DANIEL: Towards Automated Bug Discovery By Black Box Test Case Generation & Recommendation

Michael G. Peechatt
mp6510@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Abstract

Finding and documenting bugs in software systems is an essential component of the software development process. A bug is defined as a series of steps that produces behavior which differs from the software specification and requirements. Finding steps to produce such behavior requires expert knowledge of the possible operations of the software in development as well as intuition and creativity. This thesis proposes the Directed Action Node Input Execution Language (DANIEL), a language that represents test cases as directed graphs, where each node represents an action, and possible input arguments for each action are represented along the incoming directed edges. With this representation, it is possible to form a union of all recorded test cases, making a combined directed graph which represents all of the paths of interaction with the developing software. This thesis demonstrates how DANIEL can generate prioritized test cases for a web form application, while also preserving workflow context. Using a graph built on Selenium test cases, we evaluate a random walk, a weighted walk, and model-weighted walks integrating logistic regression and XGBoost to compute the relevant probabilities. We find that the weighted walk discovers the most bugs while the model-weighted walk provides the most meaningful coverage.
DANIEL: Towards Automated Bug Discovery by Black Box Test Case Generation and Recommendation

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Alex Ororbia II, Supervisor

Dr. Carlos Rivero, Reader

Dr. Michael Mior, Observer
Acknowledgments

Alex Ororbia supported me from the beginning of my Master’s thesis journey. He shared my enthusiasm for applying intelligent systems to software quality assurance and set me on the path of research. It has been a long journey, and he has been there every step of the way. I am thankful for his support and guidance.

Michael Mior helped me understand the scope and scalability of my research. I still recount those meetings I spent during his office hours, where I first pitched the idea and showed my first demo.

Carlos Rivero saw the potential in my research early on and was there to help guide my graph traversal methods. His lectures on breaking down algorithms from research papers were invaluable throughout this process.

Special thanks to Avezou Petit-Frere for providing the web form application used in the testing process. That day we met at Java’s truly was a divine appointment.
Dedicated to my beloved parents and to the Prince of Peace.

“Go thy way, Daniel: for the words are closed up and sealed till the time of the end.”

Jesus Christ, Daniel 12:9
## Contents

1 **Introduction**  
   1.1 The Cost of Software Testing ................................ 1  
   1.2 The Anatomy of a Bug ........................................... 2  
   1.3 White Box Testing .............................................. 3  
   1.4 Black Box Testing .............................................. 4  
   1.5 Gray Box Testing .............................................. 4  
   1.6 Current Approach .............................................. 5  
       1.6.1 Flaws With Current Approaches ......................... 5  

2 **Literature Review**  
   2.1 Preventing Bugs ............................................... 9  
       2.1.1 Model Checking With TLA+ .............................. 9  
       2.1.2 Test Driven Development ............................... 9  
   2.2 Writing Better Code ........................................... 10  
       2.2.1 Deep Bugs .............................................. 10  
       2.2.2 VulIntel: Secure Coding Recommendation System ...... 11  
   2.3 Improving the Software Development Process .............. 12  
       2.3.1 Assigning Bugs ....................................... 12  
       2.3.2 Diagnosing Bugs ....................................... 13  
       2.3.3 Software Behavior Graphs ............................ 13  
       2.3.4 Code Change Differences ............................. 14  
   2.4 Generating Test Cases ........................................ 15  
       2.4.1 Springer Verlag’s Online Conference System .......... 15  
   2.5 Recommending Test Cases .................................... 16  
       2.5.1 Sampling By Topic Similarity ........................ 16  
       2.5.2 Recommendation as a Search Problem ................. 16  

3 **The Directed Action Node Input Execution Language**  
   3.1 Modeling Test Cases as Graphs ............................. 18
# CONTENTS

3.2 The Language Syntax ........................................... 19
3.3 Graph Definition .................................................. 22
   3.3.1 Graph Weights ............................................. 23
3.4 Traversing the Graph ............................................. 23
   3.4.1 The Random Walk Traversal ................................. 24
   3.4.2 Weighted Walk .............................................. 25
   3.4.3 Model-Weighted Walk ....................................... 26

4 Experimental Results 31
   4.1 Recording Test Cases ........................................... 31
      4.1.1 Converting to DANIEL Graphs ............................... 32
      4.1.2 Executing Generated Tests .................................. 34
   4.2 Implementing the Model-Weighted Walk .......................... 35
      4.2.1 The Test-Case Embedding Function .......................... 35
      4.2.2 Generating Data with Active Learning ...................... 36
      4.2.3 Sampling Strategy ......................................... 36
      4.2.4 Labeling Methodology ..................................... 37
   4.3 Bug Classification Estimator Performance ...................... 37
      4.3.1 Benefits of Active Learning ................................. 38
   4.4 Graph Traversal Performance ................................... 40
      4.4.1 Implications of Performance ................................. 41

5 Conclusion 42
   5.1 Wider Application .............................................. 42
   5.2 Bug Classification .............................................. 43
   5.3 Scalability ..................................................... 43
   5.4 Embedding and Sampling ....................................... 44
   5.5 Closing Thoughts .............................................. 45

Appendices 51

A Appendix 52
   A.1 Python Graph Dictionary Structure ............................. 52
      A.1.1 Graph Structure ........................................... 52
      A.1.2 Ending Weights ........................................... 54
List of Figures

1.1 Categories of testing based on access to source code [3] 3

2.1 How data was generated from a code corpus for DeepBugs [30] 10

2.2 How bug assignment can be informed both by previous lifetime prediction data and developer performance [16] 12

2.3 The SVM classifier identifies suspicious regions of function calls based on the change in classification accuracy [22] 14

3.1 First test case example represented as a directed graph 20

3.2 The union of the first and second example allows for the creation of test cases that combine actions and inputs from both 22

3.3 DANIEL test case can be represented as both a graph and a vector 27

4.1 Web form application that stores different kinds of cheese 32

4.2 Recording test cases and bugs using Selenium IDE 33

4.3 Converting a Selenium test script to DANIEL test case, using selenium2daniel.py 34

4.4 The iterative Active Learning cycle 35

4.5 Increasing model performance is related to the number of labeled test cases, chosen by Entropy Sampling 39

5.1 RESTful APIs are stateless [39] 43

5.2 Neo4j is a scalable, open source graph database [37] 44
List of Tables

3.1 Syntax of the DANIEL language. .......................... 19

4.1 Logistic Regression Performance. To the left is the
confusion matrix and to the right are metrics com-
puted from the confusion matrix, i.e., precision, re-
call, and the harmonic mean of both (F1 score) –
note that higher values are better. ......................... 38

4.2 XGBoost Performance. The confusion matrix is shown
on the left and the precision, recall, and F1 score
metrics are reported to the right. ........................ 39

4.3 Logistic Regression Validation Performance. The con-
fusion matrix is shown on the left and the precision,
recall, and F1 score metrics are reported to the right. 40

4.4 XGBoost Validation Performance. The confusion ma-
trix is shown on the left and the precision, recall,
and F1 score metrics are reported to the right. .... . 40

4.5 Percent of Predetermined Bugs in Generated Test
Cases ................................................................. 40
Chapter 1

Introduction

1.1 The Cost of Software Testing

An integral part of the software development process is testing. Poor quality assurance can lead to expensive post-release issues, ranging from extra development costs for emergency updates, to poor reviews, and even to lack of user confidence. According to the 2020 IBM Security Cost of Data Breach Study, conducted by the Ponemon Institute, the average total cost of a data breach is USD $3.86 million, with the United States being the most expensive country having a cost of USD $8.64 million [40]. While the cause of these data breaches is rooted in code-level vulnerabilities, attacks are often performed without access to the source code.

To combat this, a vast amount of resources and time are spent on software quality assurance [18]. Comprehensive testing requires using the application under different configurations with different inputs and different possible workflows. In addition to making sure that the functionality is matched to the specified requirements, non-functional issues such as performance, security, and usability require additional testing in the form of stress testing and acceptance testing. Generating test cases for all these types of testing requires expert domain knowledge of the software system under development. Additionally, during each release cycle, new software features can cause feature interactions which can invalidate previous test case automation. This occurs when newly implemented program features change the workflow of old test cases [26].

While developers contribute to the code base, quality assurance testers use the daily software builds to verify the implemented features and to look for abnormal behavior. If the testers discover software behavior that differs from
the specified requirements, they file issues or, what these are more commonly referred to as, bugs. Finding bugs in software, however, is not simply an optimization problem. This is because it is not possible to completely and accurately quantify the number of bugs introduced from a particular code commit. For this reason, we will design our system and methods in this thesis with the perspective that this type of problem appears to be similar to that of automated discovery.

