



The
University
Of
Sheffield.

Guiding Random Graphical and Natural User Interface Testing Through Domain Knowledge

Thomas White

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy.

The University of Sheffield
Faculty of Engineering
Department of Computer Science

2019



The
University
Of
Sheffield.

This thesis contains original work undertaken whilst at The University of Sheffield between October 2015 and July 2019 for the degree of Doctor of Philosophy.

“Guiding Random Graphical and Natural User Interface Testing Through Domain Knowledge”

Copyright © 2019 by Thomas D. White

Abstract

Users have access to a diverse set of interfaces that can be used to interact with software. Tools exist for automatically generating test data for an application, but the data required by each user interface is complex. Generating realistic data similar to that of a user is difficult. The environment which an application is running inside may also limit the data available, or updates to an operating system can break support for tools that generate test data. Consequently, applications exist for which there are no automated methods of generating test data similar to that which a user would provide through real usage of a user interface. With no automated method of generating data, the cost of testing increases and there is an increased chance of bugs being released into production code. In this thesis, we investigate techniques which aim to mimic users, observing how stored user interactions can be split to generate data targeted at specific states of an application, or to generate different sub-areas of the data structure provided by a user interface. To reduce the cost of gathering and labelling graphical user interface data, we look at generating randomised screen shots of applications, which can be automatically labelled and used in the training stage of a machine learning model. These trained models could guide a randomised approach at generating tests, achieving a significantly higher branch coverage than an unguided random approach. However, for natural user interfaces, which allow interaction through body tracking, we could not learn such a model through generated data. We find that models derived from real user data can generate tests with a significantly higher branch coverage than a purely random tester for both natural and graphical user interfaces. Our approaches use no feedback from an application during test generation. Consequently, the models are “generating data in the dark”. Despite this, these models can still generate tests with a higher coverage than random testing, but there may be a benefit to inferring the current state of an application and using this to guide data generation.

Acknowledgements

I am profoundly grateful to my supervisors Gordon Fraser and Guy J. Brown, who made all the research in this thesis possible and who pondered many a meeting with me trying to figure out why something was not working. Heartfelt thanks go to my family and friends, especially my partner Dolser for her endless commitment and encouragement when the hours were long and Carol White for her ever loving support and care.

I would like to thank my fellow PhD students at The University of Sheffield including those who provided data: M. Foster, G. O'Brien, G. Hall, N. Albulian, A. Alsharif, J.C. Saenz Carrasco, Y. Ge, B. Clegg, and T. Walsh. Thanks to S. Falconer, S. Smith, J. Bradley, D. Paterson and M. Dewhirst for their support and many an entertaining night. Finally, a thanks to those who spent I spent countless hours talking to over a beer on a Friday night: A. Warwicker, J. Perez Heredia, M. Hall, S. Shamshiri, J. Medeiros de Campos, J. Miguel Rojas, W. Ward, N. Walkinshaw and Mr and Dr A. Poulston.

Publications

The following work presented in this thesis has been published in peer reviewed conferences:

- [1] Thomas D. White, Gordon Fraser, and Guy J. Brown. Improving Random GUI Testing with Image-Based Widget Detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.
- [2] Thomas D. White, Gordon Fraser, and Guy J. Brown. Modelling Hand Gestures to Test Leap Motion Controlled Applications. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 204–213, April 2018. **A-MOST 2018 Best Paper Award.**

In addition, the following has also been published during the PhD programme but the work therein does not appear in this thesis:

- [1] José Miguel Rojas, Thomas D. White, Benjamin S. Clegg, and Gordon Fraser. Code Defenders: Crowdsourcing Effective Tests and Subtle Mutants with a Mutation Testing Game. In *Proceedings of the 39th International Conference on Software Engineering. IEEE Press*, pages 677-688, May 2017. **ACM SIGSOFT Distinguished Paper Award.**

Contents

Publications	ix
Contents	xi
List of Figures	xv
List of Tables	xix
1 Introduction	1
1.1 Overview	1
1.2 Test Data generation	3
1.2.1 Motivation for Automated Test Data Generation	4
1.2.2 Summary	8
1.3 Structure and Contributions	9
2 Literature Review	15
2.1 Introduction	15
2.2 Software Testing	16
2.2.1 What is a Software Error, Bug and Failure?	17
2.2.2 Testing Methods	18
2.2.3 Test Levels	24
2.2.4 The Oracle Problem	28
2.3 Automated Test Generation	29

Contents

2.3.1	Random Testing	30
2.3.2	Dynamic Symbolic Execution	32
2.3.3	Search-based Software Testing	32
2.3.4	Model-based Testing	37
2.3.5	User Guidance	39
2.4	User Interface Testing	43
2.4.1	Types of User Interfaces	43
2.4.2	Random GUI Testing	48
2.4.3	Dynamic Symbolic Execution in GUI Testing	49
2.4.4	Model-based GUI Testing	49
2.4.5	Machine Learning in GUI Testing	55
2.4.6	Image Processing in GUI Testing	56
2.4.7	Mocking in NUI Testing	57
2.4.8	Modelling User Data for NUI Testing	59
2.5	Summary	59
3	Guiding Testing through Detected Widgets from Application Screen Shots	61
3.1	Introduction	61
3.2	Automatic Detection of GUI Widgets for Test Generation	64
3.2.1	Identifying GUI Widgets	67
3.2.2	Generating Synthetic GUIs	73
3.2.3	A Random Bounding Box Tester	78
3.3	Evaluation	79
3.3.1	Widget Prediction System Training	80
3.3.2	Experimental Setup	83
3.3.3	Threats to Validity	87
3.3.4	Results	89
3.4	Discussion	100
3.5	Conclusions	106

4	Testing By Example: Graphical User Interfaces	109
4.1	Introduction	110
4.2	Modelling GUI Interaction Events	111
4.2.1	GUI Events	112
4.3	Generating GUI Interaction Events	113
4.3.1	Random Events Monkey Tester	114
4.3.2	Generating User Models	115
4.3.3	Application Window-based Models	119
4.4	Evaluation	121
4.4.1	Experimental Setup	121
4.4.2	Threats to Validity	127
4.5	Results	128
4.6	Conclusions	138
5	Testing By Example: Natural User Interfaces	141
5.1	Introduction	141
5.2	Natural User Interfaces	145
5.2.1	Natural User Interface Testing	146
5.2.2	Leap Motion Controller	148
5.3	Modelling Leap Motion Data	149
5.3.1	Model Generation	151
5.4	Generating Leap Motion Data	153
5.4.1	Executing Leap Motion Tests	156
5.5	Evaluation	157
5.5.1	Experimental Setup	157
5.5.2	Threats to Validity	161
5.5.3	RQ5.1: How beneficial is NUI testing with n-gram models generated from real user data when gen- erating tests for Leap Motion applications?	163

Contents

5.5.4	RQ5.2: How does the quantity of training data influence the effectiveness of models generated for NUI testing?	167
5.5.5	RQ5.3: How beneficial is a model with separation of NUI data, which can generate parts of the data structure in isolation, compared to a model trained on a whole snapshot of the NUI data? . . .	169
5.6	Discussion	172
5.7	Conclusions	175
6	Conclusions and Future Work	179
6.1	Summary of Contributions	179
6.1.1	Identifying Areas of Interest in GUIs	180
6.1.2	Generating Natural User Data	182
6.2	Future Work	183
6.2.1	Graphical User Interface Patterns	184
6.2.2	Natural User Interface Patterns	186
6.3	Final Remarks	187
	Bibliography	189
A	Participant task sheet	201

List of Figures

1.1	Windows Media Player, a GUI-based application for interacting with media files such as audio or video.	7
2.1	A behaviour driven test case to log into a website	23
2.2	Navigating the file directory through a command line interface and getting the line count of a file.	43
2.3	Windows File Explorer, allowing file system navigation through a graphical user interface (GUI).	45
2.4	The Leap Motion Controller, a Natural User Interface allowing computer interaction through the use of hand tracking (source: “Modelling Hand Gestures to Test Leap Motion Controlled Application” [145]).	47
3.1	The web canvas application MakePad (https://makepad.github.io/) and its corresponding DOM. The highlighted “canvas” has no children, hence widget information cannot be extracted.	65
3.2	Objects in a screen shot from the Ubuntu print application’s GUI.	68
3.3	YOLO predicting object class and dimensions in an image. (Source: “You Only Look Once: Unified, Real-Time Object Detection” [112])	69

List of Figures

3.4	A generated Java Swing GUI. Each element has a random chance to appear. (Source: White et al. 2019 [146])	71
3.5	Generated GUIs by two techniques: random widget selection and placement (left), and random GUI tree generation (right).	73
3.6	A generated GUI tree and the corresponding GUI derived from the tree.	76
3.7	Data inflation transformations. The right images contain a brightness increase; the bottom images contain a contrast increase. The top left image is the original screen shot.	81
3.8	The loss values from the YOLOv2 network when training over 100 epochs.	82
3.9	Intersection over Union (IoU) values for various overlapping boxes.	85
3.10	Precision and recall of synthetic data against real GUIs from Ubuntu/Java Swing applications.	88
3.11	Manually annotated (a) and predicted (b) boxes on the Ubuntu application “Hedge Wars”.	88
3.12	Confusion matrix for class predictions.	89
3.13	Predicted bounding boxes on the OSX application “Photoscape X”.	91
3.14	Precision and recall of synthetic data against real GUIs from Mac OSX applications.	92
3.15	Predicted widget boxes and the corresponding heatmap of predicted box confidence values. Darker areas indicate a higher confidence of useful interaction locations, derived from the object detection system. (Source: self)	93

3.16	Branch Coverage achieved by a random clicker when clicking random coordinates, guided by predicted widgets positions and guided by the Swing API.	95
3.17	Interaction locations for a random tester guided by different approaches.	101
3.18	Precision and recall of a widget prediction system trained on a selected subset, or the entire training dataset of labelled GUIs.	102
3.19	Precision and recall of synthetic data against manually annotated application GUIs.	103
3.20	Precision and recall of the widget prediction modeling using an area overlap metric	105
4.1	Branch Coverage of techniques <i>APPMODEL</i> , <i>WINMODEL</i> , and <i>WINMODEL-AUT</i>	128
4.2	Interaction cluster centroids for the “LEFT_DOWN” event when using the <i>WINMODEL</i> approach.	129
4.3	Interaction cluster centroids for the “LEFT_DOWN” event when using the <i>APPMODEL</i> approach.	129
4.4	Interaction cluster centroids for the “LEFT_DOWN” event in the <i>Simple Calculator</i> application.	130
4.5	Interaction cluster centroids for the “LEFT_DOWN” event when using the <i>WINMODEL-AUT</i> approach.	131
4.6	Branch Coverage of techniques <i>APPMODEL</i> and <i>RANDOM</i>	132
4.7	Branch Coverage of techniques <i>APPMODEL</i> , <i>PREDICTION</i> , and <i>GROUNDTRUTH</i>	133
4.8	Branch Coverage of techniques <i>USERMODEL</i> , <i>RANDEVENT</i> , and <i>RANDCLICK</i>	137
5.1	The data structure used by the Leap Motion Controller	146

List of Figures

5.2	Interaction with the application “PolyDrop” through the Leap Motion	148
5.3	Recording user interactions with the Leap Motion Controller and splitting data into subsets.	151
5.4	Our approach: generating features before combining them into a realistic data object.	152
5.5	Sequences of hands generated as input data using different techniques	154
5.6	The five applications under test, used to empirically evaluate our framework.	159
5.7	Line Coverage for different data generation techniques for each application.	163
5.8	Line Coverage for different data generation techniques for each application across time.	165
5.9	Line Coverage for models using either multiple or single person data when training for each application.	168
5.10	Line Coverage for models using either single or multiple model generation for each application.	169
5.11	Line Coverage for models using either an application model or a window model for test data generation for each application.	173
5.12	Line Coverage over time for models using either an application model or a window model for test data generation for each application.	173

List of Tables

3.1	Widgets that the model can identify in a Graphical User Interface	74
3.2	The applications tested when comparing the three testing techniques.	87
3.3	Branch coverage of random guided by no guidance, widget prediction and the Java Swing API. Bold is significance.	98
4.1	Features of each event type	115
4.2	The applications tested when comparing the three testing techniques.	122
4.3	Techniques of using user data to generate GUI events for interactions.	124
4.4	Techniques of using user data to generate GUI events for interactions.	125
4.5	Techniques of using user data to generate GUI events for interactions.	126
4.6	Branch Coverage of techniques WINMODEL, APPMODEL, and WINMODEL-AUT. Bold indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).	128
4.7	Branch Coverage of techniques APPMODEL, RANDOM, and WINMODEL-AUT. Bold indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).	132

List of Tables

4.8	Branch Coverage of techniques USERMODEL, PREDICTION, and APIGROUNDTRUTH. Bold indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).	134
4.9	Branch Coverage of techniques USERMODEL, RANDEVENT, and RANDCLICK. Bold indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).	137
5.1	Code coverage for different data generation techniques for each application. Bold is significant ($P < 0.05$).	163
5.2	Code coverage difference between single and merged data sources for each application. Bold is significant ($P < 0.05$). .	168
5.3	Code coverage for single or multiple model generation for each application. Bold is significant ($P < 0.05$).	170

1 Introduction

1.1 Overview

Sets of instructions understandable by computers can be combined to perform complex functions. This is known as software, and plays a vital role in every aspect of all areas of the world. Software acts as a high level layer, allowing complex calculations to be performed on a computer without needing the low level knowledge of a computer's internal workings. There is a high dependency on software in society and a need for software to work robustly. This is even more important in high risk domains such as control systems or finance, where a single mistake could cost greatly or even put life at risk.

Users interact with software through a user interface (UI), which allows users to execute desired actions and observe the results. There are many kinds of UIs, allowing different and unique methods of interaction with software. The most common interface is the graphical user interface (GUI), which displays the events possible in an application through visual entities ("widgets") like buttons and text fields. However recent advances in technologies such as virtual reality have also exploited natural user interfaces to allow application interaction, which uses body tracking or information from the real world as input.

Developers often have to interact with the UI of an application during development to ensure that an application has correct functionality. This is a form of manual testing.

Testing is the process of finding problems in an application that are not intended (“bugs”) and is needed to reduce the cost of fixing faults in software during development, before problems can occur at a later point. Testing is an expensive part of software development, but is needed to reduce the chance of unintended behaviour being present in a released application. Testing is an essential part of the software development life cycle, and quality control of software is a key difference between software projects that succeed, and ones which are cancelled or fail [70].

As software is increasing in complexity, the cost of developing software is also increasing [22]. Due to this growing expense, any issues encountered on released software can have negative economic consequences. Because of this, the quality of testing during software development can have a significant effect on the total cost of software development and maintenance [49]. However, the increase in software complexity, as well as the range of available inputs that users can interact with, provide a challenge for automated methods of software testing.

To counteract the high cost of testing, it is often desired to automate parts of the testing stage. One technique which aids this automation is the ability to generate input data for an application.

1.2 Test Data generation

Tests involve executing an application under some condition and observing the application's behaviour. Construction of tests can be made easier or even fully automatic by generating test input data for an application. For example, if a value inside the input domain of a function causes the function to throw some exception which is left uncaught, it could indicate a bug inside the function.

Many tools exist that can generate test data for an application, each using some technique of deriving information to aid in test data generation [45, 101], or using purely random generation [105, 37]. It is not practical to try all combinations of inputs from the input domain of a function to exhaustively test for bugs, especially when it is not clear to an automated test generator what the actual expected functionality of a program should be [12].

The aim of test data generation is to product test inputs which use a *representative* subset of the available input domain. If some specification of the application is provided, then the output for each generated input can be evaluated against the specification to check for correctness, or other forms of error detection can be used (e.g., if the application halts/never returns when it is given some input, there is a high probability that this is due to a bug). Usually, an unhandled exception in an application indicates that unintended use cases and possible side effects could be present in the source code.

1.2.1 Motivation for Automated Test Data Generation

Many tests manually created during software development use hand-picked input values to ensure that a function is performed correctly under different scenarios. As stated before, it is unrealistic to test a function using every combination of values from the input domain, and evaluating these against the correct, expected output. As a motivating example, take the following Java class:

```
1  public class PointHelper {
2      public float getMagnitude(float x, float y, float z){
3          return (float) Math.sqrt((x*x) + (y*y) + (z*z));
4      }
5
6      public float [] normalisePoint(float x, float y, float
7          z){
8          float magnitude = getMagnitude(x, y, z);
9          if (magnitude == 0){
10             throw new IllegalArgumentException(
11                 "Magnitude of point must be > 0"
12             );
13         }
14         return new float []{
15             x/magnitude,
16             y/magnitude,
17             z/magnitude
18         };
19     }
```

The *normalisePoint* function will take a three dimensional vector, and return the corresponding unit vector representing the vector's direction. A unit vector always has a magnitude of one. Unit vectors are used in equations such as lighting in computer graphics. If we are testing

the *normalisePoint* function declared on line 6, there are a few testing objectives which we may try to achieve, for example:

- *normalisePoint* should return the appropriate mathematical sign for all inputs (i.e., if a positive value is passed in for x , the first value in the returned array should also be positive);
- when *getMagnitude* returns 0, an exception should be thrown;
- *getMagnitude* of the three values returned from *normalisePoint* should always equal 1.

This list is not exhaustive but is a good start to adequately testing the *normalisePoint* function. A test designer can then manually construct a test which calls *normalisePoint* with certain values, and checks the output returned.

However, it may also be possible to automatically generate inputs to this function and observe the output [4]. By analysing the code, a test data generation tool can observe that there is a branching condition on line 8, when $magnitude = 0$. Therefore, a test generator can attempt to solve the *getMagnitude* function, which the *magnitude* variable is assigned to on line 7. Generating inputs such that the value returned from *getMagnitude* equals 0 could involve trying different input values and observing the output. To solve this function with respect to 0, mathematical techniques can be used to find the real root of this equation, where $x=0, y=0, z=0$. This is an easily solvable function, and a test can be automatically generated asserting that an *IllegalArgumentException* is thrown when *normalisePoint(0, 0, 0)* is called.

Having this test automatically generated can save time for developers, and tests like this can enhance the current set of tests, helping fill gaps

in a test suite missed during test construction. A developer can read the generated test, and ignore it, accept it, or modify and accept it. The automatic generation of this test was cheap, involving no input from the developer until it is either accepted or rejected into the existing test cases. A developer also has to validate the correctness of the generated test, as it could have been generated on a function containing faults. As stated previously, an unhandled exception can signify erroneous behaviour in the application, but for this function, was expected behaviour.

Automatically generating test data is not always easy. We studied the *getMagnitude* function, which was a simple case that can be solved using well known mathematical techniques of finding real roots of an equation. However, there are far more complex functions where solving the equation using the constraints extracted from a function is difficult or even impossible. For example, when testing a visual application, any pixel on the computer monitor can become a point of interaction, possibly triggering events in the underlying application.

For natural user interfaces, how can realistic tests be generated to sufficiently test an application?

Why Test Through a User Interface?

Users interact with software through a user interface. The output from the software is present in the interface, and any possible inputs are processed through the interface. When software interactions are processed through a user interface, the system acts as a single entity [144] and certain faults can be revealed that were not detected by other testing levels. Testing the software as a complete system is also the only method

1.2 Test Data generation

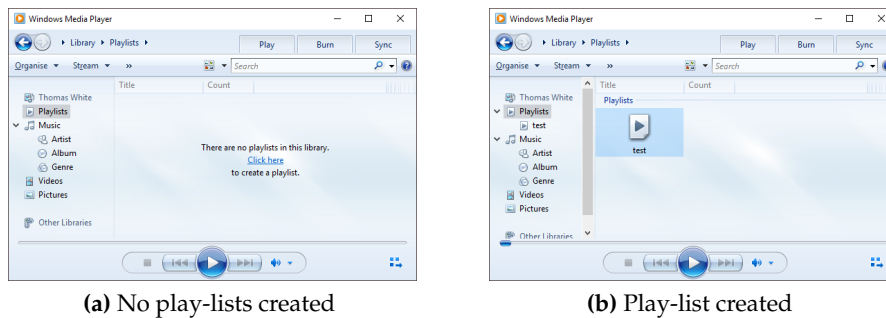


Figure 1.1. Windows Media Player, a GUI-based application for interacting with media files such as audio or video.

of acceptance testing [64], which demonstrates that the finished software meets the requirements of the customer. Testing traditional software typically consist of sequences of API calls and checks ensuring that the output from each call is correct. However, testing through a user interface is more difficult. A user interface provides predefined data structures to an application, and these data structures are much more challenging to generate automatically. For example, the input for an application controlled by a Microsoft Kinect input device consists of a collection of points in 3D space, which collectively represent the body of the programs user. Without the ability to automatically generate test data for a user interface, the cost of testing increases and more bugs may be present in the final system.

As an example, graphical user interfaces provide a visual method of interacting with applications. Figure 1.1a shows Windows Media Player, an application which allows user interaction through a graphical user interface. One test for this application could ensure that new play-lists can be created:

CLICK "playlists"

CLICK "click here"

TYPE "test"

PRESS ENTER

ASSERT "test" exists

Executing this test would result in Figure 1.1b, and the assertion on the last step would pass. This test executes many components in the application together, and if one component fails then the test may also fail.

It is difficult to automatically generate a test similar to that above due to many reasons. For example, there needs to exist some mapping such that all available interactions on the graphical user interface are known. Then, an interaction can be selected. Without this mapping, testing tools can fall back to an approach which interacts with random coordinates on the screen, but these sequences of interactions are unlikely to be effective. Even knowing this information, some custom widgets could limit the potential for GUI test generators, which usually achieve less than 70% code coverage on average [57, 90, 103], with one recent study showing test generators on the Android platform to achieve less than 40% coverage after one hour generation time [33]. A smarter approach of deciding the positions of generated interactions is needed.

1.2.2 Summary

Testing is an important aspect of software engineering, providing confidence that software will perform as expected. However, manually writing tests for software can be expensive and repetitive for develop-

ers. Generating tests and test data can be a cheap method of enhancing a test suite, filling gaps in the tests that were missed during manual test construction. Generating tests is cheap relative to the cost of manual construction, but generated tests still need inspecting to assert the correctness of functionality of the subject under test, especially when test assertions are checked against the current functionality and not the expected functionality that would be provided by specifications.

In this thesis, we propose techniques which do not need to find and solve an application's constraints to generate tests. These techniques have no access to the source code behind an application, and do not know the different conditions in the function which tests are targeting. Instead of relying on a tester to manually construct test cases, it may be possible to build a model of the interactions between a user and an application. This model can then be used to generate novel test data. For this, we observe interactions from a real user during application use, or exploit the image processing and object detection area of machine learning to interact with an application using the only information available to users: screen shots.

1.3 Structure and Contributions

Chapter 2: Literature Review

An overview of current work in software testing including current practices, automated test generation, and how to test an application through the many devices that users have access to for interaction. We focus more on testing an application through simulating these devices, and how to generate tests when little information about the target appli-

cation and environment is known. We found that there was very little work on interacting with an application when certain assumptions are removed, such as knowing what framework or structure is used to process user interactions. Users do not need this information to understand how to interact with an application.

Chapter 3: Guiding Testing through Detected Widgets from Application Screen Shots

Without the assumption of a pre-known programming language/framework for handling user interactions, it is difficult to generate test data for an application. In this chapter, we focus on using machine learning to create a model from generated data. The contributions for this chapter are:

Contribution 1: A technique for generating synthetic screen shots of GUIs in large quantities that are automatically annotated. The generated screen shots can be used by machine learning algorithms to train a model capable of automatically annotating screen shots of real GUIs. The generated data is cheaper than manually gathered data, and the properties of the whole dataset can be finely controlled during generation.

Contribution 2: Creating and training a system based on machine learning and applying this model to the context of testing an application through interaction with its graphical user interface (GUI). Using a screen shot, areas of interest in an application's GUI can be identified and provided to a test generator.

Contribution 3: An improvement to a random GUI tester through ex-

exploitation of interesting areas of a GUI, identified from the model trained in contribution 2. By modifying a random testing algorithm, and focusing generated interactions on those parts of the screen that are likely to be relevant to a real user, more targeted tests can be generated.

Contribution 4: An empirical evaluation that compares a random tester guided by interesting areas from contribution 3 against current testing approaches. Current approaches assume knowledge of the source code or data structures in an application, and hence, can target generated interactions in even more specific areas when testing.

Now interactions can be generated using only the information presented to users, in the next chapter, we compare the models we created using these machine learning techniques to models we derived from real user interactions with an application.

Chapter 4: Testing By Example: Graphical User Interfaces

With crowd sourcing platforms like Amazon’s Mechanical Turk [26] and even crowd testing platforms like Code Defenders [117], it is plausible that user interaction data with an application already exists or can be purchased. In this chapter, we look at exploiting this user interaction data to create models capable of automatically generating tests for an application. The contributions of this chapter are as follows:

Contribution 5: An identification of low-level interaction events that are needed to replicate user interactions using a keyboard and mouse. What data is needed to replay a user’s interaction with an application through an application’s GUI? We identify four key event types that need to be stored to reconstruct and replay a sequence of user events.

Contribution 6: A technique of processing user GUI interaction data into a model capable of generating interaction data that is statistically similar to that of the original user's interactions. The resulting model can have different levels of abstraction, from a single user model of the whole application, to separate models for each window that appears in an application.

Contribution 7: An empirical study comparing different techniques of generating models from user interactions, and comparing them to current testing techniques of testing an application solely through its GUI. Here, we compare the approaches from chapter 3 against the approaches that are derived from user data, observing the impact that sequential data can have on generated sequences of interactions with an application.

We have generated tests for interaction with a graphical user interface through two techniques: learning from synthesised data, and learning from real user data. Now, we investigate if these techniques can also be applied to a different type of interface: natural user interfaces.

Chapter 5: Testing By Example: Natural User Interfaces

In this chapter, we automatically generate test data for the Leap Motion, a device which enables interactions with an application through hand tracking.

Contribution 8: A framework for storing, modelling, and generating sequences of hand movements similar to that which the Leap Motion would provide under real usage. By learning from real users, we create a model that can generate statistically similar data to a real user.

Contribution 9: An empirical evaluation of different techniques of exploiting user data when testing 5 Leap Motion applications. We propose different approaches to using user data which change the model created, and here we identify which approaches are effective and when. We also compare these approaches to previous approaches.

Contribution 10: An empirical evaluation of the impact that training data can have on a derived model, and how this influences the generated test data when testing an application. By combining data from multiple users, it is possible to create a model capable of generating more diverse data and therefore testing an application more thoroughly.

Contribution 11: An empirical evaluation showing the effects of splitting user data into related sub-groups, and training a model for each sub-group. The sub-groups are then recombined to form the generated test data to input into the application under test. Certain areas of user data are related to others. We look at these relationships and how user data can be isolated and combined to allow the possibility of generating more data than that which was originally provided by the user.

Conclusions and Future Work

The conclusion contains a summary of the work undertaken in this thesis, with the results achieved. Finally, the end of the thesis presents future work, extension points and problems identified which need to be addressed.

2 Literature Review

2.1 Introduction

When interacting with an application, users often get frustrated when the application acts in an unexpected way. This usually occurs through a bug or fault in the application introduced through developer error in the development or maintenance phase of the program. Throughout this chapter, software testing, which aids developers in detecting bugs, will be outlined. Afterwards we study automated techniques which exist to reduce the high cost of testing.

2.1	Introduction	15
2.2	Software Testing	16
2.3	Automated Test Generation	29
2.4	User Interface Testing	43
2.5	Summary	59

2.2 Software Testing

Software testing consists of observing software executions, validating that the execution behaves as intended, and identifying any errors during the execution [20]. Software testing aims to uncover bugs in computer programs. When a program is given some input, the program's output can be compared against the expected output to check for correctness. The comparator of these outputs is known as an oracle [12] and decides whether a test passes or fails.

Software testing is not an easy task. Each program has varying levels of testability. A program with high testability means that bugs are easier to detect, if any exist. Low testability applications increase the difficulty in detecting bugs and testing, often having scenarios which are not common. These scenarios can cause behaviour which conflicts with the specification for certain inputs if these inputs are not specially handled (*edge cases*). It is known that certain programming paradigms such as object oriented programming (OOP) have a lower testability than other paradigms (e.g., procedural programming) [139].

Software testability varies across different applications. If all bugs in software caused a program to crash or fail, then software would have high testability [138] and testing would be easier. However, this is not the case, and the chance of detecting bugs is lowered as software may only return incorrect data in specific scenarios. This increases the cost of testing and the chance of missing bugs which can be released into a production system, increasing the overall cost of software development. During development, there are various types of mistakes in software that can be detected, and various techniques of testing different aspects of software.

2.2.1 What is a Software Error, Bug and Failure?

When developing software, it is possible that developers make *errors* through misunderstandings of specifications, misconceptions, or innocent mistakes [27]. If an error impacts the software such as to break the specification, then it is now a *fault* [27] or *bug* [50]. If part of the software cannot perform the required functionality within the performance requirements of the software specification, then this is known as a *failure* [27]. But how does an error by a developer propagate and become a failure in the application?

An application has a data-state, a map containing each variable and their respective values [76]. When a buggy line of code, through developer error, is executed, the data-state could become *infected*. The data-state now contains an incorrect value (data-state error). This can lead to incorrect program behaviour or a failure if the infected data-state *propagates* through the application, rendering the application unusable.

To ensure that developers and users maintain confidence in an application, it is important to check for correctness of an application. Some applications can only have guaranteed correctness if exhaustively tested, i.e., the output for all possible inputs is checked against the expected output [138]. Given a simple program that takes a single signed 32-bit integer, the domain of possible inputs is 4,294, 967,295 in size (assuming -0 and 0 are equivalent). It is infeasible to check the expected output against actual output for all possible inputs. Instead, a method of selecting a subset from the input domain is needed. By analysing the source code and using values that execute different parts of the source (white box testing) or by following a program's specifications and use cases (black box testing), a representative test set can be built. These tests can

be categorised into different types.

2.2.2 Testing Methods

The two main types of tests we will discuss are white box and black box tests. White box testing involves knowledge of the internal workings of an application.

White Box Testing

White box tests are created to ensure that the logic and structure powering a program is valid [75]. White box testing consists of a developer inspecting the source code of an application and designing tests to achieve some form of coverage over the code base [13]. To guide white box tests, it is important to have some percentage of the system which has been tested, so testing effort can be most effectively targeted at parts of the system likely to contain bugs and which are not already covered by a test.

Coverage Criteria

If a bug exists, then it can only impact a program if the corresponding statement is executed. When executing a buggy statement, program failure can occur, or an infected data-state could propagate and cause other issues [138].

Applications have various operations which cause different areas of source code to execute. Different inputs trigger different areas of code execution. By tracing all possible different executed areas and the paths between areas, it is possible to create a graph. This is known as a control flow graph.

Because a bug needs to be executed in order to impact software, it is important that most of the source code or states in software is executed (i.e., a high coverage is achieved) when testing. There are a number of different coverage criteria that could be used to assess the quantity of code executed by a test, and each coverage metric reflects specific use cases triggered by the inputs into a function:

- Function coverage – also known as method coverage, the percentage of all functions in an application that have been called.
- Line coverage – also known as statement coverage, the percentage of source lines of code (SLOC) executed during testing.
- Condition coverage – the percentage of boolean expressions that have been assigned both a true and a false value at some point during test execution.
- Branch coverage – also known as edge or decision coverage, the percentage of edges in an application’s control flow graph that have been traversed.

Some coverage criteria subsume others. If complete line coverage (100%) is achieved, then it must also be the case of complete function coverage. However, on the contrary, complete function coverage could leave many lines uncovered by a test suite. Throughout this chapter, we will focus mainly on line and branch coverage, which can be calculated cheaply at test-time by instrumenting the application’s compiled byte code and have been used in various other studies (e.g., [45, 116]).

Line Coverage

Line coverage is a measurement of how many lines in a program are executed when the program is tested. Lines can be uncovered for various

reasons including a function which is never called during testing, or a branching condition never evaluating to one of the two possible values with the current test suite.

If a bug is present on a line of code, it can only impact the program if that line of code is executed. A common requirement for adequate testing is having all executable lines in a program covered by at least one test [156]. However, this does not mean that complete (100%) line coverage will detect all bugs [72].

“ If you spend all of your extra money trying to achieve complete line coverage, you are spending none of your extra money looking for the many bugs that wont show up in the simple tests that can let you achieve line coverage quickly. ”

Cem Kaner, 1996 [72]

Branch Coverage

A branching statement is one which can execute different instructions based on the value of a variable [72]. For example, an *if* statement could go inside the *if* body if some condition is true, but skip over the body if the same condition is false. The following Java code shows a function which calculates the absolute value of some input:

```
1 int abs (x){
2     if (x < 0)
3         x = -x;
4     return x;
5 }
```

If we called the function *abs* with parameter $x=-1$, we execute lines 2, 3 and 4 achieving 100% line coverage. However, we have only covered one of two branches. Line 2 is a branching condition, and has two

possible outcomes depending on the value of x : jump to line 3 and execute line 4; jump to line 4. Only the first of these outcomes has been tested when using $x=-1$ as input. We also need to call the function *abs* with a parameter $x \geq 0$ to test the other branch.

Guiding Testing Effort

Different coverage metrics can be used to guide testers when creating a test set for an application. There is no golden metric that applies to all applications [72] and that indicates that testing is complete once complete coverage in this metric is attained. To guide testing effort, there are other methods that can be utilised.

To help testers identify locations in the source code with a high probability of masking bugs, Voas [137] presents PIE, a technique for analysing an application for locations where faults are most likely to remain undetected if they were to exist. PIE does not reveal faults, but randomly samples inputs from the possible input domain and uses statement coverage to identify areas where bugs are most likely to be hidden. It can see how resilient an application is to small changes (mutations) and use this to provide feedback to testers.

One criticism of this technique is that random sampling of the input domain frequently achieves a shallow level of coverage, and more systematic approaches to generating test data have been shown to increase coverage achieved [57, 91]. By using a random sample of the possible input domain, certain metrics predicted by Voas (e.g., propagation and infection estimates) fall outside the confidence interval bounds when the same metrics are calculated using the entire input domain [76]. This is due to the PIE technique being overly sensitive to minor variations

of parameters and input values. Also using random sampling of the input domain assumes that a function will take a uniform distribution of inputs from the entire domain. However, in regular software usage, certain values and functions appear more often than others. It might be more beneficial to use an operational profile of the software to target testing effort into functions which will be executed the most [78].

In summary, white box testing involves designing tests to cover different coverage criteria and relies on knowledge of low level implementation details of the application under test. If these details are unavailable, black box testing is a possible option.

Black Box Testing

Black box tests do not require low level knowledge of an application. Instead, the specifications of an application are used when designing tests. Black box (also known as functional) testing is a methodology where testers do not use existing knowledge of the internal workings of the system they are testing, instead creating tests and test inputs from the specifications of the application [13]. When tests are executed, the expected result from the specifications can be compared to the actual result from the application. In depth knowledge of the system is not required to create black box tests, and tests can be designed solely from the systems specifications and requirements.

Black box tests can be written in a specification-independent language, for example, behaviour driven tests when using Behaviour Driven Development (BDD) [110]. See Figure 2.1 for an example of a behaviour driven black box test. This test does not require knowledge of the logic behind the system, and can be written by anyone with a specification of

GIVEN Gordon is on the login page
WHEN he enters a correct user name
AND he enters a correct password
THEN he successfully logs into the system.

ure

Figure 2.1. A behaviour driven test case to log into a website

what the system should do. This black box test is easy to understand by anyone, even those without prior programming knowledge. This test will execute on the final system, consisting of all components working together and interacting with the system as an end user would. It is possible to track the coverage achieved by black box tests, but more difficult to use this information to guide testing effort.

