussed a prototype capable of leveraging these ML-based systems in order to generate and execute test cases against the Spring Pet Clinic application, resulting in defects being found in the application. The web classification subsystem was only trained on 2 example data points from one of the pages of the Spring Pet Clinic application, and the test flow generation system was not trained on any information from the Spring Pet Clinic application. Based on our evaluation, our presented approach is able to better generalize as more data is collected and trained on, and the approach generalizes to new systems with little effort.

The first sub-objective of the thesis is to develop a trainable classifier which perceives application state and allows for the extraction of web page components in a way that is agnostic to a web application. This sub-objective was evaluated using multiple pages from several disparate web applications using accuracy, precision, and recall. As discussed in Section 5.1, the classifier was evaluated against 7 different websites, comprising 95 web pages and 17,360 web elements. Among the labeled data were 122 elements labeled as page titles, 384 elements labeled as widget labels, 91 labeled as error messages, and 16,762 noise data points. Using percentage

split and cross-validation, our evaluation results show an F1-Score of 96.3% for widget label classification, 99.4% for error message classification, and 71.6% for page title classification. Overall, the results were favorable for the different classification problems evaluated.

The second sub-objective is to develop a language that enables the generalized description of web application test flows and to develop an ML classifier for generating test flows. This sub-objective was evaluated by taking a representative set of hand-crafted test cases for a web application and checking the expressiveness of the language and the ability of the language to express each test case as an abstract test flow. In addition, this sub-objective was evaluated by testing the ability of an ML classifier to generate a comparable set of abstract test flows for a previously unseen web application. As discussed in Section 5.2, our evaluation shows that our language was able to express a total of 250 test cases for different types of application screens. In addition, our ML implementation was able to generate abstract flows with 93.85% accuracy. Each of the valid generated abstract flows were applicable to new unseen applications such as the Spring Pet Clinic application, as discussed in Section 5.3.

The third sub-objective is to develop a test execution engine that uses ML-generated test flows and seeded test inputs to produce executable tests for a previously unseen web application. This sub-objective was evaluated by comparing the number of defects found by our approach compared to that of the number of defects found by a human testing approach. As discussed in Section 5.3, our approach was able to find 12 defects on the Spring Pet Clinic application, compared to 20 defects found by a human tester on the same application. While there is room for improvement, the results are generally favorable.

5.5 Limitations

While the overall results were favorable with regards to the development of a trainable web classification system, the machine learning experiments were conducted on limited data sets, and as such the trained ML models may have over-fit during training. Only 7 different websites were considered for data collection and training evaluation, and only 17,360 web elements were considered. In addition, only 3 different classification problems were investigated: widget labels, error messages, and page titles. In addition, only a finite set of machine learning classifiers and hyper-parameter configurations were evaluated. As such, it is difficult to predict whether the approach will generalize as more data is collected.

Although the results were favorable for both language expressiveness and ability to generate test flows, there are also a few threats to validity. The language was only expanded to accommodate test cases from 5 different types of web application screens. In addition, a limited set of test cases were selected for each of the 5 types of screens. One of the challenges observed when training LSTMs is instability in the accuracy of the trained model. Significant fluctuations in accuracy were observed during training. In addition, instability was also observed in the ability for the LSTM to learn patterns and generalize across training examples. Also, only a finite set of hyper-parameter configurations and LSTM layer configurations were evaluated.

While we were able to show that our approach could generate test flows, execute test flows, and find defects against an unseen web application, namely the Spring Pet Clinic Application, the evaluation was limited and only attempted to execute generated test flows against a single application. As such, it is difficult to predict how generalizable the approach is. In addition, although the test coordination subsystem

performs de-duplication of generated test cases, and does not en-queue a test case in the event of an execution failure (outside of expectation mismatches), the system does not learn from these events in order to improve test generation. In addition, current test planning algorithms do not implement high level planning with test goals and exit criteria in mind. Instead, they focus on filtering out generated tests that are not executable against the concrete SUT. Overall, this thesis presents an approach that aims to reduce the gap between the testing capabilities of humans and those of machines. Although the results are generally positive, the research presented in this thesis is still limited and only examines a limited scope of each of the presented sub-problems.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis focused in detail on a specific approach for allowing human testers to express tests as reusable test flows, and for leveraging AI and ML to enable the creation of a trainable test flow generation system. The primary objective of the thesis is to investigate the feasibility of using machine learning approaches such as decision trees and neural networks to automatically generate tests in a way that is independent of any specific web application. Based on our evaluation, our presented approach is able to better generalize as more data is collected and trained on, and the approach generalizes to new systems with little effort.