1.2 The Anatomy of a Bug

A bug consists of various attributes. From the perspective of quality assurance, we list the following key attributes of a bug as:

- **ID**: An identifying number used to store the report.
- **Title**: Text that concisely describes the issue.
- **Timestamp**: The date and time of when the bug was filed.
- **Steps to Reproduce**: Terse steps that describe how to create the bug.
- **Expected Behavior**: What should occur when following the steps.
- **Observed Behavior**: What actually occurs when following the steps.
- **Regression Information**: The current configuration and which prior software builds do not exhibit the issue.
- **Type**: The category of the bug, ranging from crashes to usability issues.
- **Priority**: Signifies the importance of the issue on a discrete scale.
- **Tags**: Identifies the affected components of the system.
- **Attachments**: Files such as screenshots, videos, and logs.
- **Related Bugs**: The IDs of bugs that have a similar issue.
- **Duplicates**: The IDs of bugs that have the same underlying issue.

The most important part of a bug report is the steps to reproduce section. When an issue is assigned to a developer to fix, the developer runs through the series of steps to try and reproduce the issue. The developer will often do this with debugging tools to place breakpoints that aid in identifying the
causes of the abnormal behavior. After committing a change to the code base to fix the issue, the original tester is assigned the bug. The tester will then run through the steps again in order to see if the issue still occurs. Additionally, they will also test features surrounding the change, to make sure no new abnormal behavior was introduced by the fix. This is particularly important given that changes to software can introduce unexpected behavior, which is a specific kind of issue called a regression bug.

The two main categories of testing are broadly defined as bottom-up testing and top-down testing. The former involves looking for issues at the source code (lower) level while the latter involves looking for issues at the highest, functional level. These two categories are also called white box versus black box testing. The following sections will detail the specifics of each. Figure 1.1 presents the spectrum of the main kinds of software testing.

1.3 White Box Testing

White box quality assurance is a type of testing where the source code is accessible. Knowledge of how the software is implemented can inform the tester as to what areas of the build they should inspect. Testing automation and unit test cases that execute low level functions fall under this category [28]. The issues documented in this kind of testing are related to the underlying code base that the software is built on. These types of bugs are often centered around how the code framework handles certain interactions and inputs. Examples of these kinds of bugs include code refactoring tasks, buffer overflows, or boundary value issues.

The kinds of bugs discovered through white box testing are usually filed by the developers themselves and not the quality assurance testers. As we
will discuss in the Chapter 2, white box testing is what most of the existing literature in the area of intelligent quality assurance focuses on.

1.4 Black Box Testing

Black box quality assurance is a type of testing where the tester does not have access to the source code. Testers, under this paradigm, have little to no knowledge regarding how the features/components of the program were implemented. Instead, they are far more versed with the program features that will ship in the release that they are testing. Black box testers are to qualify these new features by ensuring that functionality matches the specified requirements. They do this by using the features based off test-cases defined in the documentation.

However, black box testers are also tasked with using these features in novel ways. These individuals will deliberately try to break the feature with boundary value inputs, with novel setting configurations, and with different steps leading up to the usage of new features. Black box testers essentially act as the end user before the software ships. The kind of bugs found in this setting are what end users would find, especially since they too would not have access to the source code. Black box testing prescribes how to use the program at its highest level with no regard for the underlying libraries or functions which are being used to facilitate such behavior. As we will see in Chapter 2, little research has been conducted with regards to investigating the application of machine learning to this kind of testing.

1.5 Gray Box Testing

As shown in Figure 1.1, there is a hybrid of the two main testing styles known as gray box testing. This kind of testing is where the runner has access to system architecture or design documentation in addition to the executable program. This can inform a potential hacker which action path they should take in order to crack the software. As mentioned in Section 1.1, most data breaches are performed by hackers who do not have access to the original source code. For example, if an attacker knows that a website is using a SQL database for its back end, they may try to use SQL injection in input forms that make calls to the database.
1.6 Current Approach

With each new daily build, the quality assurance team will run through smoke tests, or a set of simple test cases that cover most of the program’s features in a superficial way. If any serious break was pushed into the code base the day before, it will likely be found here. Team members are assigned to specific features and spend the remainder of the day looking for new bugs in their area, as well as verifying the bugs that were fixed. Throughout this process, issues are documented in a database of bug reports.

Near the end of a software release cycle, the bugs — those that were reported and fixed — are collected. The team then begins regression testing, which is a process to verify that previously fixed bugs no longer occur. The project manager will reassign filed issues to different members of the team. Each member is tasked with running through the steps to reproduce each of the fixed bugs to ensure that the issue no longer occurs.

1.6.1 Flaws With Current Approaches

In the book How Google Tests Software [38], author James Whittaker provides insight into the true value of bug reports and test plans:

“Every artifact generated by testers is secondary to the source code. Test cases are less important; the test plan is less important. Bug reports are less important. The activities that go into making these artifacts are what actually provide value.”

In other words, this is to say that it is not the actual number of bugs filed by the quality assurance team that should be valued. Instead, it is the feature coverage caused by the act of writing bug reports that should be paramount. The common/frequent problem of duplicate bug reports illustrates this concept where multiple bug reports can be written about the same underlying issue. A raw count of fixed bugs is a sub-optimal metric for software quality. For further inspiration, we refer to a quote from Edsger Dijkstra:

“Testing shows the presence, not the absence of bugs.” [9]

This is not to say that testing is useless. On the contrary, testing can show us the presence of bugs. Therefore, testers must expand their usage paths and input combinations in an intelligent manner. This necessity illustrates the biggest problem with current approaches to software testing and quality assurance: quality assurance testers all too often traverse the same steps/action
pathways. Little is done to diversify the testing paths taken and expand feature coverage. A project manager may assign bugs to different team members that were not involved with the original issues. Nonetheless, while doing this will provide new perspectives on the steps taken in currently examined bug reports, this small remedy fails to find issues outside of these steps. The only mechanism truly available for potentially identifying out-of-report issues is Ad-hoc testing, or unstructured, undocumented testing which the (development) team members do of their own accord.

Little consideration is given to what more can be done with the steps from the collection of previously filed bugs. This is the main impetus for DANIEL (Directed Action Node Input Execution Language). Our proposed language abstracts the components of test cases — actions, inputs, and states — and represents their relationships as directed graphs (or, from another perspective, finite graph automaton). This standardized representation, as we will show, can be translated into different scripting languages and frameworks such as Selenium [31]. We will see the benefits of modeling software behavior with the DANIEL language by applying it to a simple web form application.

The rest of this thesis will be organized as follows:

• The literature review in Chapter 2 discusses the existing research in the application of intelligent systems to software quality assurance. We will see various case studies that apply different techniques to assist with preventing bugs, assigning bugs, diagnosing bugs, and recommending test cases.

• The core contribution described in Chapter 3 is where we explain the Directed Action Node Input Execution Language (DANIEL), its syntax and usage. We define various graph traversal methods and algorithms, which can be used on the combined graph representation to generate new test cases and search for bugs.

• The experimental results of Chapter 4 describe the application of the language to a simple web form application. We showcase how the language can be used to model test cases and how the language can be embedded into a vector representation for use in training estimators that will classify a series of steps as either a test case $y = 0$ or a bug $y = 1$. We then rank the different graph traversals by the percentage of generated test case paths which include predetermined bugs.

• Finally, Chapter 5 details the possible future application of the language to different frameworks and scripting. Scalability is discussed by way
of database solutions for storing action node graphs. This chapter also proposes new areas of research in the way of different embedding and uncertainty sampling strategies.
Chapter 2

Literature Review

For decades there have been debates on how to improve the reliability of software. When the field of computer science was in its infancy, there was the notion that programming should strive to become more like mathematics [8]. Many argued that programmers should adopt formal reasoning and proofs into their software design specifications.

One of the biggest proponents of this mindset was the prolific computer scientist Edsger Dijkstra, who argued in favor of formal verification processes [11]. Dijkstra emphasized that software quality should be built from the bottom up, rather than from the top down. His emphasis was on preventing bugs rather than discovering them. The following quote is reproduced from his 1972 ACM Turing Lecture, *The Humble Programmer*:

> “Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.” [10]

Dedicated software quality assurance teams did not exist in Dijkstra’s day, nor did many of the practices that we will discuss throughout this chapter. However, we will see how this mode of thinking has largely shaped the landscape of literature with respect to the subject of testing. Research, in essence, has been focused on preventing bugs from a bottom up, white box perspective, while the task of intelligently finding bugs from a top down, black box perspective has been largely unexplored.
2.1 Preventing Bugs

In this section, we will review several ideas, concepts, and approaches related to the effort of preventing bugs from the start of the software design process.