The main difference between white box and black box testing is that white box testing is mainly used to assert that the underlying logic in an application is correct, and black box testing relies on ensuring correctness in the specifications of an application. Both white box and black box tests can be executed against an application, and may reveal faults if a test performs differently when executed against an application with a changed code base (e.g., one with a new feature added).

Test Automation

There are several techniques and coverage criteria which can be used to focus testing effort for an application. Knowing where to focus testing efforts aids in construction of new tests. However, having a developer manually performing the same tests on an application is tedious and increases the likelihood of mistakes in the tests. There are several methods of repeating tests automatically. These tests can run against

new releases of an application, cutting down the manual testing cost for newly implemented features and future program releases.

It is common to design test suites that can be executed repeatedly in the development process of software. These suites consist of various tests which execute and validate the functionality of the current application version. If a test which passed on a previous version starts failing after a new version is released, then this could indicate that a bug has been introduced by the changes between versions. This is known as a software regression [133]. Tests which have failed previously due to a regression have an increased chance of failing when executed against future releases of the software [85].

There are various frameworks which aid in writing regression tests, like JUnit [93], TestNG [30] and qunit [134]. These testing frameworks have methods of running whole test suites, reporting various statistics such as failing tests, and can be added directly to a developer's integrated development environment. These frameworks come with standard functions such as *assertTrue*, which takes a single boolean parameter and fails a test if the parameter evaluates to false. All of these frameworks are used to construct a type of white box test called a unit test.

Both white box and black box tests can be used to reveal software regressions, but there are also various levels of an application that can be tested, and each level may reveal different faults in an application.

2.2.3 Test Levels

Applications often have many layers (levels), for example, with some form of back end layer responsible for storing and providing data to a

front end layer that a user can interact with. Testing an application at different levels can find faults in each layer, or in interactions between different layers. Here, we will talk about three test levels: unit, integration, and system.

Unit Testing

Unit testing aims to test an application by parts, ensuring each component functions correctly. The different parts an application can be split into are: functions; classes; modules; etc. A unit is the smallest testable part of a program. For example, in object-oriented programming, a unit can be a class or set of classes [151]. A unit test is a set of instructions that ensures the behaviour of a unit is correct, and observes the output for correctness using a developer's judgement. The following unit test is targeted at the *abs* function declared earlier:

```
1 void test_abs_positive(){
2     int x = 10;
3     int r = abs(x);
4     assert(10 == r);
5 }
6 void test_abs_negative(){
7     int x = -5;
8     int r = abs(x);
9     assert(5 == r);
10 }
```

The test suite above shows two unit tests for the *abs* function, which calculates and returns the absolute value of the input. To evaluate whether the functionality of the class is correct, developers use *assertions*. The *assert* function throws an exception if called with an input parameter of false. Usually, this will fail the test case and alert the developer of a pos-

sible bug. For the test functions to fully execute and pass, the assertions need to always evaluate to true.

Unit tests are popular for regression testing. Tests written for one version of a unit can also run on newer versions of a unit (e.g., if a new feature is introduced). If the output of a test differs for the old and new versions, then this could indicate that a bug has been introduced by the newly implemented feature (or a bug has been fixed). Because unit tests focus on small units of an application, they may also guide developers to the location or area of the application's source code containing a bug when they fail.

Integration Testing

It is possible to combine multiple units in an application and test this combination. This is testing one level above unit tests. Testing the interaction of components and the effect one component has when functioning in a system is known as integration testing.

Integration testing involves writing dummy units called stubs that the current testing target uses in place of the actual component, so only the functionality of the current testing target is checked [79]. The main effort of integration testing is writing stubs. The next test level involves testing a whole integrated system. This is known as system testing.

System Testing

Unit and integration testing is efficient at testing small parts of an application, but bugs could exist in the final system that cannot be detected from unit tests alone. Sometimes, interactions between components in

an application can cause other issues. To complement unit testing, a complete program can be interacted with and tested to ensure that all the components function correctly when working together (i.e., a system test). The system can be seen as an opaque, black box, where tests are designed to target the specifications of the complete system. These specifications and tests can be designed even before development begins.

Executing system tests can be automated using capture and replay tools. The tools observe some form of user interaction, and can replay the interactions at some point in the future. However, manually creating tests is expensive. Yeh et al. present Sikuli [t.yeah2009-sikul], a tool which uses image processing to increase the robustness of capture and replay tests by searching for the target elements of interactions in screen shots of the application. Alégroth et al. [152] show that using automated tools like Sikuli improved test execution speed at Saab AB by a factor of 16. Using an image processing library like Sikuli can also aid in maintenance of test cases. By matching image representations of GUI widgets, modifications to the source code of an application are less likely to produce a false positive failing test case. However, there is still a high probability that a change in theme or widget palette will make these tests fail when no regression has been introduced into the code base.

Systems tests can function similarly to an end user interacting with an application. Under normal application usage, users interact with a user interface which allows interaction with an application without knowledge of the application's internal workings. Developers create end-points to execute functionality of the application at a high level from the user interface. Applications can usually be controlled solely

through their interface so long as a developer has linked code in the software to the interface. There are many types of UIs available, but three have become more prominent. The three most popular types of UI are the command line interface (CLI), graphical user interface (GUI) and natural user interface (NUI). Each UI offers unique benefits and drawbacks.

One issue with all levels of testing involves how a test checks for correct functionality. For example in unit tests, choosing correct assertions relies on knowledge, or a formal set of the specifications of a unit. Because specifications of a unit or system are not always known, it is difficult to choose correct assertions. This is known as the oracle problem.

2.2.4 The Oracle Problem

An oracle in software testing is an entity which is able to judge the output of a program to be correct for some given inputs [61]. The oracle problem occurs because automated techniques of testing cannot act as an oracle: automated tools may not have prior knowledge or assumptions of specifications of a system, and so cannot decide if an output is valid and correct.

We previously saw a unit test for an *abs* function, which returned the mathematical absolute value of an input. We know the specifications to this function: it returns the positive representation of any input it sees. We can easily create assertions knowing this, acting as an “oracle”. Automated tools can call the *abs* function with random numbers as parameters and observe the output. It is also possible to automatically generate assertions from the observed output values. This is quicker than a developer having to think of values and manually creating a new unit

test. However, when using this approach of automatically generating tests, any bugs in the function will then also be present in the test suite. This bug can only be detected when an oracle with more knowledge of the specifications of the function manually validates the assertions generated, or a test can be checked against formal specifications.

Pastore et al. [107] show that it is possible to use crowd-sourcing as an oracle. When given some specification, the crowd decides which generated tests should pass and which should fail. This made it possible for tests to be generated automatically by developers with a single button click. Code Defenders [117] also produces crowd sourced oracles. Developers compete as players in a testing game, with half the players writing a test suite and the other half introduced subtle bugs into the application (mutations). A mutant is a simple change to one or several lines of source code used to assess the quality of a test suite [68]. The mutating (*attacking*) team score points by creating mutants that survive the current test suite written by the testers (*defenders*), who score points by killing mutants.

Although it is possible to crowd source the oracle problem, automated solutions to the oracle problem have received little attention and need to be studied further. This will allow automated testing techniques to reach full potential [12].

2.3 Automated Test Generation

In the previous section, different forms of testing were outlined. It is possible for developers and testers to manually create each type of these tests. However, techniques for automating creation of tests exist.

For example, unit tests can be generated [45, 105] or system tests for GUIs can also be generated [91].

Many tools exist that can automate the creation of tests for some software. For instance, AutoBlackTest can be used to simulate users interacting with a GUI [91] and CORE [2] can emulate network interactions with an application. Over the next section, we will look at tools that can generate test data for different types of UIs. Test generation tools are good for producing tests that achieve a high code coverage.

Automated testing works well for producing tests that cover a high proportion of a program in terms of code coverage. Despite this, automated testing often fails to reach program areas which rely on complex interaction, and are limited by the oracle problem [12]. Manually written automated tests are carefully designed to target specific areas of the source code. Rojas et al. [118] found a significant increase in coverage in 37% of applications when seeding manually written unit tests into a test generation tool, guiding generation of new tests. However, automatically generated unit tests do have their benefits, such as augmenting an existing test suite and are incredibly cheap compared to manually written tests.

The easiest form of test generation to implement involves sampling random values from the available input domain. This is known as random testing.

2.3.1 Random Testing

Random testing is a black box testing technique [40], having no biases in generated data through exposure to the internal workings and logic

of an application. Two basic types of random testing would be sampling from a numerical distribution [66] or generating random character sequences [98] for the numeric and character data types respectively.

Unit tests can also be generated using random testing. As an example, Randoop is an application which automatically generates unit test suites for Java applications [105], generating two types of test suites:

1. Contract violations – a contract violation is a failure in a fundamental tautology for a given language. An example contract is $A = B \iff B = A$.
2. Regression tests – tests that can run on an updated version of a program's unit to see if the functionality has changed.

To generate test suites, Randoop uses random testing and execution feedback, randomly selecting and applying method calls to incrementally build a test. As method arguments, Randoop uses output from previous method calls in the sequence. Because of the random nature, the domain of available method call sequences is infinite. Each sequence of method calls in the available method call domain can be a possible test for the program, but finding good sequences is a challenge. Randoop found a contract violation in the Java collections framework. This was found in the *Object* class, where *s.equals(s)* was returning false [105]. Random testing is not only applicable to object oriented languages. QuickCheck is a random test data generator, which generates tests for programs written in the functional language Haskell. Claessen and Hughes [34] found that random testing is suitable for the functional programming paradigm as properties need to be declared with great granularity, giving a clear input domain to sample test data from.

Random testing is cheap but can also aid more complex test generation methods. In the DART tool, Godefroid et al. use randomness to resolve difficult decisions, where automated reasoning is impossible [52]. When DART cannot decide a method of proceeding, e.g., it cannot find a value to cover a particular branch in a program, random testing is used.

2.3.2 Dynamic Symbolic Execution

Dynamic symbolic execution is a technique that can generate test cases with a high level of code coverage. One example of dynamic symbolic execution involves executing the application under test with randomly generated inputs [155] whilst collecting constraints present in the system through symbolic execution. Symbolic execution is a method of representing a large class of executions [77], and which parts of a program a class of inputs will execute. Symbolic execution is useful for program analysis, such as test generation and program optimisation.

Dynamic symbolic execution which relies on random testing still has a large quantity of values that need to be input through an application to find the input classes. However, there are methods of selecting points in the inputs domain that are not purely random, but guided by some function.

2.3.3 Search-based Software Testing

Search-based software testing (SBST) is an area of test generation that tries to search the large input landscape through a function which slightly changes the inputs over time using a guidance metric. SBST samples

and improves one or more individuals from a search space over time using meta-heuristic search techniques. An individual is a possible solution to the problem being solved (i.e., a test suite for test generation). Many individuals are tracked during a search, and are grouped into a *population*.

To improve the population over time, some method of comparing the *fitness* of an individual against other individuals in the population is required. The fitness of an individual corresponds to the effectiveness of the individual of solving the problem at hand [100].

A basic form of search algorithm is hill climbing. Hill climbing uses a population size of 1, taking a single random individual from the search space, and investigating the immediate neighbours in the population. The neighbours are individuals closest to the current individual, accessed through minor modifications to the current individual. Hill climbing is a greedy algorithm; if a neighbour has a higher fitness, then this neighbour replaces the original individual [94]. The search then continues from the new individual.

One disadvantage of a greedy approach is related to the shape of a fitness landscape. Fitness landscapes in test generation usually have peaks and troughs. Because of this, hill climbing does not always improve over random search and can get stuck following *local optima*, a peak in the fitness landscape that is not the highest. A local optimum is where no neighbours have a fitness that increases over the current best individual [69]. Random search has a high chance of producing fitter individuals than hill climbing in a search landscape that is rugged or when using multiple objectives [60]. However, there are other search algorithms which avoid this problem.

Evolutionary Algorithms (EAs) are a set of such algorithms. EAs are useful for solving problems with multiple conflicting objectives [157]. EAs maintain a population of individuals, which evolve in parallel. Each individual in the population is a possible solution to the task at hand, with their own fitness value. One form of EA is the Genetic Algorithm.

A Genetic Algorithm (GA) works on balancing exploitation of individuals in the current population, against exploration of the search landscape by looking for new, fitter individuals [63]. Exploitation is a local search, finding the fittest individual that can be exploited from the current population through crossover: the combination of multiple individuals. Exploration is exploring a wider range of the search domain, mutating an individual's genes to introduce new traits into the population.

An example tool which utilises SBST is EVOSUITE [45], a tool that automatically generates unit tests using a genetic algorithm. EVOSUITE generates tests for any program which compiles into Java Byte-code. The output is a JUnit test suite which can be used as regression tests against future changes, and can also find common faults, e.g., uncaught exceptions or contract violations.

Search-based software testing can generate more than unit tests. Jeffries et al. [67] found that using meta-heuristic algorithms to generate system tests requires several experts in order to be effective. However, they conclude that the many years of experience between the experts aids the meta-heuristic's performance, and that using meta-heuristics for user interface testing reveals a large amount of low priority or extremely specific problems. This could limit the effectiveness of search-based approaches when generating system tests.¹

Meta-heuristic algorithms work with a population of individuals and apply operators like mutation and crossover to increase the solving-capability of the population to a given problem. However, results using meta-heuristics in user interface testing are highly dependent on skilled testers. In order to solve a problem effectively, meta-heuristics rely on some method of calculating how effective an individual is to solving said problem. This is represented as an objective function.

Objective Functions

Objective functions evaluate an individual and calculate the fitness of the individual against the given problem. Individuals in a population can be directly compared using this calculated fitness value, and individuals with a minimal fitness can be discarded. An objective function can identify the fitter individuals in a population (i.e., the individuals which are closer to solving the problem). This can be used to guide a search through the problem domain [136].

For automated testing, many objective functions are used. The most common objective function directly correlates fitness of a test suite with a single or combination of code coverage achieved when executing all tests in the suite [143, 11, 97]. Using this objective function will result in test suites with high code coverage given enough search time.

Other criteria can also be integrated into objective functions, as code coverage is not the only important factor in a test suite. One issue with generated tests is that tests are often difficult for developers to understand. This increases the maintenance cost of generated tests and also makes it difficult for developers to identify why a generated test is failing when one does, and whether the fault is with the test case or with

the actual system. Generating tests and method call sequences can test robustness or contract violations effectively, but do not take readability into account [47]. This can often incur a high human oracle cost, but it is possible to increase readability by adding more criteria to the fitness function during test generation.

To increase readability of generated tests, SBST and user data can be exploited to select more realistic function inputs. One example by Afshan et al. [1] applies a natural language model to generate strings which are more readable for users, therefore reducing the human oracle cost. This is achieved through applying an objective function derived from a natural language model into the fitness function of a search-based test generator, giving each generated string of characters a *score*. The score is the probability of that string appearing in the language model, and can be used to alter the fitness value of a test suite.

To evaluate the effectiveness of using a language model when generating strings, Afshan et al. generated tests for Java methods which take string arguments. The candidate methods were selected from 17 open source Java projects. Tests were then generated for each method with and without use of the language model, and evaluated in a human study. Participants of the study were given the input to a method and in return they provided the output they expected to return. It was found that in three of the Java methods, using a language model significantly improved the correct output over not using a language model. Language models are not the only type of model that can be used to aid in test generation.

2.3.4 Model-based Testing

Model-based testing can generate tests, but a model of the target system is required. A model is represented as a specification, acting as an oracle and knowing expected outputs for given inputs to a function. The aim is to lower the labour cost of testing through test generation using a model [102], but creating the model comes with its own labour cost. Takala et al. [132] found that from an industrial point of view, developers may be unwilling to spend a significant amount of effort to learn model-based testing tools, and there should be future work invested into making these tools easier to use.

Model-based testing can identify the relevant parts of an application to test, and can even generate a formal model, e.g., a Finite State Machine (FSM) or Unified Modelling Language (UML) Diagram [38]. Formal models could also be created manually by developers instead of inferred automatically.

Using formal models, some forms of coverage criteria can be derived such as state coverage and transition coverage [135]. Apfelbaum and Doyle apply models in the system test phase of a Software Development Life-cycle (SDLC) [6]. With the system completed and built, interaction as an end user is needed to validate correct functionality. Due to requirements and functional specification often being incomplete and ambiguous, applying model based testing in the system test phase can reduce ambiguity and errors [6]. In this sense, modelling is similar to flow charting, describing the behaviour of the system that can occur during execution.

Model based testing can also be used without a specified model of the

system under test. One such example, MINTEST [106], is a black-box test generation approach where models are inferred from stored user execution traces. The inferred model can be used to derive novel notions of test adequacy. To evaluate the approach, mutation testing was used to measure test adequacy across three applications. It was found that the resulting test sets were more adequate than random test sets, and were more efficient at finding faults. However, a program trace for every possible output of the program was required to infer the model used when generating tests.

To reduce the number of program traces required to infer models, Walkinshaw and Bogdanov [141] present a technique which can execute passively, with a model provided in advance, or actively, where the developer is asked questions iteratively about the intended system behaviour. The active run configuration forces developers to think about different scenarios and edge cases. This technique infers a state machine of the application but using less input from the developer, and can generate counter examples, i.e., inputs and outputs which do not hold in a model of a program.

It is also possible to use models to aid in writing integration tests. To generate stubs, it may be beneficial to use a formal model of the system. For example, Harel [59] proposes statecharts, which model the flow of states and transitions in an application. Using state charts, it is possible to model components of a system. One such technique of modelling components is by Hartman et al. [62], with an aim to minimize the testing cost of the initial test stubs and test cases. It was found that whilst statecharts allow modelling of components in different states of the system, internal data conditions (i.e., the global state machine's variables) and concurrent systems were not supported by this technique. How-

ever, it is possible to model a system with no access to the source code, only the statechart and component interactions.

2.3.5 User Guidance

To overcome a local optimal, Pavlov and Fraser [108] ask for assistance from developers during test generation. To aid in the meta-heuristic search, developer feedback is included in EVOSUITE's search. If EVOSUITE's genetic algorithm stagnates in the search, then the best individual is cloned and the user is allowed to edit this individual. The edited individual is inserted into the genetic algorithm's population and the individual with the poorest fitness is removed. To evaluate user influence in the search, Pavlov and Fraser semi-automatically generated tests for 20 *non-trivial* classes [108]. They gave their own feedback when the search stagnated until no further improvement could be made. It was found that semi-automatic test generation improved branch coverage over automatic test generation by 34.63%, whilst reducing the amount of developer written test statements by 77.82% over manual testing.

It is possible for the search landscape in certain classes to hinder a genetic algorithm's search. When test generation for a given subject under test cannot be guided by a fitness value, a meta-heuristic algorithm falls back to a random search algorithm. However, Shamshiri et al. [125] found that as random generation executes around 1.3 times faster than a GA, random search can quite often outperform a genetic algorithm for certain classes.

One issue with generated test data is that sequences of calls and the parameters passed into functions is not representative of real usage of the

source code. Often, users will perform similar or identical tasks, and some functions will be called many times more than others. It may be more important to find bugs in the more commonly called areas of the code base before those in niche areas. To generate more representative test data, real operational data can be exploited. It is possible to use previous knowledge or sample test data from specific distributions representing the real data, rather than sampling randomly from the input's domain.

For example, Whittaker and Poore show how exploiting actual user sequences of actions taken from an application specification can be used when creating structurally complete test sequences [148], representing a path from an uninvoked application state to a terminating application state. For this, a Markov chain is used where each state of the chain represents a value from the application's input domain. Further, Whittaker and Thomason generate Markov chain usage models [147]. This chain contains values from the expected function, usage patterns, or previous program versions. The chain can then generate tests that are statistically similar to the operational profile of the program under test. It has also been shown by Walton et al. that usage models are a cost effective method of testing [142].

Common object usage exploits objects in source code and replicates them in unit tests. The objects were originally created by developers and have intrinsic insight into the application. Fraser and Zeller use common object usage to aid in generating test cases, making tests more similar to the developer source code [47]. To achieve this, Fraser and Zeller study the source code and any code available to clients from an API. Afterwards, a Markov chain is derived representing the interactions of classes in the users code.

To generate a test, Fraser and Zeller select a random method as the target of a test case and then follow the Markov chain backwards until the selected class has been initialised. Following the chain forward constructs an object similar to one observed in the source code, and this object can then be used as a parameter for methods when generating tests.

This technique was evaluated using the Joda-Time library, and it was found that using no user information (*no model*) achieved the highest coverage, but generated tests which violated preconditions for the methods of Joda-Time. Due to a lack of knowledge when using no model, parts of the Joda-Time specifications are ignored and so unrealistic branches are set as goals to be covered in the search. Realistically, these runtime exceptions would not be expected in regular application usage.

It is clear that using user data has a benefit in guiding test generation tools. User data can also provide operational profiles of an application. An operational profile contains the probabilities of an operation occurring through a system according to user interactions [8]. Finding the most commonly executed areas of an application may help in identifying bugs which have a high probability of occurring under normal usage. The parts of a system that the user executes are logged and probabilities of areas being executed can be calculated. These probabilities can be used to guide test planning and reveal areas of the system with high usage which may need more testing effort.

A threshold occurrence probability is assigned as $0.5/N$ where N is the total number of test cases for an operation. Test cases are allocated based on the probability of an operation occurring. After probability rounding, it is possible for an operation to have zero test cases assigned (i.e., when the probability is below the threshold occurrence probab-

ity) [8]. Operational profile driven testing is useful for ensuring that the most commonly used operations of a program have been tested efficiently. This is useful if, for example, the program has to be shipped early due to other constraints such as lack of funding or time [127].

Using these user-guided techniques, it is possible to generate objects with complex data structures. Feldt and Poulding combine engineer expertise, fault detection and simulated operational profiles to influence test data generation of complex objects [43]. Poulding and Feldt later found that this approach to generating complex objects is more efficient than an equivalent data sampler, whilst still being able to sample uniformly to maintain test data diversity [109].

Evaluating Automatically Generated Tests

The goal of testing is to expose faults or failures in an application, and this can only happen when tests fail. Tests only fail when an oracle's check evaluates to an incorrect value. One issue with generated unit tests is the quality of their assertions. Automated tools have a difficult time constructing strong assertions when specifications or a model of an application is missing. To evaluate the effectiveness of automated testing technique at exposing faults, there exists datasets of analysed and reproducible real software faults. Defects4J contains 357 bugs from five real world applications [71]. This also includes the test cases that expose these bugs. Test generators can create unit tests for an application on a non-buggy version and see if the tests detect the real bug. Another set of real faults is AppLeak, a dataset of 40 subjects focusing on Android applications which contain some resource leak from the Android API [115]. AppLeak contains the applications and tests to

```
thomas@DESKTOP-6K4UBPI: ~/work/thesis
thomas@DESKTOP-6K4UBPI:~$ ls
work
thomas@DESKTOP-6K4UBPI:~$ cd work
thomas@DESKTOP-6K4UBPI:~/work$ cd thesis
thomas@DESKTOP-6K4UBPI:~/work/thesis$ wc -l thesis.log
3754 thesis.log
thomas@DESKTOP-6K4UBPI:~/work/thesis$
```

Figure 2.2. Navigating the file directory through a command line interface and getting the line count of a file.

reproduce the leaks. Using these real faults, it is possible to evaluate the fault finding capability of automated software testing techniques through the oracle they use.

2.4 User Interface Testing

Users interact with software through a user interface (UI). When interacting with software through a UI, many components in the software may be working together, and faults between software components could be revealed. There are many types of UIs available for developers to integrate into their application.

2.4.1 Types of User Interfaces

Command Line Interfaces

A command line interface (CLI) allows access to a program solely through textual input via the computer keyboard. An example of an application controlled through a CLI is the UNIX *wc* command, which counts the number of words, lines, or bytes in a file (see Figure 2.2). CLI applica-

tions take parameters from a system's command line, and use textual output for communication with users. There are various methods of learning about a CLI application, with the most common being a help flag that can be passed into the application (e.g., `wc -help`). Using the help flag returns the documentation of an application, with commands that are possible and the arguments for each command. Each command maps to the corresponding source code controlling that functionality when a command is provided by the user through standard input.

CLIs are more commonly used by expert users, and can be quicker than other forms of interfaces [119]. The expert knowledge required to use a CLI-based application presents a problem for automated testing techniques. Tools such as *expect* [83] exist for writing tests that feed data into the standard input stream of an application, and monitor the standard output stream, comparing the output against a predefined expected value [84]. For applications without expert users, it may be beneficial to store common combinations of these commands and allow execution in a graphical, more memorable, way.

Graphical User Interfaces

Figure 2.3 shows Windows File Explorer. This application has similar functionality to the applications used in the CLI section, but is easier to interact with. Some notable differences between the CLI and GUI applications are that the GUI version has support for mouse interaction. The bar at the top of the application window allows quick and easy access to common functionality such as New Folder and Delete and the favourites bar allows quick navigation to common areas. The clickable icons are more user friendly than keybindings, and icons are

2.4 User Interface Testing

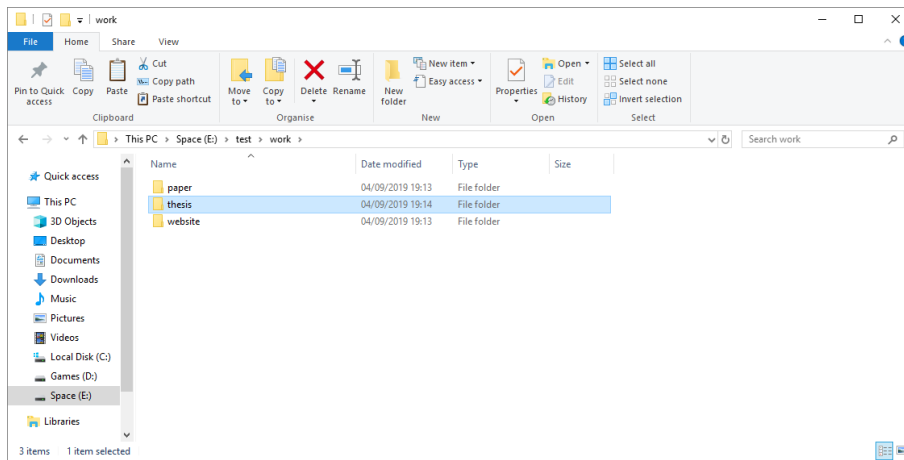


Figure 2.3. Windows File Explorer, allowing file system navigation through a graphical user interface (GUI).

usually common across a number of applications with each icon performing a similar task. In the CLI, a user has to remember the specific combination of commands to quickly access these same functions.

Other GUI conventions are present in Windows File Explorer too, like a scroll bar on the left hand side of the screen. By clicking and dragging the scrollbar up or down the screen, more content can be seen. Finally, a menu can be seen at the top of the screen. Menus are good at storing lots of functionality in a small, compressed space. When clicked, the menu expands revealing many actions useful to users. An example usage of the menu in Windows File Explorer would be *File – New Window*, which opens a new Explorer window targeting the same directory as the current one.

A GUI contains various widgets that can be interacted with, like buttons, scrollbars and text input fields. These widgets can have action listeners attached which execute certain code when a user interacts with a specific widget.

To create a GUI, some type of framework is usually used. An example

of a GUI framework is Java Swing [41]. Java Swing allows a GUI to be created by extending the Java class *JFrame*. A subclass of *JFrame* has methods like *add*, which can add common GUI widgets to the GUI, and widgets have the method *addActionListener* which has parameter of type *ActionListener*. For example, in Figure 2.3, we could recreate the *New Folder* button. We would instantiate the Java Swing button class (*JButton*), add it to the *JFrame* and implement the delete functionality in the *ActionListener* linked to this button. When the button is clicked, Java Swing will call the *ActionListener's* *actionPerformed* method. For more information on Java Swing and *ActionListeners*, see *The definitive guide to Java Swing* [158].

Testing an application through its GUI can be challenging. It is commonplace for capture and replay tools to be used [96], requiring lots of manual testing effort and producing tests which can easily break when a GUI is modified. However, there are approaches which can automatically interact with an application's GUI [92, 15, 103, 37]. These tools aim to find faults in the underlying application, and the event handlers which process user input into the application.

Natural User Interfaces

A Natural User Interface (NUI) allows interaction with a computer through more intuitive techniques, such as body tracking. NUIs provide a constant stream of data to an application which can react to certain events present in the data, such as predefined gestures. An example of a NUI is the Leap Motion Controller, which tracks a user's hands and allows applications to be controlled by displacing the hand in 3 dimensional space, or performing gestures like swiping the hand in a specific direction. Figure 2.4 shows the Leap Motion Controller and an application

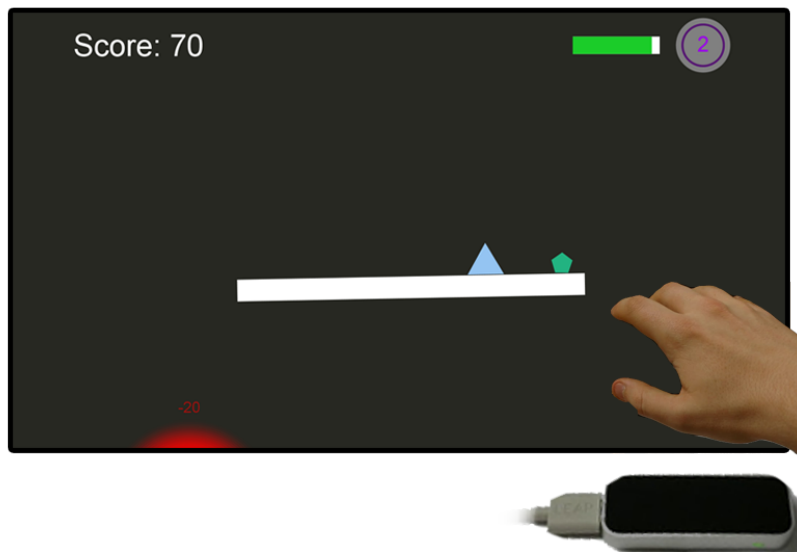


Figure 2.4. The Leap Motion Controller, a Natural User Interface allowing computer interaction through the use of hand tracking (source: “Modelling Hand Gestures to Test Leap Motion Controlled Application” [145]).

being controlled by a user’s hand.

The Leap Motion Controller tracks user’s hands through three cameras [80] and converts inputs into a Frame. A Frame is built of various related substructures, including predefined gestures the user is performing, hands that were tracked, *Pointable* objects being held (e.g. a pencil). Because of the relationship and limits of these substructures, a Leap Motion Frame is a complex structure which is difficult to automatically generate.

UIs often provide complex data to applications, and automatically generating this data is not a trivial problem. However, there are a few approaches that can be used, such as random testing.

2.4.2 Random GUI Testing

The simplest approach to generating system tests is via random testing. For example, GUI tests can click on random places in a GUI window, and this is known as *monkey testing*. With every click, there is a small chance that a widget will be activated. Miller et al. [99] developed an approach of testing UNIX X-Windows applications. Their testing program sits between the application and the X-Window display server, and can inject random streams of data into the application as well as random mouse and keyboard events. Using this technique, over 40% of X-Windows applications crashed or halted. Forrester and Miller [44] later extended this study to Windows NT based applications, finding that on application crashes, the user often was not given a choice to save their work or open a different file. Applications also produced error messages to users showing technical aspects such as memory addresses and a memory dump.

Monkey testing has been shown to be effective at finding faults when testing through an application's GUI, finding many crashes. Monkey testing is also cheap as no information is required. Android Monkey is an example monkey testing tools for the Android platform [37], which generates system tests. Despite being cheap, Choudhary et al. [33] declared Android Monkey "the winner" between multiple Android testing tools. However, monkey testing was effected by the execution environment. There is a distinct difference between the Android platform and Java applications in terms of code coverage, as random testing is less efficient in Java applications. The reason for this difference is unknown. Zeng et al. [153] evaluated Android Monkey on the popular application "WeChat" and found that the random technique often spent long periods of time exploring the same application screen. There

were two main reasons for this: 1) Monkey triggers interactions at random screen coordinates, having no knowledge of widgets, and this can waste time; 2) Monkey does not keep track of states already explored, and can cycle repeatedly between states.

2.4.3 Dynamic Symbolic Execution in GUI Testing

Random testing for GUIs is effective at finding faults, however, it may be more beneficial to generate more targeted interactions when specific inputs are required by an application. As an example, Salvesen et al. [123] use dynamic symbolic execution (DSE) to generate specific input values for text boxes in an application's GUI. After a program has finished execution, input values are grouped together depending on the path navigated when input into the program. It was found that using DSE significantly increased code coverage in both applications used in the case study compared to generation without DSE. Saxena et al. [124] use DSE in the tool "Kudzu", which automates test exploration for JavaScript applications. Kudzu uses a "dynamic symbolic interpreter", which records real inputs of an application and symbolically interprets the execution, recording the semantics of executed JavaScript bytecode instructions. Kudzu can generate high coverage test suites for JavaScript applications using only a URL as input. Kudzu revealed 11 client-side code injection vulnerabilities, of which two were previously undiscovered.

2.4.4 Model-based GUI Testing

To guide GUI test generators, Memon et al. [35] models the interactions that occur in GUI applications. Several definitions aid in creating a

model:

1. Modal Window: A window which takes full control over user interaction, restricting events to those available in the Modal Window.
2. Component: A pairwise set containing all Modal Windows along with all elements of the Modal Window which can trigger events through interaction, and a set of elements which have no Modal window (Modeless).
3. Event Flow Graph: Can represent a GUI component. An Event Flow Graph represents all possible events from a component.
4. Integration Tree: A tree representing components. Starting at the *Main* component, component *A* invokes component *B* if component *A* has an event that leads to component *B*.

Memon et al. also mentions other GUI criteria such as Menu-open Events and System-interaction Events. Using these definitions, it is possible to model GUI applications so long as they are deterministic and have discrete frames (e.g., no animations such as a movie player). From the created model of a target application, Memon et al. outline two new coverage criteria, specifically for applications which use GUIs:

- Event coverage – the percentage of events triggered by a test through an application's GUI.
- Event interaction coverage – The percentage of all possible sequences of interactions executed between pairs of some quantity of widgets in a GUI.

The new criteria provide feedback on a test suite interacting with a GUI. This can aid in producing a more complete final test suite. Memon et al. found that the properties of GUIs are different than conventional software and that different approaches were required to test them [35]. One reason for this is because of the complex structures which are constructed when interaction with a GUI widget occurs. The functionality that the widget is mapped to takes these complex data structures as input. These data structures are difficult to randomly generate. However, it may be possible to exploit knowledge of the underlying framework and the data structures used by a GUI to generate new interactions.

GUI Testing Specific Frameworks

Testing an application through a GUI requires interaction with widgets displayed on the computer screen. No prior knowledge of the application is needed, and often GUI testing is a black box approach.

However, identifying widgets in a GUI is not trivial. Each GUI framework uses different data structures and appearances. Consequently, most methods for testing GUIs rely on predefined underlying structures. Once the structures are known, it is possible to automatically rip a model of the system, or use random testing to generate GUI tests. For instance, Gross et al. proposed a tool which relies on a Java Swing testing framework [58]. Though the technique may be more abstract, any tool following the proposed technique will also need to rely on a similar GUI framework. Bauersfeld and Vos [15] present GUITest, a tool which relies on the MacOSX Accessibility API. GUITest constructs a widget tree of all widgets currently on the screen. Then, sensible default actions are performed. GUITest is a tool which can automatically

test an application via its GUIs, including complex functionality such as drag and drop. GUITest became Test*, a tool which aids GUI testers by deriving a GUIs structure automatically [120] and has been applied to different industrial scenarios [140]. However, the reliance on a GUI framework or an accessibility API means that many applications are still not support by Test*.