The first sub-objective of the thesis was to develop a trainable classifier which perceives application state and allows for the extraction of web page components in a way that is agnostic to a web application. As discussed in Section 4.2, we frame the problem of understanding webpage components as a supervised learning classification problem. Using an ML-based approach allows us to improve the system by providing additional data, and allows us to raise the level of abstraction by which we interact with webpage components, ultimately promoting test re-usability across systems. As discussed in Section 5.1, the classifier was evaluated against 7 different websites, 95 web pages, and 17,360 web elements. Among the labeled data were 122 page title elements, 384 widget label elements, 91 error message elements, and 16,762 noise data points. Overall, the results were favorable for the different classification problems evaluated, averaging an overall classification F1-Score of 89.1%.

The second sub-objective was to develop a language that enables the generalized description of web application test flows and to develop an ML classifier for generating test flows. As discussed in Sections 4.3, 4.4, and 4.5, we developed a test flow specification language that may be used to manually create test flows

for training, and for parsing automatically generated test flows. We also developed an ML-based approach to test flow generation via the use of techniques previously applied to text generation. Our evaluation shows that our language was able to express a total of 250 test cases for different types of application screens. In addition, our ML implementation was able to generate abstract flows with 93.85% accuracy.

The third sub-objective was to develop a test execution engine that uses ML-generated test flows and seeded test inputs to produce executable tests for a previously unseen web application. As discussed in Section 4.6, we developed a system that consists of a collection of agents and coordinators that work in cooperation with our test flow generation and web classification systems in order to automatically explore a System Under Test, automatically perceive application state across various pages, and automatically generate and execute test cases. In addition, our test execution system is distributed and scalable as testing is more efficient as more test agents are deployed. Our approach was able to find 12 defects on the Spring Pet Clinic application within 10 minutes using 5 test agents.

While the results are promising, our work is limited. There are still outstanding research challenges, such as the oracle problem. Although we show the ability to generate abstract test flows that include expectation information, our training data and prototypes are limited in that they are focused on generic problems, and not domain-specific details. To construct domain-specific oracles, more training data with higher variety must be collected, and more analysis must be done.

Further work must be done on expanding the feature synthesis work for the web classification approach in order to improve the classification power of the approach. While our work focused on classifying individual elements on web pages, remaining work must be done in order to classify a group of elements as belonging to a particular class. Techniques that utilize image data and convolutional neural networks

will be explored. In addition, we also plan to explore spatial encoding techniques for expressing webpage positional information in a sparse form that does not require the use of images [GHS16].

To improve the generalization of the approach, additional training and test data must be collected, and the scope of the problem domains from which the training website data belongs to must be expanded. Additional remaining work involves further development of a tool that functions as a web browser plugin, allowing a human user to supply labeled information about any webpage component on any webpage. This labeled data may be used to iteratively improve the underlying machine learning models. In addition, it would be useful to create a web browser plugin that is capable of observing a human user as they carry out testing activities. The plugin will work in tandem with the web component classification system to record information about web components being interacted on by testing activity. The observations that are automatically collected from human testers may be encoded into strings that belong to the test flow specification language. Once the test flows have been created, they may be used to train the LSTM networks. The trained LSTM models may then be used to generate new test flows.

A promising research direction for future work involves the creation of a hierarchical LSTM-based test flow generation model, whereby higher-level models generate abstract domain-specific concepts. The trained LSTM model detailed in our work would serve as the foundation of the hierarchy. While base-level LSTMs enable the generation of fine-grained test cases against low-level web elements, higher level LSTMs focus on generating coarse-grained test cases that would be useful for testing higher-level domain-specific application work-flows. The LSTM models at each level of the hierarchy are all independently trainable and serve specific purposes, and may be composed to increase test coverage of the generated test case set.

BIBLIOGRAPHY

[ABC+13] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[AH04] Kandel Abraham and Bunke Horst. *Artificial intelligence methods in software testing*, volume 56. World Scientific, 2004.

[AIS18] AIST. IEEE International Workshop on Automated and Intelligent Software Testing, 2018. `http://paris.utdallas.edu/AIST18/`.

[AMRS11] Pekka Aho, Nadja Menz, Tomi Räty, and Ina Schieferdecker. Automated java gui modeling for model-based testing purposes. In *Information technology: New generations (itng), 2011 eighth international conference on*, pages 268–273. IEEE, 2011.

[ANT+18] Jason Arbon, Chris Navrides, Ken Toley, Ryan Bedino, Vicky Fan, and Michelle Petersen. Abstract Intent Test (AIT) Syntax, 2018. `https://goo.gl/XbwgkL`.

[AO17] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2 edition, 2017.

[Arb17] Jason Arbon. AI for Software Testing. In *Pacific NW Software Quality Conference*. PNSQC, 2017.

[BDTD+16] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[Bro18] Brownlee, Jason. Text Generation With LSTM Recurrent Neural Networks in Python with Keras. `https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/`, 2018. Accessed: 2018-04-10.

[Cap17] Capgemini. World quality report, 2017. `https://www.capgemini.com/world-quality-report-2016-17/`.

[Cho17] Francois Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017.

[De'07] Glenn De'Ath. Boosted trees for ecological modeling and prediction. *Ecology*, 88(1):243–251, 2007.