2.1.1 Model Checking With TLA+

Leslie Lamport helped to establish the groundwork for formal verification methods used in distributed systems today. One of his contributions is TLA+, a specification language that defines software states in terms of set theory and temporal logic, which can be verified through the use of hierarchical proofs [7, 42]. Using discrete math and set notation, software specifications written in TLA+ can be used as blueprints for system implementation. This paradigm helps prevent bugs by identifying possible issues at the system design stage. By ensuring the model specification can handle boundary value conditions at the time of implementation, many issues can be prevented in advance.

Amazon Web Services has reported benefits from using the formal methods provided by TLA+ [27]. The development of DynamoDB used this system to model its system specification. According to the authors, the tool helped discover subtle bugs that went unnoticed in code reviews and missed during static code analysis. The main weakness of TLA+ is its inability to detect sustained performance degradation. The authors expressed an intent to model this as a safety property, even though it was already considered an error by their testing automation.

Additionally, formal model checking helps engineers with writing assertions for testing automation. Model checking makes use of system invariants, or functions that must always evaluate to True based on the current system state. This helps to expose vulnerabilities before the system is even implemented in any programming language.

2.1.2 Test Driven Development

There have been several advancements in testing automation that seek to prevent bugs from being introduced. One key example is Test Driven Development (TDD) [4]. With TDD, developers write unit test cases before they implement features. In this way, the unit tests act as a kind of checklist for the developer, ensuring that a feature is only fully implemented when all of its tests pass. It encourages the developer to consider the edge cases before implementing any part of their design. This also helps with regression testing
CHAPTER 2. LITERATURE REVIEW

Figure 2.1: How data was generated from a code corpus for DeepBugs [30].

Given that the unit tests can be re-run every time a new change has been committed to the code base. This ensures proper functionality at the lowest level of the development stack.

While both TLA+ and TDD may ensure that a particular software component or algorithm can handle boundary value inputs, it does little to prevent issues which can occur from feature interactions. Even though both are invaluable bottom up quality assurance techniques, black box testing is still necessary for complete coverage.

2.2 Writing Better Code

In this section, we will examine several approaches to bug prevention through the process of controlled and up-front intelligent code writing.

2.2.1 Deep Bugs

Michael Pradel and Koushik Sen applied the advancements in deep learning to generating static code analyzers with their research into DeepBugs [30]. Their model was trained on a corpus of JavaScript code from a source repository. The main problem with static code analyzers is their high false positive rates. Typically, this can be minimized by manually generating heuristics that filter out such false positives. DeepBugs sought to overcome this by training a statistical learning model on existing code examples thus automatically embedding such heuristics into their generated static code analyzer.

The biggest drawback to this approach is the need for millions of training examples. These examples are not only difficult to generate, but any approach to data creation must also take into account the unbalanced distribution of the code/program samples. In most code bases, there are far more positive examples than negative ones. To solve this problem, DeepBugs generated positive and negative coding examples from an existing code corpus. The transforma-
tions on the generated code were compliant with the language, so as not to cause compilation errors. These included mutations such as swapped arguments, incorrect use of binary operators, etc. After categorizing the generated examples as positive or negative, these code snippets were embedded into vector representations with appropriate labels. See Figure 2.1 for an illustration of the DeepBugs pipeline.

While powerful, this kind of automated bug discovery/prevention does not take into account the high level features of the software release. It is only focused on code implementation and white box bugs. Most released software compiles and passes unit tests. Most detrimental bugs are not usually discovered in the source code but rather from observing high level behavior which differs from the requirements. This is why black box testing is still an invaluable part of quality assurance and deserves further, more concentrated exploration.

2.2.2 VulIntel: Secure Coding Recommendation System

Recommending secure coding practices — tailor-made for a specific code repository — can help developers to incorporate security standards into their programming. Most code analysis tools have poor usability and are rarely used by developers. The VulIntel (Vulnerability IntelliSensor) system proposed by Nembhard, Carvalho, and Eskridge combines text mining of secure coding documentation with IntelliSense to create a recommendation system that helps developers with secure Java coding [25]. Unlike DeepBugs, these recommendations are not self referential; they are not based off of self-generated code examples. Instead, the training examples are based off mined code samples from the National Vulnerabilities Database (NVD) and the Common Vulnerabilities and Exposures (CVE) dictionary.

VulIntel runs through two phases: The first phase involves collecting code recommendations from the NVD and CVE databases, and then breaking down those code examples into features. The second phase involves evaluating the code that is being written in real time, querying the recommendation system, and serving the programmer possible secure coding practices. Java code is tokenized using an abstract syntax tree. Once the code is tokenized, VulIntel can see which secure coding database has the closest match. Fixes are ranked based on the similarity of the current code context, and the top ranking one is served by IntelliSense.

By serving the developer relevant secure coding examples, the developer is more likely to implement industry safety standards. Much like TLA$^+$ and TDD, this system helps with preventing bugs rather than discovering them.
CHAPTER 2. LITERATURE REVIEW

2.3 Improving the Software Development Process

Assigning and prioritizing bugs is another task in the software quality assurance pipeline. The bug dispatcher must assess the impact of an issue by considering its severity as well as how long it will take to fix. The dispatcher then assigns the bug to a developer who specializes in the affected component. The developer then creates a fix for the assigned issue and pushes their changes into the next build. Finally, the bug dispatcher then re-assigns the bug report to the original tester to verify if the issue is truly fixed and to check to see if the fix introduced any new issues, or regressions.

2.3.1 Assigning Bugs

Because determining which team member to assign an issue to depends on the various attributes of the bug report (see Section 1.2), this can be defined as a classification problem. Other information, such as software failure state log
reports, can also be used in predicting the impact and priority of a bug [29]. Finding the best developer to assign the issue to, as well as the best tester to verify, can further minimize the time that a bug remains inactive. This was the motivation behind the market-based bug allocation mechanism developed by Hosseini, Nguyen, and Godfrey [16].

Their approach involved training a model to predict how long it would take to fix a bug by using features based on the various attributes of a report (priority, type, etc.). This prediction, in conjunction with the dispatcher’s assigned category and the past performance and workload of a developer, can be used to offer data driven assistance in the assignment of bugs. Figure 2.2 illustrates this.

Given the rich semantic information in bug reports and their relationships to other issues, it is possible to form a knowledge graph of an issue database [43]. Such a representation could prove to be useful in classifying reports, as well as predicting what approach developers should take. Recommendations and search queries conducted that are informed by a bug knowledge graph would help dispatchers to better assign bugs to appropriate team members.

2.3.2 Diagnosing Bugs

In addition to assigning bugs to team members, another important task in the test life cycle is diagnosing bugs. This entails isolating which software component is affected by the issue [22].

2.3.3 Software Behavior Graphs

Some approaches to improved software development involve building approaches based on so-called software behavior graphs. One notable approach was documented by the IBM Watson Research Center [22]. Their technique involved mining and classifying the function call traces of program executions, which taken together were composed to create graphs summarized program execution at the function level (each node was one function). These software behavior graphs were labeled and classified using a support vector machine (SVM).

Ultimately, their overall system model was then run against the function calls that were executed when performing the steps to reproduce a bug. The change in classification accuracy could isolate the region of code that likely caused the issue (see Figure 2.3). Their biggest challenge involved embedding their behavior graphs into a vector feature space. Ultimately, they found that such a representation quickly became intractable/infeasible. This brought into question the scalability of their solution.
CHAPTER 2. LITERATURE REVIEW

Figure 2.3: The SVM classifier identifies suspicious regions of function calls based on the change in classification accuracy [22].

2.3.4 Code Change Differences

Another technique for isolating buggy components of software systems involves identifying changes/differences ("diffs") in the abstract syntax tree representations of code. This approach was documented in DeepDelta [23], which attempted to capture the changes to code that would resolve build errors. Their model was trained with input that was compiler diagnostic information and the target to model was a recommended change delta. DeepDelta’s performance was exceeded by a more recent approach known as Graph2Diff, which also used the abstract syntax tree representations for code diffs [35].

These examples of research show attempts to improve the process of assigning, classifying, and diagnosing bugs and compilation errors. However, the primary issue with this line of work is that the issues must already have been documented a priori (in order to build a good model of change deltas, for example). We find that there is a relative dearth of research in the way of exploring software execution trees for the purpose of discovering bugs.
CHAPTER 2. LITERATURE REVIEW

2.4 Generating Test Cases

Creating and documenting test case paths is crucial in the software development process. Similar to a bug, test cases document a series of steps of software usage along with its expected behavior. However, this process takes time away from searching for abnormal behavior. Furthermore, not all test cases are equally important. Generating prioritized test cases — that is a series of steps that are relevant to the latest code changes — can facilitate the exploration process in discovering new bugs.