Borges et al. [23] link event handlers at a source code level to corresponding widgets displayed in a GUI. This involved mining the interface of many Android applications and deriving associations between event handlers and UI elements. These associations were gathered through crowd sourcing and can then be applied to new applications. Borges et al. found a coverage increase of 19.41% and 43.04% when supplying these associations to two state of the art Android testing tools.

Su et al. present FSMDroid [129], a tool which builds an initial model of an Android mobile application by statically analysing the source code. FSMDroid then automatically explores the application, updating the model as the tool tests. FSMDroid could increase the coverage achieved by the tests it generated by 84% over other Android model based testing tools, and also needed 43% less tests to achieve this increase. Choi presents the SwiftHand algorithm [32], which learns a model of the application being tested by utilising machine learning. SwiftHand will then exploit this model, focusing on generating tests for unexplored areas of the application. Both FSMDroid and SwiftHand learn how to interact with an application through exploring the available interactions available, and updating a model of the system.

When the specific framework used to create a GUI is known, so are the underlying data structures of events, or how to interact with a specific GUI. This opens up the opportunity to use search-based algorithms.

Search-based GUI Testing

GUI test generators can also be guided using search-based approaches. Mao et al. present Sapienz [89], an approach to generating system tests for Android applications. Sapienz has been deployed by Facebook as part of their continuous integration process, reporting crashes in the applications deployed by Facebook back to developers. Sapienz can handle a large amount of commits in parallel by utilising a network of mobile devices. To generate tests, Sapienz uses a hybrid approach, utilising both random and search-based algorithms.

When it is possible to get all events present on the screen through an API or known framework, and know specific GUI states, it is also possible to guide interaction generation through search-based approaches. One example of this is by Bauersfeld et al. [16], which presents the problem of GUI testing as an optimisation problem. Using a metric of reducing the size of existing test suites to guide the search-based generation approach, Bauersfeld et al. generate GUI interactions which aim to maximise the number of leafs in a call sequence tree. To achieve this, four important steps are needed.

Firstly, the GUI of the application under test is scanned to obtain all widget information. Secondly, interesting actions are identified (e.g., if a button is enabled). Thirdly, each action is given a unique name. Finally, sequences of actions are executed. The tool runs until a pre-defined quantity of actions has been generated. Bauersfeld et al. found that given enough time, this search-based approach could find better sequences of events than random generation.

This approach of generation was then extended. When GUI states can be extracted from an application's source code, the interactions between

different states can be represented as a graph. Then, search-based approaches to solving graph theory problems can also be applied to the problem of GUI testing. Bauersfeld et al. [17] use ant colony optimisation as a solution to this graph theory representation of the GUI testing problem. By generating sequences of events during application execution, no model of the application's GUI is needed and no infeasibility can occur. Carino and Andrews [28] evaluate using ant colony optimisation to test GUIs. It was found that using ant colony optimisation could increase the code coverage achieved by generated tests, and also the number of uncaught exceptions found.

Su et al. [130] use a different approach in their tool *Stoat*, a model-based testing tool which uses search-based techniques to refine the actual model involved in generating event sequences. *Stoat* uses a two phase approach to test generation. Firstly, it takes an application as input and reverse engineers a model of the application's GUI using static analysis. This is possible by exploiting the structures in the Android API. The second phase involves mutating this model to increase the coverage achieved and the diversity of generated event sequences in tests. *Stoat* was found to be more effective than other techniques of generating Android tests.

Another search-based approach which exploits a model of the application is by Mahmood et al. [88]. Their tool, *EvoDroid*, takes an application's source code as input, and can extract two models: an internal model of the application's call graph, and an external model of the application's interfaces. *EvoDroid* uses these models to begin an evolutionary search of the application's test input domain, keeping a population of individuals which represent a sequence of events in the application under test. However, *EvoDroid* is hindered by a known

limitation of search-based approaches: lack of reasoning when selecting input conditions [88]. It may be possible to overcome this lack of reasoning using machine learning techniques to process large quantities of user data and traces when creating a testing model.

2.4.5 Machine Learning in GUI Testing

Models can be generated from user data that can aid in GUI testing. For instance, Ermuth and Pradel [42] propose a tool which learns how to interact with GUI elements by observing real user interactions, adapting data mining techniques into automated testing. When a user interacts with a GUI, the execution trace is saved and exploited to simulate complex interactions, such as filling in a drop down box. This approach, and many other recent approaches to testing an application through its GUIs, work through machine learning techniques with data gathered through data mining.

It is also possible for a test generator to learn from their own interactions. Mariani et al. presents AutoBlackTest, a technique for generating test cases for interactive applications through GUI testing [91]. AutoBlackTest generates GUI tests and explores the application through Q-Learning, a machine learning technique which selects optimal decisions based on a finite Markov decision process. AutoBlackTest performs an event from the current GUI widget set, observes the results, and incorporates the results back into the Q-Learning algorithm for selecting future events. AutoBlackTest generates tests that achieve a high coverage but can also detect faults in the application missed by developers [92]. Further, Becce et al. [18] extend AutoBlackTest to search above and to the left of data widgets for static widgets that can provide

more information for testers about the type of data to input. Coverage increases of 6% and 5% were found in the two applications tested when providing context about data widgets to the tool AutoBlackTest.

Another example of a test generator learning from itself was by Degott et al. [36], who apply a reinforcement learning approach to testing Android applications. This was through presenting the problem of Android GUI testing as the Multi-armed Bandit Problem. This problem consists of some budget and various gambling machines. Users can learn which machine has the greatest chance of a high return on budget investment [9]. Degott et al. present each possible interaction with a widget as a gambling machine, and the budget as the execution time or interactions remaining. It was found that using two forms of reinforcement learning could lead to a coverage increase of 18% and 24% over the crowd sourcing approach by Borges et al.

It is also possible to apply areas other than machine learning to automated test generation. One area that links directly with GUI testing is image processing.

2.4.6 Image Processing in GUI Testing

To improve assertions for applications which use a web canvas, Bajamal et al. present an approach of generating assertions using image processing techniques [10]. A Document Object Model (DOM), can be exploited, extracting a map of all widgets in a website to their graphical representation on the screen. However, the contents of web canvasses do not have entries in the DOM, being drawn directly to an image buffer usually using JavaScript. Consequently, this structure cannot be exploited by testing tools, which see only a single “canvas” element.

Users can identify the elements inside the canvas as they appear similar to a normal widget, so have no issues interacting with the application. Bajammal et al. identified common shapes in web canvasses. However, many iterations of image processing techniques are needed to identify isolated shapes, each occurring an overhead in computation time. Once shapes have been identified, assertions can be generated and used in regression testing.

Sun et al. [131] also use image processing to guide a random GUI test generator. They investigate an approach of guiding a monkey tester by identifying interesting interaction locations from screen shots. Their tool, Smart Monkey, detects salient regions (i.e., interesting areas in the screen shot for interaction) of a rendered GUI using colour, density and texture. It was found that Smart Monkey can increase the likelihood of generating an interesting interaction (i.e., one that interacts with a widget on the GUI) over Android Monkey, although the actually hit ratio was still fairly low, between 21-55% depending on the application under test. However, the increased likelihood of interesting interactions enabled Smart Monkey to find crashes using on average 8% less testing budget than Android Monkey.

2.4.7 Mocking in NUI Testing

Natural User Interfaces (NUIs) allow users to interact with applications in a more intuitive method without physical contact to keyboard and mouse or a game controller [24]. NUIs commonly rely on techniques such as body tracking, requiring efficient algorithms that take up minimal computer resources but track users in real time [25]. An example of a NUI is the Leap Motion Leap [81].

Only a minimal amount of work exists for automatically testing an application via its NUI. This is worrying given their growth in popularity with systems like virtual reality (VR), and use in different domains such as medical [73], robotics [128], and touch screen devices [149].

A commonly used NUI is present in mobile devices: sensor and location based information. Mobile applications that rely on external sensors present an interesting problem for automated testing. Mobile phones contain a high number of sensors that contribute towards a Natural User Interface. Griebe et al. test context aware mobile phone applications through their NUI [55]. A context aware application is one which uses the physical environment around the device as an input, e.g., the device's current location. Griebe et al. firstly *enrich* the UML Activity diagrams with context specific events, then automatically generate tests using the *enriched* UML Activity Diagrams [55]. To evaluate this approach, tests were generated for the "call-a-cab" application, which relies on a devices current location and also a valid location being entered by the user as a destination. 32 total tests were generated, representing all paths through the system with regards to only these two inputs. Griebe et al. then extended this work with a test framework to allow simulation of sensor-based information [56]. This allows user motion of the device to be mimicked and used in test suites. Using the new motion test suites, it is possible to cover the source code which handles user movement interaction with the application (i.e., the user physically moving the mobile phone). When the new tests run, generated sensor-based information is used opposed to the real sensor-based API. The implication of this is that data can be generated and inserted into the application through the mimicked API, allowing this functionality to be tested through generated test suites.

2.4.8 Modelling User Data for NUI Testing

To generate data more like that provided by a user, stored user interactions can be exploited. The Microsoft Kinect was an infrared camera that enabled interactions with applications through full body tracking, providing information such as location of each joint in the body. The Kinect is now discontinued, but the technology exists inside devices such as the HoloLens and Windows Hello biometric facial ID systems. Hunt et al. [65] worked on automatically testing a web browser with Kinect support. Different methods of generating test data based on real user data were compared. The first approach was purely random testing: random coordinates for each joint. The second technique was using random snapshots of clustered user data. The third approach included temporal data, using a Markov chain to select the next cluster to seed. When comparing the first purely random approach to selecting snapshots of training data, Hunt et al. found that selecting snapshots gave a coverage increase over purely random. Further, it was found that using the Markov chain to include temporal information in data generation gave a further increase in coverage over snapshots.

2.5 Summary

Software testing is a vital part of the software development life cycle. This chapter has outlined the main practices of software testing including white and black box testing, unit testing, system testing, and the oracle problem. Many tools and techniques exist that can automatically generate tests or test data for applications simulating different types of user interfaces. There are many approaches to generating test data,

varying from exploiting available user data, using a model of expected behaviour of an application, or statically analysing a program and solving constraints.

It may be beneficial to test an application in a similar way to how a user would interact with an application. Tests can be generated that interact with the fully built system using the same input techniques that a real user would have. This type of test can find bugs missed during lower level testing techniques, like inter-component interaction.

There is lack of work in interacting with an application directly through a user interface. Current techniques rely on prior assumptions about framework usage for user interactions or exploit an accessibility API to interact with a GUI. Consequently, random testing is used as a fall back to automatically detect crashes in applications. However, random testing has disadvantages, often only achieving coverage on lines of code that are “easy” to execute, missing out complex branching conditions and edge cases.

In the next chapter, we will extend the use of image processing in GUI applications, presenting a technique with uses only the information available to users. This information consists of only the visual information of the application (i.e., a screen shot) and points of interaction need to be identified.

3 Guiding Testing through Detected Widgets from Application Screen Shots

This chapter is based on the work “Improving Random GUI Testing with Image-Based Widget Detection” published *In the Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019 [146]*.

3.1	Introduction	61
3.2	Automatic Detection of GUI Widgets for Test Generation	64
3.3	Evaluation	79
3.4	Discussion	100
3.5	Conclusions	106

3.1 Introduction

A Graphical User Interface (GUI) enables events to be triggered in an application through visual entities called widgets (e.g., buttons). Users

interact using keyboard and mouse with the widgets within a GUI to fire events in the application. Automated GUI test generation tools (e.g., AutoBlackTest [92], Sapienz [89], or GUITAR [95]) simulate users by interacting with the widgets of a GUI, and they are increasingly applied to test mobile and desktop applications. The effectiveness of these GUI test generation tools depends on the information they have available. A naïve GUI test generator simply clicks on random screen positions. However, if a GUI test generator knows the locations and types of widgets on the current application screen, then it can make better informed choices about where to target interactions with the program under test.

GUI test generation tools tend to retrieve information about available GUI widgets through the APIs of the GUI library of the target application, or an accessibility API of the operating system. However, relying on these APIs has drawbacks: applications can be written using many different GUI libraries and widget sets, each providing a different API to access widget information. Although widget information can be retrieved by accessibility APIs, these differ between operating systems, and updates to an operating system can remove or replace parts of the API. Furthermore, some applications may not even be supported by such APIs, such as those which draw directly to the screen, e.g., web canvasses [10]. These challenges make it difficult to produce and to maintain testing tools that rely on GUI information. Without knowledge of the type and location of widgets in a GUI, test generation tools resort to blindly interacting with random screen locations.

To relieve GUI testing tools of the dependency on GUI and accessibility APIs, in this chapter we explore the use of machine learning techniques in order to identify GUI widgets. A machine learning system

is trained to detect the widget types and positions on the screen, and this information is fed to a test generator which can then make more informed choices about how to interact with a program under test. However, generating a widget prediction system is non-trivial: Different GUI libraries and operating systems use different visual appearance of widgets. Even worse, GUIs can often be customized with user-defined themes, or assistive techniques such as a high/low contrast graphical mode. In order to overcome this challenge, we randomly generate Java Swing GUIs, which can be annotated automatically, as training data. We explore the challenge of generating a balanced dataset that resembles GUIs in real applications. The final machine learning system uses only visual data and can identify widgets in a real application's GUI without needing additional information from an operating system or API.

In detail, the contributions of this chapter are as follows:

- We describe a technique to automatically generate labelled GUIs in large quantities, in order to serve as training data for a GUI widget prediction system.
- We describe a technique based on deep learning that adapts machine learning object detection algorithms to the problem of GUI widget detection.
- We propose an improved random GUI testing approach that relies on no external GUI APIs, and instead selects GUI interactions based on a widget prediction system.
- We empirically investigate the effects of using GUI widget prediction on random GUI testing.

In our experiments, for 18 out of 20 Java open source applications tested, a random tester guided by predicted widget locations achieved a significantly higher branch coverage than a random tester without guidance, with an average coverage increase of 42.5%. Although our experiments demonstrate that the use of an API that provides the true widget details can lead to even higher coverage, such APIs are not always available. In contrast, our widget prediction library requires nothing but a screen shot of the application, and even works across different operating systems.

3.2 Automatic Detection of GUI Widgets for Test Generation

Interacting with applications through a GUI involves triggering events in the application with mouse clicks or key presses. Lo et al. [87] define three type of widget which appear in GUIs:

- Static widgets in a GUI are generally labels or tooltips.
- Action widgets fire internal events in an application when interacted with (e.g. buttons).
- Data widgets are used to store data (e.g., text fields).

Static widgets do not contribute towards events and interactions, often only providing context for other widgets in the GUI. We focus on identifying only action and data widgets. One widgets have been identified, interactions can be automatically generated to simulate a user using an application's GUI. The simplest approach to generating GUI tests is through clicking on random places in the GUI window [44], hoping to

3.2 Automatic Detection of GUI Widgets for Test Generation

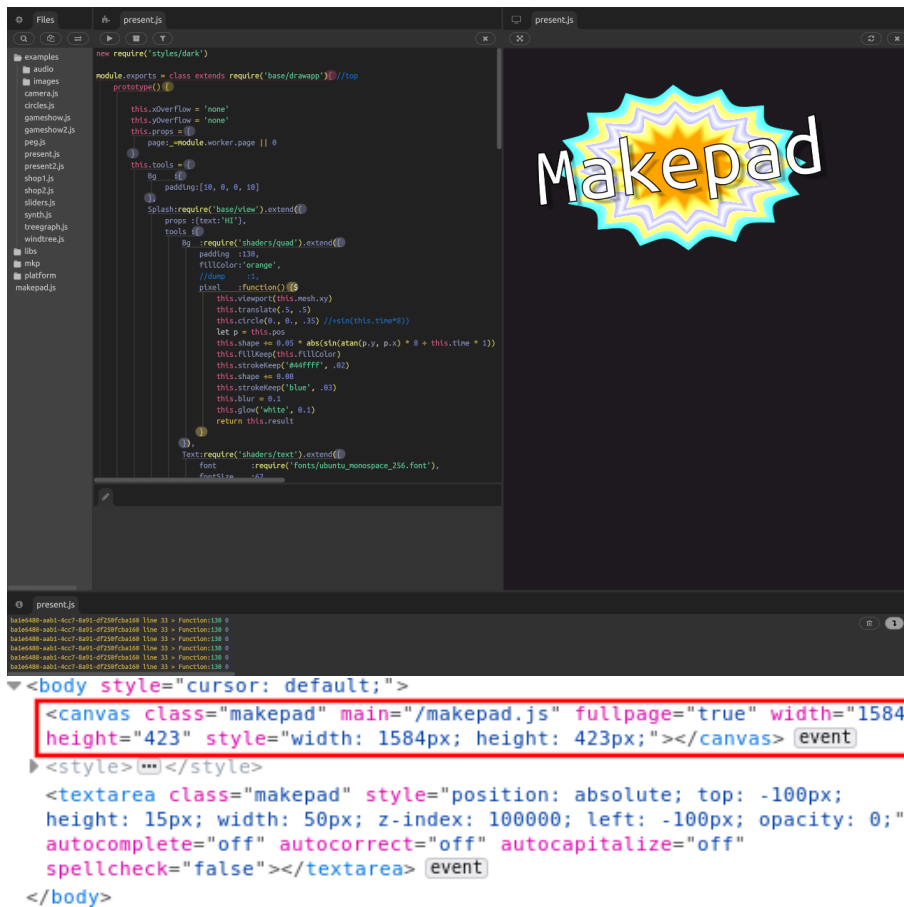


Figure 3.1. The web canvas application MakePad (<https://makepad.github.io/>) and its corresponding DOM. The highlighted “canvas” has no children, hence widget information cannot be extracted.

hit widgets by chance (e.g., Android Monkey [37]). This form of testing (“monkey testing”) is effective at finding crashes in applications and is cheap to run; no information is needed (although knowing the position and dimensions of the application on the screen is helpful).

GUI test generation tools can be made more efficient by providing them with information about the available widgets and events. This information can be retrieved using the GUI libraries underlying the widgets used in an application, or through the operating system’s accessibility API. For example, Bauersfeld and Vos created GUITest [14] (now

known as TESTAR), which uses the operating system's accessibility API to identify possible GUI widgets to interact with, and Mariani et al. present AutoBlackTest [92], which relies on a commercial testing tool (IBM Rational Functional Tester) to retrieve widget information. GUI ripping could also be used [95] to identify *all* GUI widgets in an application to permit systematic test generation. GUI ripping enables effective testing and flexible support for automation, but there are drawbacks [103]. GUI trees have to be manually validated, and component identification issues can lead to inaccurate GUI trees being generated.

Current approaches to testing an application through its GUI rely on an automated method of extracting widget information from a GUI. Applications and application scenarios exist where widget information cannot be automatically derived (as seen in Figure 3.1), and tools may fall back to random testing. However, object detection and image labelling may be able to help with this.

In order to improve random GUI testing, we aim to identify widgets in screen shots using machine learning techniques. A challenge lies in retrieving a sufficiently large labelled training dataset to enable modern object recognition approaches to be applied. We produce this data by (1) generating random Java Swing GUIs, and (2) labelling screen shots of these applications with widget data retrieved through GUI ripping based on the Java Swing API. The trained network can then predict the location and dimensions of widgets from screen shots during test generation, and thus influence where and how the GUI tester interacts with the application.

3.2.1 Identifying GUI Widgets

Environmental factors such as the operating system, user-defined theme, or choice of application designer effect the appearance of widgets. Each application can use a unique widget palette. When widget information cannot be extracted through use of external tools, e.g., an accessibility API, then this diversity of widgets presents a problem for GUI testing tools. For example, applications that render GUIs directly to an image buffer (e.g., web canvas applications) generally cannot have their GUI structure extracted automatically. Pixels are drawn directly to the screen and there is no underlying XML or HTML structure to extract widget locations. We propose a technique of identifying GUI widgets solely through visual information. This is an instance of object detection.

Machine Learning and Object Detection

Machine learning is a field of computer science that aims to produce statistical models which rely on patterns in data to perform specific tasks. One such approach to this is through using a neural network. A neural network is a function that has trainable weights (parameters). By using a labelled set of data, it is possible for the function to predict information about some input and compare it to the corresponding correct label in the training data. Then, the weights can be updated in an attempt to improve the predictions of the network. Seeding all the training data through the network is known as an epoch, and many epochs can be performed to improve the predictions of the network. An area which uses machine learning is object detection.

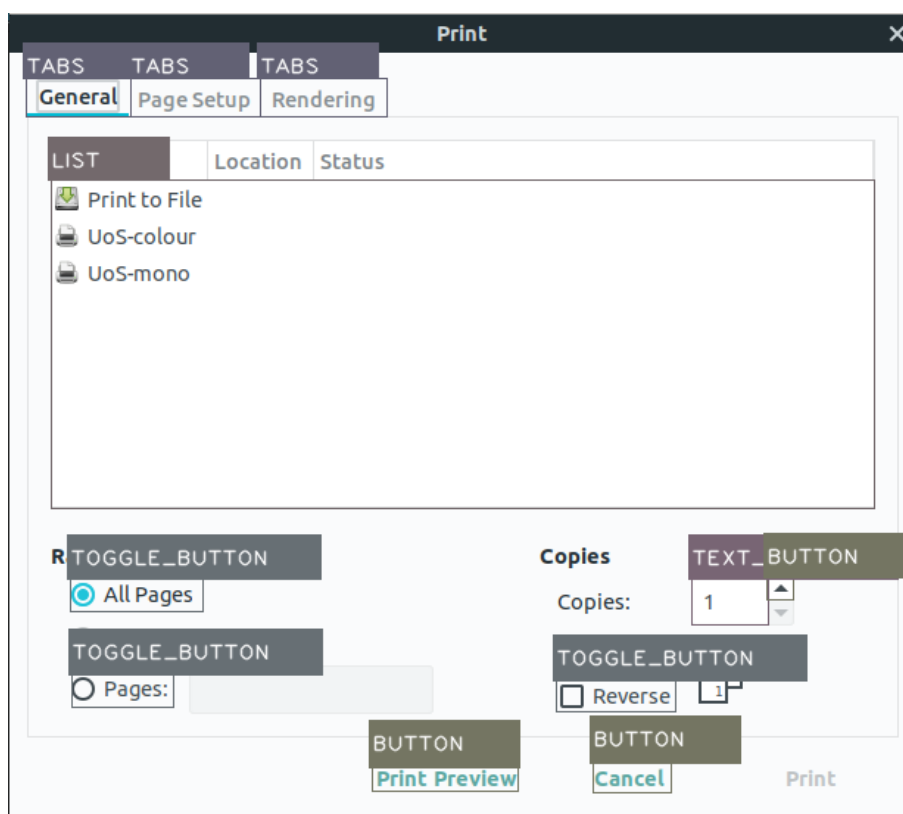


Figure 3.2. Objects in a screen shot from the Ubuntu print application's GUI.

Object detection is the act of identifying objects and their location in an image. This is a popular area of research in machine learning groups, with much attention from Facebook [19], Google [53] and Amazon [3].

By using manually annotated data, it is possible to create a machine learning system that will automatically tag learned classes of objects in new images. Figure 3.2 shows objects we manually annotated in a screen shot of a GUI in the Ubuntu print settings application. Girshick et al. [51] propose the Region-based Convolutional Neural Network (RCNN). A convolutional neural network is similar to other neural networks, but makes the assumption that the input data is an image. With this assumption, the network can have less parameters due to the encoding of properties in an image [5].

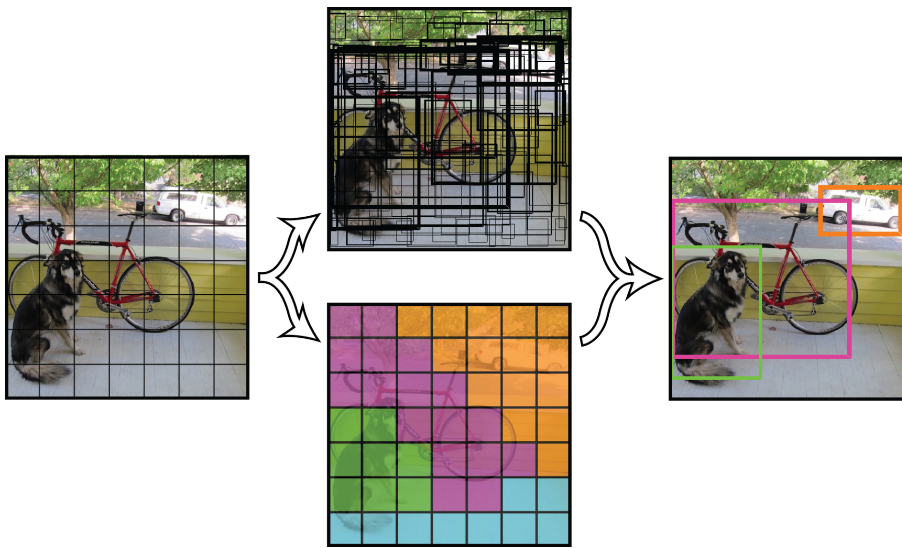


Figure 3.3. YOLO predicting object class and dimensions in an image. (Source: “You Only Look Once: Unified, Real-Time Object Detection” [112])

R-CNN works through a CNN and a sliding window, which captures a small area in the image. Using a sliding window with a CNN has the consequence of inputting images through the network many times, depending on the size of the image and the step size of the sliding window. CNNs identify patterns in image data and are expensive to train, which is exacerbated by the many inputs of a sliding window. However, there are CNNs that do not rely on a sliding window, instead processing an entire image at a time. This allows quicker processing of images and is beneficial in scenarios with limited processing time.

Redmon et al. proposed You Only Look Once (YOLO) [112], a network which can be trained for object detection and processes a whole image at once, removing the complications of a sliding window. YOLO predicts the object type (*class*), position and dimension of an object. It also gives each prediction a confidence score.

YOLO takes an image as input, with width and height being a multiple

of 32 pixels. This is because YOLO downsamples the image by a factor of 2 for multiple layers of the network. The output of the network is a $N \times N$ grid of predicted objects. N is equal to the width or height divided by 32. Figure 3.3 shows the raw image and corresponding output from the YOLO network. The image is split using two techniques: the first one predicts the dimensions of the box surrounding possible objects in the image (the top image). The thicker the border of the box, the more confident that YOLO is of the prediction. The second technique predicts the class of any object with centre point inside this grid cell (the bottom image). Each colour represents a different class having the highest probability. Then, these two datasets are merged, and only boxes with a predicted confidence above a certain threshold (i.e., boxes with a thick border in the top image) are output as a prediction. The centre point of these boxes is then used to assign the box a class, depending on the class with the greatest predicted value from the bottom image.

The original YOLO algorithm was extended by Redmon and Farhadi, resulting in YOLOv2 [113]. YOLOv2 predicts B boxes per grid cell, resulting in the final number of predictions being $N \times N \times B$. YOLOv2 has been used to train many object detection systems, such as Chinese traffic sign recognition [154] and ship detection [31].

During GUI testing, it is beneficial to recognise widgets quickly. By processing screen shots at a faster pace, more actions can be generated when giving the same time constrain for test generation (i.e., the same testing budget). Therefore, we use You Only Look Once (YOLOv2), proposed by Redmon et al. [112], which labels an image by seeding the whole image through a CNN once. YOLO is capable of predicting the position, dimension, and class of objects in an image. YOLOv2 [113] is

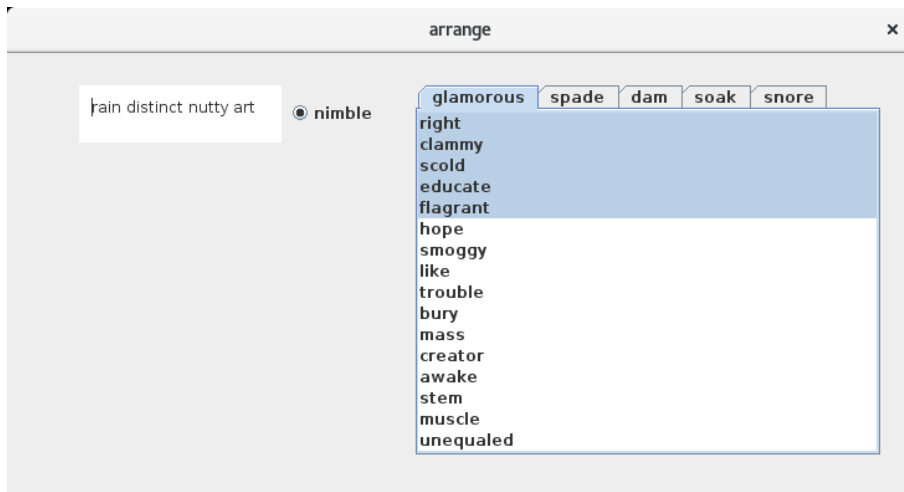


Figure 3.4. A generated Java Swing GUI. Each element has a random chance to appear. (Source: White et al. 2019 [146])

an extension to YOLO. YOLOv2 predicts B boxes per grid cell. Each box contains five predicted values: the location (x, y) , the dimension ($width$, $height$) and a confidence score for the prediction (c). Predicting multiple boxes per grid cell aids in training as it allows different aspect ratios to be used for each box in each cell. YOLOv2 exploits that many objects have similar shaped boxes in an image. For example, a box which is thin but wide may repeatedly appear and YOLOv2 would have a prediction per grid cell representing this common box aspect ratio. The aspect ratios are passed to the algorithm to modify the predicted width and height of each box. To calculate aspect ratios tuned on the objects present in GUIs, the dimensions of all boxes in the training data are clustered into N clusters, where N is the number of boxes predicted per cell. The centroid of each cluster represents the aspect ratios to supply to YOLOv2.

A single class is predicted from C predefined classes for each grid cell. In total, this makes the size of the network's output $N \cdot N \cdot (B \cdot 5 + C)$. We can now filter the predicted boxes using the confidence values.

Algorithm 3.2.1: RANDOMWIDGETTREE(*nodeCount*)

```
nodes = [Container]
while |nodes| < nodeCount
  do {nodes ← nodes ∪ RANDOMWIDGETTYPE()}
  while (|nodes| > 1)
    do {
      node ← sample(nodes, 1)
      parent ← sample(nodes, 1)
      if isContainer(parent) and node ≠ parent
        then {
          parent.children ← parent.children ∪ node
          nodes ← nodes \ node
        }
    }
return (nodes[0])
```

Boxes with a confidence value close to zero may not be worth investigating, and can be eliminated using a lower confidence threshold. This will be discussed further in section 3.2.3.

Using the YOLOv2 convolutional neural network, we can automatically identify GUI components in a screen shot. We chose the YOLOv2 algorithm for our network due to the speed at testing it can process entire images, and the accuracy it achieves on predictions. Our implementation of YOLOv2 only uses greyscale images, so the first layer of the network only has a single input per pixel opposed to the three (*r,g,b*) values proposed in the original network [113]. This decision was firstly as a preprocessing step for GUIs. GUIs often have clear edges between widgets to aid in the user experience when using an application. Therefore, a greyscale image should suffice when identifying widgets. Secondly, neural networks can over tune themselves to certain inputs. By converting to greyscale, we eliminate two thirds of the input data and therefore reduce the possibility of this occurring.

Algorithm 3.2.2: RANDOMJFRAME(*width, height, nodeCount*)

```

procedure ApplyWidget(container, widget)
  swingComponent ← COMPONENTFROMWIDGET(widget)
  i ← 0
  while i < | widget.children |
    do { child ← widget.children[0]
        APPLYWIDGET(swingComponent, child)
        i ← i + 1
      }
  container.add(swingComponent)

jframe ← JFrame<INIT>(width, height)
rootNode ← RANDOMWIDGETTREE(nodeCount)
APPLYWIDGET(jframe, rootNode)
return (jframe)

```



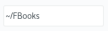

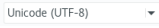

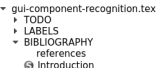

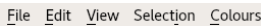

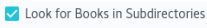
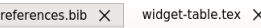
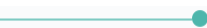
Figure 3.5. Generated GUIs by two techniques: random widget selection and placement (left), and random GUI tree generation (right).

3.2.2 Generating Synthetic GUIs

One issue with using a neural network is that it requires large amounts of labelled data for training. To obtain labelled screen shots, we generate synthetic applications. A synthetic application is one with no event handlers, containing only a single screen with random placements of widgets. Generating GUIs allows precise control over the

Chapter 3 Guiding Testing through Detected Widgets from Application Screen Shots

Table 3.1. Widgets that the model can identify in a Graphical User Interface

Widget	Description	Example
Text Field	Allows input from the keyboard to be stored in the GUI to be used later.	
Button	Allows an event to be triggered by clicking with the mouse.	
Combo Box	Allows selection of predefined values. Clicking either inside the box or the attached button opens predefined options for users to select.	
List	Similar to a Combo Box, allows selection of predefined values. Values are present at all times. Scrolling may be needed to reveal more.	
Tree	Similar to a list but values are stored in a tree structure. Clicking a node may reveal more values if the node has hidden child elements.	
Scroll Bar	A horizontal or vertical bar used for scrolling with the mouse to reveal more of the screen.	
Menu	A set of usually textual buttons across the top of a GUI	
Menu Item	An individual button in a menu. Clicking usually expands the menu revealing more interactable widgets.	
Toggle Button	Buttons that have two states toggled by clicking on them.	
Tabs	Buttons changing the contents in all or part of the GUI when activated.	
Slider	A button that can be click-and-dragged in a certain axis, changing a value which is usually a numeric scale, e.g., volume of an application.	

training data, such as ensuring that the generated dataset contains a balanced number of each widget, mitigating against the “class imbalance” problem which can negatively effect trained networks by countering the machine learning assumption of an equal misclassification cost for all classes [82]. An example generated GUI can be seen in Figure 3.4. We use 11 standard types of widgets in generated GUIs, which are shown in Table 3.1.

To generate synthetic applications, we use the Java Swing GUI framework. Initial attempts at generating GUIs by entirely random selection and placement of widgets yielded GUIs that were not representative of ones encountered in real applications. The GUIs were unstructured, with uneven spacing. GUIs could also have far too many, or very few

widgets, creating packed or sparse GUIs. Real GUIs have structure, and spacing is used to aid in widget separation and identification. Consequently, the resulting prediction system performed poorly on real GUIs. To create more realistic GUIs as training data, our approach therefore generates an abstract tree beforehand, and uses this tree as a basis, assigning widgets to each node or leaf before generating the GUI. Figure 3.5 shows the difference in generated GUIs. With the previous, random widget selection and placement technique, the layout manager is responsible for organising the generated GUI and certain areas can become crowded. If we select a widget which can contain children (e.g., a list), the same random technique would be called recursively (with a hard coded maximum depth to stop infinite looping and unrealistically large/populated GUIs). Certain types of widgets that are children of the root container are also placed on their own, making most GUIs fairly sparse. Generating an abstract tree and deriving the GUIs from this tree creates much more structured, realistic GUIs. The possibility of looping indefinitely is also removed by generating the tree in advance, as we predetermine the quantity of nodes that will be present in the tree, removing the recursive call. By extension, the quantity of widgets is also predetermined in the GUI that will be generated from the tree.