[Fow13] Martin Fowler. Page objects. `https://martinfowler.com/bliki/PageObject.html`, 2013. Accessed: 2018-03-10.

[GG05] Cyril Goutte and Eric Gaussier. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In *European Conference on Information Retrieval*, pages 345–359. Springer, 2005.

[GHS16] Tomas Gogar, Ondrej Hubacek, and Jan Sedivy. Deep neural networks for web page information extraction. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 154–163. Springer, 2016.

[Gra13] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[Gri18] Ilya Grigorik. Render-tree construction, layout, and paint. `https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction`, 2018. Accessed: 2018-03-10.

[GS01] Felix A Gers and E Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

[Gut13] Allan Gut. *Probability: A Graduate Course*. Springer-Verlag, New York, 2nd edition, 2013.

[KA18] Tariq M. King and Jason Arbon. AI for Software Testing Association, 2018. `https://www.aitesting.org/`.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Lar18] Lark. Lark. `https://github.com/lark-parser/lark`, 2018. Accessed: 2018-03-10.

[LO17] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2017.

[Lui18] Angel Luis. Introduction to the DOM, 2018. `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction`.

[LW+02] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[Mar14] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[MBN03] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. Gui ripping: Reverse engineering of graphical user interfaces for testing. In *null*, page 260. IEEE, 2003.

[Mem04] Atif M Memon. Automated gui regression testing using ai planning. In *Artificial Intelligence Methods In Software Testing*, pages 51–99. World Scientific, 2004.

[Moy16] Christopher Moyer. How Google's AlphaGo beat a Go world champion. *The Atlantic*, 28, 2016.

[MR] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK.

[Ng18] Andrew Ng. Machine learning, 2018. `https://online.stanford.edu/course/machine-learning-1`.

[Ona16] Aytuğ Onan. Classifier and feature set ensembles for web page classification. *Journal of Information Science*, 42(2):150–165, 2016.

[Pet09] Leif E Peterson. K-nearest neighbor. *Scholarpedia*, 4(2):1883, 2009.

[Pet13] Spring PetClinic. Petclinic. `https://github.com/spring-petclinic`, 2013. Accessed: 2018-03-10.

[PS13] Tina R Patil and SS Sherekar. Performance analysis of Naive Bayes and J48 classification algorithm for data classification. *International Journal of Computer Science and Applications*, 6(2):256–261, 2013.

[Qui96] J. R. Quinlan. Bagging, boosting, and c4.s. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, pages 725–730. AAAI Press, 1996.

[RMPM12] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[Sch97] Robert J Schalkoff. *Artificial neural networks*, volume 1. McGraw-Hill NY, 1997.

[Sci18] ScikitLearn. Decision trees, 2018. `http://scikit-learn.org/stable/modules/tree.html`.

[Sel18] Selenium Development Team. Selenium - Web Browser Automation. `https://www.seleniumhq.org/`, 2018. Accessed: 2018-03-10.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. 2002.

[TH12] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[VKCF+15] Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. Testar: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.

[Wan05] Lipo Wang. *Support vector machines: theory and applications*, volume 177. Springer Science & Business Media, 2005.

[Whi00] James A Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.

VITA

DIONNY SANTIAGO

2015	B.Sc., Computer Science Florida International University Miami, Florida
2015–Present	Test Architect Ultimate Software Weston, Florida
2014–2015	Iteration Manager Ultimate Software Weston, Florida
2010–2014	Software Test Engineer Ultimate Software Weston, Florida
2010	Software Engineering Intern Ultimate Software Weston, Florida
2010–2011	Research Assistant Florida International University Miami, Florida
2009–2010	Engineering Intern Infotech Soft Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Santiago, Dionny, et al. *Abstract Flow Learning for Web Application Test Generation.* Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '18). ACM, 2018.

Santiago, Dionny, et al. *AI-Driven Test Generation: Machines Learning from Human Testers.* Proceedings of the 36th Pacific NW Software Quality Conference. PNSQC, 2018.

Patent: System For Autonomously Testing A Computer System, Filed 2018

Patent: System For Optimizing System Resources And Runtime During A Testing Procedure, Filed 2018

Patent: System For Understanding Navigational Semantics Via Hypothesis Generation And Contextual Analysis, Filed 2018

Patent: System For Providing Autonomous Discovery Of Field Or Navigation Constraints, Filed 2018

Patent: System For Discovering Semantic Relationships In Computer Programs, Filed 2018

Patent: System For Providing Intelligent Part Of Speech Processing Of Complex Natural Language, Filed 2018

King, Tariq M., et al. *Towards a Bayesian Network Model for Predicting Flaky Automated Tests.* 2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, 2018.

King, Tariq M., et al. *Legend: an agile DSL toolset for web acceptance testing.* Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014.

Santiago, Dionny, et al. *Towards Domain-Specific Testing Languages for Software-as-a-Service.* MDHPCL@ MoDELS. 2013.