2.4.1 Springer Verlag’s Online Conference System

The process of automatic test case generation was the driving impetus behind the development of DANIEL. Specifically, the main ideas were inspired by a paper that documented the development of Springer Verlag’s Online Conference System (OCS) [26]. The authors in this work used automated continuous quality assurance to drive their software testing with active automata learning using the $L^*$ algorithm [2].

Notably, the authors represented the possible work flows of the Online Conference System using Mealy machine notation. This means that their target finite automata was a state representation of current build of the software. Using the $L^*$ algorithm, the Mealy machine models and queries generated during training would be evaluated against predefined rules. This technique discovered several bugs early on in the development cycle. While the work was promising, the authors did note that there were drawbacks to this approach.

The biggest setback related to the interpreting of the trained output. Expert knowledge of the Online Conference System was still necessary in order to determine what should be done to refine the system. It was further stressed that their model was a significantly simplified version of the application itself. If every possible program state was modeled, it would lead to the state explosion problem [6]. The authors noted that the advantage to their system was its flexibility. When a new release cycle defined new features, they only needed to change the target model to include the new features and the training process would readily automatically generate new test cases based on the target.
2.5 Recommending Test Cases

2.5.1 Sampling By Topic Similarity

As mentioned in Section 1.6, an important part of the release cycle comes near the end, when fixed bugs are reassigned for regression testing. This stage is crucial to prevent ostensibly fixed issues from creeping into the final build. However, not all bugs and test cases are equally important. This principle is what drove the research behind the Topic Model Recommendation Method [1].

In this approach, the authors aimed to increase the effectiveness of regression test case recommendation by assessing the similarity of the test case topics picked by the dispatcher. This was accomplished by leveraging topic modeling using the latent Dirichlet allocation (LDA) framework where latent variables were extracted from each test case with the topicmodels package in R. Their system then recommended test cases (most) similar to the manually-selected ones. The remaining recommendations were influenced by the application’s testing history, pushing the test cases that were not covered to the top. This combination of hand-picked tests, similar reports, and under-exposed tests created a diverse sample. The biggest setback with their strategy was its reliance on the dispatcher’s sample selection. A different dispatcher may select an entirely different set of test cases. This brought into the question the generality of their results.

While this is one of the very few studies that have focused on improving testing from a black box perspective, it unfortunately only focused on test case recommendation at the regression stage, rather than test case generation. More importantly, it did little with respect to the creation of new test cases.

2.5.2 Recommendation as a Search Problem

Atkinson et al. [17] proposed an Eclipse plug-in, Test Tenderer, that would recommend JUnit test cases to the developer based on the code that they wrote. This was accomplished by searching for existing JUnit test cases that were related to the given Java class interface in the current code. Their system then broke down the current code into an abstract syntax tree and looked for semantically-related test cases. The search results were reported to the IDE which helped to inform the programmer which JUnit test cases should be written for the specific operation being implemented. It should be noted that this system and the VulIntel system, described in Section 2.2.2, appear to be quite similar.

Unit test case generation can also be formulated as a search problem. How-
ever, searching the space of possible test cases requires a *fitness function* to
guide the stochastic search and to evaluate the effectiveness of the recommenda-
dtions. Such a function should favor test cases that are most likely to cause
faults. This need is what drove the research of Almulla et al. [33]. Their results
suggested that the tests that generated the highest level of code branch cover-
age were the most effective criteria. Their study affirmed that test generation
should be based around dependency utilization and code coverage.

In both search examples, it is important to observe that software quality
assurance is still being addressed from a white box perspective. The unit
tests that were generated and recommended to the developer were informed
by useful metrics, but, again, these examples came from the realm of bug
prevention rather than bug discovery.
Chapter 3

The Directed Action Node Input Execution Language

In this chapter, we will discuss the motivation for creating the Directed Action Node Input Execution Language (DANIEL) as well as how to write test cases in this proposed language. We will see how test cases written in DANIEL are represented as directed graphs and how it is then possible to combine the test cases into a single, unified graph. Lastly, we will look at and formally define different graph traversal methods to be used in exploring action pathways in our resulting finite graph automata.

3.1 Modeling Test Cases as Graphs

The purpose of software quality assurance is to find and report bugs in development builds. It has long since been an elusive goal to create an intelligent agent that can do this for developers. When considering the components of a bug report, we find that the most important section is the steps to reproduce, which describes the steps that were taken to cause the abnormal behavior. Therefore, we must create an agent that generates a series of steps.

State machines are typically used to model software behavior, where each node represents a program state. As discussed in Section 2.4, this can lead to issues such as the state explosion problem [6]. Furthermore, such a representation requires underlying knowledge of the program implementation. DANIEL graphs are not typical state machines, because each node represents a high level, black box action, not a state. Even though program state can be stored in an action node, it is not necessary. At best, DANIEL graphs could be likened to finite graph automata [5, 19, 36]. Writing these kinds of tests does
not require any knowledge of the underlying software implementation.

The main assumption behind DANIEL is that a certain level of abstraction is necessary to define steps. Each step must have an action, such as clicking on a button. These actions are sometimes accompanied by inputs, such as the words that will be typed into a text field. Certain actions and inputs can alter the program state, such as logging into a website. This is especially true for programs which change the available actions depending on the state. Recording the current state can help to separate action paths in multi-state programs. For example, actions available to the user when logged differs from the available actions when being logged out.

Since a bug is composed of a series of steps and a step is, in essence, an action (optionally accompanied by an input and state), we can encapsulate these elements in a directed graph. Each action in a test case is represented as a node and the possible input arguments for this action are represented as its incoming edges. Depending on the application test case, the action nodes may also include a state token, showing the current program state when executing the action. Once the test case is represented as a graph, it can be combined with other test graphs, forming a union of all recorded test cases – this will model all of the recorded actions, inputs, and states of the developing software. It is then possible to form new test cases by traversing this combined graph and combining actions and inputs in a way that preserves workflow. This, we argue, is one of the primary benefits of the directed graph representation.

3.2 The Language Syntax

Each DANIEL test case begins with a title line. The title text is written after the (Start) token. This token represents the start node, which every test case must begin with. Each new line following the title is a step. Each step begins with a sequential number, identifying how far it is into the test case (sequence). Every step requires an action node, but may optionally contain
Figure 3.1: First test case example represented as a directed graph

an input and a state. Refer to Table 3.1 for a depiction of the syntax for each part of a step. Each action node forms a directed connection to the next step’s action node.

Given the above language syntax, we illustrate DANIEL’s usage in a simple example. Specifically, the following sample presents a slightly abstract/generic way of defining a DANIEL test case:

(Start) First Test Case Example
1. {<State 1> Action 1} [Input 1]
2. {<State 1> Action 2}
3. {<State 1> Action 3}
4. {<State 1> Action 1}

The (Start) node is connected to the {Action 1} node through a directed edge with an input. This edge signifies {Action 1} is performed using [Input 1]. Each successive step is performed without an input. We see a directed edge from the {Action 1} node to the {Action 2} node. Likewise, there is a directed edge from the {Action 2} node to the {Action 3} node, and finally a connecting edge goes from the {Action 3} node back to the {Action 1} node. Steps 2 through 4 have no inputs. The <State 1> token, which is present in every action node, shows that all actions are performed while the program is in the same state. Figure 3.1 illustrates the final directed graph representation of the First Test Case Example. Notice how the <State 1> token is embedded into each action node.

To illustrate some of the complexity one could introduce to a DANIEL test case, we will now consider a second example which contains new inputs,
CHAPTER 3. THE DIRECTED ACTION NODE INPUT EXECUTION LANGUAGE

actions, and a new state. Specifically, consider the sample below:

(Start) Second Test Case Example
1. {State 1} Action 4
2. {State 2} Action 5 [Input 3]
3. {State 1} Action 1 [Input 2]
4. {State 1} Action 4

In the above Second Test Case Example, the edge from the (Start) node to the first action node {Action 4} does not have an input. Performing {Action 4} causes a change in program state, as we see {State 1} of the first step changes to {State 2} in the second step. The directed edge from {Action 4} to {Action 5} does have an input, showing that {Action 5} is performed with [Input 3]. Executing this step causes the program to return to {State 1}, as shown by the action node in step 3. The final step demonstrates that there is a directed edge from {Action 1} to {Action 4}, with no input.

Both of the test cases presented so far would be stored in a .daniel file. This file is parsed by daniel.py, which then synthesizes a directed graph by combining the test cases. The power of the DANIEL representation can be seen when we form a union of both test case examples. Figure 3.2 illustrates the union between these two examples, i.e., the action nodes and inputs are combined into a single graph.