First, we randomly choose a Swing layout manager. A layout manager controls how widgets appear on the GUI. One example layout manager is a grid layout, which divides the space in the GUI into an $I \cdot Y$ grid. Widgets can be placed in each grid cell. With the layout manager chosen, we then generate a random tree where each node represents a GUI widget. Only widgets which can contain other widgets can be assigned child nodes in the tree, for example, a tab pane can have children representing other GUI widgets assigned to it, but a button cannot. Algorithm 3.2.1 shows how a random abstract tree of GUI widget types

Chapter 3 Guiding Testing through Detected Widgets from Application Screen Shots

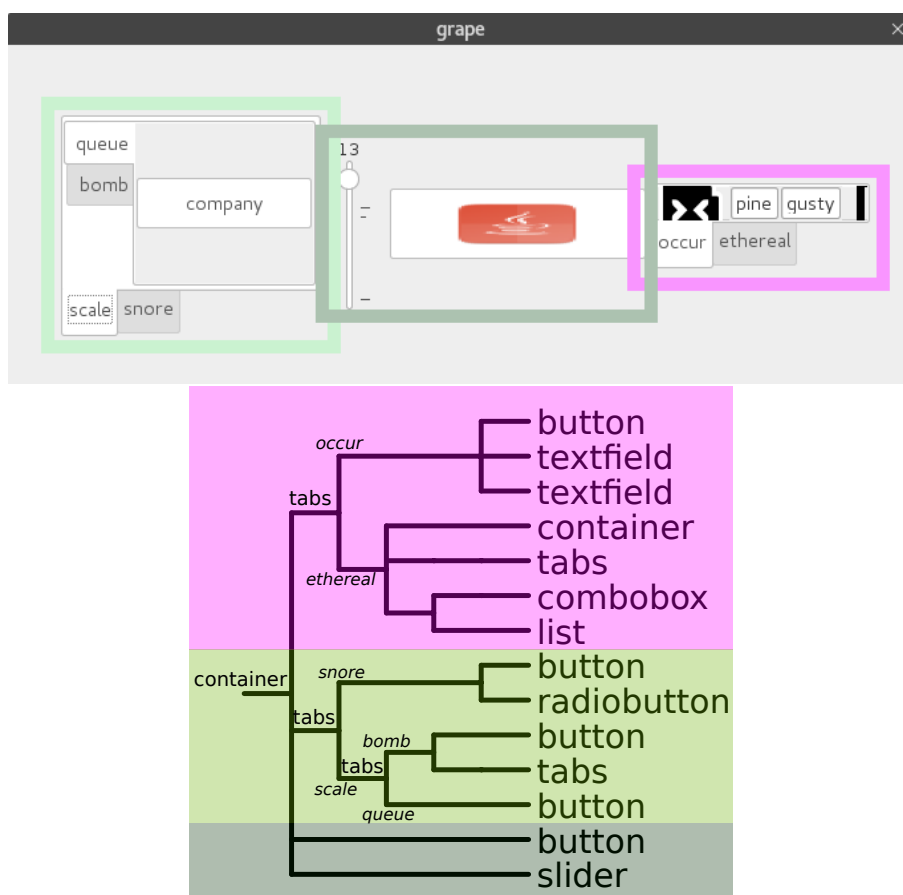


Figure 3.6. A generated GUI tree and the corresponding GUI derived from the tree.

is generated. Here, the nodes list initially contains a “Container” widget type which is to eliminate an infinite loop later if the *RandomWidgetType()* function returns no containers. The call to *RandomWidgetType* randomly returns one of the 11 types of widgets. Each widget has the same probability of appearing but we found that some GUI widgets are constructed of others when using Java Swing, e.g., a Combo Box also contains a button, and a scroll bar contains two buttons, one at each end. When generating data, we have a unique advantage that we can balanced the number of widgets in GUIs, evenly distributing the widget types through the generated dataset. We found that *menu_items*

has a dependency with a *menu*. Weighting *menu_items* with an equal probability to appear forced *menus* to also appear on nearly all generated GUIs. To balance the dataset, we lowered the probability for *menu_items*.

To generate a Swing GUI, Algorithm 3.2.2 walks through the generated tree. Each node is assigned a random position and dimension inside its parent. The position is randomly selected based on the layout manager. For example, with a `GridLayout`, we randomly assign the element in the current node to a random x, y coordinate in the grid. However, with a `FlowLayout`, the position does not matter as all widgets appear side by side in a single line. In algorithm 3.2.2, the *container.add* method call in the *ApplyWidget* procedure is from the `JComponent` class of Java Swing, and the random position is seeded here depending on the current layout manager. Figure 3.6 shows an example generated tree, and the corresponding GUI derived from the tree. An interesting observation about this Figure is that even though only a single tab is selected at any one time, the contents for the other tabs have also been generated. These contents are never seen however, as only one snapshot of the GUI is every taken.

Once a Swing GUI has been generated, Java Swing allows us to automatically extract information for all widgets. This includes the position on the screen, dimension and widget type. This is similar to the approach current GUI testing tools use when interacting with an application during test execution.

3.2.3 A Random Bounding Box Tester

Once widgets can be identified, they may be used to influence a random GUI tester. We created a tester which randomly clicks inside a given bounding box. At the most basic level, a box containing the whole application GUI is provided, and the tester will randomly interact with this box. One of three actions is executed on the selected box: a) left click anywhere inside the given box; b) right click anywhere inside the given box; c) left click anywhere inside the given box and type either a random string (e.g., "Hello World!" in our implementation) or a random number (e.g., between -10000 and 10000 in our implementation). We use these two textual inputs to represent the most common use for text fields: storing a string of characters or storing a number. Our random tester is a version of Android Monkey [37] we implemented that uses conventional GUIs in place of Android ones. Algorithm 3.2.3 shows how the tester can interact with a provided box. In this algorithm, $rand(x, y)$ returns a random number between x and y inclusive. $LeftClick(x, y)$ and $RightClick(x, y)$ represent moving the mouse to position x, y on the screen and either left or right clicking respectively. $KeyboardType(string)$ represents pressing the keys present in $string$ in chronological order.

We can refine the box provided to this random tester using the trained YOLOv2 network. We randomly select a box with a confidence greater than some value C . When seeded to the tester, the tester will click on a random position inside one of the predicted widgets from the network.

Finally, we can provide the tester with a box directly from Java Swing. This implementation currently only supports Java Swing applications but will ensure that the GUI tester is always clicking inside the bound-

Algorithm 3.2.3: RANDOMINTERACTION(*box*)

```

interaction ← rand(0, 2)
x ← box.x + rand(0, box.width)
y ← box.y + rand(0, box.height)
if interaction == 0
  then LEFTCLICK(x, y)
  else if interaction == 1
    then RIGHTCLICK(x, y)
    else if interaction == 2
      then {
        LEFTCLICK(x, y)
        inputType ← rand(0, 1)
        inputString ← ""
        if inputType == 0
          then inputString ← "Hello World!"
        else {
          inputNumber ← rand(-10000, 10000)
          inputString ← inputNumber.toString()
        }
        KEYBOARDTYPE(inputString)
      }

```

ing box of a known widget currently on the screen. Our tool *GUIDance* is open-source and can be found and contributed to on GitHub¹.

3.3 Evaluation

To evaluate the effectiveness of our approach when automatically testing GUIs, we investigate the following research questions:

RQ3.1 How accurate is a machine learning system trained on synthetic GUIs when identifying widgets in GUIs from real applications?

RQ3.2 How accurate is a machine learning system trained on synthetic GUIs when identifying widgets in GUIs from other operating system and widget palettes?

¹<https://github.com/thomasdeanwhite/GUIDance>

RQ3.3 What benefit does random testing receive when guided by predicted locations of GUI widgets from screen shots?

RQ3.4 How close can a random tester guided by predicted widget locations come to an automated tester guided by the exact positions of widgets in a GUI?

3.3.1 Widget Prediction System Training

In order to create the prediction system, we created synthetic GUIs on Ubuntu 18.04, and to capture different GUI styles, we used different operating system themes. We generated 10,000 GUI applications per theme and used six light themes: the default Java Swing theme, *adapta*, *adwaita*, *arc*, *greybird*; two dark themes: *adwaita-dark*, *arc-dark*, and two high-contrast themes which are default with Ubuntu 18.04. These are all popular themes for Ubuntu and were chosen so that the pixel histograms of generated GUI images were similar to that of real GUI images.

In total this resulted in 100,000 synthetic GUIs, which we split as follows: 80% of data was used as training data, 10% as validation data, and 10% as testing data. To train a machine learning system using this data, the screen shots are fed through the YOLOv2 network and the predicted boxes from the network are compared against the actual boxes retrieved from Java Swing. If there is a difference, the weights of the network are updated using gradient descent to improve the predictions next epoch.

It is important to have a validation dataset to determine whether the network is over-fitting on the training data. This can be done by checking the training progress of the network against the training and valida-

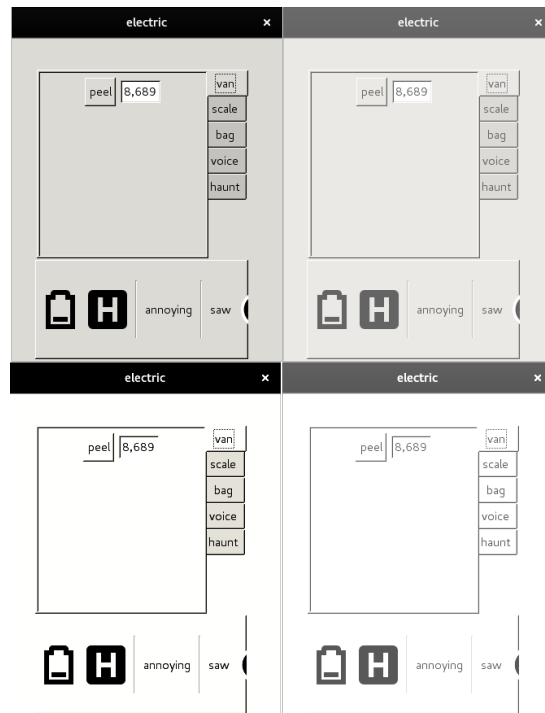


Figure 3.7. Data inflation transformations. The right images contain a brightness increase; the bottom images contain a contrast increase. The top left image is the original screen shot.

tion dataset. During training, the network is only exposed to the training dataset, so if the network is improving when evaluated against the training dataset, but not improving on the validation dataset, the network is over-fitting.

With the isolated training data, we trained a network which uses the YOLOv2 network. It has been observed that artificial data inflation (augmentation) increases the performance of neural networks, exposing the network to more varied data during training [122, 39]. During training, we artificially increased the size of input data using two techniques: brightness and contrast adjustment. Before feeding an image into the network, there is a 20% chance to adjust the image. This involves a random shift of 10% in brightness/contrast and applies to

Chapter 3 Guiding Testing through Detected Widgets from Application Screen Shots

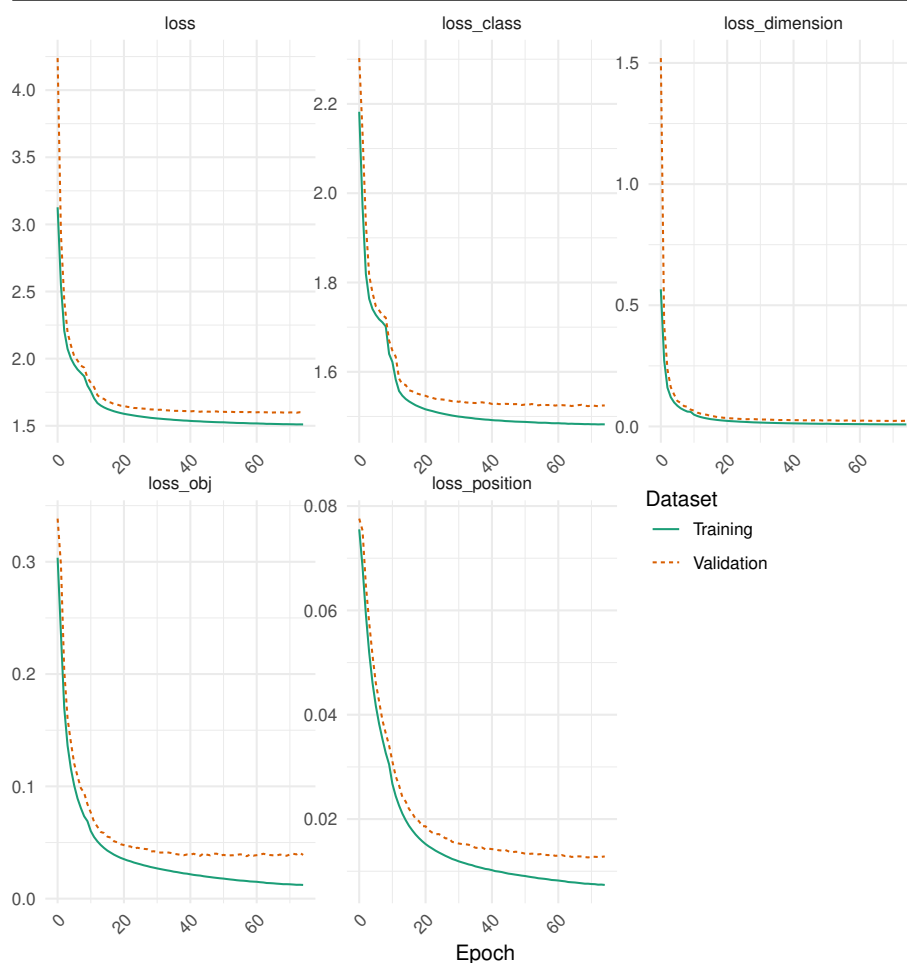


Figure 3.8. The loss values from the YOLOv2 network when training over 100 epochs.

only a single training epoch. For example, an image could be made up to 10% lighter/darker and have the pixel intensity values moved up to 10% closer/further from the median of the image's intensity values. These transformations can be seen in Figure 3.7. The top left image is the original, with images to the right containing an increase in brightness, and images below, an increase in contrast.

YOLOv2 trains by reducing the loss value of five different areas involved in the prediction of the bounding boxes:

- *loss_class*: The loss value associated with incorrect class predictions;
- *loss_dimension*: The loss value associated with inaccurate dimensions of predicted Boxes
- *loss_obj*: The loss value associated with incorrect object detection (i.e., predicting an object in a grid cell when one is not there, or missing an object);
- *loss_position*: The loss value associated with inaccurate predictions of the centre point of a bounding box;
- *loss*: an aggregation of all of above.

In Figure 3.8, the loss value for two datasets can be seen during training of the network. The network performance stagnates somewhere between 40 and 50 epochs on the validation dataset. At this point, the network is overfitting to the training dataset, where performance is still slowly improving. In our experiments, we used the weights from epoch 40, taking the overfitting into account. There are very minor improvements against the validation set after epoch 40, but we use 40 also as an early stopping condition [29].

3.3.2 Experimental Setup

RQ3.1

To evaluate RQ3.1, we compare the performance when predicting GUI widgets in 250 screen shots of unique application states in real applications against performance when predicting widgets in synthetic applications. Screen shots were captured when a new, unseen window

was encountered during real user interaction. 150 of the screen shots were taken from the top 20 Swing applications on SourceForge, and annotated automatically via the Swing API. The remaining 100 screen shots were taken from the top 15 applications on the Ubuntu software centre and manually annotated. The network used to predict widget locations was trained on only synthetic GUIs, and in RQ3.1 we see if the prediction system is able to make predictions for real applications.

YOLOv2 predicts many boxes, which could cause a low precision. To lower the number of boxes predicted, we pruned any predicted boxes below a certain confidence threshold. To tune this confidence threshold, we evaluated different confidence values against the synthetic validation dataset. As recall is more important to us than precision, we used the confidence value with the highest F2-measure to compare synthetic against real application screen shots. We found this value C to be 0.1 through parameter tuning on the synthetic *validation* dataset. However, the actual comparison of synthetic data against real GUI data was performed on the isolated *test* dataset, to avoid biases in this value of C being optimal for the validation dataset.

After eliminating predicted boxes with a confidence value less than C , we can compare the remaining boxes with the actual boxes of a GUI. In order to assess whether a predicted box correctly matches with an actual box, we match boxes based on the intersection over union metric.

Intersection-over-union (IoU) calculates the similarity of two boxes in two dimensional space [111]. We calculate the IoU between the predicted boxes from the YOLOv2 network, and actual boxes in the labelled test dataset. The IoU of two boxes is the area that the boxes intersect, divided by the union of both areas. An IoU value of one indicates that the boxes are identical, and an IoU of 0 indicates the boxes

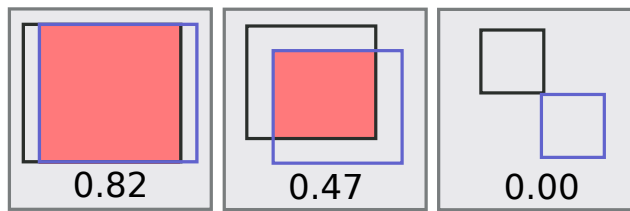


Figure 3.9. Intersection over Union (IoU) values for various overlapping boxes.

have no area of overlap. See Figure 3.9 for an example of IoU values for overlapping boxes. The shaded area indicates overlap between both boxes. We consider a predicted box to be matched with an actual box when the IoU value is greater than 0.3.

RQ3.2

To evaluate RQ3.2, we use the same principle as in RQ3.1, however, the comparison datasets are the synthetic test data-set and a set of manually annotated screen shots taken from the applications in the Apple store. We gathered 50 screen shots of unique application states, five per application of the top 10 free applications on the store as of 19th January 2019. Again, each screen shot was taken when a new, previously unknown window appeared during real user interaction. The screen shots were manually annotated with the truth boxes for all widgets present.

RQ3.3

To evaluate RQ3.3, we compare the branch coverage of tests generated by a random tester to tests where interactions are guided by predicted bounding boxes. The subjects under test are 20 Java Swing applications,

including the top six Java Swing applications from SourceForge and the remaining ones from the SF110 corpus by Fraser and Arcuri [46]. Table 3.2 shows more information about each application. We limited the random tester to 1000 actions, and conservatively performed a single action per second. Adding a delay before interactions is common in GUI testing tools, and using too little delay can produce flaky tests or tests with a high entropy [48]. Because of the delay, all techniques had a similar runtime. On a crash or application exit, the application under test was restarted. Each technique was applied 30 times on each of the applications, and a Mann-Whitney U-Test was used to test for significance. Although all the applications use Java Swing, this was to aid conducting experiments when measuring branch coverage and allow retrieval of the positions of widgets currently on the screen from the Java Swing API for RQ3.4. Our approach should work on many kinds of applications using any operating system.

RQ3.4

To answer RQ3.4, we compare the branch coverage of tests generated by a tester guided by predicted bounding boxes, to one guided by the known locations of widgets retrieved from the Java Swing API. The API approach is similar to current GUI testing tools, which exploit the known structure of a GUI to interact with widgets. We use the same applications as RQ3.3. We allowed each tester to execute 1000 actions over 1000 seconds. On a crash or application exit, the application under test is restarted. Each technique ran on each application for 30 iterations.

Table 3.2. The applications tested when comparing the three testing techniques.

Application	Description	LOC	Branches
Address Book	Contact recorder	363	83
bellmanzadeh	Fuzzy decision maker	1768	450
BibTex Manager	Reference Manager	804	309
BlackJack	Casino card game	771	178
Dietetics	BMI calculator	471	188
DirViewerDU	View directories and size	219	90
JabRef	Reference Manager	60620	23755
Java Fabled Lands	RPG game	16138	9263
Minesweeper	Puzzle game	388	155
Mobile Atlas Creator	Create offline atlases	20001	5818
Movie Catalog	Movie journal	702	183
ordrumbox	Create mp3 songs	31828	6064
portecle	Keystore manager	7878	2543
QRCode Generator	Create QR codes for links	679	100
Remember Password	Save account details	296	44
Scientific Calculator	Advanced maths calculator	264	62
Shopping List Manager	List creator	378	62
Simple Calculator	Basic maths calculator	305	110
SQuiz	Load and answer quizzes	415	146
UPM	Save account details	2302	530

3.3.3 Threats to Validity

There is a chance that our object detection network over-trains on the training and validation synthetic GUI dataset and therefore achieves an unrealistically high precision and recall on these datasets. To counteract this, we use the third test dataset when calculating precision and recall values for the synthetic dataset which has been completely isolated from the training procedure.

To ensure that our real GUI screen shot corpus represents general applications, the Swing screen shots were from the top applications on SourceForge, the top rated applications on the Ubuntu software centre, and the top free applications from the Apple Store.

In object detection, usually an IoU value of 0.5 or more is used for a

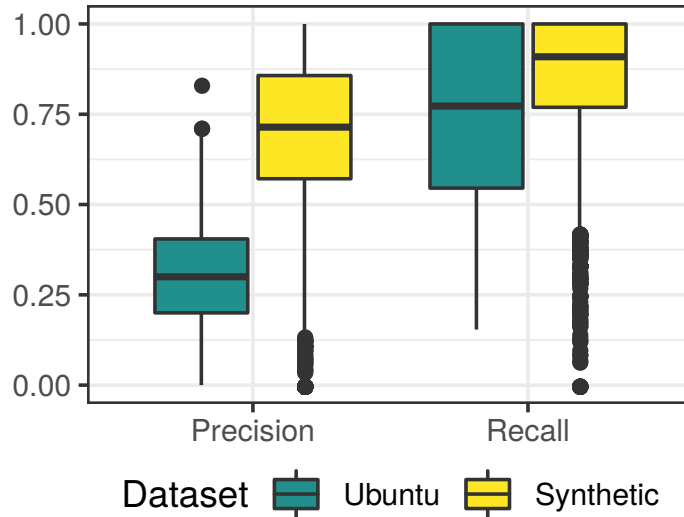


Figure 3.10. Precision and recall of synthetic data against real GUIs from Ubuntu/Java Swing applications.



Figure 3.11. Manually annotated (a) and predicted (b) boxes on the Ubuntu application “Hedge Wars”.

predicted box to be considered a true positive (a “match”). However, we use an IoU threshold of 0.3 as the predicted box does not have to exactly match the actual GUI widget box, but it needs enough overlap to enable interaction. Russakovsky et al. [121] found that training humans to differentiate between bounding boxes with an IoU value of 0.3

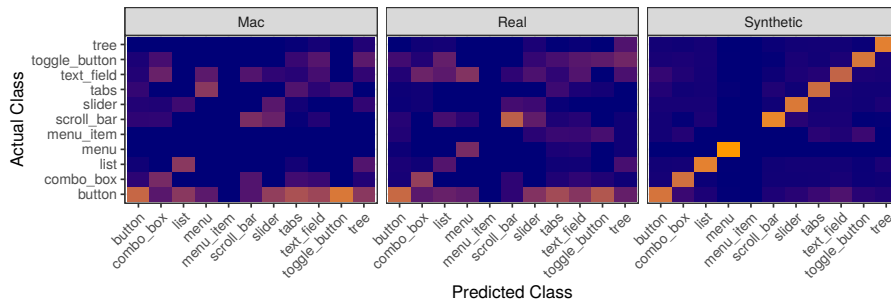


Figure 3.12. Confusion matrix for class predictions.

or 0.5 is challenging, so we chose the lower threshold of 0.3.

As the GUI tester uses randomized processes, we ran all configurations on all applications for 30 iterations. We used a two-tailed Mann-Whitney U-Test to compare each technique and a Vargha-Delaney A_{12} affect size to find the technique likely to perform best.

3.3.4 Results

RQ3.1: How accurate is a machine learning system trained on synthetic data when detecting widgets in real GUIs?

Figure 3.10 shows the precision and recall achieved by a network trained on synthetic data. We can see that predicting widgets on screen shots of Ubuntu and Java Swing GUIs achieves a lower precision and recall than on synthetic GUIs. However, the bounding boxes of most widgets have a corresponding predicted box with an IoU > 0.3 , as shown by a high recall value. A low precision but high recall indicates that we are predicting too many widgets in each GUI screen shot. Figure 3.11a shows an example of a manually annotated image, and Figure 3.11b shows the same screen shot but with predicted widget boxes.

The precision and recall values only show if a predicted box aligns with an actual box. Figure 3.12 shows the confusion matrix for class predictions. An orange (light) square indicates a high proportion of predictions, and blue (dark) square a low proportion. We can see that for synthetic applications, most class predictions are correct. However, the prediction system struggles to identify *menu_items* and this is most likely due to the lower probability of them appearing in synthesized GUIs. The network would rather classify them as a button which appears much more commonly through all synthesized GUIs. However, as a menu item and a button has the same functionality (i.e., when clicked, performs some action), the interaction generated will be equivalent regardless of this misclassification. From the confusion matrix, another problem for classification seems to be buttons. Buttons are varied in shape, size and foreground. For example, a button can be a single image, a hyper-link, or text surrounded by a border. Subtle modifications to a widget can change how a user perceives the widget's class, but are much harder to detect automatically.

While this shows that there is room for improvement of the prediction system, these improvements are not strictly necessary for the random tester as described in Section 3.2.3, since it interacts with all widgets in the same manner irrespective of the predicted type. Hence, predicting the correct class for a widget is not as important as identifying the actual location of a widget, which our approach achieves. However, future improvements of the test generation approach may rely more on the class prediction and handling unique classes differently may be beneficial.

RQ3.1: In our experiments, widgets in real applications were detected with an average recall of 77%.

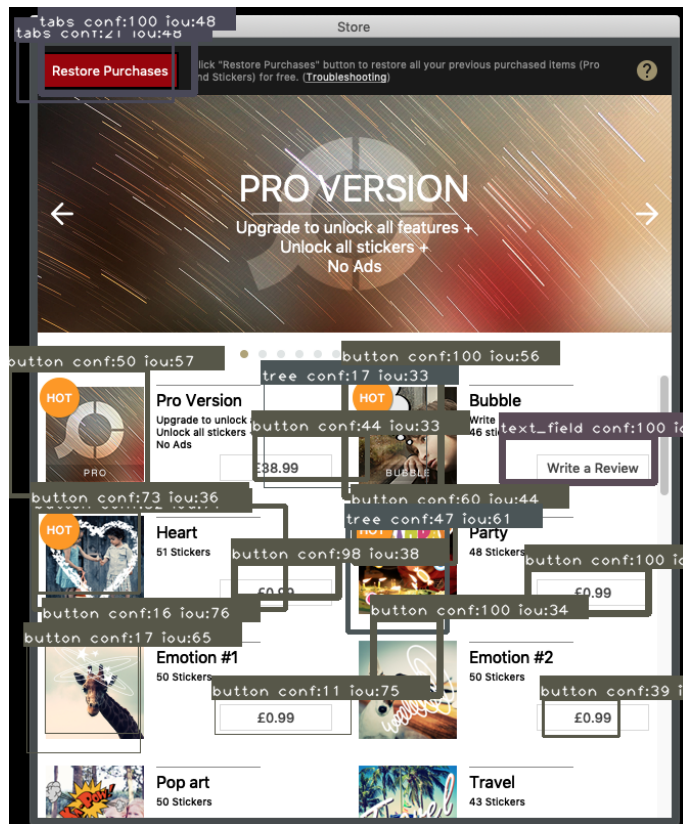


Figure 3.13. Predicted bounding boxes on the OSX application “Photoscape X”.

RQ3.2: How accurate is a machine learning system trained on synthetic data when detecting widgets on a different operating system?

To investigate whether widgets can be detected in other operating systems with a different widget palette, we apply a similar approach to RQ3.1 and use the same IoU metric, but evaluated on screen shots taken on a different operating system and from different applications.

Figure 3.14 shows the precision and recall achieved by the prediction system trained on synthetic GUI screen shots. We again see a lower precision and recall on OSX (Mac) GUI screen shots compared to synthetic

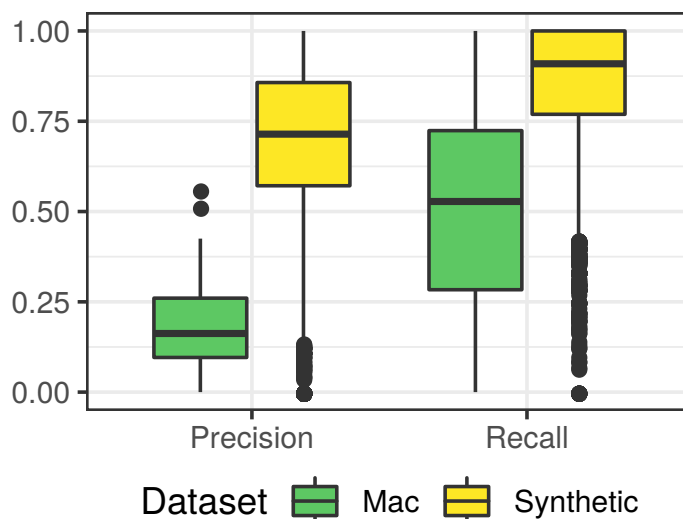


Figure 3.14. Precision and recall of synthetic data against real GUIs from Mac OSX applications.

GUIs, but we still match over 50% of all boxes against predicted boxes with an IoU > 0.3.

A lower precision indicates many false positive predictions when using the OSX theme in applications. An observable difference between predictions on OSX and on Ubuntu is that our machine learning system has greater difficulty in predicting correct dimensions for bounding boxes on OSX. See Figure 3.13 for correctly predicted boxes in the OSX application “Photoscape X”.

One observation of applications using OSX is that none use a traditional menu (e.g. File, Edit, etc.). OSX applications instead opt for a toolbar of icons that function similar to tabs. Our prediction system could be improved by including this data in the training stage.

For the purposes of testing, an exact match of bounding boxes is less relevant so long as the generated interaction happens somewhere within

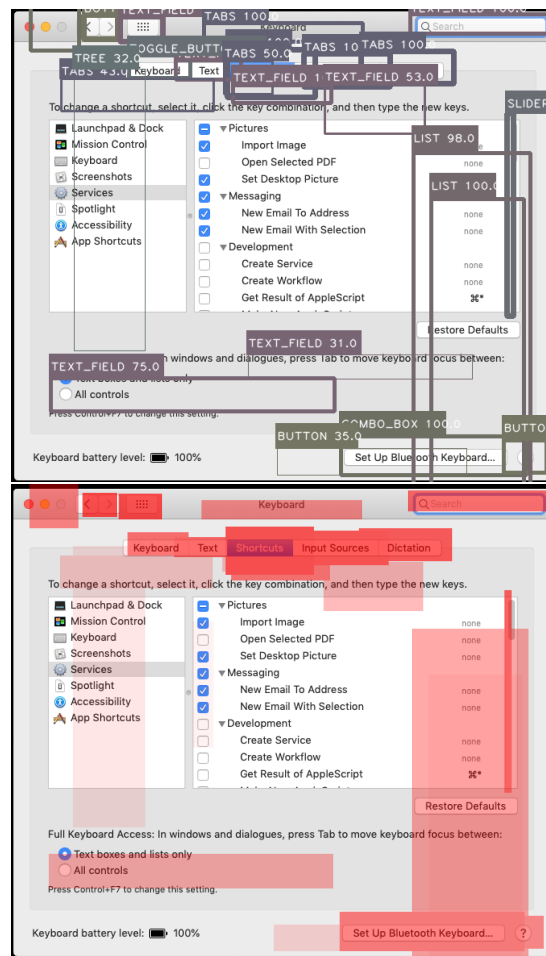


Figure 3.15. Predicted widget boxes and the corresponding heatmap of predicted box confidence values. Darker areas indicate a higher confidence of useful interaction locations, derived from the object detection system. (Source: self)

the bounding box of the actual widget. For example, if a predicted box is smaller than the actual bounding box of a widget, interacting with any point in the predicted box will trigger an event for the corresponding widget. IoU does not take this into account, and it is possible that a box will be flagged as a false positive if $\text{IoU} < 0.3$ but the predicted box is entirely within the bounds of the actual box. In this case, the predicted box would still be beneficial in guiding a test generator. Figure 3.15 shows predicted boxes on an OSX application, and the correspond-

ing heatmap by plotting the confidence values of each box. It is clear from this image that the predictions can be used to interact with many widgets in the GUI.

RQ3.2: GUI widgets can be identified in different operating systems using a widget prediction system trained on widgets with a different theme, achieving an average recall of 52%.

RQ3.3: What benefit does random testing receive when guided by predicted locations of GUI widgets from screen shots?

Figure 3.16 shows the branch coverage achieved by the random tester when guided by different techniques. Table 3.3 shows the mean branch coverage for each technique, where a bold value indicates significance. Here we can see that interacting with predicted GUI widgets achieves a significantly higher coverage for 18 of the 20 applications tested against a random testing technique. The A_{12} value indicates the probability of the tester guided by predicted widget locations performing worse than the comparison approach. If $A_{12}=0.5$, then both approaches perform similarly (i.e., the probability of one approach outperforming another is 50%); if $A_{12}<0.5$, the tester guided by predicted widget locations usually achieves a higher coverage. If $A_{12}>0.5$ then the tester guided by predicted widgets would usually achieve a lower coverage. For instance, take Address-book: $p_v(Pred,Rand) < 0.001$ and $A_{12}(Pred,Rand) = 0.032$. This indicates that the testing approach guided by predicted widgets would achieve a significantly higher code coverage than a random approach around 96.8% of the time when testing this application.

Overall, guiding the random tester with predicted widget locations lead to a 42.5% increase in the coverage attained by generated tests.

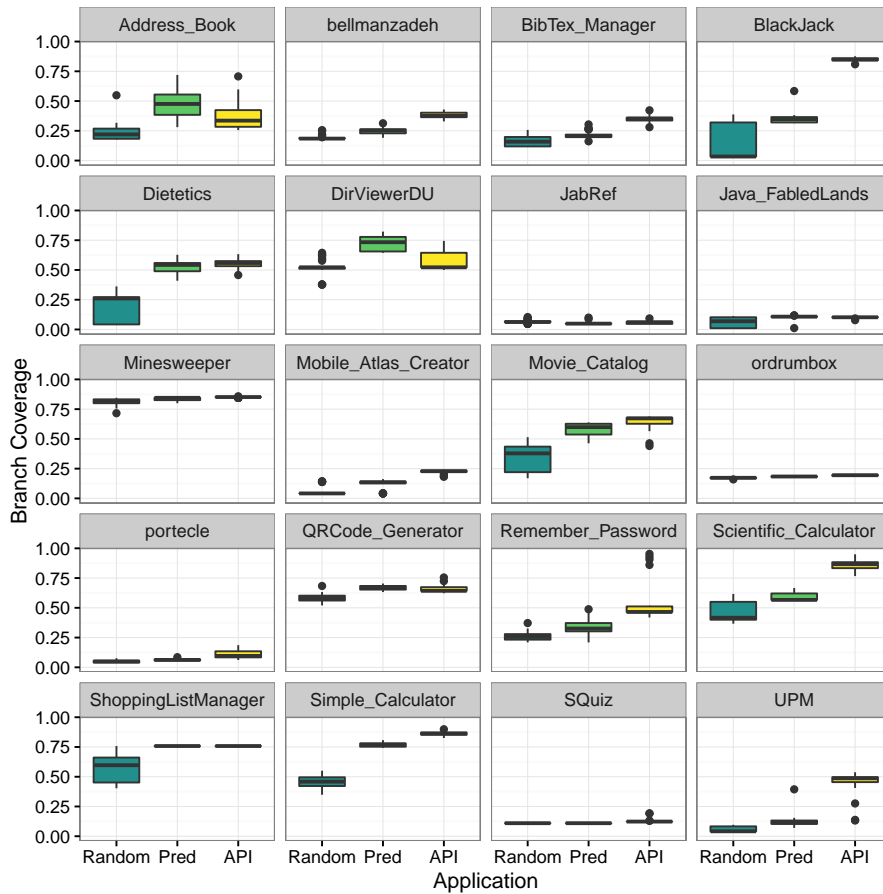


Figure 3.16. Branch Coverage achieved by a random clicker when clicking random coordinates, guided by predicted widgets positions and guided by the Swing API.

We can see that even on applications that use a custom widget set (e.g., `ordrumbox`), using predicted widget locations to guide the search achieves a higher coverage. The main coverage increases were in applications with sparse GUIs, like `Address Book` (24%→48%) and `Dietetics` (20%→54%). The predicted widgets also aided the random tester to achieve coverage where complex sequences of events are needed, such as the `Bellmanzadeh` application (22%→28%). `Bellmanzadeh` is a fuzzy logic application and requires many fields to be created of different types. Random is unlikely to create many variables of unique

types but, when guided by predicted widget locations, is more likely to interact with the same widgets again to create more variables. The random tester is similar to Android Monkey, but achieves a lower level of coverage to that Choudhary et al. [33] observed. The coverage levels achieved by the random tester show that it spends more time performing uninteresting actions, whereas it is far more likely to interact with an actual widget when guided by widget predictions

One notable example is the application JabRef, where unguided random achieved 6.6% branch coverage, significantly better than random guided by widget predictions which achieved 5.2%. JabRef is a bibtex reference manager, and by default it starts with no file open. The only buttons accessible are “New File” and “Open”. The predicted boxes contain an accurate match for the “Open” button and a weak match for the “New File” button. If the “Open” button is pressed, a file browser opens, locking the main JabRef window.

As we randomly select a window to interact with from the available, visible windows, any input into the main JabRef window is ignored until the file browser closes. There are two ways to exit the file browser: clicking the “Cancel” button or locating a valid JabRef file and pressing “Open”. There are, however, many widgets on this screen to interact with lowering the chance of hitting cancel, and it is near impossible to find a valid JabRef file to open for both the prediction technique and the API technique. Even if the “Cancel” button is pressed, there is a high chance of interacting with the “Open” button again in the main JabRef window.

On the other hand, the random technique has a low chance of hitting the “Open” button. When JabRef starts, the “New” button is focused. We repeatedly observe the random technique click anywhere in the tool

bar and type “Hello World!”. As soon as it presses the space key, it would trigger the focused button and a new JabRef project would open. This then unlocks all the other buttons to interact with in the JabRef tool bar

RQ3.3: In our experiments, widget prediction lead to a significantly higher attained coverage in generated tests, achieving on average 42.5% higher coverage over random testing. However, widget prediction can get stuck in a loop if the amount of identified widgets is low.

Table 3.3. Branch coverage of random guided by no guidance, widget prediction and the Java Swing API. **Bold** is significance.

Application	Prediction Cov.	Random Cov.	p_v (Pred, Rand)	\hat{A}_{12} (Pred, Rand)	API Cov.	p_v (Pred, API)	\hat{A}_{12} (Pred, API)
Address-Book	0.484	0.235	<0.001	0.032	0.370	<0.001	0.237
bellmanzadeh	0.276	0.215	<0.001	0.048	0.425	<0.001	1.000
BibTex-Manager	0.214	0.160	<0.001	0.145	0.347	<0.001	0.998
BlackJack	0.355	0.167	<0.001	0.143	0.848	<0.001	1.000
Dietetics	0.544	0.197	<0.001	<0.001	0.564	0.067	0.640
DirViewerDU	0.728	0.522	<0.001	<0.001	0.576	<0.001	0.089
JabRef	0.052	0.066	<0.001	0.768	0.060	0.608	0.540
Java-FabledLands	0.105	0.056	<0.001	0.098	0.102	<0.001	0.122
Minesweeper	0.837	0.811	<0.001	0.170	0.850	<0.001	0.859
Mobile-Atlas-Creator	0.120	0.059	<0.001	0.199	0.224	<0.001	1.000
Movie-Catalog	0.581	0.328	<0.001	0.007	0.643	<0.001	0.826
ordrumbox	0.192	0.181	<0.001	0.056	0.203	<0.001	0.905
portecle	0.063	0.049	<0.001	0.121	0.106	<0.001	0.948
QRCode-Generator	0.673	0.582	<0.001	0.024	0.658	0.010	0.304
Remember-Password	0.333	0.255	<0.001	0.182	0.535	<0.001	0.968
Scientific-Calculator	0.588	0.469	<0.001	0.129	0.863	<0.001	1.000
ShoppingListManager	0.758	0.563	<0.001	0.032	0.758	1.000	0.500
Simple-Calculator	0.769	0.460	<0.001	<0.001	0.864	<0.001	1.000
SQuiz	0.111	0.111	1.000	0.500	0.130	<0.001	1.000
UPM	0.125	0.060	<0.001	0.040	0.460	<0.001	0.986
Mean	0.395	0.277	0.050	0.135	0.479	0.084	0.746

RQ3.4: How close can a random tester guided by predicted widget locations come to an automated tester guided by the exact positions of widgets in a GUI?

Using GUI ripping to identify actual GUI widget locations serves as a gold standard of how much random testing could be improved with a perfect prediction system. Therefore, Figure 3.16 also shows branch coverage for a tester guided by widget positions extracted from the Java Swing API. It is clear that whilst predicted widget locations aid the random tester in achieving a higher branch coverage, unsurprisingly, using the positions of widgets from an API is still superior. This suggests that there is still room for improving the prediction system further.

However, notably, there are cases where the widget prediction technique *improves* over using the API positions. One such case is DirViewerDU. This is an application consisting of only a single tree spanning the whole width and height of the GUI. If a node in the tree is right clicked, a pop-up menu appears containing a custom widget not supported or present in the API widget positions. However, the prediction approach correctly identifies this as an interactable widget and can generate actions targeting it.

Another example of this is in the Address Book application. Both guidance techniques lead the application into a state with two widgets: a text field and a button. To leave this GUI state, text needs to be typed in the text field and then the button needs to be clicked. If no text is present in the text field when the button is clicked, an error message is shown and the GUI state remains the same. However, the information of the text field is not retrieved by the Swing API as it is a custom

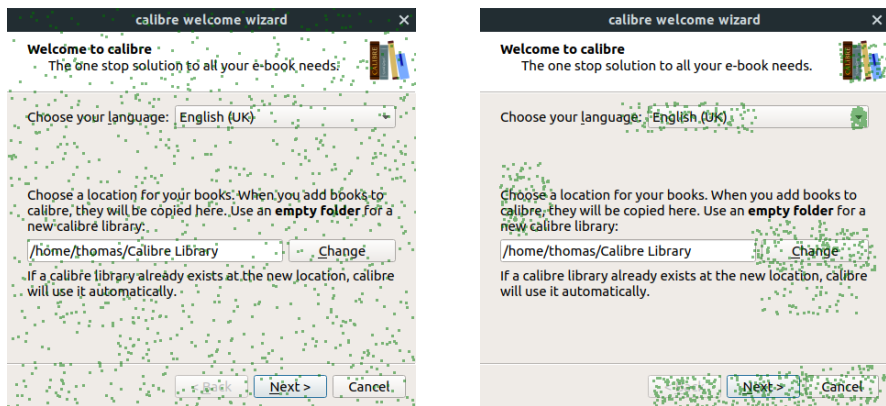
widget. The API guided approach then spends the rest of the testing budget clicking the button, producing the same error message. Predicted widget guidance identifies the text field, and can leave this state to explore more of the application. Similar behaviour was observed one more application.

A final observation is with the Java Fabled Lands application. This application is a story-based game, with links embedded in the text of the story, only identifiable by users due to the links being underlined. The Swing API guided approach only identifies the overall text field, having to fall back to a random strategy to interact with these links. However, the detection approach can identify a few of these links and can navigate through certain scenarios in the story, achieving a higher code coverage than the API approach in this instance.

RQ3.4: Exploiting the known locations of widgets through an API achieves a significantly higher branch coverage than predicted locations, however widget prediction can identify and interact with custom widgets not detected by the API.

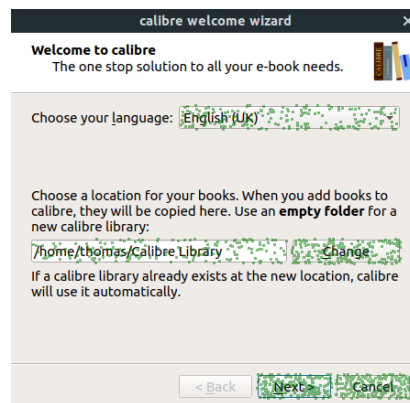
3.4 Discussion

To further investigate the quality of generated interactions using the random tester, we plotted 1000 interactions with a single GUI. Figure 3.17 shows the points of interaction for each technique. The random approach has an expected uniform distribution of interaction points in the GUI. The detection approach refines this and targets each widget. However, we can see that the disabled “next” button has also been identified and targeted, as well as some text. Finally, the gold standard



(a) Interaction locations of an unguided random GUI tester

(b) Interaction locations of a random GUI tester guided by predicted widgets



(c) Interaction locations of a random GUI tester guided by the Java Swing API

Figure 3.17. Interaction locations for a random tester guided by different approaches.

API approach always interacts with a valid widget.

The quality of the prediction approach is directly related to the training data of the YOLOv2 network. We tried to improve the precision and recall of the prediction system. One such method was by selecting a subset of the training data that had similar statistics to the corpus of real GUIs. For this, we looked at the following statistics:

Chapter 3 Guiding Testing through Detected Widgets from Application Screen Shots

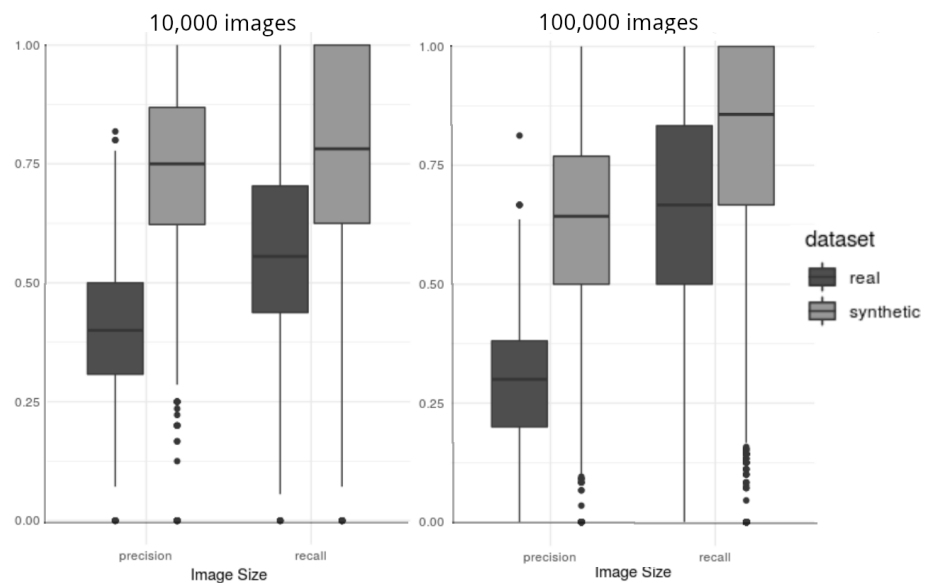


Figure 3.18. Precision and recall of a widget prediction system trained on a selected subset, or the entire training dataset of labelled GUIs.

- *widget location*: the probability of a widget appearing in a cell of the 13x13 grid used by YOLOv2;
- *widget dimension*: the proportion of the total space in the GUI that a widget is occupying;
- *widget probability*: the probability of a widget appearing in a GUI;
- *image pixel intensity*: the shape of the histograms of all images in the set combined.

Then, we calculated each metric on the real dataset. These values were plugged into a genetic algorithm which would select N images from the training dataset, calculate these metrics and compare against the real dataset. The fitness function was directly correlated to the difference in statistics, with an aim to minimise this difference. The crossover function was single point, swapping elements from the two selected individuals. The mutation function would replace a random quantity

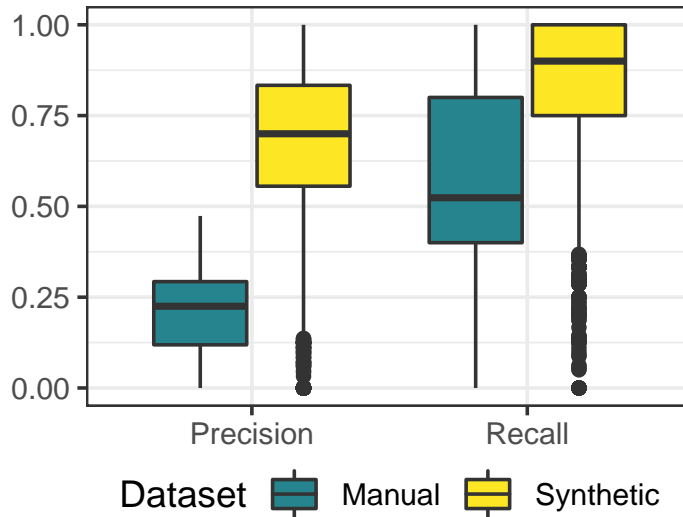


Figure 3.19. Precision and recall of synthetic data against manually annotated application GUIs.

of GUIs in the list with random GUIs taken from the set of training data that do not currently appear in the list.

Figure 3.18 shows the precision and recall when comparing a network trained on the data selected by the genetic algorithm, to a network trained on the entire dataset. From this Figure, the system trained on more training data has an increased recall on real GUIs, and the system trained on a subset of the training data has higher precision.

The relationship between recall, precision and code coverage is interesting. It is not immediately obvious whether sacrificing recall for precision will aid in code coverage, or vice-versa. Future work is needed into this relationship, and the best method of sampling a subset of data to maximise the quality of the trained prediction system for object detection. For our experiments, we decided to use the entire set of training data to expose the network to more variety, but it is entirely possible

that selecting a subset could produce a better object detection system.

To further investigate if our model can predict widgets in applications which are not supported by current testing tools, we ran our model on GUIs where widget information cannot currently be extracted. Figure 3.19 shows the precision and recall of the model on GUIs that had to be manually annotated. These applications mainly consist of unknown GUI framework usage, or drawing widgets directly to the application's screen buffer. This usually involves a unique theme created by the application's designer, which is unique from other applications on the same platform. It is interesting that on these GUIs, our model achieves a level of performance which is in-between the Ubuntu applications, and the Mac OSX applications. This could be because these manually annotated applications ran on Ubuntu so parts of the theme could appear in the screen shots, and our model has a slight increase in performance when detection widgets in these screen shots.

When evaluating the performance of the widget prediction model on screen shots of applications, we currently use the intersection over union metric, which relies on accurate bounding boxes being predicted. However when testing applications using their GUI, it is not necessary to predict good bounding boxes, so long as the bounding boxes have a high chance of allowing interaction with the widget. One issue with intersection over union is that a predicted box which is entirely encapsulated by the actual widget may be perceived as a false positive, but would have a 100% chance of interacting with a widget if exploited by a testing tool. To evaluate the difference between the object detection intersection over union, and the likelihood of just clicking a widget, we calculated precision and recall using a new metric: area overlap. This metric calculates the area of intersection between the predicted bound-

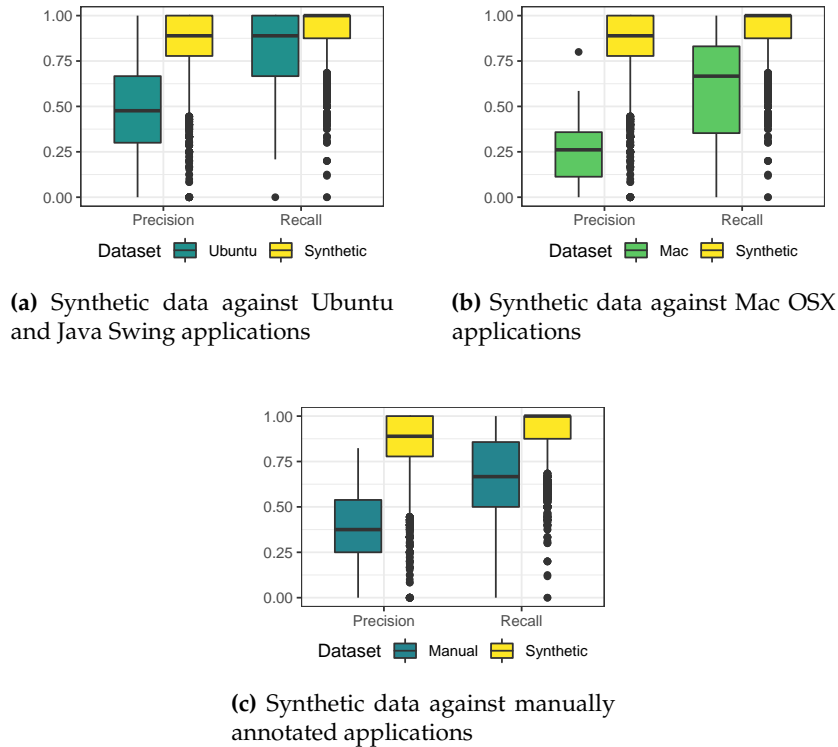


Figure 3.20. Precision and recall of the widget prediction modeling using an area overlap metric

ing box, and the actual widget, and divides it by the area of the predicted bounding box to give the probability that a randomly generated coordinate inside the box will click on the actual widget.

Figure 3.20 shows the precision and recall for different datasets using the new area overlap metric with the same threshold as for the previous intersection over union metric. Here, we can observe a sharp increase in both precision and recall in all three dataset comparisons. In our experiments, we used intersection over union, as this is standard practice in object detection. However, intersection over union may not be the best metric to evaluate how effective a widget prediction model is at guiding a tester to click on widgets in a GUI.

3.5 Conclusions

When applications have no known exploitable structure behind their GUI, monkey testing is a common fail-safe option. However, it is possible to identify widgets in a GUI from screen shots using machine learning, even if the prediction system is trained on synthetic, generated GUIs. Applying this machine learning system during random GUI testing led to a significant coverage increase in 18 out of 20 applications in our evaluation. A particular advantage of this approach is that the prediction system is independent of a specific GUI library or operating system. Consequently, our prediction system can immediately support any GUI testing efforts.

Comparison to a gold standard with perfect information of GUI widgets shows that there is potential for future improvement:

- Firstly, we need to find a better method of classifying GUI widgets. A tab that changes the contents of all or part of a GUI's screen has the same function as a button, so they could be grouped together.
- We currently use YOLOv2 and this predicts classes exclusively: if a button is predicted, there is no chance that a tab could also be predicted. Newer methods of object detection (e.g. YOLOv3 [114]) focus on multiple classification, where a widget could be classified as a button and as a tab. This could improve the classification rate of widgets that inherit attributes and style.
- The relationship between precision and recall needs further investigation. It is not clear what role each metric plays in achieving code coverage. A higher precision should ensure that less time

is spent on wasted interactions, but if certain widgets can never be interacted with as a result, then this could negatively impact test generation. What is the trade off between having less false positives, or a higher recall and increasing test generation time?

- Whilst labour intensive, further improvements to the widget prediction system could be made by training a machine learning system on a labelled dataset of *real* GUIs. To lower effort costs, this dataset could be augmented with generated GUIs. The performance of the prediction system is dependent on the quality of training data.
- Furthermore, in this chapter we focused on a single operating system with various themes. However, it may be beneficial to train the prediction system using themes from many operating systems and environments to improve performance when identifying widgets across different platforms.

Besides improvements to the prediction system itself, there is potential to make better use of the widget information during test generation. For example, if there are a limited number of boxes to interact with, it may be possible to increase the efficiency of the tester by weighting widgets differently depending on whether they have previously been interacted with (e.g., [130]). This could be further enhanced using a technique like Q-learning (cf. AutoBlackTest [92]) or using solutions to the multi-armed bandit problem [36].

4 Testing By Example: Graphical User Interfaces

In the last chapter, we focused on using deep learning to create a prediction system that can identify patterns in screen shots of GUIs. We used many screen shots of generated GUIs to train the network powering the system. However, it might be possible to model how real users interact with an application, and exploit the interactions of these users to create application-specific models. Adopting the growing trend of crowd-sourcing would be a useful method of processing and augmenting the large quantities of gathered user data.

4.1	Introduction	110
4.2	Modelling GUI Interaction Events	111
4.3	Generating GUI Interaction Events	113
4.4	Evaluation	121
4.5	Results	128
4.6	Conclusions	138

4.1 Introduction

We live in an age of data. It is common for most applications to log information about the environment they are running in, problems encountered and even how users interact with parts of the system. In the last chapter, we proposed testing techniques which take the current application state at a single point in time, identifying interesting parts of the GUI where interactions may trigger underlying events in the source code, and then performing these interactions. These techniques have no concept of time. They do not know if an interaction led to a new GUI state, or had any impact, and do not differentiate between the first and last interaction performed. However, a user interacting with a GUI thinks about future interactions, as well as past and present. Usually a user has some objective, and performs a sequence of interactions over time to achieve their goal. For example, when a user is filling out a form, they usually start from the top and work their way down, finally clicking a *submit* button. Analysing this sequential information could be beneficial for testing tools to generate more realistic event sequences as test data.

The key contributions of this chapter are:

1. An identification of low-level interaction events that are needed to replicate user interactions using keyboard and mouse for black box testing.
2. A technique of processing user GUI interaction data into a model capable of generating interaction data that is statistically similar to that of the original user's interactions.
3. An empirical study comparing different techniques of generat-

ing models from user interactions, and comparing them to current testing techniques of testing an application solely through its GUI.

In this chapter, we will focus on how effectively a model trained on user interactions can explore an application versus the approaches from the previous chapter.

4.2 Modelling GUI Interaction Events

To simulate a user interacting with a GUI, it is necessary to understand what actions a user can perform that trigger events in the application. The event-driven programming paradigm enables applications to remain in an idle “sleep” state until some event triggers parts of the system to execute. As discussed in chapter 2, widgets can have event handlers linked to them, executing code in the application when interaction occurs. These interactions are triggered through usage of the application’s GUI.

To interact with a GUI, two devices are primarily used: keyboard and mouse. The mouse can move a cursor on the screen, and click to target specific elements shown to users. A standard mouse usually has a maximum of five buttons: left click, right click, middle click and button 4/5 that is usually on the side of the mouse. A mouse usually has a scroll wheel which can be used to easily navigate a page when contents are taller than the monitor’s resolution. Keyboards are usually very specific to locale, but function similarly. Each press and release of the buttons on the keyboard sends corresponding events to an application, firing events in the system.

Because of the diversity of keyboards and the large amount of interactions that can occur with the mouse, it is important to employ a general set of events that are not locked to one specific keyboard layout or mouse.

4.2.1 GUI Events

There are many methods of interacting with a Graphical User Interface. We define 4 categories of user GUI events which encapsulate standard keyboard and mouse interactions. By combining these events, most forms of user interaction can be generated for black box testing without any prior application assumptions.

1. **Mouse Event:** With the cursor over a GUI's widget, this event occurs when any of five buttons on the mouse is pressed or released. This has 10 sub-events across the five buttons on the mouse, a *button pressed* event and a *button released* event. Mouse position was encoded into the event (e.g., `MOUSE_DOWN[100, 100]` for a mouse button press at pixel location 100, 100). Mouse movement without clicking could also be encoded as a mouse event, but was not tracked during our experiments as most applications do not take mouse movement into account, only the location when a mouse button is pressed.
2. **Key Event:** Pressing any key on the keyboard. Each key event is encoded with corresponding triggering key and is split into two sub events for each key: *key down* and *key up*. This leads to many sub-events, depending on the number of available input keys to users but allows complex interactions such as past-

ing via short cut key combination (e.g., CTRL-V in four events: `KEY_DOWN['CTRL']`, `KEY_DOWN['V']`, `KEY_UP['V']`, `KEY_UP['CTRL']`).

3. **Scroll Event:** With the mouse over a certain position in a GUI, this event is triggered by the scroll wheel moving. This is split into two types of scroll event: *up* or *down*. Again, the position of the mouse when scrolling is encoded into the event, as well as the mouse wheel displacement (e.g., `SCROLL_DOWN[100, 100, 10, 0]` for scrolling the mouse wheel down 10 pixels and across 0 pixels at pixel location 100, 100).
4. **Window Change Event:** Although not directly triggered by a user, this event occurs when the title of the currently in-focus window changes. This could be through an application opening a new window, or the user clicking on a different window.

These events allow us to record, store and replay any interactions that a user performs in a GUI through a traditional keyboard and mouse. In the next section, we look at how we can learn from the events stored from users, and generate new events.

4.3 Generating GUI Interaction Events

The approaches from the last chapter are incapable of generating some events that could be recorded from users. For example, we described a random “monkey” tester that could interact with random positions in the application through clicking or typing. This could never generate drag and drop functionality, or a user pressing a short cut key (e.g., CTRL-V) as every generated interaction is only for that specific point in time, interactions can never overlap. Dragging and dropping

is an example of two interactions: mouse button up, and mouse button down. Because of this limitation of the previous approach, a new random baseline is needed capable of generating these newly identified interactions.

4.3.1 Random Events Monkey Tester

The random approach discussed in the last chapter may have a disadvantage over any model which learns from user data due to the new events that a user model can perform. Models based on user data can perform more interactions such as scrolling, click-and-drag or holding down keys.

We created a new random approach which is very similar to the random monkey tester in the previous chapter. Firstly, we decide on what type of action to generate from the ones listed in the previous section, excluding window change events as these are not directly triggered by the user, but as a response by the application itself. When an event type has been decided, we can generate the data that the event type needs. For example, a mouse event needs three components: a mouse button, whether the button was pressed or released, and a position on the screen for the interaction.

This new monkey tester can generate events such as scrolling, click and drag or holding down keys, but does not have temporal data such as that of a real user. Interactions generated by the random tester are not related to the previous, or the next interaction. To include this sequential-based relationship information, an analysis of user data is needed.

Table 4.1. Features of each event type

Event Type	Features	Description
Mouse Event	2	$(position_x, position_y)$ of cursor in GUI when the interaction occurred.
Key Event	0	No keyboard data is clustered
Scroll Event	4	$(position_x, position_y)$ of mouse when scrolling started, and $(displacement_x, displacement_y)$ of scroll wheel for current scroll event.

4.3.2 Generating User Models

When users interact with an application, for each event described in the previous section, we store all relevant data from the interaction. This gives a large sequence of data for each user, with corresponding timestamps so a time-line of interactions can be constructed.

We group the interactions by type, for example, one group containing all events where the left mouse button was pressed down. Each event type has explicit metadata that can be used to replay the interaction, for example, the position of the cursor in the GUI when the left mouse button was pressed. Grouping by type allows us to find patterns for each type of interaction in a user's data, and this is done through clustering an interaction type's metadata. This creates subgroups in the metadata, and is a form of unsupervised learning. Each subgroup represents similar user interactions, and by analysing the user data, we can identify transitions between subgroups which can influence the interactions we generate.

Clustering Events

Table 4.1 shows events recorded from user interactions, the features of each event, and the description of each feature. In this table, the event type is a generalisation of user interactions, for example, *Mouse Event* represents any interaction occurring through the mouse which could be *LEFT_DOWN*, *LEFT_UP*, *RIGHT_DOWN*, etc.

With each event type's features stored, they can easily be clustered. We used K-means [21] in our implementation. Selecting an appropriate value of K is important for our test generation technique to perform at an optimal level. Too many clusters, and there will be a limited number of paths through the system (there will be a low number of transitions between each cluster). Too few clusters and some widgets will become uninteractable, having a large distance between the original point of user interaction and the cluster centroid. How we select the value of K is influenced by the amount of data available for us to cluster.

As we store data in chronological order, we can apply techniques from other areas of computer science which also work with ordered data. As an example, we look at how human language is processed. Human language is an ordered sequence of data. This data consists of words (i.e., sequences of characters), and sentences (i.e., sequences of words) with temporal data present between elements in each of the sequences. Natural Language Processing (NLP) is a subfield of computer science, with a goal of processing and studying the patterns present in large amounts of human language data. To achieve this, NLP develops novel practical applications, intelligently processing written human language. NLP involves the processing of sequences of text to infer models describing the characteristics, syntax, and semantics of the text. However, the tech-

niques applied are not only limited to textual input. Sequences of user data gathered from various user interfaces can be analysed using NLP techniques. For example, one such technique of calculating the transition probabilities of elements in a sequence is to use an n-gram model.

N-gram - N-grams are language models which store sequential information about text. N-grams contain the probabilities of transitions between each element in sequential data (e.g., a sentence). When an n-gram is created from a sentence, the data contained would be the probabilities of one word following another. The n of n-gram is actually a parameter that can be changed to modify the length of sequences generated. With $N=2$, the probabilities shown would be for a sequence of two words appearing one after the other in a given corpus of text. For example, take the following sentence:

She sells sea shells by the sea shore. The sea shells that she sells are sea shells for sure.

If we assume $N = 2$, we can construct the following word pairs:

She sells; sells sea; sea shells;
shells by; by the; the sea; sea shore
The sea; sea shells; shells that;
that she; she sells; sells are;
are sea; sea shells; shells for; for sure

We can calculate the probabilities that one word follows another using

these groupings, and observe some details about the sentence, for instance:

1. The probability of the word “shells” following the word “sea” is 0.75 (i.e., $P(\text{shells}|\text{sea}) = 0.75$)
2. The word “sea” always follows the word “the”: $P(\text{sea}|\text{the}) = 1$

We can also apply this technique to the sequential data recorded from users. By replacing each event in the sequence of user data with the cluster centroid assigned to an events metadata, we can generate a probabilistic model. To generate interaction data from this model, we use a Markov chain. We start by selecting a random element from the n-gram model and seed the event represented by the cluster centroid linked to this group. Then, we look at the probabilities of the n-gram of transitions from this cluster to another. We decide which cluster to jump to next, weighted by the n-gram probabilities. Now we can look at the last two interaction groups and the probabilities of jumping to each group from these two previous groups.

This model can run indefinitely, and has the added advantage of never tiring as a human tester would. The model represents all user interactions with an application. However, the full set of user interactions with an application can generate models that are large in size. Given that unique pages and screens in an application use different interaction events, it may be beneficial to generate smaller, more targeted models that interact with the application at specific times during testing.

4.3.3 Application Window-based Models

Having a separate model for each state of an application would aid in guiding test generation by interacting with certain states more specifically as a user did than with the application as a whole. For example, if an application requires a positive integer entered into a textbox at a certain point in the execution, it will be far easier to generate this event with a model trained from user data that consists of the subset of user data which clicks inside the textbox and types a positive integer, opposed to all possible interactions with the application across all states. However, inferring an application's state from the appearance of its GUI is difficult.

To generate models for unique parts of an application, we chose to identify parts of an application with a simple, naïve approach. We use only the title of the currently in-focus window. The advantages of this are that it requires low computation, but the disadvantage is that applications which only use a single window but hot-swap the contents of said window will have inflated, non-specific models.

Now, we follow the same process for the application user model, but the data provided to the model is filtered to only include interactions that users performed in the current application's window. We can grab the title of the current window, and select the models depending on this title to guide the generated interaction data. This works so long as we have seen the current window during training, but we also need a method of generating interactions for windows that have not been previously seen.

Using separate models per application window limits the amount of

training data each model has access to, and this alters the clustering process and the available values of K in K -means. This creates a challenge when generating models. Each window in an application has a diverse quantity of interactions, so a static value of K is not sufficient. For instance, a confirmation dialog may only have a few interactions, with an “OK” or “Cancel” button, but the main window may have an interaction count many magnitudes higher. It is also not practical to manually choose the number of clusters for each window of an application. It is important to balance the number of clusters for each window to provide the temporal sequential model with sufficient transitions between clusters, but also to ensure there are enough clusters to interact with the same widgets that users did by minimising the distance between each cluster and its centroid.

We experimentally tried many equations to calculate the number of centroids that should be used. Firstly, we tried dividing the number of cluster centroids by various numbers, e.g., by 10. The problem with this approach were application screens which had very little interaction points (e.g., pop-up confirmation menus). With a low number of interaction points, the cluster centroids often appeared in the middle of groups of widgets present in the application state, and so the test generators can get stuck in these states. Also, application screens with many interaction points still have clusters with little data points and a language model which is sparse. To fix this issue, we changed the division to a square root, so as the number of interactions for an application’s screen increases, so does the amount of data points per cluster. Eventually, we select the following equation, where x denotes the number of user interaction data points available to a specific application’s window: $\text{floor}(\min(x, \max(5, 3 \times \sqrt{x})))$. We need to allow a high number of clusters in relation to a small dataset to enable interactions to be gen-

erated at all points in a window. However, with a large dataset, we can have a much lower proportion of clusters with many more data points in each. This equation has a good balance of allowing many clusters to appear when only a limited number of interaction points are present in a window, but will also flush out the transition probabilities when there are many interaction points by assigning more events to each cluster.

4.4 Evaluation

To observe the impact that user data can have on a model that generates GUI interaction data, we ask the following questions:

- RQ4.1: How beneficial is using a model trained on GUI interactions within specific windows compared to models trained on interactions with the whole application, or a number of different applications?
- RQ4.2: What is the best approach to generate GUI tests when encountering a window for which no user data was available to create a model?
- RQ4.3: Does a GUI testing approach in which interactions with GUI components are guided by a model based on user data provide better code coverage than an approach in which interactions are purely random?

4.4.1 Experimental Setup

To create models based on real user data, we have to record users interacting with applications. Test subject applications were taken from the last chapter, but in order to maximise the quantity of user data per

Table 4.2. The applications tested when comparing the three testing techniques.

Number	Application	Description	LOC	Branches
1	bellmanzadeh	Fuzzy decision maker	1768	450
2	BlackJack	Casino card game	771	178
3	Dietetics	BMI calculator	471	188
4	JabRef	Reference Manager	60620	23755
5	Java Fabled Lands	RPG game	16138	9263
6	Minesweeper	Puzzle game	388	155
7	ordrumbox	Create mp3 songs	31828	6064
8	Simple Calculator	Basic maths calculator	305	110
9	SQuiz	Load and answer quizzes	415	146
10	UPM	Save account details	2302	530

application, we halved the number of subjects to 10 applications. This allows more transitions to be present in each Markov chain, and more cluster centroids to be used per application and window. Table 4.2 shows the applications selected. The criteria for application selection was as follows:

- The application should be complex enough to not be fully explorable in under three minutes;
- The application should have minimal input/output commands (e.g., saving and loading to disk);
- The application should be simple to understand and take minimal training for users, having a small amount of unique states and help menus included in the application.

With 10 applications to test, we recorded 10 participants interacting with each application. Participants were randomly selected PhD students from The University of Sheffield, each studying different areas of computer science. User recording was split into two phases: a warm up “training” phase, followed by a “task-driven” phase.

The warm up phase was the first time any user had interacted with an application. This was designed to simulate a novice user who is learning and exploring the application with no real goal of what to achieve with their interactions. Each participant interacted with each application for three minutes during the warm up phase.

With the warm up phase complete, each participant was given a list of tasks to perform. The tasks were in a randomized order and if a user had already performed a task during the task-driven phase, they could skip over the task. See Appendix A for an example task sheet given to participant one.

Next, the user data can be used in various ways to generate user models. Each combination is presented as a different technique which will be described through each research question.

The techniques used in this chapter are a little different to the techniques from the last chapter. The user models can generate events such as click and drag, but can also generate the events from the previous chapter (click and key pressed). In order to ensure that the techniques from both chapters generate a similar number of events, we have to define the events from the last chapter. Firstly, a “click” is a mouse button press and mouse button release. The event generators that are derived from user data would need to perform two interactions to perform an identical interaction as that of event generators from the last chapter. Because of this, we have to make any event that corresponds to a “down” and “up” event only perform 0.5 actions. This gives each technique a fair testing budget (i.e., all techniques could perform 500 mouse clicks which correspond to 1000 mouse down and mouse up interactions).

Table 4.3. Techniques of using user data to generate GUI events for interactions.

Acronym	Resource Cost	Description
<i>APPMODEL</i>	User interaction data for AUT	Derive a sequential model from user interaction data on an application
<i>WINMODEL</i>	User interaction data for AUT	Derive a sequential model from user interaction data for each seen window of an application. If currently at an unseen window, fall back to <i>APPMODEL</i>
<i>WINMODEL-AUT</i>	User interaction data for multiple apps, not including AUT	Derive a sequential model from user interaction data for each seen window of a set of applications, not including the AUT. If currently at an unseen window, fall back to an aggregated <i>APPMODEL</i> of the applications.