Once we have the combined representation/union, it is possible to generate new test cases by traversing this graph. For example, the test case below could be generated from the final graph:

(Start) Generated Test Case From Union Graph
1. {State 1} Action 1 [Input 1]
2. {State 1} Action 4
3. {State 2} Action 5 [Input 3]
4. {State 1} Action 1 [Input 2]
5. {State 1} Action 2

Note that this particular combination of steps was not explicitly written in the first and second example. Rather, this test case was generated by traversing the graph. As we will see in Chapter 4, modeling application test cases in this manner allows for the generation of novel action and input combinations.
3.3 Graph Definition

We now formally define an arbitrary individual test case graph $T$ as a tuple of $V$ and $E$ specifically as follows:

$$T = (V, E)$$  \hspace{1cm} (3.1)

where the set of vertices $V$ and the set of edges $E$ are explicitly defined as:

$$V = \{\text{Set of all action nodes in the test case}\}$$ \hspace{1cm} (3.2)

$$E = \{\text{Set of all directed input edges in the test case}\}.$$ \hspace{1cm} (3.3)

Given the definition of an individual test case, the union of all $T$ test cases yield the full graph $G$. The graph can be fully expressed as follows:

$$G = \bigcup_{i=1}^{N} T_i$$ \hspace{1cm} (3.4)

where $N$ represents the total number of test cases used to create the graph. By forming a union of all test cases, we will construct a graph that models all of the recorded interactions with the program. An example of this kind of union is illustrated in Figure 3.2.

![Diagram of graph union](image-url)
### 3.3.1 Graph Weights

A DANIEL graph $G$ also has various weights used to direct a weighted graph traversal. Each edge has a corresponding edge weight and each input along that edge has an input weight. When daniel.py initially reads in test cases, the edge weights are all initialized to 1. See Appendix A.1.1 for details on the underlying graph structure (including edge/input weights). Each node has an ending weight which is associated with the number of test cases that have that node in their last step. Refer to Appendix A.1.2 for more on this attribute.

The category of the .daniel file will influence the edge and input weights when building the graph. The set of test cases that are known issues will multiply each weight along its execution path by 0.5. This is done in order to lower the probability of generating duplicate bugs. The test cases that are new features will multiply each weight along its execution path by 1.5. This is to increase the frequency of new feature actions appearing in the generated test cases. Lastly, the collection of test cases that are fixed bugs have the highest multiplication factor. Each weight along the execution path of fixed bugs are multiplied by 2.0. This is done since, in the event that a bug is fixed, a change must have been committed to the code base. The software components that are impacted by the change to source code are most likely to have bugs.

We will later discuss how these weights are used to influence the construction of test cases during the weighted traversal in Section 3.4.2.

### 3.4 Traversing the Graph

Once we have the unified graph representation $G$, there are different types of graph traversals we can use to generate new test cases. This thesis will examine the following three types of walks:

1. Random Walk
2. Weighted Walk
3. Model Weighted Walk

Each walk differs in how they calculate the probability for selecting the next action node and input. However, regardless of the traversal type, all traversals come to an end when any one of the following conditions are fulfilled:

- The current node has no out degree neighbors
- Test case has reached $maxSteps$ number of steps
CHAPTER 3. THE DIRECTED ACTION NODE INPUT EXECUTION LANGUAGE

Algorithm 1 The Random Walk Traversal Algorithm

1: procedure RandomWalk(numTests, maxSteps)
2: for i ∈ {1 . . . numTests} do
3:     > % Write out (Start) Test Case #i %
4:     previousNode ← “Start”
5: for j ∈ {1 . . . maxSteps} do
6:     neighbors ← getNeighbors(previousNode)
7:     if |neighbors| > 0 then
8:         nextNode ← randomChoice(neighbors)
9:         > % Write out the jth step with the nextNode action %
10:            > % j. {nextNode} %
11:     inputs ← getInputs(previousNode, nextNode)
12:     inputVal ← randomChoice(inputs)
13:     if inputVal \neq “lambda” then
14:         > % Write out input for the action if not lambda %
15:            > % j. {nextNode} [inputVal] %
16:     previousNode ← nextNode

• Ending weight probability of current node evaluates to True

Note that every traversal must begin at the (Start) node. It is not possible to form a valid test case starting elsewhere. This limits the sampling strategies that might otherwise be used in graphs that do not need to preserve “directionality” [21].

3.4.1 The Random Walk Traversal

When randomly traversing graph G, the probability for choosing the next action is based on the count of out degree neighbors. This calculation does not take the edge and input weights into account. Formally, the calculation of this probability is simply:

\[ p(n_k) = \frac{1}{|N|} \]  \hspace{1cm} (3.5)

where \( p(n_k) \) represents the probability of selecting the \( k \)-th action node, i.e., \( n_k \), next. \( |N| \) represents the cardinality of the set of neighbors \( N \), i.e., the number of neighbor action nodes. This is the calculation that occurs in the function call on line 7 in the Random Walk Algorithm 3.4.1.\(^1\)

\(^1\)We re-label \( N \) in the algorithm as neighbors & \( n_k \) as nextNode for clarity of presentation.
3.4.2 Weighted Walk

As mentioned in Section 3.3.1, the nodes in graph $G$ have corresponding edge and input weights. Depending on the category of the test case, the edge and input weights along its underlying execution path will be updated. These weights are incorporated into the probability calculation for selecting the next action. A summary of the different categories of .daniel files and their corresponding multiplicative factors are shown in the following usage statement:

usage: daniel.py [-fb FIXED_BUGS] [-nf NEW_FEATURES] [-ki UNKNOWN_ISSUES] test_cases max_steps num_tests

positional arguments:
  test_cases    daniel file of test cases for graph
  max_steps     number of max steps for each test case
  num_tests     number of test cases to be generated

optional arguments:
  -fb FIXED_BUGS, --fixed_bugs FIXED_BUGS
    daniel file with test cases of fixed bugs; edge weights * 2.0
  -nf NEW_FEATURES, --new_features NEW_FEATURES
    daniel file with test cases of new features; edge weights * 1.5
  -ki UNKNOWN_ISSUES, --known_issues UNKNOWN_ISSUES
    daniel file with test cases of known issues; edge weights * 0.5

The edge and input weights of test cases in test_cases.daniel are all initialized to 1.0. When building a graph $G$, the weights in the execution paths of the tests from fixed_bugs.daniel, known_issues.daniel, and new_features.daniel are multiplied by their corresponding multiplicative factor.

After the graph $G$ has been constructed using the updated edge and input weights, the weighted traversal can then be applied. Algorithm 3.4.2 constructs test cases in a similar manner as in Algorithm 3.4.1. The two only differ in how they calculate the probability of selecting a neighbor action node. This can be expressed formally as follows:

$$p(n_k) = \frac{\text{weight}(n_k)}{\sum_{n \in N} \text{weight}(n)} \quad (3.6)$$
Algorithm 2 The Weighted Walk Traversal Algorithm
1: procedure WeightedWalk(numTests, maxSteps)
2: for i ∈ {1...numTests} do
   ▷ % Write out (Start) Test Case #i %
3:     previousNode ← “Start”
4:     for j ∈ {1...maxSteps} do
5:        neighbors ← getNeighbors(previousNode)
6:        if |neighbors| > 0 then
7:           neighWeights ← getNeighborWeights(previousNode)
8:           nextNode ← weightedChoice(neighWeights)
   ▷ % Write out the jth step with the nextNode action %
9:        inWeights ← getInputWeights(previousNode, nextNode)
10:       inputVal ← weightedChoice(inWeights)
11:      if inputVal ≠ “lambda” then
12:         ▷ % Write out input for the action if not lambda %
13:            % j. {nextNode} [inputVal] %
14:     previousNode ← nextNode

where $p(n_k)$ represents the probability of selecting the $k$-th neighbor action node among the set of all neighbor action nodes $N$. This is the calculation that occurs on line 8 of Algorithm 3.4.2.

3.4.3 Model-Weighted Walk

Beyond a purely random or weighted traversal through a DANIEL graph, we propose a more intelligent means of generating neighbor probabilities by replacing the hand-coded probability calculation with an estimator, specifically by normalizing a model that is trained to predict whether a test case sequence (in a DANIEL graph) is a bug or not, using features extracted from test case instances. This means that we must define two functions – an embedding function that will convert a DANIEL test case in its string/text representation to a vector form and a function that will estimate a scalar probability given this vector representation.
CHAPTER 3. THE DIRECTED ACTION NODE INPUT EXECUTION LANGUAGE

Embedding Test Cases

The embedding function will, in effect, take in as input a sequence of tokens (which means that any given DANIEL test case must be appropriately converted to an array of token objects) and return a $D$-dimensional vector. Formally, this means we introduce a set of parameters $\Theta^3$ and define the function:

$$t = \sum_{w \in W} f_e(w, \Theta) \quad (3.7)$$

where $f_e(\cdot)$ is a function based on a pre-trained single-token vector embedding model such as one taken from the Word2Vec family [24]. Note that the above function is simply the sum of the individual tokens in a test case (in tokenized text form) $W = \{w_1, w_2, \ldots, w_{|W|}\}$. Alternatively, we could modify the above embedding function to compute the average over token embeddings instead.

---

Figure 3.3: DANIEL test case can be represented as both a graph and a vector

\[\begin{align*}
\text{daniel.py} & & \text{daniel2vec.py} \\
\text{Start} & \rightarrow \text{Action 1} & \rightarrow \text{Action 2} & \rightarrow \text{Action 3} & \rightarrow \text{Start} \\
\end{align*}\]
meaning that the function would take the following form:

\[ t = \frac{\sum_{w \in W} f_e(w, \Theta)}{|W|}. \]  

Further implementation details related to our embedding function will be provided in Chapter 4.

The Estimator

Once we have a test-case embedding function defined, as described in the previous section (Equation 3.7 or 3.8), we may then train an estimator of the probability that a test case sample is a bug or not. Specifically, given a test case vector embedding \( t \), we would like to decide if the test case is a bug \( y = 1 \) or not \( y = 0 \) (these binary labels are provided either manually by the experimenter and/or with methods based on active sampling, a variant of which we design in the next chapter). Given a set of labeled set of test case embeddings, we can then train an estimator of \( p_\phi(y = 1|t) \), where \( \phi \) are the parameters of the estimator. Note that the probability that a test case is not a bug is simply: \( p_\phi(y = 0|t) = 1 - p_\phi(y = 1|t) \).

The form that the estimator \( p_\phi(y = 1|t) \) can take could be as simple as a logistic regressor to as complex as a deep feedforward artificial neural network [20]. We will describe our estimator function choices in the next chapter. Ultimately, this probability produced by this estimator will then be used to determine which neighbor action node to choose next, instead of the edge and input weights themselves. This will allow us to craft what we call a “model-weighted walk” (see Algorithm 3.4.3), which is a traversal through a DANIEL graph that is driven by an estimator based on statistical learning.

In order for the traversal to be driven by the estimator described above, we must build temporary test cases at the current node for each possible neighbor action and input combination (in other words, we must create a vector \( t \) at each step in the action sequence defined by a test case). When the model evaluates a temporary test case \( t \), it will generate a (continuous) probability value in the range \([0, 1]\) that indicates how likely the current “intermediate” test case vector belongs to the bug class. However, in order to fulfill the requirement that all the probability values of each neighbor sum up to one, yielding a valid distribution over next-step neighbors, we must include a normalization
Algorithm 3 The Model-Weighted Walk Traversal Algorithm

1: procedure MODELWEIGHTEDWALK(model, numTests, maxSteps)
2:     for i ∈ {1 . . . numTests} do
3:         ▷ % Write out (Start) Test Case #i %
4:             previousNode ← “Start”
5:             ▷ % Strings for building up steps for current test %
6:             currentStep ← “ ”
7:             previousSteps ← “ ”
8:             for j ∈ {1 . . . maxSteps} do
9:                 ▷ % Create a test case for each neighbor input combination %
10:                    for inputVal ∈ getInputs(previousNode, neigh) do
11:                        ▷ % Store current step as a possible next step %
12:                           nextSteps ← nextSteps ∪ {currentStep}
13:                           ▷ % Get embedding of test with current next step %
14:                           currentTest ← previousSteps ∪ currentStep
15:                           vectorTest ← getEmbedding(currentTest)
16:                           probs ← normalizeProbs(probs)
17:                           ▷ % Write out jth step with nextStep action and input %
18:                           previousSteps ← previousSteps ∪ nextStep
19:                           previousNode ← getNodeFromStep(nextStep)
20:     end for
21: end for

constant. Formally, we require the following conditions to be met:

\[
\sum_{t \in T} p_{\phi}(y = 1|t) \neq 1 \tag{3.9}
\]

\[
k \sum_{t \in T} p_{\phi}(y = 1|t) = 1 \tag{3.10}
\]

where k is the normalization constant. To solve for k, we divide both sides of
the equation by the summation of all the possible test case probabilities:

\[ k = \frac{1}{\sum_{t \in T} p_\phi(y = 1|t)} \]  

(3.11)
yielding the desired normalization constant. Once we have determined the value of \( k \), we may finally compute the probability of selecting a particular \( n_k \) action node (as before) via the following:

\[ p(n_k) = k \cdot p_\phi(y = 1|t). \]  

(3.12)
Note that the above equation is implemented by line 20 in Algorithm 3.4.3.
Chapter 4

Experimental Results

In this chapter, we will see the practical application of DANIEL to a simple web database program, allowing us to demonstrate how DANIEL is concretely implemented and used in practice. We will demonstrate how to record test cases with Selenium, how these recordings can be converted to DANIEL graphs, and how generated test cases from various graph traversals can be converted back into Selenium scripts.

Next, we will define our choices for the test case embedding and bug-probability estimator functions needed to fully realize the proposed model-weighted traversal algorithm of the previous chapter. To train our probability estimators, we will also introduce our approach to readily generate useful labeled examples based on an active learning labeling strategy. Finally, we will evaluate our proposed, different graph traversals and discuss the implications of their performance.

The web database program we will be referring to throughout this chapter allows users to create an account on a site that stores different kinds of cheese. When logged out, the user is able to view the cheeses stored in the database. When logged in, the user is able to add new cheeses into the database through a simple form submission. They are also able to modify their profile information. A screenshot of this web application is visually depicted in Figure 4.1.

4.1 Recording Test Cases

The DANIEL language can be used to model any test case which is distilled into a series of steps, composed of actions and inputs. Handwriting these steps is not only a cumbersome process, it is also inefficient and error-prone. Therefore, it is necessary to define a standardized recording process which can
CHAPTER 4. EXPERIMENTAL RESULTS

32

Figure 4.1: Web form application that stores different kinds of cheese.

capture user activity.

In the world of web application development, Selenium is an invaluable framework for automating black box testing [15]. Selenium IDE records user interactions with the web browser and the currently opened website’s Document Object Model (DOM). This recording can be exported as a pytest script. These same scripts can be run using Selenium’s web driver Python library, as we will discuss in Section 4.1.2. When exporting a Selenium script, we annotate the steps and define whether or not the recorded test case is a bug. As illustrated in Figure 4.2, the country flag icon should render on the page. Because it does not, this test case will be saved with a label of $y = 1$ class, a bug.

4.1.1 Converting to DANIEL Graphs

Once a Selenium test case has been saved as a Python script, we then convert it into DANIEL syntax. The interactions with elements from the DOM are considered actions, while the words typed into input fields are considered the inputs.
Refer to Figure 4.3, where the DOM elements selected \texttt{By.ID} are considered the actions and are wrapped in curly braces. The \texttt{click()} function calls are not given an input argument, and the text passed into the \texttt{send_keys()} function calls are the argument inputs for the text field actions. Note the comments in the Selenium Python script that clearly separate the chunks of code into different states. All steps before clicking the Sign In button are given the \texttt{<Logged Out>} state, while all the steps after are given the \texttt{<Logged In>} state.

While recording state is not necessary, it can be important for generating test cases for applications where the available operations change depending on the program state. In the case of this simple web form, the navigation bar at the top had different options depending on whether or not the user was signed in. After forming a union of all $T$ test case graphs into graph $G$, the generated test cases were impossible to execute. This is because without recording whether or not the user was \texttt{<Logged Out>}, the actions — which were only available when \texttt{<Logged In>} — were accessible even when the user had not signed into the web application. This is also helpful in separating different user login paths; it would be useless to generate test cases that have mismatched login credentials.

For the current implementation, program state is manually determined by the test case writer, but it is possible to scrape the program state from the web page at the time of the recorded action. What information from the DOM would constitute this state string would depend on the type of web application.
4.1.2 Executing Generated Tests

After generating DANIEL test cases from one of the various graph traversals mentioned in Section 3.4, it is then possible to convert them into Selenium Python scripts with `daniel2selenium.py`. These scripts can then be executed using Selenium’s web driver library. This is especially useful for determining whether or not a generated test case is a bug, as this can help the tester to visualize the performed steps. Though, if a tester is already aware of a known issue, it is also possible to make this determination by only looking at the DANIEL steps without executing the script.
4.2 Implementing the Model-Weighted Walk

4.2.1 The Test-Case Embedding Function

If we are to use a set of DANIEL steps using a model-weighted traversal, we must next define the embedding function $f_e(\circ)$ described in the last chapter. In this thesis, we chose the form of $f_e(\circ)$ to be the continuous bag of words (CBOW) model of the Word2Vec family [24], trained on a tokenized corpus representation of all $T$ input test cases for its vocabulary [32]. We set the dimensionality of the model’s embedding output to be $D = 10$ (meaning that $f_e(\circ)$ returned a 10-dimensional vector for each token in a tokenized test case sequence $W$). This value was determined experimentally. We can then leverage this model whenever we need an embedded representation for a DANIEL test case.