RQ4.1: How beneficial is using a model trained on GUI interactions within specific windows compared to models trained on interactions with the whole application, or a number of different applications?

This first question evaluates the impact that training data has on the quality of generated models. This research question aims at comparing a model that is trained on specific data (e.g., data from the application under test) against general data (e.g., data taken from applications excluding the application under test). Also, even if a model is trained specifically on data from the application under test, it may be possible to split the training data and generate multiple models, or a model for each window of the application.

Table 4.3 shows three techniques that will be compared to evaluate the impact that the source of user data can have on a generated model. Comparing *APPMODEL* and *WINMODEL* will give insight into the benefits of generating a model per application window. *WINMODEL* and *WINMODEL-AUT* are identical in technique, but the data used to generate the models differs. *WINMODEL-AUT* should have much more data, with many more interaction options. Whilst this can be beneficial, it can also hinder the number of useful interactions that

Table 4.4. Techniques of using user data to generate GUI events for interactions.

Acronym	Resource Cost	Description
<i>APPMODEL</i>	User interaction data for AUT	When encountering an unseen window, use a model derived from user interaction data on an application.
<i>RANDOM</i>	None	When encountering an unseen window, generate random event in the bounds of the application's window, similar to what user could, including scrolling, click and drag, and press and holding keys.
<i>PREDICTION</i>	Screen shot	Predict locations of widgets from screen shot and interact inside the bounds of a random predicted box. Does not use user data.
<i>GROUNDTRUTH</i>	Supported underlying widget structure	Use exact known positions from Swing API to interact with widgets. Does not use user data.

WINMODEL-AUT performs. On the other hand, *WINMODEL* should have a limited set of interactions available to generate, with most of them having some impact when performed on the AUT, as users rarely perform uninteresting actions.

RQ4.2: What is the best approach to generate GUI tests when encountering a window for which no user data was available to create a model?

During testing, it is possible that the window title of an application's window will not have been seen during user interactions. This *unseen window* will have no corresponding model to generate events, so another method of guiding interactions is needed. This could occur if user submitted content (e.g., the name of the current document being edited) is included in the window's title, a new version of the application has added additional windows, or various other reasons.

Table 4.4 shows four possible techniques to handle unseen application windows. To find the better technique of handling an unseen window in an application, we first compare the two user approaches: *RANDOM*

Table 4.5. Techniques of using user data to generate GUI events for interactions.

Acronym	Resource Cost	Description
<i>RANDCLICK</i>	None	A random tester from the last chapter that can click or type at random positions on the screen.
<i>RANDEVENT</i>	None	Generate event similar to what user could, including scrolling, click and drag and press and holding keys.
<i>WINMODEL</i>	User interaction data for AUT	The best techniques of using user data combined. Derive a sequential model from user interaction data for each seen window of an application. If currently at an unseen window, fall back to <i>APPMODEL</i> .

and *APPMODEL*. The better technique will be taken forward to RQ4.3, and named as *USERMODEL*. We will also compare the *USERMODEL* against the approaches from the previous chapter which do not use user data, but can still handle unseen application windows: *PREDICTION* and *GROUNDTRUTH*.

RQ4.3: Does a GUI testing approach in which interactions with GUI components are guided by a model based on user data provide better code coverage than an approach in which interactions are purely random?

The testing technique with the cheapest resource cost is random testing. To see whether models generated from user data can outperform random testing, we compared the techniques shown in table 4.5. *RANDCLICK* is the random technique from the last chapter, with *RANDEVENT* being the new random tester which can generate events similar to the models generated from user data. *USERMODEL* is a combination of the best techniques taken from RQ4.1 and RQ4.2.

4.4.2 Threats to Validity

There is a risk of users learning during application interaction, and this influencing data recorded in the later stages of the experiment. To mitigate this, each user interacted with a random ordering of applications, and performed a random order of tasks. Each user interacted with different applications before and after encountering any specific application, but the same set of tasks in a randomised order.

As each testing technique uses randomised processes, there is a risk that one technique may outperform another due to generating lucky random numbers. To mitigate against this, each technique ran on each application for 30 iterations and we used appropriate statistical tests to evaluate the effectiveness of each technique.

To ensure that all techniques have a fair testing budget, each technique seeded on average a single interaction per second for 1000 actions. As the techniques proposed in the last chapter generate interactions that are a combination of some events of the techniques from this chapter (e.g., “clicking” instead of “mouse button down” followed by “mouse button up”), we modified the delay of the techniques for this chapter appropriately. For example, instead of waiting a full second after pressing the mouse button down, a half second delay would occur and when the mouse button is released, the remaining half second delay will synchronise the time scale of both techniques. The “button down” and “button up” events also count as half the testing cost of a full “click” event.

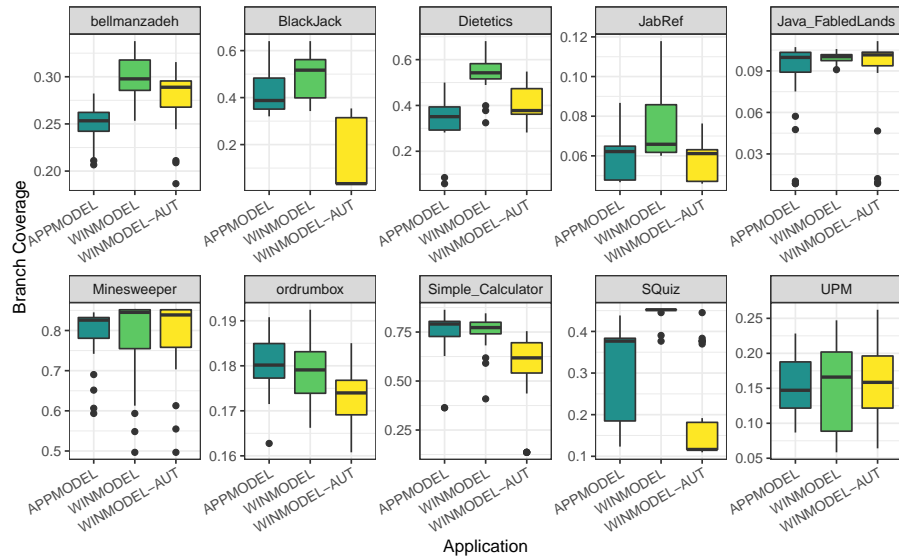


Figure 4.1. Branch Coverage of techniques *APPMODEL*, *WINMODEL*, and *WINMODEL-AUT*.

Application	WINMODEL	APPMODEL	WINMODEL	WINMODEL-AUT
bellmanzadeh	***0.298	0.253	**0.298	0.289
BlackJack	***0.517	0.388	***0.517	0.034
Dietetics	***0.543	0.351	***0.543	0.378
JabRef	***0.066	0.062	***0.066	0.061
Java-FabledLands	0.100	0.100	0.100	0.102
Minesweeper	***0.845	0.826	0.845	0.839
ordrumbox	0.179	0.180	***0.179	0.174
Simple-Calculator	0.773	0.791	***0.773	0.618
SQuiz	***0.452	0.377	***0.452	0.116
UPM	0.166	0.147	0.166	0.158
Mean	0.383	0.336	0.383	0.282

Table 4.6. Branch Coverage of techniques *WINMODEL*, *APPMODEL*, and *WINMODEL-AUT*. **Bold** indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).

4.5 Results

RQ4.1: How beneficial is using a model trained on unique windows against models trained on whole applications or all applications?

Figure 4.1 shows the branch coverage achieved by *WINMODEL*, *APPMODEL* and *WINMODEL-AUT*. We can see from table 4.6 that using



Figure 4.2. Interaction cluster centroids for the “LEFT_DOWN” event when using the *WINMODEL* approach.



Figure 4.3. Interaction cluster centroids for the “LEFT_DOWN” event when using the *APPMODEL* approach.

models constructed on separate windows (*WINMODEL*) achieves a significantly higher coverage in six of 10 applications when compared to using a single model for an entire application. Figure 4.2 and 4.3 show the centroids available for application interaction for the same window, but using either a window specific or application specific model. These figures show that the window specific model has less choice for interaction, and therefore a higher probability of selection a cluster centroid that will generate an interesting interaction. From this, we can conclude that using models based on an application’s windows is better than a single model based on user data for an entire application. However, there is one interesting observation. If we observe the applications *JabRef* and *ordrumbox*, both techniques perform identically. This

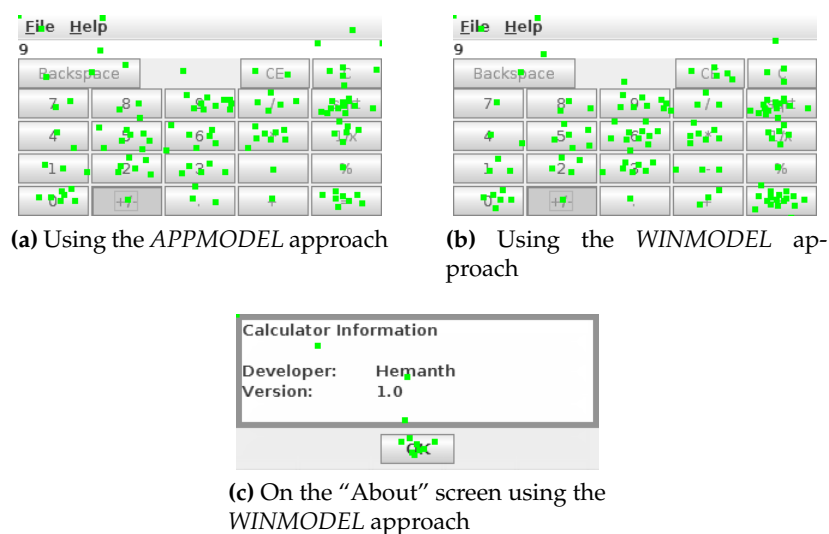


Figure 4.4. Interaction cluster centroids for the "LEFT_DOWN" event in the *Simple Calculator* application.

is an interesting case and is not unexpected. Both of these applications draw directly to the screen buffers of the application. The window-based models rely on a unique title for each window in the AUT, but as these only use a single window with the title never changing, in these two applications, both of these approaches are equivalent.

The *Simple Calculator* application is the only application where using the *APPMODEL* technique achieves a higher code coverage than *WINMODEL*. However, the coverage increase here is small, equating to less than two more branches covered on average by using the *APPMODEL* technique. Figures 4.4a, 4.4b and 4.4c show the reason behind this. *Simple Calculator* has two screens: the main calculator screen, and an about screen. The "OK" button on the about screen is in the same location in the GUI as the decimal point button in the main calculator screen. This gave *APPMODEL* more clusters and transitions to this location on the screen, allowing the button to be pressed more frequently than with the *WINMODEL* technique, which generally struggled to achieve branch



Figure 4.5. Interaction cluster centroids for the “LEFT_DOWN” event when using the *WINMODEL-AUT* approach.

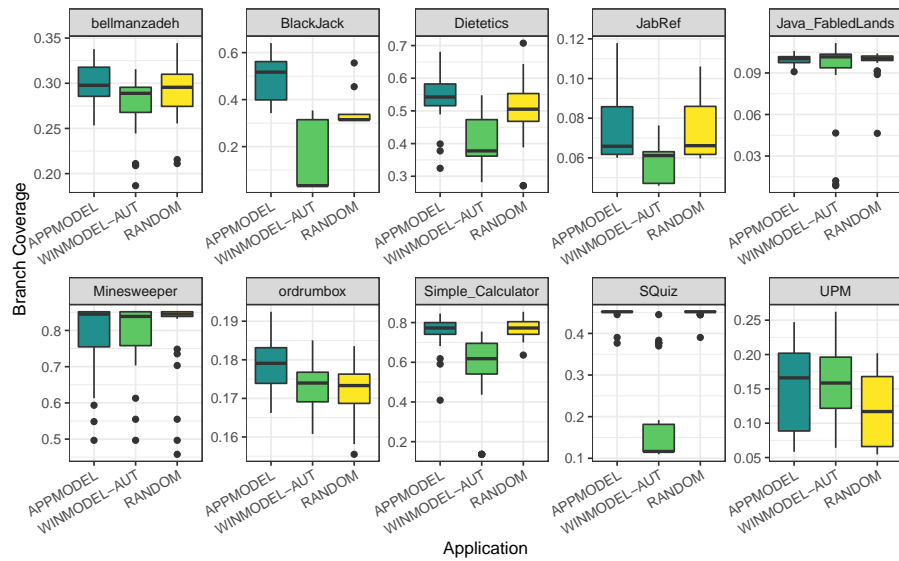
coverage in the function responsible for handling decimal points.

The second part of this question involves a model trained on different applications than the one under test. Using *WINMODEL* achieves a significantly higher coverage in seven of 10 applications when compared to using *WINMODEL-AUT*. This is not surprising, as using data derived from the subject under test creates models more targeted and specific for that subject, being able to interact with the specific widgets on each screen quicker than a model without the application specific knowledge. Figure 4.5 shows a large number of cluster centroids for a basic GUI, and with this large number, the chances of interesting events being generated is diminished.

RQ4.1: In our experiments, using unique user models per application window to guide a GUI tester led to tests with a significantly higher coverage when compared to a single model per application or unique window models trained on many applications other than the AUT, with an average coverage increase of 14.0% and 35.8%.

RQ4.2: What is the best approach to generate GUI tests when encountering a window for which no user data was available to create a model?

Figure 4.6 shows the branch coverage achieved by techniques *APP-MODEL*, *RANDOM* and *WINMODEL-AUT*. The difference between


Figure 4.6. Branch Coverage of techniques *APPMODEL* and *RANDOM*

Application	APPMODEL	RANDOM	APPMODEL	WINMODEL-AUT
bellmanzadeh	0.298	0.296	**0.298	0.289
BlackJack	***0.517	0.315	***0.517	0.034
Dietetics	**0.543	0.505	***0.543	0.378
JabRef	0.066	0.066	***0.066	0.061
Java-FabledLands	0.100	0.100	0.100	0.102
Minesweeper	0.845	0.845	0.845	0.839
ordrumbox	***0.179	0.173	***0.179	0.174
Simple-Calculator	0.773	0.773	***0.773	0.618
SQuiz	0.452	0.452	***0.452	0.116
UPM	**0.166	0.117	0.166	0.158
Mean	0.383	0.361	0.383	0.282

Table 4.7. Branch Coverage of techniques *APPMODEL*, *RANDOM*, and *WINMODEL-AUT*. **Bold** indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).

the *APPMODEL* and *RANDOM* techniques is the method of generating interactions in an application state not seen during user recording. *APPMODEL* falls back to an application model when coming across a previously unseen window, whereas *RANDOM* instead generates random events. *APPMODEL* achieves a significantly higher coverage in four of 10 applications, and a similar level of coverage in six of 10 applications. This leads us to believe that widget information in a GUI application may be influenced by other windows (i.e., some form of

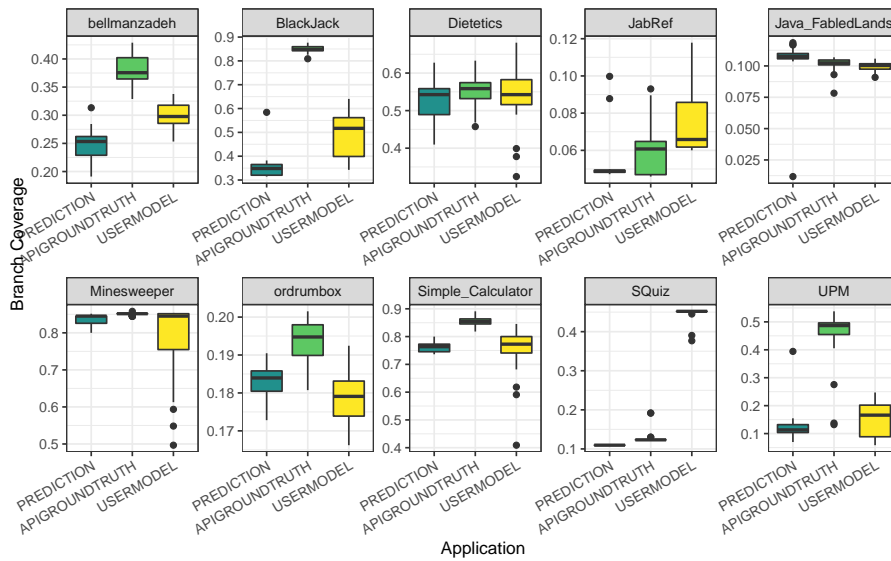


Figure 4.7. Branch Coverage of techniques *APPMODEL* , *PREDICTION* , and *GROUNDTRUTH* .

template exists for each window), and that if a widget is in a certain place in one window, it has a higher chance of being in the same place in another window. This is also supported by the result from *Simple Calculator* in the last research question, where the knowledge of a button in the “About” screen increased the probability of a button being pressed in the main screen when the user data was aggregated.

The *WINMODEL-AUT* technique is capable of generating tests for unseen application windows, and contains a large amount of data. However, it again under-performs compared to both other testing techniques. The *APPMODEL* technique achieves the greatest branch coverage and will be used later to compare models learned from user data against other approaches. *USERMODEL* will be the name used to refer to *APPMODEL* from this point.

As discussed in the previous chapter, there are testing techniques which can work without user data and these could be used on unseen ap-

plication states. Figure 4.7 shows the branch coverage achieved by the test generator *USERMODEL* which uses a model based on user data against two approaches from the last chapter: *PREDICTION* and *GROUNDTRUTH*. Table 4.8 shows a statistical comparison of these techniques.

Application	USERMODEL	PREDICTION	USERMODEL	APIGROUNDTRUTH
bellmanzadeh	***0.298	0.253	0.298	***0.376
BlackJack	***0.517	0.348	0.517	***0.848
Dietetics	0.543	0.543	0.543	0.559
JabRef	***0.066	0.049	***0.066	0.061
Java-FabledLands	0.100	***0.108	0.100	***0.102
Minesweeper	0.845	0.845	0.845	***0.852
ordrumbox	0.179	***0.184	0.179	***0.195
Simple-Calculator	0.773	0.764	0.773	***0.855
SQuiz	***0.452	0.110	***0.452	0.123
UPM	0.166	0.113	0.166	***0.487
Mean	0.383	0.331	0.383	0.442

Table 4.8. Branch Coverage of techniques *USERMODEL*, *PREDICTION*, and *APIGROUNDTRUTH*. **Bold** indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).

Using the *USERMODEL* leads to tests with a significantly higher branch coverage in four of 10 applications, and significantly lower in two of 10 compared to *PREDICTION*. When compared to the *GROUNDTRUTH* approach, which extracts exact coordinates of widgets from the Java Swing API, *USERMODEL* achieves a significantly higher branch coverage in two of 10 applications, and significantly lower in seven of 10. For the applications “JabRef” and “SQuiz”, using a user model achieves a significantly higher branch coverage when compared to both other approaches. As discussed in the last chapter, the *PREDICTION* and *GROUNDTRUTH* approaches struggle with JabRef due to a button opening a modal pop-up which locks the main window, and using a user model has a much lower probability of triggering this event as users rarely interacted with the application in this manner, instead creating a new document. SQuiz uses a menu that controls everything in the application, from starting a new quiz, to viewing the high score-

board. Using the sequence data from users aided to achieve a higher coverage than both other approaches as interaction with elements in the menus requires sequential events to be triggered. First, the menu header needs to be clicked, then an element in the newly expanded menu needs to be pressed. This has a higher probability to occur when exploiting information stored in sequences of user events, compared to the other approaches which have a high probability of clicking a widget outside the expanded menu.

The applications where widget detection performs better are *ordrumbox* and *Java-FabledLands*. These applications swap the contents of the screen without changing the title of the window. This negated the benefits of using models based on unique window titles. For example, *Java-FabledLands* opens a window with a large amount of text, with links embedded in the text. The text changes when a link is pressed. When using a widget detection approach, these links can be identified as they have a unique appearance from the rest of the text. However, a user model based approach has no method of tracking what state it is in, and struggles to interact with the text. This is only exacerbated when clustering the position of user clicks is introduced, moving possible points of interaction further away from the required position. This is similar with *ordrumbox*, where windows are introduced, but are painted to the buffer of the current window so the title remains the same. The user model approach loses track of what it can interact with in the window rapidly as it has no feedback from the application.

It is no surprise that *GROUNDTRUTH*, which uses the known widget locations of all GUI elements outperforms a user model based approach which is generating events “in the dark”, with no insight into the response of applications to generated events other than the title

of the currently focused window. Users can click buttons in different locations, and clustering leads to some user positions being slightly adjusted which can make the difference between triggering the event bound to a button, or not. It is clear that using an API based approach to generate tests is superior, but as discussed earlier, there are still applications where a user model performs better. Also, as stated in the previous chapter, there are applications which are not supported by an API based technique, and using a user model could be beneficial to test these applications, if user data is available.

RQ4.2: The APPMODEL technique achieves a significantly higher coverage than using RANDOM , with an average coverage increase of 6.1%. This is the highest performing model learned from user data, and will be referred to as USERMODEL for future comparisons. The new USERMODEL approach lead to tests with a significantly higher coverage compared to PREDICTION in four of 10 applications, and significantly worse in two of 10. Unsurprisingly, GROUNDTRUTH performed significantly better in most cases than other approaches.

RQ4.3: Does a GUI testing approach in which interactions with GUI components are guided by a model based on user data provide better code coverage than an approach in which interactions are purely random?

Figure 4.8 shows the branch coverage achieved by three techniques: *RANDCLICK* , *RANDEVENT* and *USERMODEL* . Table 4.9 shows a pair-wise statistical comparison of these approaches.

We can see from this table that the *USERMODEL* approach achieves a significantly higher branch coverage than both randomised approaches.

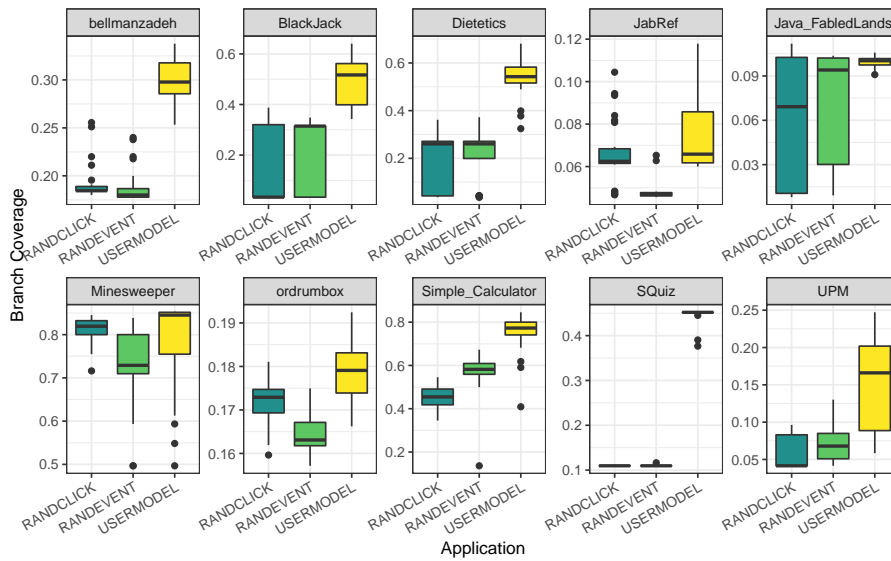


Figure 4.8. Branch Coverage of techniques *USERMODEL*, *RANDEVENT*, and *RANDCLICK*.

Application	USERMODEL	RANDEVENT	USERMODEL	RANDCLICK
bellmanzadeh	***0.298	0.180	***0.298	0.184
BlackJack	***0.517	0.315	***0.517	0.034
Dietetics	***0.543	0.261	***0.543	0.261
JabRef	**0.066	0.047	**0.066	0.062
Java-FabledLands	**0.100	0.094	**0.100	0.069
Minesweeper	***0.845	0.729	**0.845	0.819
ordrumbox	***0.179	0.163	***0.179	0.173
Simple-Calculator	***0.773	0.582	***0.773	0.455
SQuiz	***0.452	0.110	***0.452	0.110
UPM	***0.166	0.068	***0.166	0.042
Mean	0.383	0.241	0.383	0.228

Table 4.9. Branch Coverage of techniques *USERMODEL*, *RANDEVENT*, and *RANDCLICK*. **Bold** indicates significance. * Indicates Effect Size (*Small, **Medium and ***Large).

This is due to the user model approach having a higher probability of useful interactions compared to a randomised approach, and therefore it spends less time performing uninteresting actions on the application under test. When compared to the randomised approaches, *USERMODEL* achieves a significantly higher branch coverage in 10 of 10 applications. There are some differences between *CLICKRAND* and *EVENTRAND*, but these are minor and application specific.

RQ4.3: In our experiments, using a user model to guide a GUI tester lead to tests with a significantly higher coverage when compared to randomised approaches, with an average increase of 58.9% and 68.0%.

4.6 Conclusions

There are currently limited techniques of interacting with an application without exploiting some underlying data structure of the GUI. In this chapter, we investigated the creation of models from real user interaction data with the application under test.

We found that the best approach to generating interaction data is by splitting the user data and creating separate models targeting different windows of an application. If a previously unseen window is found, it is best to fall back to an aggregated version of the window models (i.e., a model generated from data of the entire application under test).

We also found that models generated from user data can outperform the widget detection approach outlined in the previous chapter. This is rather interesting, as the user models approach are generating data “in the dark”, with no insight into an application’s reaction to generated data. However, collecting user data to generate models is very expensive compared to the widget detection approach, which requires only a screen shot.

The most expensive approach, *GROUNDTRUTH*, relies on the application using a supported GUI framework or exposing the location of its widgets through an accessibility API. This expensive approach, unsurprisingly, still achieves the highest branch coverage, using feedback from the application and wasting very little interactions on uninterest-

ing events.

There is still much work to investigate:

- By combining user models with widget detection, it may be possible to merge the advantages of both approaches. The random widget selection of the detection based approach could be guided by the bounding boxes that users interact with, and the state inference weakness of using a window title could be improved by widget detection by including widget information to infer unique states.
- Using an application model as fall back achieves a higher branch coverage than using a random technique as fall back. This suggests that there is some common structure in GUIs and that interactions in one GUI screen state can aid in other screen states. This needs further investigation to identify the relationship between interactions at a window level and interactions at an application level.
- The clustering technique for user interactions can often generate cluster centroids that are not actually inside the interaction box of a widget. Further investigation is needed in order to develop a non-destructive clustering technique that maintains information about the original points of interactions.
- It may be possible to use other GUI testing tools to gather an initial set of user data. Tools such as FSM Droid [129] or Swift-Hand [32] explore Android applications and dynamically build a probabilistic interaction model. Using the interaction data generated from these tools may aid in gathering user data and improving the models created in this chapter.

5 Testing By Example: Natural User Interfaces

This chapter is based on the work “Modelling Hand Gestures to Test Leap Motion Controlled Applications” published elsewhere in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2018*.

5.1	Introduction	141
5.2	Natural User Interfaces	145
5.3	Modelling Leap Motion Data	149
5.4	Generating Leap Motion Data	153
5.5	Evaluation	157
5.6	Discussion	172
5.7	Conclusions	175

5.1 Introduction

Natural User Interfaces (NUIs) present a difficult problem for test data generators. The data required by applications which use such devices is derived from the real world. Generating data resembling real usage

of the NUI is a challenge. Random testing techniques are unlikely to generate this data, and creating a constraint-based model is expensive and will only apply to a single target NUI, as each unique NUI uses complex and unique data structures.

To aid in data generation, we look at an approach similar to the last chapter, which learns from data provided by end users and generates models capable of generating statistically similar data.

Natural User Interfaces (NUIs) allow users to interact with software through methods such as body tracking, gestures, or touch interfaces [149]. They are a primary input source for virtual reality applications, which are increasing in popularity. NUIs are also crucial for interacting with computers in environments where using a keyboard and mouse is not an option (e.g., surgeons navigating between x-ray images in an operating room). Testing applications controlled by NUIs is a challenge: no frameworks exist to automatically generate test data meaning that manual testing is the main form of testing. This requires developers having to interact with the application for every new feature developed. This increases the chance of regressions occurring in the application's code base.

Testing manually can be repetitive, and there is an increased chance of bugs making it into the final system when it is the only form of testing. To aid in testing an application, tests can be generated and ran against an application. For example, there are tools that can automatically generate tests for many different types of applications [45, 105, 91], lessening the testing effort of designing test data and test cases.

Typically, standard software consists of API calls or the event driven paradigm when interacting with GUI widgets. However, the inputs

expected by NUI applications are much more challenging to produce automatically. For example, the input for an application controlled by a Microsoft Kinect input device consists of a collection of points in 3D space which collectively represent the body of the program's user; the input for an application controlled by a Leap Motion input device consists of data representing the user's hand and finger joint positions in 3D space. The data structures contain complex relationships (e.g., each body part needs to be connected at the right place). Here lies a challenge for test data generators in generating data which abides by the constraints in the NUI API. Documentation for the API usually does not give a list of the constraints, or how data is derived from the input, only increasing the challenge of data generators.

One technique to generate test data automatically is to learn models of realistic user input, and then to sample these models for new sequences of input. Hunt et al. [65] demonstrated that this approach can be effective at generating test data for a web browser application which used the Microsoft Kinect as input. However, previous work focused only on one aspect of the Microsoft Kinect input, the body joint positions.

Throughout this chapter, we extend the approach by Hunt et al. applying it on the Leap Motion controller, which derives data about a person's physical hand including: hand positions, finger positions, finger gestures, and various other aspects. The data from the Leap Motion is more complex with more constraints, increasing the test generation problem. We present a framework to apply NUI test generation to applications based on the Leap Motion, and evaluate our approach on five candidates taken from the Leap Motion app store.

Using only a single data model, as Hunt et al. did, to snapshot all the different aspects of the NUI input data at a single point in time may

not be the most effective approach. For example, in the Leap Motion Controller, the hand movement and finger joints shape are encoded together, but training one model on the combined result eliminates the possibility of identifying similar finger joint shapes at different positions in 3-D space. As the hand moves through time, it may form common shapes but at different positions, and encoding the position and the shape of the hand separately allows generation of specific hand shapes at many more positions in 3-D space.

In order to evaluate the benefits of representing the complex NUI data with multiple models, we present a methodology in which we split the NUI data into subsets, and learn separate models for each subset. In our experiments we contrast test data generated from these *multiple models* with data generated from a *single model* of the input data.

In detail, the contributions of this chapter are as follows:

- A framework to model hand interactions, and automatically generate and replay test cases for the Leap Motion NUI.
- An empirical evaluation of NUI testing on five applications controlled by the Leap Motion controller.
- An empirical evaluation of the influence of the training data on the resulting code coverage.
- An empirical comparison of generating NUI data from multiple models vs. a single model.

Our experiments show that our approach to automated NUI testing can handle the complexity of the Leap Motion controller well, and produces sequences of test data that achieve significantly higher code cov-

erage than random test generation approaches. We show that the training data has a large influence on these results, while the benefits of splitting NUI data into multiple models are small and application dependent.

5.2 Natural User Interfaces

Natural User Interfaces (NUIs) provide a means of controlling software by recording a continuous stream of data that represents the position or motion of the user's body. For example, the Microsoft Kinect employs a depth camera that allows a user to interact with an application through body tracking. Similarly, the Leap Motion Controller is a small desktop device which tracks the hand and finger positions of the user, thus providing a natural interface in which the user can point, draw or gesture with their hands.

NUIs have been used to solve a range of problems. For example, The Microsoft Kinect has been used in medicine for effective stroke rehabilitation, giving doctors access to the body profile of patients from a patient's own home, and making exercises more fun and motivating for patients [150]. The Leap Motion has been used for controller robotic arms [128] and in many Virtual Reality (VR) applications, being mountable on many VR devices.

NUIs rely on a user's existing knowledge for interactions with applications: if a menu is on the screen, it is intuitive to reach out and touch a desired button for progression through an application. Although intuitive for real users, NUIs are difficult to test with an automated approach. Another issue for test generators is the complexity of data pro-

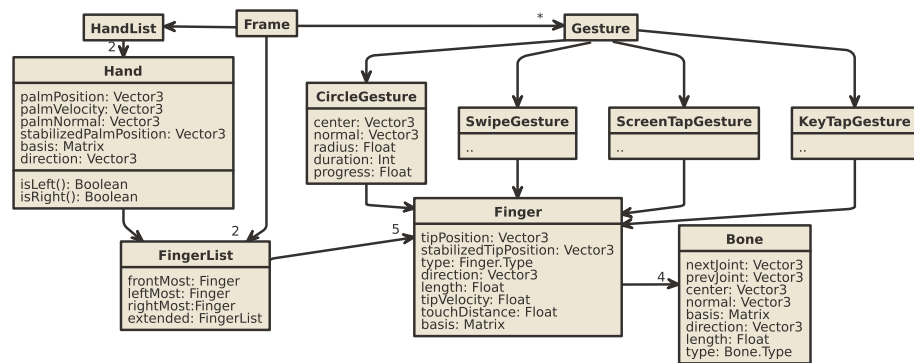


Figure 5.1. The data structure used by the Leap Motion Controller

vided by NUIs. Figure 5.1 shows the data structure received from the Leap Motion API. A *Frame* has many components, and it is not immediately clear how to create a *Frame* for testing purposes. Mostly, NUI applications are manually tested by the developer of an application, and only recently was support for capture and replay tools implemented. It may be beneficial to automatically generate test data, and there exists work in generating test data for other kind of Natural User Interfaces.

5.2.1 Natural User Interface Testing

Mobile applications use combinations of regular program inputs (e.g., via touch displays), and NUI inputs (e.g., via external sensory data). To test mobile applications, Griebe et al. describe a framework in which location information [55] and accelerometer data [56] can be replaced with mocked data by developers.

Hunt et al. automatically generated test sequences for the Microsoft Kinect [65]. To generate data, Hunt et al. trained models on data recorded by users. Similarities in the data are identified through clustering, and sequences of clustered data are used to generate a Markov chain. The Markov chain is a probabilistic model that can be used to de-

cide which cluster to seed next during test generation. Clustering was performed on all features of the data structure but this assumes that all features have a static (time-unconstrained) relationship.

Hunt et al. used branch coverage to assess the effectiveness of different NUI data generation methods. The application under test (AUT) was a web browser adapted for Kinect support. Hunt et al. found that using a purely random approach for generation, i.e. using randomly sampled values for each variable in the data structure, performed the worst. Second was an approach involving seeding randomly sampled processed user data. To increase performance further, Hunt et al. generated an n-gram model from the sequence information collected when recording data and used this model during data generation. A single model of NUI data may link different independent aspects of the user movement, resulting in biased test generation. For example, if a NUI was to capture a person running, the body gestures (i.e., the shape of the body relative to the centre point of the body, without location taking into account) observed may be a repeated sequence of body positions, but the actual data will never be repeated as running displaces the body in 3-D space. Potentially, representing the body location and the joint movement as separate models could be a more effective means of generating new, realistic test input not observed in the training data. New, realistic data will still resemble the body part being tracked and also moves smoothly through time. This approach may be particularly important in the case of NUIs where many potentially independent features are present in the common input data structure.