We strip out the (Start) token and title (as this has no bearing on the composition of the graph), and then we tokenize the remaining steps. We then add or average each corresponding word vector together to form our final test case vector embedding (refer to Section 3.4.3). This methodology is used in daniel2vec.py, as illustrated in Figure 3.3. Adding word vectors together (see Equation 3.7 of the last chapter) performed better for our estimator when it took the form of logistic regression while averaging word vectors together (see Equation 3.8 of the last chapter) performed better when the estimator

![Figure 4.4: The iterative Active Learning cycle](image-url)
CHAPTER 4. EXPERIMENTAL RESULTS

took the form of an XGBoost model.

4.2.2 Generating Data with Active Learning

Now that we have embedded vector representations for our DANIEL test cases, we can begin using them to train our bug-probability estimator(s). However, we will still need to label these data points. Traditional supervised learning techniques require a multitude of labeled examples to perform well [13]. Advancements made in the domain of machine learning known as active learning (AL) have been shown to greatly alleviate the need for the manual labeling of prohibitively large databases [34].

Rather than manually labeling all generated test cases in a brute force manner, active learning utilizes uncertainty sampling to select the unlabeled test cases that the model will benefit the most from having been labeled. Using a human oracle to label these (chosen) data points — the ones the model had the most difficulty with — subsequently trained statistical models achieve the highest accuracy using the smallest amount of the data. The driving principle behind this technique is the notion that not all data is equally important. This is especially true for test cases that have the same steps as labeled tests.

An example of the loop illustrated by Figure 4.4 is as follows: given a large unlabeled pool of test cases \( U \), 10% of these are manually labeled and added to the labeled training set \( L \). A statistical model is trained using this small labeled test case set \( L \). That same model is then used to predict the labels on the newly generated set of unlabeled test cases \( U \). Each probability vector prediction is accompanied by an uncertainty calculation.

The predictions with the highest uncertainty represent the samples that the model had the most difficulty labeling. These test cases are manually labeled and added to the labeled training set \( L \). This cycle continues until the model reaches an adequate performance level. We next detail the sampling strategy we implemented to incorporate active learning into the training of DANIEL’s probability estimators.

4.2.3 Sampling Strategy

Entropy is typically used as a measure for uncertainty in multi-class classification. It is defined as follows:

\[
H(t) = - \sum_{c=1}^{C} p_\phi(y = c|t) \log p_\phi(y = c|t). \tag{4.1}
\]
As seen above, \( p_\phi(y = c|t) \) represents the probability score that the current model, with parameters \( \phi \), assigns to class \( y = c \) when determining the classification for \( t \). From this definition, we see that the model is most uncertain when its assigned probability value is equal to the inverse of the number of \( C \) classes, or \( p_\phi(y = c|t) = \frac{1}{C} \). For our data set, we had two classes: one for test cases \( y = 0 \) and another for bugs \( y = 1 \). Therefore, the model is most uncertain when it assigns a probability value of 0.5 to both classes. In this case, the entropy value will be at its maximum value of 1.

Conversely, the model is most certain or confident when it assigns a probability score close to or equal to 1, i.e., \( p_\phi(y = c|t) = 1 \). The entropy value will then be at its minimum value of 0.

**4.2.4 Labeling Methodology**

The generated test cases output their corresponding entropy value depending on the model’s probability estimation. The cases are sorted in descending order according to their entropy values, and the top 10% data points with the highest uncertainty value are selected for manual labeling by the human oracle. An example program output (for logistic regression as the chosen estimator), where \(|T| = 100\) generated test cases, is shown below:

```plaintext
=== Highest Uncertainty ===
Test Case #87: 0.6930849750521243
Test Case #20: 0.6930571132235872
Test Case #27: 0.6929926181130432
Test Case #84: 0.6929215165222784
Test Case #95: 0.6926281325526888
Test Case #69: 0.691109738172516
Test Case #5: 0.6909153373689629
Test Case #7: 0.6903552156731039
Test Case #1: 0.6901978188125775
Test Case #3: 0.689547866666781552
```

**4.3 Bug Classification Estimator Performance**

For our model-weighted walk’s probability estimator, we investigated two models: scikit-learn’s implementation of logistic regression and XGBoost’s ensemble model. Both models were trained with data labeled using the active learning methodology discussed in the previous section.
Table 4.1: Logistic Regression Performance. To the left is the confusion matrix and to the right are metrics computed from the confusion matrix, i.e., precision, recall, and the harmonic mean of both (F1 score) – note that higher values are better.

<table>
<thead>
<tr>
<th></th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>30</td>
<td>15</td>
</tr>
<tr>
<td>Yes</td>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.75</td>
<td>0.81</td>
<td>0.78</td>
</tr>
</tbody>
</table>

The classification performance of each of these estimators is shown in Table 4.1 and Table 4.2. From these results, we can see that XGBoost performed better overall in classifying the training set. The XGBoost Model was initialized with the following parameters:

```python
# XGBoost model parameters
param = {
    'max_depth': 10,
    'eta': 0.3,
    'objective': 'multi:softprob',
    'num_class': 2
}
```

The ‘max_depth’ parameter was experimentally set to 10, as increased performance leveled off around that value. Conversely, performance began to decrease when lowering the max tree depth. Considering the model performs best when the max tree depth has the same value as the number of dimensions used in the Word2Vec embedding, it is possible that the performance between the two may be related.

4.3.1 Benefits of Active Learning

Figure 4.5 shows us the effectiveness of Active Learning. We can see when the training set \( \mathcal{L} \) is made up of \( |\mathcal{L}| = 40 \) labeled test cases, the scikit-learn Logistic Regression model accuracy is 0.55 for the test set and 0.42 for the validation set. However, when 60 additional labeled test cases are added — selected by uncertainty sampling — we see the performance of the LR model increases by 20% and then begins to level off. Even when we add another 400 labeled test
Table 4.2: XGBoost Performance. The confusion matrix is shown on the left and the precision, recall, and F1 score metrics are reported to the right.

<table>
<thead>
<tr>
<th>Predicted</th>
<th>No</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>31</td>
<td>10</td>
</tr>
<tr>
<td>Yes</td>
<td>7</td>
<td>52</td>
</tr>
</tbody>
</table>

Precision 0.84  
Recall 0.88  
F1 0.86

cases, the performance only fluctuates between 70% and 80%. That initial spike in performance shows the benefits of this Active Learning, which is particularly important for labeling this kind of data, given how expensive it is to interpret. Unlike labeling bounding boxes in images or classification tasks, labeling whether or not a series of steps is a test case $y = 0$ or a bug $y = 1$ requires a tester to perform the steps in the program and categorize its behavior.

Figure 4.5: Increasing model performance is related to the number of labeled test cases, chosen by Entropy Sampling.
CHAPTER 4. EXPERIMENTAL RESULTS

Table 4.3: Logistic Regression Validation Performance. The confusion matrix is shown on the left and the precision, recall, and F1 score metrics are reported to the right.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>37 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>19 39</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: XGBoost Validation Performance. The confusion matrix is shown on the left and the precision, recall, and F1 score metrics are reported to the right.