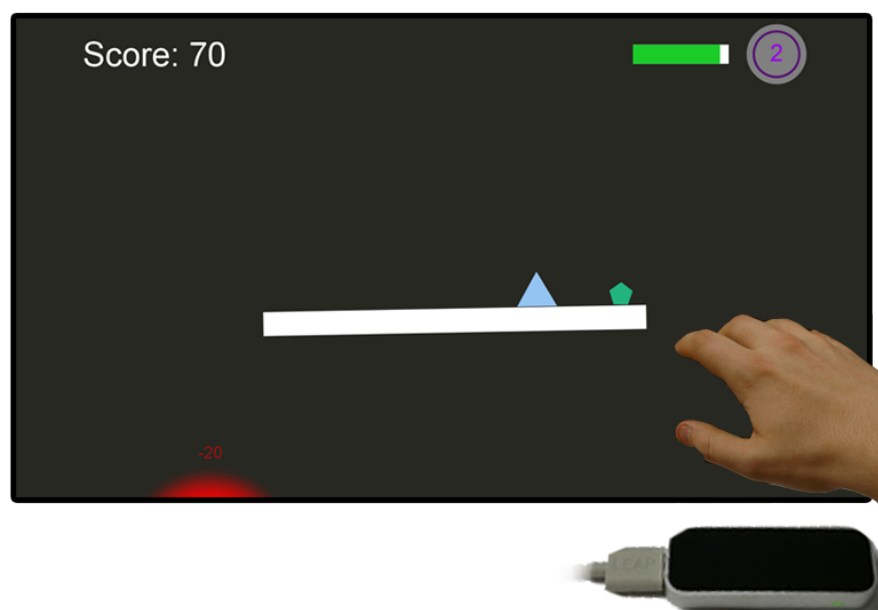


Figure 5.2. Interaction with the application “PolyDrop” through the Leap Motion

5.2.2 Leap Motion Controller

The Leap Motion Controller is a NUI which tracks a user’s hand movements and gestures. The device (see Figure 5.2) is placed on a desk, and users place hands above the device to interact with software. The *Controller* tracks properties of a hand such as position in 3-D space, the location of all joints in each finger, the position of the tips of each finger and many other things. Each data frame received from the Leap Motion Controller contains a snapshot of the user’s hands at the current time, providing data up to 200 times per second. Because applications expect data at this rate, it is important that testing techniques can match this speed, whilst generating realistic data. Expensive overheads for testing techniques when generating data could hinder the performance of a technique, providing less frames to an application than expected. The Leap API gives applications a complex relational data structure.

The top level of the structure is a *Frame*, which contains all relevant information observed by the Leap at the current time. However, some aspects of the structure are not only reliant on the current time of capture. For example, to interact with 2-D applications, a developer replaces the virtual cursor of the mouse with the *Tip Positions* of each finger. However, there is also a *Stabilized Tip Position* for each finger which returns a smoothed version of the fingers tip position, directed at 2-D application interaction, and updates according to the speed which the finger tip was moving. Stabilized positions allow for more consistent 2-D GUI interactions, specifically with micro movements, but how the values are calculated does not appear in the API documentation. Due to a lack of documentation with how some values are calculated in the Leap Motion API, it may be beneficial to learn from real data observed in user interactions with applications. This eliminates the need for reverse engineering of the algorithms producing these values.

5.3 Modelling Leap Motion Data

We split the Leap Motion data into 5 parts, where each part is modelled using an n-gram model. An n-gram model represents the probabilities of one element following the n previous elements in a sequence of data [104]. Using such models has distinct advantages and disadvantages in testing compared to manual testing by users. Once created, a model is cheaper to execute than having a developer manually test an application, and can generate long sequences of test data without tiring. However, a model is only as good as the data used to train it, and may not generalize to novel kinds of interaction that were not encountered during training. This may limit the extent to which a NUI application

can be explored by model-based test generation.

This section shows how user data from the Leap Motion is split into five separate models, each model representing a unique aspect of the Leap Motion data structure. The five models are as follows:

- **Position:** The 3-D position of the palm, relative to vector (0, 0, 0) in Leap Motion Controller space. This is the physical position of the hand in 3-D space. Position data is denoted in Figure 5.3 by three axes and a point labelled with (x, y, z).
- **Rotation:** The rotation of the palm, stored as Euler angles by the Leap Motion, we convert to quaternions for modelling. A quaternion is a 4-D unit vector that represents an object's rotation in 3-D space. Rotational data is denoted in Figure 5.3 by a circle with an arrow through, representing the quaternion angle of rotation.
- **Joints:** The 3-D position of each bone joint in the fingers of each hand, respective to the palm position. All fingers were stored in the same feature to preserve anatomical constraints between fingers. Joints are denoted in Figure 5.3 by the circles on the fingers of hands.
- **Gestures:** The sequence of pre-defined Leap Motion gesture types performed by the user (Circle, Swipe, Key tap and Screen tap). This is also split into four child models, one per gesture type. For the applications tested here, only the circle gesture is used, which triggers when a *Finger* performs a circular motion. A circle gesture consists of a circle centre, normal, radius and the duration that the gesture has been performed for. Circle gestures are denoted in Figure 5.3 by a green circle with an arrow.

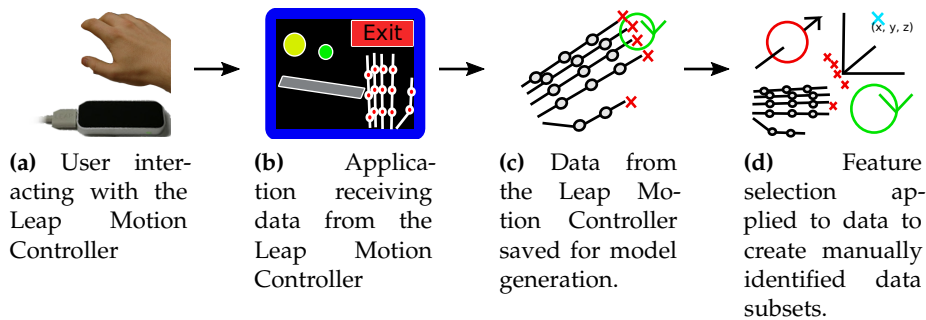


Figure 5.3. Recording user interactions with the Leap Motion Controller and splitting data into subsets.

- **Stabilized Positions:** Each hand also has stabilized data, which are vectors targeted towards 2-D menu interactions and rely on time. One example are stabilized tip positions for the tips of each finger, being a variable amount of time behind the actual hand data. Stabilized positions are stored in a separate model to preserve 2-D interactions. Stabilized data are denoted in Figure 5.3 by red "X"s representing the stabilized tips of each finger.

For the five models defined, we use user data to generate models. Figure 5.3 shows how user data is stored as separate data subsets, one subset per model. User Recording is the process of capturing user interaction with the Leap Motion and hence the application under test. We intercept this data and use it to train models. First, the data is split into data subsets. This involves applying feature selection [126] to the data and training each model on the selected features.

5.3.1 Model Generation

For each data subset, the same technique is applied to generate models. Firstly, the volume of user data is reduced using K-means clustering.

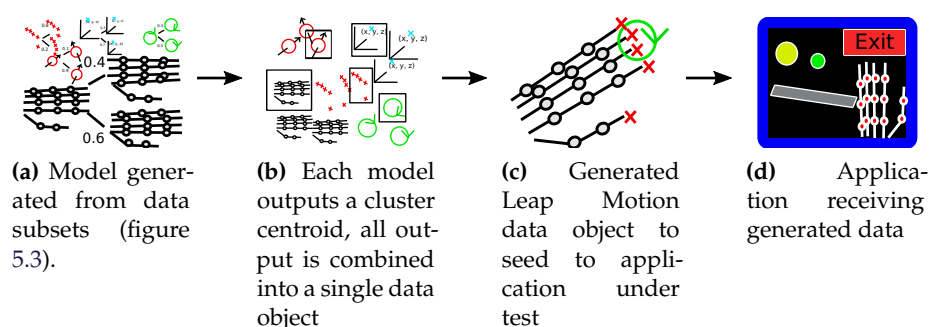


Figure 5.4. Our approach: generating features before combining them into a realistic data object.

This is similar to the technique from the previous chapter. K-means clustering groups together related records by Euclidean distance, using all features in the calculation. The result labels each record with a cluster $0..k$ where the label is the cluster with the nearest centroid (mean of all elements in the respective cluster).

Each record is now labelled but the quantity of data has not changed. To reduce the data, we substitute each record with the centroid of the assigned cluster. This reduces the total amount of user data to K centroids.

As with the last chapter, the chronological sequence in which each record was received is stored when recording user data. This sequence can be replaced by the assigned cluster labels and used to train an n-gram model, a model containing the probabilities of all transitions of length N in a sequence.

To generate an n-gram model, a probability tree is constructed from sequences of data. The tree is of depth N and contains all transitions of length N from one element of the sequence to other neighbouring elements. For example, assume that the cluster label sequence is as fol-

lows: 1,2,1,2,1,3. Using $n = 2$, the probability of observing a record in cluster 2 following a record in cluster 1 is $2/3$, and the probability of observing 1 after observing 2 is 1.0. Values of K and N were chosen through parameter tuning. As each model was trained on the target application, so were the values of K and N. To tune K, values were sampled from 100-1200 clusters and used to test the target application by recreating the original user data but using clustered data. The value of K with the highest coverage was used in experiments. A similar approach was used with N, but with values 2-5.

Each Leap Motion data frame can have an undefined amount of gestures, linked to different fingers i.e. it is possible for a single Leap Motion frame to have three circle gestures and a swipe gesture. We use an additional n-gram model to decide which gestures go in which frames. Specifically, the gesture n-gram model gives the following information: when to start and stop a gesture; which finger each gesture should be linked to; and the types of each gesture.

5.4 Generating Leap Motion Data

Once models are trained, Figure 5.4 shows how data produced from each model can be recombined into valid Leap Motion data. Each model produces a cluster centroid for the area of the Leap Motion data structure the model is representing. The centroids from all models are combined into one data object, and seeded back to the application during test generation. This is where our approach differs from the technique by Hunt et al. [65], which only uses a single model to reconstruct data. Our approach to testing NUI applications has the advantage that generated data still resembles the original user data, but is also diverse

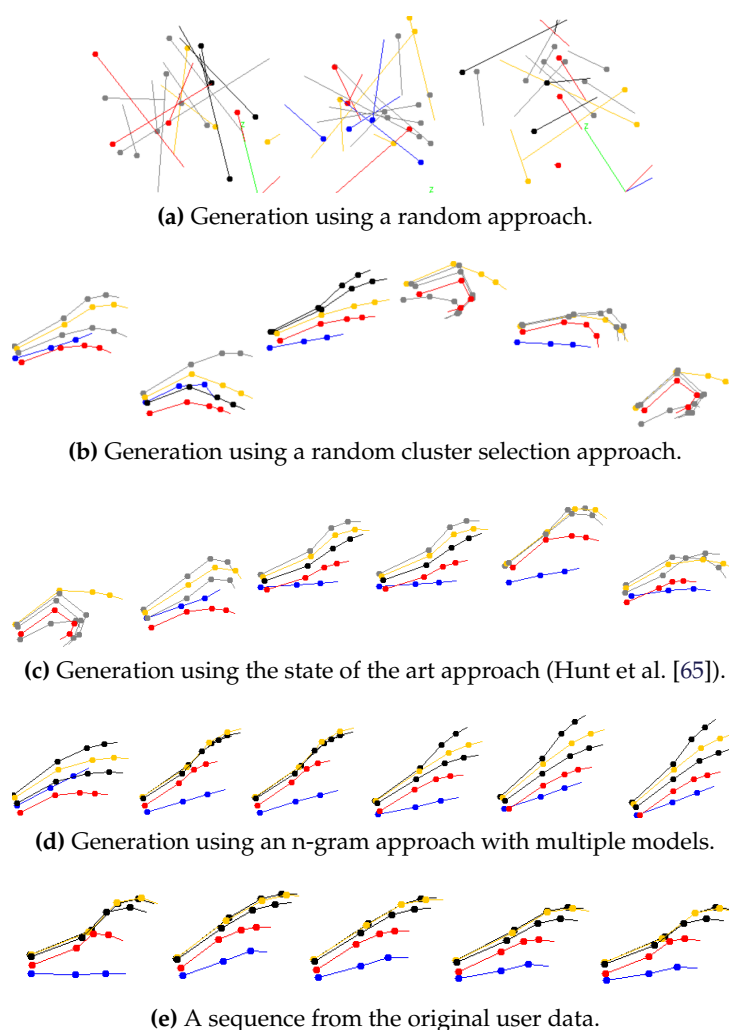


Figure 5.5. Sequences of hands generated as input data using different techniques

enough to test parts of a program not necessarily tested by users. We identify common patterns in the user data per model, retaining some relationships that would otherwise be lost when using a single model.

Using the models generated, we propose three methods of generating mock Leap Motion data:

1. Random: Sample numbers in the Leap Motion’s view range do-

main for all properties of all features. See Figure 5.5a for an example sequence of hands generated using this approach.

2. Random Clusters: Randomly select a cluster for all models and seed the centroid of the clusters. This produces realistic Leap Motion data at a single point in time, but not over time. All time-related data is discarded producing ‘mechanical’ hands with no animation. See Figure 5.5b for an example sequence of hands generated using this approach.
3. N-gram Model Generation: Use the generated n-gram models to select the next cluster centroid to seed. This preserves time-related data, but using separate models eliminates the static relationships between models preserved in the single model method by Hunt et al. However, the benefit is that a more diverse range of data can be produced, e.g., a single hand shape at various positions in the Controller’s 3-D space can be generated, including positions that the user did not provide for the respective hand shape. See Figure 5.5c for example data generated using a single model (Hunt et al.) or Figure 5.5d for an example using multiple models.

Each technique reconstructs hands using the following method: 1) generate model data in isolation by selecting cluster centroids; 2) combine generated features into single data objects. Using an n-gram model produces a sequence of data that are statistically similar to the order of hands seen during user recording. This has a higher probability of generating realistic sequences of animated data (e.g., a closed hand slowly opening). In contrast, selecting random clusters produces more uniform data but with no animation (e.g., the hand will instantly open in 5 milliseconds, from one from to the next).

5.4.1 Executing Leap Motion Tests

Our technique generates tests for applications which use the Leap Motion Controller [81]. The Controller allows interaction with applications through hand tracking. The Leap API supports many target source code languages, and works through a background service installed on a machine, which provides a continuous stream of data to applications registered as listeners.

Our framework functions as a layer that sits between the application under test and the Leap Motion background service, replacing the Leap Motion's stream of data with automatically generated data. We use a full mock of the Leap Motion's Java API. During test generation, when applications register as a listener for the Leap Motion, our framework now provides a stream of data in place of the Leap Motion background service.

To save tests, we store the ordered cluster labels of each model and the execution time which the generated data frame was seeded to the AUT. Replaying a test involves using these stored cluster labels to select the cluster centroids for all models at the appropriate point in time, before combining all centroids into a data frame. Tests produced by our tool currently produce sequences of hands that can be played back into an application. This is useful for regression testing: ensuring that the current program state after seeding data on the modified application is equal to the state seen during generation. However, in this chapter, we do not focus on the problem of state inference.

5.5 Evaluation

To study NUI testing on the Leap Motion in more detail, we investigated the following research questions:

- RQ5.1: How beneficial is NUI testing with n-gram models generated from real user data when generating tests for Leap Motion applications?
- RQ5.2: How does the quantity of training data influence the effectiveness of models generated for NUI testing?
- RQ5.3: How beneficial is a model with separation of NUI data, which can generate parts of the data structure in isolation, compared to a model trained on a whole snapshot of the NUI data?

5.5.1 Experimental Setup

To answer RQ5.1, we compare the test generation techniques seen previously in in Figure 5.4, i.e., random test data, random clusters, and n-gram based test generation. For the Microsoft Kinect, Hunt et al. [65] observed that the use of an n-gram model resulted in substantial code coverage increases over the random baselines, and the main question is whether this effect can also be observed on Leap Motion applications, where input data is more complex than on the Microsoft Kinect.

To answer RQ5.2, we compare models created using only a single user's data, against models created using data from many users. Intuitively, assuming an equal value of K when clustering, using data from many users should lead to n-gram models which are less sparse, and have

a higher diversity in the set of centroids. However, anatomical differences (e.g., different hand sizes) could have unexpected effects in the clustering process. To evaluate the effect of user data in test generation, we use the n-gram model technique with multiple models.

To answer RQ5.3, we evaluate the effects of splitting the Leap Motion data structure into multiple models. The baseline is the approach outlined by Hunt et al. [65] for the Microsoft Kinect, i.e., creating a single model with the complete Leap Motion data structure interpreted as a flattened vector of features. To evaluate the effectiveness of splitting Leap Motion data into multiple models, we use the n-gram model generation technique with models trained on data from many users for each application.

Our metric for comparison is line coverage; the amount of lines executed in an application divided by the total lines of the application. We measured line coverage using instrumentation provided by an open source tool¹. To test for significance, we used a Wilcoxon rank-sum test, with a significant result occurring when $p < 0.05$. To find the better approach, we use a Vargha-Delaney \hat{A}_{12} effect size, with a value trending towards 1 indicating an improvement over the baseline, 0.5 being no improvement and trending towards zero being a negative impact. To account for the random nature of our generation techniques, we ran each configuration for 30 iterations [7], and the code coverage achieved at the end of a one hour period was used in comparisons.

A one-hour generation time was selected as by this time, most techniques were stuck in a certain application state, unable to progress. However, coverage increases were still being attained in certain applications. To select the value of K for clustering, tests were generated

¹<https://github.com/thomasdeanwhite/Scythe>

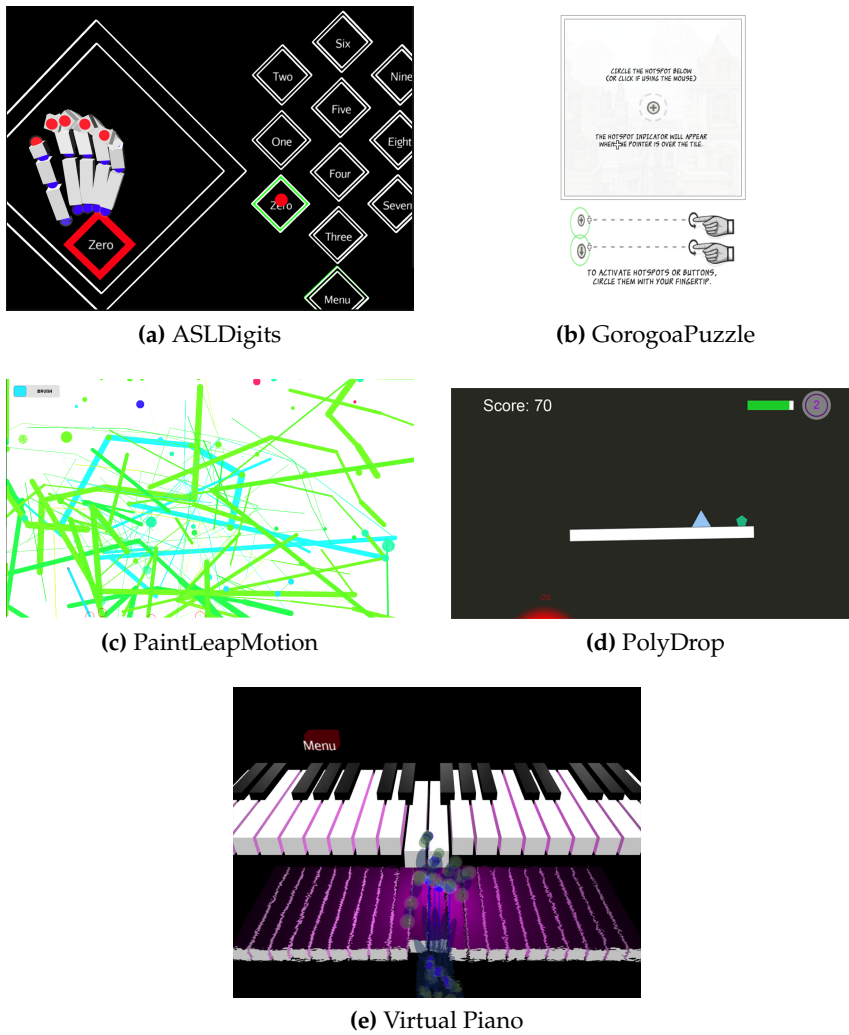


Figure 5.6. The five applications under test, used to empirically evaluate our framework.

using predefined sets of clusters between 200 – 1400 and the value of K achieving the highest coverage was used in experiments. The value of N was tuned in the same manner but for values between 2 and 4. As data is recorded separate for each application, values are also tuned separate.

For evaluation, we chose five applications: four from the Leap Motion

Airspace Apps Marketplace, and one open source application.

- ASLDigits (Figure 5.6a) is an educational game teaching American Sign Language for the numbers 0-9. There is also a game in which users score points for using the correct signs for numbers displayed in a given time limit. ASLDigits contains around 4213 Lines of Code (LOC)
- GorogoaPuzzle (Figure 5.6b) is a puzzle game where unique interactions are performed with the Leap Motion in order to advance the story, and thereby move to different program states. GorogoaPuzzle contains around 19633 LOC.
- PaintLeapMotion (Figure 5.6c) is an open source app published on GitHub. This application allows users to paint onto a canvas with a selection of tools using the Leap Motion. PaintLeapMotion contains around 1579 LOC.
- PolyDrop (Figure 5.6d) is a physics game in which blocks fall on to the screen and the player needs to catch them on a bridge controlled by Leap Motion interaction. PolyDrop contains around 8212 LOC.
- Virtual Piano for Beginners (*VPfB*, Figure 5.6e) is an application which allows users to play an “air piano”. There is a free play mode and also an educational mode which teaches users to play certain songs. *VPfB* contains around 2276 LOC.

We chose these applications due to their variety of use with the Leap Motion API. These applications include use of the gestures API, 2-D menu interactions, advanced processing of the Leap Motion data structures and other areas. The applications are also dissimilar to one an-

other. The only information that our technique has of each application is the data from the Leap Motion background service when user interaction occurred.

Data was recorded from five users for each application. The users first practiced interacting with the Leap Motion on the “Leap Motion Playground”, a training app provided by Leap Motion. Then, users explored each application in sequence for five minutes. When recording user data in the last chapter, we provided them with a set of tasks to complete. For these experiments, we did not instruct them to perform specific tasks with the applications but allowed them to freely explore applications.

5.5.2 Threats to Validity

We chose a subset of available applications which use the Leap Motion Controller. To decide if our framework was applicable to an application, we use the following criteria: 1) the applications must be in Java, and use the Leap Motion Java API; 2) the application must be available publicly, either on the Leap Motion Airspace Apps Store² or open source. The applications chosen use different areas of the Leap Motion API. Some applications, like PolyDrop, make use of the stabilized vectors for menu interactions, whereas others like ASLDigits use the raw finger joint positions. Only GorogoaPuzzle uses gestures, and only a circle gesture. The variance in usage of the API means that our technique can be used on a wide range of applications which use the Leap Motion Java API.

A threat to external validity is whether the data used in training models

²The Airspace App Store closed in June 2017

is representative of data that actual users would provide. To mitigate this, we use data from five users, each interacting with the application under no guidance. Users were given a short training session on how to use the Leap Motion Controller, but not on how to use each application. Users recorded data for five minutes per application, with breaks in between each app. It is possible that the order of data recording gave users a chance to learn more about the Leap Motion Controller and improve usage on later applications. The order in which application data was recorded changed per person to mitigate against this.

As we are recreating and mocking an API, there is a question as to whether our mimic API represents the real Leap API. The version of the Leap API used for these experiments does not support replay of data, so playback of data through the physical device cannot occur, therefore the mock API must be used. The Leap API is sparsely documented, and it is infeasible to recreate the API exactly without knowledge and calculations that are missing from the documentation. To mitigate against this threat, we have techniques of reconstructing the raw data using our API before clustering occurs and we ensure that the reconstructed data seeded through our framework performs similar to the original user data.

With any developed software there is a potential for faults to occur. To mitigate against this threat, we have a unit test suite and also make all the artefacts available publicly on GitHub³.

³<https://github.com/thomasdeanwhite/NuiMimic/tree/nuimimic>

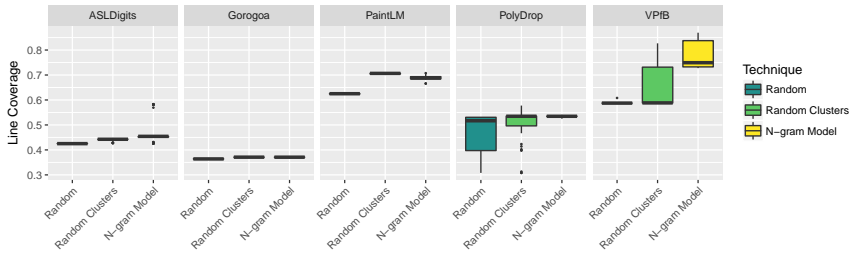


Figure 5.7. Line Coverage for different data generation techniques for each application.

Table 5.1. Code coverage for different data generation techniques for each application. **Bold** is significant ($P < 0.05$).

Application	Random Cov.	Random Clusters Cov.	N-gram Model Cov.	N-gram Model Comparison			
				Random A12	P-value	Random Clusters A12	P-value
ASLDigits	0.425	0.441	0.468	0.963	< 0.001	0.884	< 0.001
Gorogoa	0.364	0.371	0.371	1.000	< 0.001	0.366	0.109
PaintLM	0.625	0.706	0.689	1.000	< 0.001	0.080	< 0.001
PolyDrop	0.459	0.505	0.534	1.000	< 0.001	0.513	0.838
VPfB	0.589	0.663	0.778	1.000	< 0.001	0.849	0.002

5.5.3 RQ5.1: How beneficial is NUI testing with n-gram models generated from real user data when generating tests for Leap Motion applications?

Table 5.1 shows the line coverage achieved by different techniques of data generation. The two right-most columns show the *A12* effect size when comparing the n-gram model technique to random and random clusters respectively. Random generation achieves the overall lowest code coverage, compared to both random clusters and n-gram based generation. Generating random points in 3-D space and inserting them into the Leap Motion data structure generates data that is unrealistic and changes far more rapidly than an application expects. Unsurprisingly, adding structure to the generated data and providing data that resembles a hand performs significantly better than purely random generation. Statistical comparison between n-gram based generation and

random generation shows that the difference is significant in all five cases, as can be seen in Figure 5.7. Using random clusters for test generation leads to substantial coverage increase on all 5 apps. The difference between the random and random clusters approach is that the random clusters approach exploits domain knowledge, selecting random cluster centroids from the model generation stage. Combining these centroids generates data similar to that which the original user provided i.e. real data that the Leap Motion could provide to an application. However, the random approach generates unrealistic hands and is very unlikely to generate something resembling actual human data from the Leap Motion under normal use. This demonstrates how important it is to generate realistic data.

Compared to random clusters, the n-gram based generation adds temporal data (i.e., realistic movements) across time. N-gram based generation allows not only the current hand to appear realistic, but a sequence of hands to be more human-like. This leads to a significant coverage increase in two of the applications. For Paint Leap Motion the use of the n-gram interestingly leads to a significant decrease in coverage; for GorogoaPuzzle and PolyDrop there is no significant change. While overall there is a small average coverage increase, this result justifies a closer look at the individual applications under test. ASLDigits and Virtual Piano for Beginners (VPfB), the applications where the n-gram based approach performs best, use a Java game engine with Leap Motion integration. They both require the hand data to represent specific positions and gestures. For example, ASLDigits uses a machine learning approach to determine if signs are correctly shown, and Virtual Piano for Beginners requires specific hand shapes with minute changes over time. Furthermore, both applications use complex menus which require precise interactions with menu elements. All these as-

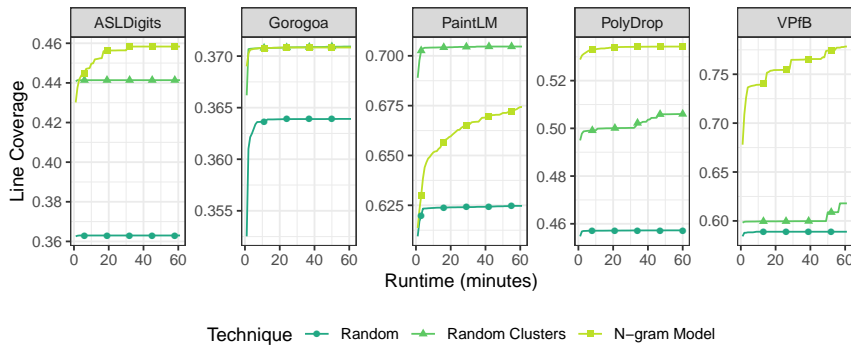


Figure 5.8. Line Coverage for different data generation techniques for each application across time.

pects are more likely to occur with n-gram based test generation, leading to around 114 and 262 more lines of code being covered for ASLDigits and VPfB respectively. PaintLeapMotion, the application where the random cluster technique achieved higher coverage than the n-gram based approach, is a painting application, where users paint on a canvas using hand gestures. While the n-gram based approach generates more realistic hand sequences, these do not matter for this application: PaintLeapMotion only uses the finger tips provided by the Leap Motion API. Users can change tools by moving their hand towards the back of the Leap Motion Controller’s view and selecting a new tool from the pop-up menu. Here is a code snippet from PaintLeapMotion:

```

1 if (minDepth < ...DRAWING_DEPTH) {
2   menuPanel.hide();
3   draw(i);
4 } else if (minDepth > ...MENU_DEPTH) {
5   menuPanel.show();
6 } else {
7   tool.stopDrawing();
8   menuPanel.hide();
9   setLastPosition(i, null, null);
10 }

```

In this code, `minDepth` is the minimum position of a finger tip. The Leap Motion API uses a negative Z-axis so this is the front-most part of the hand. The selection of random clusters more uniformly samples combinations of cluster centroids, therefore more rapidly changing between the branches in this function. In the application, this is reflected by alternating between showing the menu, selecting new tools, and painting on the canvas very quickly. This leads to an increase of around 43 lines of code over the n-gram model approach. Using the n-gram model technique can also change tools, but does so at a much slower speed, following realistic movement. Random generation of test data is unlikely to move all points in the hand behind the threshold to activate the pop-up menu so can only paint on the screen using the default tool, and thus performs poorly on this application. For `GorogoaPuzzle` and `PolyDrop` the likely reason that coverage does not increase with the use of n-gram models is that both apps require very specific and complex interactions (e.g., balancing elements on a horizontal bar in `PolyDrop`). While n-gram based generation may produce more realistic data sequences, these sequences would need to be tailored to the specific state of the gameplay. Consequently, both random clusters and n-gram based generation are likely stuck at the same point in the application. Overall, the benefits of using an n-gram model in data generation are application specific. On applications such as `PaintLeapMotion`, which do not rely on a steady stream of data with small change over time, random clusters performs well. Other applications such as `Virtual Piano For Beginners` require precise gestures and slow interactions with menu items, which are more commonly generated with the n-gram based approach, while it is unlikely that the random cluster approach will generate a hand that remains still long enough to activate a button. Figure 5.8 shows the change in line coverage during test gener-

ation. In three of five applications, line coverage is still increasing after 30 minutes for an n-gram model based approach to generation. In two of five, line coverage is still increasing after 50 minutes. Given more time, it is plausible that the n-gram model based approach will achieve an equal level of coverage than random clusters on the PaintLeapMotion application.

Although code can be executed by seeding NUI data, it is impossible to achieve 100% coverage in certain circumstances. For example, GoroGoaPuzzle has defensive programming when loading images, ensuring that the image exists. The cases where an image does not exist cannot be executed by seeding Leap Motion data alone. Another example of unreachable code is in PaintLeapMotion, which contains both NUI and mouse interactions. For our experiments, no mouse interaction could take place hence there is no possible way to test this code.

RQ5.1: NUI test generation approaches increase coverage on Leap Motion applications by an average of 14% when compared to a purely random generation approach, but applications may only use subsets of the complex NUI input data structures, limiting benefits achievable with n-gram modelling.

5.5.4 RQ5.2: How does the quantity of training data influence the effectiveness of models generated for NUI testing?

Table 5.2 shows the mean coverage for different generation techniques using models trained on both a) a single user's data or b) all users' data for the respective application. For all five applications, the mean coverage was greater for a 'merged' model that was trained on all users' data, as confirmed in Figure 5.9. Of the five applications tested, three appli-

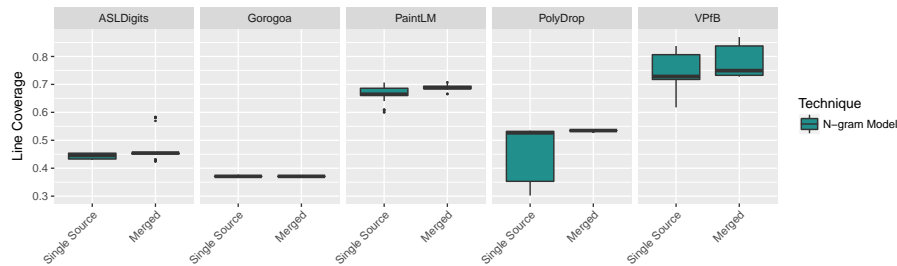


Figure 5.9. Line Coverage for models using either multiple or single person data when training for each application.

Table 5.2. Code coverage difference between single and merged data sources for each application. **Bold** is significant ($P < 0.05$).

Source Application	Single Source Cov	Merged Source Cov	A12	P-value
ASLDigits	0.444	0.468	0.725	0.017
Gorogoa	0.371	0.371	0.536	0.590
PaintLM	0.672	0.689	0.759	< 0.001
PolyDrop	0.467	0.534	0.965	< 0.001
VPfB	0.738	0.778	0.711	0.080

cations achieved a significantly higher code coverage when tested with the merged model. From this, we can make two conclusions. Firstly, models that have been trained with more data yield higher code coverage. Secondly, a greater volume of training data is beneficial even when it originates from a number of different users.

Increasing the amount of data available to produce models increases the data points assigned to each cluster, producing a more diverse set of cluster centroids to be chosen by models when generating data. Also, as each cluster contains more elements, the n-gram models representing transitions between clusters are less sparse, allowing a greater variance in the sequences generated. The finding that a benefit accrues from a larger amount of training data, even when it originates from a diverse pool of users, is not entirely expected. Users interacting with the

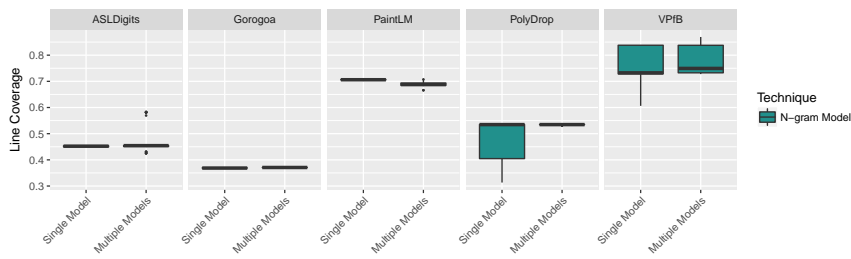


Figure 5.10. Line Coverage for models using either single or multiple model generation for each application.

Leap Motion have different anatomy (e.g., hand sizes, finger lengths) and may interact with the controller in specific ways. Apparently, the benefits of generalizing over a diverse pool of data outweigh the disadvantages that might be expected from anatomical differences. This suggests that in future work, crowd-sourcing interactions from a large pool of users should be an effective way of building models for NUI testing.

RQ5.2: Test generation using models trained with more than one source of training data outperformed those using only a single data source, leading to an average coverage increase of 5.5%. This suggests that pooling data across a number of users is beneficial, even though the users differ in their anatomy (e.g., their hand sizes and finger lengths).

5.5.5 RQ5.3: How beneficial is a model with separation of NUI data, which can generate parts of the data structure in isolation, compared to a model trained on a whole snapshot of the NUI data?

Table 5.3 shows the mean code coverage after testing the two forms of model generation: a single model or multiple models. The single model approach generates entire data frames at once, by selecting a centroid

Table 5.3. Code coverage for single or multiple model generation for each application. **Bold** is significant ($P < 0.05$).