<table>
<thead>
<tr>
<th>Actual</th>
<th>Predicted</th>
<th></th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>37 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>16 42</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Percent of Predetermined Bugs in Generated Test Cases

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Random</th>
<th>Weighted</th>
<th>LR</th>
<th>LR Avg</th>
<th>XGB</th>
<th>XGB Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.30</td>
<td>0.40</td>
<td>0.40</td>
<td>0.70</td>
<td>0.70</td>
<td>0.60</td>
</tr>
<tr>
<td>50</td>
<td>0.38</td>
<td>0.60</td>
<td>0.50</td>
<td>0.46</td>
<td>0.62</td>
<td>0.66</td>
</tr>
<tr>
<td>100</td>
<td>0.47</td>
<td>0.65</td>
<td>0.49</td>
<td>0.41</td>
<td>0.49</td>
<td>0.43</td>
</tr>
<tr>
<td>200</td>
<td>0.44</td>
<td>0.66</td>
<td>0.47</td>
<td>0.42</td>
<td>0.54</td>
<td>0.52</td>
</tr>
<tr>
<td>500</td>
<td>0.41</td>
<td>0.68</td>
<td>0.47</td>
<td>0.40</td>
<td>0.45</td>
<td>0.48</td>
</tr>
<tr>
<td>1000</td>
<td>0.40</td>
<td>0.69</td>
<td>0.50</td>
<td>0.44</td>
<td>0.49</td>
<td>0.53</td>
</tr>
<tr>
<td>Total Found</td>
<td>762</td>
<td>1,261</td>
<td>907</td>
<td>795</td>
<td>910</td>
<td>956</td>
</tr>
</tbody>
</table>

4.4 Graph Traversal Performance

The graph traversal methods were evaluated by calculating what percentage of generated test cases contained steps of the predetermined bugs. This was only possible because we knew the bugs that existed in the system before running the experiments. For example, every test case that had a step which involved
viewing any cheese information, logged in or out, it would trigger the missing flag artifact bug (as shown in Figure 4.2). Likewise, any test case that had steps which involved signing up for the application was also a bug, since that feature was also broken. When running the generated test cases, we discovered an undocumented bug which involved viewing the information for cheese that was duplicated in the database.

When considering these results, it is important to remember the discussion from Section 1.6.1. It is not the raw number of test cases that contain bug that should be used to determine the best performance. Rather, we should consider the trade-offs and benefits to each walk, how they each leverage the intelligence imbued in their weights. This is particularly the case when you remember it is not the bug reports that count, but the act of searching through software features which generates the best coverage.

4.4.1 Implications of Performance

The results showing the percentage of generated bugs is shown in Table 4.5. From this we see the random walk generated the least number of bugs. The machine learning models, both with summed up and averaged vector embeddings, were out performed by the weighted walk. This superiority can be overlooked when you consider the issue of duplicate bug reports. Even though the weighted walk may have generated the most number of bugs, it is prone to converging on a known issue once it is filed.

However, the benefits of Active Learning’s labeling via entropy sampling cannot be understated. Recall that the weights in the machine learning walk are informed by the labels the model was trained on. Therefore, the generated test cases which have the highest uncertainty represent the step and input combinations that have not been explored. This is essentially using your trained model as a compass, to help guide the tester in which test cases to take note of. Intuitively, it is possible to see how this kind of guided labeling is even more valuable than a weighted or random walk, as it encourages the quality assurance team to run through steps no one has evaluated yet.
Chapter 5

Conclusion

In this thesis we introduced DANIEL, a test case modeling language that represents a series of actions and inputs as directed graphs. We demonstrated the benefits of this novel representation in generating new test cases from previously recorded test cases. We discussed how the traversal of such a graph can be informed by testing heuristics that take into account which bugs have been fixed, which new features have been implemented, and which paths are known issues. Finally, we showcased the benefits of applying machine learning to this problem domain, showing how entropy sampling can communicate which execution path and input combinations have not been explored.

5.1 Wider Application

For the proof of concept of this thesis, we showed how it was possible to convert Selenium Python scripts into DANIEL syntax. This shows the flexibility of the language. It can model any series of steps that can be expressed in terms of actions, inputs, and states. Other possible applications include parsing server logs, or RESTful APIs. The function calls can be wrapped in curly braces and the subsequent arguments can be wrapped in square brackets. Such an application would be particularly useful for DANIEL, because RESTful APIs are stateless operations [39].

The application of this language to stress testing, concurrent testing, and usage modeling have not been explored yet. The language is capable of being used in such ways.
5.2 Bug Classification

This thesis used binary classification for its proof of concept. Embedded DANIEL tests were categorized either as either a test case $y = 0$ or a bug $y = 1$. It would be interesting to see several classes. Perhaps $y = 0$ could denote not a bug and the increasing severity and priority of the bug could be represented by the increasing class number, up to $y = 5$ for example. If not ranked by priority, perhaps it could be treated as a classification problem, where the different kinds of bugs — UI/Usability, Crashing, Functional, Security, etc. — could each be represented as a separate class.

Furthermore, it is important to note that the application of machine learning to software quality assurance is unique, because when determining whether or not a series of steps is a bug, the label can change over time. In the daily new build, when previously filed bugs are fixed, the series of steps previously labeled as a bug would no longer apply. This means that every trained model and labeled data can become obsolete whenever a new release is pushed.

5.3 Scalability

As shown in Appendix A, DANIEL graphs are represented under the hood as a native Python dictionary. This is not scalable. Because DANIEL models directed graphs, it would be best to store such data in a database designed for that representation. Neo4j, the open source graph database system, would be ideal for storing action nodes and inputs [37]. The same underlying logic...
and algorithms used in the Python implementation of daniel.py can remain the same, but the calls to the $G$ graph dictionary would now interface with a Neo4j database instance.

### 5.4 Embedding and Sampling

When embedding our DANIEL test cases, this thesis used Word2Vec’s continuous bag of words (CBOW) model. Such an embedding solution is not tailor-made for this particular modeling language. With the advancements in embedding Knowledge Graphs (KGs), there are other methods that could be better suited to DANIEL’s directed graph representation [12, 44]. Perhaps it is even possible to pose an optimization problem based around evaluating the best embedded vector representation for DANIEL.

Just as the CBOW model was not specifically designed for the DANIEL syntax, neither was the sampling strategy used for the active learning sampling. There are domain-specific metrics that can be used to prioritize which unlabeled data should be sampled. In image recognition problems, uncertainty sampling can cause consecutive images in a sequence to be selected for labeling. This redundant data wastes labeling resources and does not maximize model performance. To mitigate this, a variety of diversity-based sampling methods [41] can be used. Using a similarity matrix to prioritize which unlabeled observations will be sent to the human oracle can prevent repeat images from being sampled.

In a recent paper by NVIDIA, the authors demonstrated that clustering unlabeled images by similarity and sampling the centroid images can enforce the selection of a diverse sample [14]. Only one image from each cluster of similar images are selected, removing duplicate images in the set of samples to be labeled by the human oracle. This ensures a diverse selection and the most informative query of images for the human oracle to label. Perhaps the same clustering strategy — where only the centroids are sampled — could also be applied to the unlabeled test case selection process in our DANIEL system’s
CHAPTER 5. CONCLUSION

bug-probability estimator training process.

5.5 Closing Thoughts

This thesis proposed a new test case modeling language DANIEL. It demonstrated several graph traversal methods with test cases generated for a simple web form application. It would be interesting to see DANIEL adopted in the industry. Bug reports could include .daniel files of the steps to reproduce as an attachment. A larger study — with the improvements from the sections above — performed using an enterprise level application would be the ideal way to truly evaluate the usefulness of such a modeling language. Nevertheless, the potential of DANIEL as well as its adaptability were empirically demonstrated by this thesis.
Bibliography


[22] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. Mining behavior graphs for “backtrace” of noncrashing bugs. In Proceedings of the 2005 SIAM International Conference on Data Mining, pages 286–297. SIAM, 2005.


Appendices
Appendix A

The appendix shows the underlying DANIEL graph structure.

A.1 Python Graph Dictionary Structure

A.1.1 Graph Structure

```python
== Graph Object Structure ==
{
'<State 1> Action 1': {
    'State 1> Action 2': {
        'edge_weight': 1,
        'inputs_and_weights': {
            'lambda': 1
        }
    },
    'State 1> Action 4': {
        'edge_weight': 1,
        'inputs_and_weights': {
            'lambda': 1
        }
    }
},
'<State 1> Action 2': {
    'State 1> Action 3': {
        'edge_weight': 1,
        'inputs_and_weights': {
            'lambda': 1
        }
    }
},
'&State 1> Action 2': {
    'State 1> Action 3': {
        'edge_weight': 1,
        'inputs_and_weights': {
```
'lambda': 1
}

'}

},
'Start': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'lambda': 1
}
}
},
'State 1 Action 4': {
'State 2 Action 5': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 3': 1
}
}
},
'State 2 Action 5': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 2': 1
}
}
},
'State 2 Action 5': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 3': 1
}
}
},
'State 2 Action 5': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 2': 1
}
}
},
'State 2 Action 5': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 3': 1
}
}
},
'State 2 Action 5': {
'State 1 Action 1': {
'edge_weight': 1,
'inputs_and_weights': {
'Input 2': 1
}
}
}
A.1.2 Ending Weights

```text
{ '<State 1> Action 1': 1,
  '<State 1> Action 2': 0,
  '<State 1> Action 3': 0,
  '<State 1> Action 4': 1,
  '<State 2> Action 5': 0,
  'Start': 0}
```