Data Generation Application	Single Model Cov	Multiple Models Cov	A12	P-value
ASLDigits	0.452	0.468	0.751	< 0.001
Gorogoa	0.369	0.371	1.000	< 0.001
PaintLM	0.706	0.689	0.070	< 0.001
PolyDrop	0.479	0.534	0.510	0.926
VPfB	0.781	0.778	0.483	0.879

from Leap Motion data clustered as a complete set of features. The multiple model approach generates data from models clustered from subsets of the data set, then combining data from each model into a data frame. For ASLDigits and GorogoaPuzzle the multiple model based approach achieves a significantly higher code coverage; on PaintLeapMotion the coverage is significantly lower. The coverage difference can be seen in Figure 5.10. On the other two applications the mean coverage is slightly higher with multiple models, but differences are not significant. These results show that the decision to use multiple models for generating data is application specific. The application which benefits mostly from use of a single model is PaintLeapMotion. From RQ5.1 we already know that random clusters perform better at interacting with the tool menu items of this application. Similarly, using a single model is more likely to reproduce the interactions with the tool menu in the training data, while creating separate models leads to less reproduction, and exploration of new combinations. For example, on PaintLeapMotion we used 1200 clusters, and the single model simply learns the temporal relationships between these clusters. In contrast, when splitting the data into five models, we end up with substantially more possible interactions (i.e., 1200^5 possible combinations). A single model approach explores the input space much quicker, leading to 27 more lines of code being covered than using multiple models. Con-

sequently, applications with simple interactions may be more suited to a single model approach, whereas applications which require more complex sequences of inputs are better suited for a multiple model approach.

GorogoaPuzzle benefits from the use of multiple models. It uses two main forms of interaction: circle gestures and hand movements. The first screen of GorogoaPuzzle requires a specific circle gesture before progression in the story can occur. However, advancing in the story does not necessarily increase code coverage, as the same code is used to handle all circle interactions. To achieve a higher coverage, tests need to advance far into the storyline, where complex sequences of interactions are introduced and needed to advance further. Using multiple models allows for more degrees of freedom in the generated data, and thus succeeds slightly more often in progressing in the GorogoaPuzzle storyline, achieving around 39 more lines of code covered.

ASLDigits also attained a significantly higher code coverage using a multiple model approach. Multiple models performs better than single model due to the application expecting specific finger-joint shapes corresponding to the ASL sign for 0-9, requested by the application. The single model approach merges hand positions and rotations from all interactions with the application, which decreases the amount of unique finger-joint shapes available; in contrast, the multi-model approach covers this with an explicit model, achieving around 67 more lines of code covered. Virtual Piano for Beginners is an interesting application when comparing single to multiple models. In RQ5.1, n-gram generation achieved a higher coverage than random clusters because it could generate a still hand to interact with the game menu. However, a single model approach can also generate a steady hand. Single

model works well for this application due to the position and rotation being encoded with finger positions. To play the correct key on the piano in a tutorial song, the single model n-gram has to generate a single sequence corresponding to the user pressing the key. However, similar to with PaintLeapMotion, the multiple model approach has a much higher search space, so is less likely to generate the sequence to activate the key and progress with the song, covering around 7 less lines of code than using a single model.

RQ5.3: Using multiple models is beneficial when applications use specific features in isolation. If a more precise replication of the training data is required, using a single model approach may be beneficial.

5.6 Discussion

As with the last chapter, we also tried to create models which were based on an inferred state from the application. Previously, we inferred the state by simply using the title of a window. However, most Leap Motion applications only use a single window, so this is infeasible. Instead, we investigated a method of inferring the application state through a screen shot, but this is not trivial.

We tried two approaches of inferring the application state. The first approach is using the histogram of pixel intensity values and comparing this against previously seen histogram, using some threshold to check for equality. The second approach was to use a discrete cosine transformation and a sliding window to greedily match areas of the screen per screen shot.

Figure 5.11 shows the coverage achieved by the approach which uses a

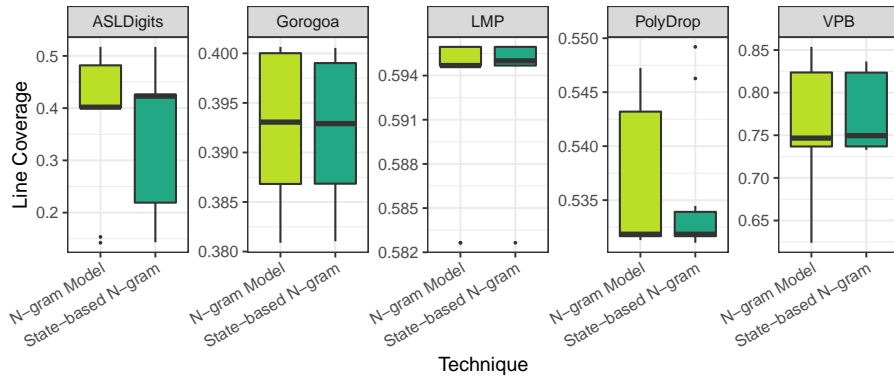


Figure 5.11. Line Coverage for models using either an application model or a window model for test data generation for each application.

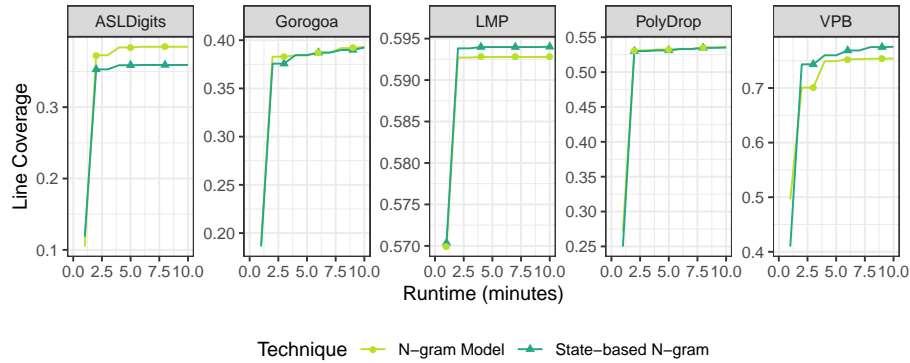


Figure 5.12. Line Coverage over time for models using either an application model or a window model for test data generation for each application.

separate model for each window. We found that the benefits from using window models was not clear over using an application model. It seemed that in most cases, a window model achieved similar or lower coverage than using an application model. This is mainly due to the time required to infer the program's state. With each screen shot identified as a new state, the cost of state inference increases over the expenditure of the testing budget. The coverage in these plots is lower than that of the empirical evaluation as the techniques were limited to only 10 minutes generation time. The implementation and performance of the tool overall was also improved after this comparison. It is

unknown whether using window models will give an advantage after 10 minutes of execution, but as the computation cost of state inference increases, this is unlikely.

Another problem when using window models was that the n-gram models were too sparse. The application models were far less sparse due to the increased amount of data in the clustering and generation of the sequence model, with more transitions between cluster groups.

When starting this comparison between application and window models, we expected that coverage using an application model should diverge, but the more specific window models should be able to overcome these difficult testing objectives that hinder progress. However, Figure 5.12 shows that this is not the case, with both approaches usually performing the same, or one being clearly better at achieving coverage from the start of testing.

Another approach we tried to increasing coverage was generating smoother sequences of hands. Instead of instantly changing between cluster centroids, we tried to linearly interpolate between clusters. However, as with the window models, the time taken for the interpolation again outweighed the benefits of generating a smoother sequence of hands. To improve the interpolation time, we also tried using a Bezier curve to interpolate between the next B clusters. Whilst the overhead was much lower, the Bezier curve changed the positions of important information in the sequence of centroids and these tests missed core functionality that could be tested without interpolation.

5.7 Conclusions

The Leap Motion allows users to interact with applications through hand tracking. We have created a model and test generation framework for the Leap Motion, capable of generating data by learning from real user interactions. This demonstrates that the idea of NUI testing generalizing to other, more complex NUI devices than the previously studied Microsoft Kinect. It is also conceivable that the approach generalizes to other systems which use complex inputs e.g., Autonomous Driver-Assistance Systems, which alert drivers to possible future hazards [54, 86].

There are various points of future work from this chapter:

- Splitting Leap Motion data structures into separate models exponentially increases the amount of data available during test generation. Each model generates data in isolation, and interacts with other models when combining data in complex ways, producing data that was never recorded from the original user. However, if applications rely on precise positioning of a user's hands for interaction as captured in the training data, then the increased quantity of possible data can be as much a hindrance than an advantage. In our experiments, two out of five applications showed a clear benefit from splitting data, but we also found an example where coverage decreased. A challenge thus lies in identifying when to split data, and when not to split data. A possible solution might be to use a hybrid approach, where data is sampled from either of the two approaches with a different probability.
- When training models from multiple sources of training data, in-

- creased data size leads to higher code coverage. This occurs even when the data is from different users. Potentially, this insight opens up the possibility to gather data through crowd sourcing from many individuals, and using that to train user-independent models for data generation.
- Currently, playback of tests takes the same time as generation, but future work is to minimize the generated tests by removing subsequences which have no impact on final code coverage. This will leave tests that are easier to understand by developers and can execute in far less time than the one hour generation budget.
 - A further angle for future work lies in the generalization of models. We limited training data to individual applications, but will it be possible to create generalized models that can be used on applications *without* previous user data to train models with?
 - Currently, our tool only provides a sequence of Leap Motion data that can be played back into the AUT. Future work involves identifying the current program state from the contents of the screen and providing regression tests with oracles. This can then be used in mutation testing.
 - We tried two additional approaches to increase coverage, of which none worked. The computational overhead needs careful consideration when implementing a new technique. If the overhead is too great, then the quantity of data is reduced substantially. However, generating more realistic data may be beneficial for testers when inspecting tests that fail and could be revealing some fault. The effects on code coverage of generating more realistic test data (e.g., through interpolation) is unknown. Further work is needed in investigating whether generating more realistic sequences can

find more faults, and the relationship and benefits of generating more realistic data.

Finally, our experiments have also shown that programs controlled with complex NUI interfaces may also have complex program behaviour, where blindly generating data may not achieve best results. In games like GorogoaPuzzle, thorough testing requires actions that are tailored towards the current state of the application. This suggests a need of identifying such program states, and learning different models for different program states.

6 Conclusions and Future Work

The data provided to applications from user interfaces varies in complexity, from basic commands of a command line interface to complex structures derived from human movement. For test generation tools, generating data to test the functionality of applications which rely on this data is a difficult task.

In this thesis, we have investigated approaches to creating models that can generate test data which resembles that which would be provided by a user interface. This data can then be seeded into applications at test time to execute the event handlers linked to specific events.

6.1	Summary of Contributions	179
6.2	Future Work	183
6.3	Final Remarks	187

6.1 Summary of Contributions

Graphical user interfaces rely on keyboard and mouse interactions to trigger event handlers in the application under test. Test generation tools need to know the coordinates of widgets in an application's GUI

to trigger events at targeted event handlers, but this relies on the application providing information about all of its widgets. Sometimes, this is not possible.

6.1.1 Identifying Areas of Interest in GUIs

The first contribution is a technique of generating synthesised data for training a system using machine learning. Generated data is automatically tagged and could be left running unsupervised to generate large quantities of training data.

The second contribution is a technique of predicting the widget information from screen shots alone, using a system trained on synthetic data. The machine learning system is capable of predicting widgets in real applications, achieving a recall of around 77% when identifying widgets on the same operating system. Using a different operating system, the same system can recall around 52% of widgets in the GUIs of 10 applications.

The third contribution compares the code coverage of tests generated by two techniques. The first is a random testing approach guided by the machine learning system which predicts the information of widgets in an application's GUI, and the second is a random GUI event generator (i.e., a "monkey tester"). We found that the widget detection system guided by predicted widget information can achieve a significantly higher branch coverage in 18 of 20 applications, with an average increase of 42.5%. This is due to the predictions by the system guiding a random tester into generating more interesting interactions, clicking locations on the screen with a higher probability of triggering an event handler and hence executing more of the application's code.

The fourth contribution compared a random tester guided by the widget prediction system to a technique which exploits the Java Swing API (i.e., a golden standard of the prediction system). From this comparison, it is clear that the prediction system can be improved substantially. One interesting observation was that the prediction system can identify custom widgets in applications not supported by the technique which exploits the Java Swing API, and can actually achieve a higher coverage in certain scenarios where the coverage of the API technique diverges or has to default to a monkey testing approach (e.g., if links were embedded inside a text field widget).

Contribution five outlines the data required to accurately store, replay, and create models that can generate events similar to that of a user interacting with an application solely through a computer's keyboard and mouse. We identify four types of events that can be generated by users.

The sixth contribution looked at real users to inspire a test data generator. Real users are likely to only interact with interesting areas of an application's GUI, and interactions generated by models derived from real user data is similar. Models were trained on data recorded from 10 users, each interacting with the same 10 applications. The models can generate events statistically similar to that of a user, but are limited by the input data.

The seventh contribution was an investigation into the effectiveness of models derived from user data. It was found that training models specific to each application's window was beneficial over using a model based on the entire dataset for an application, generating more targeted events and achieving a significantly higher branch coverage. It was also found that using a model trained on user data could outperform

the widget prediction approach by exploiting sequential information to interact with elements such as menus which require more than a single interaction at a single point in time to effectively trigger the underlying event handlers behind such widgets. However, the approach which exploits the Java Swing API again achieves a significantly higher coverage than any approach based on user data. There were applications where using a model derived from user data could achieve a significantly higher coverage than the approach that exploits Java Swing, and this was again due to the temporal information stored in the user models, giving it the ability to generate realistic sequences of events across a given time frame.

Chapter 5 presented a technique of interacting with the Leap Motion, a device where it is extremely difficult to extract event handlers,. However, the technique in Chapter 5 can be applied to general natural user interfaces.

6.1.2 Generating Natural User Data

Contribution eight was a framework capable of recording, storing, processing, and generating data for the Leap Motion device. The data generated is statistically similar to that of the real users used to derive the model.

The ninth contribution was empirical investigation into effectiveness of a model trained on real user data. The model trained on user data significantly outperformed a purely random testing technique, which samples random points in the input domain for all values in the Leap Motion data structure. However, the sequential information in the model is not always required to maximise coverage of generated tests.

The next contribution, 10, was an empirical evaluation of the impact of training data, and how the effect of combining multiple users' data into a single model. Using more user data increases the data points inside the model, and the number of data points in a models transition table. This lead to a significant increase in the coverage achieved when using multiple users' data over a single, isolated user.

Contribution eleven was an empirical evaluation of the effectiveness of splitting a user's data into smaller sub-groups, and creating a model which controlled generation of the data in that sub-group. All models would generate the data representing the area of the Leap Motion API of the data they were derived from, and then the data was combined before being seeded into an application. This technique was application specific. More data could be generated, including data which the user did not provide, but this inflated domain of possible generated data could also be a hindrance where specific sequences of data were needed.

In the next section, we look at interesting observations and future work bought about through this thesis.

6.2 Future Work

In this thesis, we lay down some foundations for testing applications without the assumption of a known source code language or data structure that could be exploited to generate test data. This has opened up several new challenges that need to be addressed.

6.2.1 Graphical User Interface Patterns

Widget Detection

When predicting widgets in a GUI, the prediction model could identify interesting areas of interaction. However, it is clear from the comparison between the prediction model and the gold standard “API” approach that the performance could be improved.

Chapter 4 shows an interesting observation when training a model on a subset of the generated screen shots of GUIs. Our aim here was to expose the model to data that has a greater similarity to that of real GUIs, and therefore increase the model’s performance on real GUIs. The subset was selected using a genetic algorithm, which aimed to reduce the difference for certain metrics between a real set of GUIs, and the generated subset. It is interesting that the trained model achieved a higher precision on real GUIs than the model trained on all data, but a lower recall. This could be due to the increased exposure to more data that the model trained on all generated GUIs had. However, this presents an interesting question: what is the best trade off between recall and precision when testing GUIs through widget detection? When presented with a smaller testing budget, it may be beneficial to use a high precision and a lower recall than when testing with a high budget, and this relationship needs further investigation.

The model could be improved by training on a real set of GUI screen-shots. However, this is expensive to gather. It may be possible to augment a set of real application GUIs with synthesised GUIs, or to begin automatically tagging a real set of application screen shots using our model and refining the automated tags. This would increase the diver-

sity of the data that the model is exposed to, whilst lowering the cost of manual labelling.

We need a better method of identifying the classes that the model predicts. The widgets in a GUI can look very similar. Users identify the widget type using context, and what is around the widget. For example, buttons usually have centred text, whereas a textfield could look identical but have left aligned text. More recent methods of object detection can predict multiple classes, and this would aid in class predictions of the model. A button and a tab may have the same effect: change all or part of the screen, so why predict them as mutually exclusive?

When studying the performance of a purely random tester on an application's GUI, we observed a lower level of coverage to that which Choudhary et al. [33] observed using Android Monkey on Android applications. This is interesting, as it reinforces the idea that random testing in Java applications is less efficient than in Android applications. One possible reason for this could be that the widgets in Android applications take up a higher proportion of the screen space than with traditional Graphical User Interfaces, increasing the likelihood that a random tester would hit a widget. This needs to be studied in greater detail, as it may be possible to improve the performance of random testing on traditional GUI applications through changing factors like window size, to allow the greatest possible chance for interaction with a widget by a random tester.

User Model Creation

In Chapter 5, we found that for certain applications, using a model trained on the whole dataset of user interactions could outperform one

trained on the current window of an application. For this to happen, there needs to be some pattern in an application's GUI. Are there general patterns in GUIs that can be identified? Is it possible to identify patterns in an application's GUI whilst testing and exploit this information?

By combining a model derived from user data with the prediction model from Chapter 3, it may be possible to strengthen both approaches and achieve a higher coverage. The detection approach can aid in state inference, which was a weakness of the models derived from user data when only a single window was used for an application as this inflated the possible interaction points that could be generated.

When clustering user interaction with a GUI, it is possible that the centroid of the cluster falls outside of the interaction box of a widget. We use K-means clustering as our data grouping algorithm. When the data is clustered, we replace user data points with the centroids of the cluster each point was assigned to. This is a destructive form of data compression, and can mean the difference between pressing a button on a GUI, and clicking an uninteresting area of the GUI due to the cluster centroid being outside the bounds of a widget. Some other form of data compression needs to be investigated, which suffers less from this problem, allowing a more representative set of points from the original user's data. This problem also extends to the clustering technique used when generating data for Natural User Interfaces.

6.2.2 Natural User Interface Patterns

In our experiments, we trained models solely on the data extracted from user interactions with the application under test. In future, we

need to remove the assumption of pre-existing user data with an application, as this is not always the case. Can a general corpus of data or model be created that can apply to any application?

We found that using more user data can lead to models which generate tests with a higher code coverage. This creates the possibility for a study on the trade-off between quantity of user data, and quality of the trained model. At some point, it is expected that the potential gain from adding more user data should be outweighed by the cost of collecting more user data.

We tried two additional approaches to guiding the data generated by the model in an attempt to increase the code coverage achieved. The first was to train models specific to the current screen contents. However, the computational overhead for this meant that far fewer interactions could be generated and consequently, the performance did not improve.

The second approach was to interpolate between the last seeded cluster and the next cluster selected from the model. Again, as the interpolation took time, the time synchronisation between the original user data and the interpolated data was destroyed, and although the same quantity of data was seeded, the actual diversity of the data was diminished.

6.3 Final Remarks

Users can be put off using software when encountering bugs in normal application use. Automated testing and data generation can help to cover different areas of an application's source code, and is valuable to augment manually written test suites.

We presented approaches to generating test data and system-level events for applications using two types of user interfaces. This thesis contributes new techniques for generating test data by either learning from synthesised data, or from real user interactions with an application.

Our approaches suffer from generating data blindly and seeding it to the application “in the dark”. If we had some method of inferring an application state through screenshot alone, it may increase the performance of the models through guiding the generated data. However, state inference is a difficult and expensive problem. Because of this, the duration of test generation and execution is often large and we currently have no method of reducing the size of the tests, as we cannot check which events trigger state changes.

In the future, with a better technique of inferring states, it may be possible to isolate specific data points in a sequence of generated test data, and link this back to a specific state change in the application. This would allow developers to see which event sequence led directly to some application state, without ever leaving their integrated development environment. The techniques in this thesis can be applied to systems which use continuous streams of data (such as NUIs) or event-driven programs (such as GUIs). It may be possible to apply these techniques to generate test data for other types of applications such as cyberphysical systems [74], emulating network requests and mocking components in software for integration testing.

Bibliography

- [1] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013, pp. 352–361.
- [2] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "Core: A real-time network emulator," in *MILCOM 2008 - 2008 IEEE Military Communications Conference*, Nov. 2008, pp. 1–7. DOI: 10.1109/MILCOM.2008.4753614.
- [3] Amazon. (2019). Amazon rekognition, [Online]. Available: <https://aws.amazon.com/rekognition/> (visited on 05/15/2019).
- [4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013, ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2013.02.061>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121213000563>.
- [5] Andrej Karpathy, *CS231n Convolutional Neural Networks for Visual Recognition*, <http://cs231n.github.io/convolutional-networks/>, Accessed: 2019-09-03.
- [6] L. Apfelbaum and J. Doyle, "Model based testing," in *Software Quality Week Conference*, 1997, pp. 296–300.
- [7] A. Arcuri and L. Briand, "A Hitchhikers Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
- [8] S. Arora and R. Misra, "Software reliability improvement through operational profile driven testing," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, IEEE, 2005, pp. 621–627.
- [9] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," in *Proceedings of IEEE 36th Annual Foundations of Computer Science*, Oct. 1995, pp. 322–331. DOI: 10.1109/SFCS.1995.492488.
- [10] M. Bajammal and A. Mesbah, "Web Canvas Testing Through Visual Inference," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2018, pp. 193–203. DOI: 10.1109/ICST.2018.00028.

- [11] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02, New York City, New York: Morgan Kaufmann Publishers Inc., 2002, pp. 1329–1336, ISBN: 1-55860-878-8. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2955491.2955736>.
- [12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.
- [13] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE transactions on software engineering*, no. 12, pp. 1278–1296, 1987.
- [14] S. Bauersfeld and T. E. J. Vos, "GUITest: a Java library for fully automated GUI robustness testing," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2012, pp. 330–333. DOI: 10.1145/2351676.2351739.
- [15] S. Bauersfeld and T. E. Vos, "Guitest: A java library for fully automated gui robustness testing," in *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, ACM, 2012, pp. 330–333.
- [16] S. Bauersfeld, S. Wappler, and J. Wegener, "A metaheuristic approach to test sequence generation for applications with a GUI," in *International Symposium on Search Based Software Engineering*, Springer, 2011, pp. 173–187.
- [17] —, "An approach to automatic input sequence generation for gui testing using ant colony optimization," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, ACM, 2011, pp. 251–252.
- [18] G. Becce, L. Mariani, O. Riganelli, and M. Santoro, "Extracting Widget Descriptions from GUIs," in *Fundamental Approaches to Software Engineering*, J. de Lara and A. Zisman, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 347–361, ISBN: 978-3-642-28872-2.
- [19] B. C. Becker and E. G. Ortiz, "Evaluation of face recognition techniques for application to facebook," in *2008 8th IEEE International Conference on Automatic Face & Gesture Recognition*, IEEE, 2008, pp. 1–6.
- [20] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*, IEEE, 2007, pp. 85–103.
- [21] H.-H. Bock, "Clustering methods: A history of k-means algorithms," in *Selected Contributions in Data Analysis and Classification*, P. Brito, G. Cucumel, P. Bertrand, and F. de Carvalho, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 161–172, ISBN: 978-3-540-73560-1. DOI: 10.1007/978-3-540-73560-1_15. [Online]. Available: https://doi.org/10.1007/978-3-540-73560-1_15.
- [22] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *IEEE transactions on software engineering*, vol. 14, no. 10, pp. 1462–1477, 1988.
- [23] N. P. Borges Jr., M. Gómez, and A. Zeller, "Guiding app testing with mined interaction models," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18, Gothenburg, Sweden: ACM, 2018, pp. 133–143, ISBN: 978-1-4503-5712-8. DOI: 10.1145/3197231.3197243. [Online]. Available: <http://doi.acm.org/10.1145/3197231.3197243>.

- [24] M. N. K. Boulos, B. J. Blanchard, C. Walker, J. Montero, A. Tripathy, and R. Gutierrez-Osuna, "Web gis in practice x: A microsoft kinect natural user interface for google earth navigation," *International journal of health geographics*, vol. 10, no. 1, p. 1, 2011.
- [25] G. R. Bradski, "Real time face and object tracking as a component of a perceptual user interface," in *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*, IEEE, 1998, pp. 214–219.
- [26] M. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's mechanical turk: A new source of inexpensive, yet high-quality, data?" *Perspectives on psychological science*, vol. 6, no. 1, pp. 3–5, 2011.
- [27] I. Burnstein, *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.
- [28] S. Carino and J. H. Andrews, "Dynamically testing guis using ant colony optimization (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 138–148.
- [29] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Back-propagation, conjugate gradient, and early stopping," in *Advances in neural information processing systems*, 2001, pp. 402–408.
- [30] Cédric Beust, *TestNG*, <https://testng.org/doc/>, Accessed: 2019-09-01.
- [31] Y.-L. Chang, A. Anagaw, L. Chang, Y. C. Wang, C.-Y. Hsiao, and W.-H. Lee, "Ship detection based on yolov2 for sar imagery," *Remote Sensing*, vol. 11, no. 7, p. 786, 2019.
- [32] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013.
- [33] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android:are we there yet? (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 429–440.
- [34] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, May 2011, ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. [Online]. Available: <http://doi.acm.org/10.1145/1988042.1988046>.
- [35] "coverage criteria for gui testing,"
- [36] C. Degott, N. P. Borges Jr, and A. Zeller, "Learning user interface element interactions," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2019, pp. 296–306.
- [37] Developers, *Android, Monkeyrunner*, 2015.
- [38] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: A systematic review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07, Atlanta, Georgia: ACM, 2007, pp. 31–36, ISBN: 978-1-59593-880-0. DOI: 10.1145/1353673.1353681. [Online]. Available: <http://doi.acm.org/10.1145/1353673.1353681>.
- [39] J. Ding, B. Chen, H. Liu, and M. Huang, "Convolutional Neural Network With Data Augmentation for SAR Target Recognition," *IEEE Geoscience and Remote Sensing Letters*, vol. 13, no. 3, pp. 364–368, Mar. 2016, ISSN: 1545-598X. DOI: 10.1109/LGRS.2015.2513754.

- [40] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on software engineering*, no. 4, pp. 438–444, 1984.
- [41] R. Eckstein, M. Loy, and D. Wood, *Java swing*. O'Reilly & Associates, Inc., 1998.
- [42] M. Ermuth and M. Pradel, "Monkey see, monkey do: Effective generation of gui tests with inferred macro events," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
- [43] R. Feldt and S. Poulding, "Finding test data with specific properties via meta-heuristic search," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2013, pp. 350–359. DOI: 10.1109/ISSRE.2013.6698888.
- [44] J. E. Forrester and B. P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," in *Proceedings of the 4th USENIX Windows System Symposium*, Seattle, 2000, pp. 59–68.
- [45] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ACM, 2011, pp. 416–419.
- [46] —, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, 8:1–8:42, Dec. 2014, ISSN: 1049-331X. DOI: 10.1145/2685612. [Online]. Available: <http://doi.acm.org/10.1145/2685612>.
- [47] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 80–89.
- [48] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, vol. 1, 2015, pp. 55–65.
- [49] D. Gelperin and B. Hetzel, "The growth of software testing," *Communications of the ACM*, vol. 31, no. 6, pp. 687–695, 1988.
- [50] A. Ghahrai. (2018). Error, fault and failure in software testing, [Online]. Available: <https://www.testingexcellence.com/error-fault-failure-software-testing/> (visited on 05/12/2019).
- [51] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013. arXiv: 1311.2524. [Online]. Available: <http://arxiv.org/abs/1311.2524>.
- [52] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, pp. 213–223, Jun. 2005, ISSN: 0362-1340. DOI: 10.1145/1064978.1065036. [Online]. Available: <http://doi.acm.org/10.1145/1064978.1065036>.
- [53] Google. (2019). Open images dataset v5, [Online]. Available: <https://storage.googleapis.com/openimages/web/factsfigures.html> (visited on 05/15/2019).
- [54] D. Greene, J. Liu, J. Reich, Y. Hirokawa, A. Shinagawa, H. Ito, and T. Mikami, "An Efficient Computational Architecture for a Collision Early-Warning System for Vehicles, Pedestrians, and Bicyclists," *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 4, pp. 942–953, Dec. 2011, ISSN: 1524-9050. DOI: 10.1109/TITS.2010.2097594.

- [55] T. Griebel and V. Gruhn, "A Model-based Approach to Test Automation for Context-aware Mobile Applications," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14, Gyeongju, Republic of Korea: ACM, 2014, pp. 420–427, ISBN: 978-1-4503-2469-4. DOI: 10.1145/2554850.2554942. [Online]. Available: <http://doi.acm.org/10.1145/2554850.2554942>.
- [56] T. Griebel, M. Heseni, and V. Gruhn, "Towards Automated UI-Tests for Sensor-Based Mobile Applications," in *Intelligent Software Methodologies, Tools and Techniques - 14th International Conference, SoMeT 2015, Naples, Italy, September 15-17, 2015. Proceedings*, 2015, pp. 3–17. DOI: 10.1007/978-3-319-22689-7_1.
- [57] F. Gross, G. Fraser, and A. Zeller, "Exsyst: Search-based gui testing," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, Zurich, Switzerland: IEEE Press, 2012, pp. 1423–1426, ISBN: 978-1-4673-1067-3. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337435>.
- [58] —, "Search-based system testing: High coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, 2012, pp. 67–77.
- [59] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.
- [60] M. Harman and B. F. Jones, "Search-based software engineering," *Information and software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
- [61] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," Department of Computer Science, University of Sheffield, Tech. Rep. CS-13-01, 2013.
- [62] J. Hartmann, C. Imoberdorf, and M. Meisinger, "Uml-based integration testing," in *ACM SIGSOFT Software Engineering Notes*, ACM, vol. 25, 2000, pp. 60–70.
- [63] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–72, 1992.
- [64] P. Hsia, J. Gao, J. Samuel, D. Kung, Y. Toyoshima, and C. Chen, "Behavior-based acceptance testing of software systems: A formal scenario approach," in *Proceedings Eighteenth Annual International Computer Software and Applications Conference (COMPSAC 94)*, IEEE, 1994, pp. 293–298.
- [65] C. Hunt, G. Brown, and G. Fraser, "Automatic Testing of Natural User Interfaces," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, Mar. 2014, pp. 123–132. DOI: 10.1109/ICST.2014.25.
- [66] D. C. Ince, "The Automatic Generation of Test Data," *The Computer Journal*, vol. 30, no. 1, pp. 63–69, 1987.
- [67] R. Jeffries, J. R. Miller, C. Wharton, and K. Uyeda, "User interface evaluation in the real world: A comparison of four techniques," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '91, New Orleans, Louisiana, USA: ACM, 1991, pp. 119–124, ISBN: 0-89791-383-3. DOI: 10.1145/108844.108862. [Online]. Available: <http://doi.acm.org/10.1145/108844.108862>.
- [68] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009, Source Code Analysis and Manipulation, SCAM 2008, ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2009.04.016>. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0950584909000688>.

- [69] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis, "How easy is local search?" *Journal of computer and system sciences*, vol. 37, no. 1, pp. 79–100, 1988.
- [70] C. Jones, "Software project management practices: Failure versus success," *CrossTalk: The Journal of Defense Software Engineering*, vol. 17, no. 10, pp. 5–9, 2004.
- [71] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014, pp. 437–440.
- [72] C. Kaner, "Software negligence and testing coverage," *Proceedings of STAR*, vol. 96, p. 313, 1996.
- [73] M. Khademi, H. Mousavi Hondori, A. McKenzie, L. Dodakian, C. V. Lopes, and S. C. Cramer, "Free-hand interaction with leap motion controller for stroke rehabilitation," in *Proceedings of the extended abstracts of the 32nd annual ACM conference on Human factors in computing systems*, ACM, 2014, pp. 1663–1668.
- [74] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2014.
- [75] M. E. Khan, F. Khan, *et al.*, "A comparative study of white box, black box and grey box testing techniques," *Int. J. Adv. Comput. Sci. Appl*, vol. 3, no. 6, 2012.
- [76] Z. Al-Khanjari, M. Woodward, and H. A. Ramadhan, "Critical analysis of the pie testability technique," *Software Quality Journal*, vol. 10, no. 4, pp. 331–354, Dec. 2002, ISSN: 1573-1367. DOI: 10.1023 / A:1022190021310. [Online]. Available: <https://doi.org/10.1023/A:1022190021310>.
- [77] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.
- [78] H. Koziolok, "Operational profiles for software reliability," 2005.
- [79] Y. Le Traon, T. Jérón, J.-M. Jézéquel, and P. Morel, "Efficient object-oriented integration and regression testing," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 12–25, 2000.
- [80] Leap Motion, *How Does the Leap Motion Controller Work?* <http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/>, Accessed: 2016-02-23.
- [81] —, *Leap Motion | Mac & PC Motion Controller for Games, Design, Virtual Reality & More*, <https://www.leapmotion.com>, Accessed: 2016-09-13.
- [82] G. Lematre, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 559–563, 2017.
- [83] D. Libes, "Expect: Scripts for controlling interactive processes," *Computing Systems*, vol. 4, no. 2, pp. 99–125, 1991.
- [84] —, *Exploring Expect: A Tcl-based toolkit for automating interactive programs*. "O'Reilly Media, Inc.", 1995.
- [85] C.-T. Lin, C.-D. Chen, C.-S. Tsai, and G. M. Kapfhammer, "History-based test case prioritization with software version awareness," in *2013 18th International Conference on Engineering of Complex Computer Systems*, IEEE, 2013, pp. 171–172.

- [86] D. F. Llorca, V. Milanes, I. P. Alonso, M. Gavilan, I. G. Daza, J. Perez, and M. Á. Sotelo, "Autonomous pedestrian collision avoidance using a fuzzy steering controller," *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 2, pp. 390–401, Jun. 2011, ISSN: 1524-9050. DOI: 10.1109/TITS.2010.2091272.
- [87] R. Lo, R. Webby, and R. Jeffery, "Sizing and estimating the coding and unit testing effort for gui systems," in *Proceedings of the 3rd International Software Metrics Symposium*, Mar. 1996, pp. 166–173. DOI: 10.1109/METRIC.1996.492453.
- [88] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2014, pp. 599–609.
- [89] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 94–105.
- [90] L. Mariani, M. Pezze, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Apr. 2012, pp. 81–90. DOI: 10.1109/ICST.2012.88.
- [91] L. Mariani, M. Pezz, O. Riganelli, and M. Santoro, "Automatic Testing of GUI-based Applications," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 341–366, 2014, ISSN: 1099-1689. DOI: 10.1002/stvr.1538. [Online]. Available: <http://dx.doi.org/10.1002/stvr.1538>.
- [92] L. Mariani, O. Riganelli, and M. Santoro, "The AutoBlackTest Tool: Automating System Testing of GUI-based Applications," *ECLIPSE IT 2011*, p. 78, 2011.
- [93] V. Massol and T. Husted, *JUnit in action*. Manning Publications Co., 2003.
- [94] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.
- [95] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, Nov. 2003, pp. 260–269. DOI: 10.1109/WCRE.2003.1287256.
- [96] A. M. Memon, "Gui testing: Pitfalls and process," *Computer*, no. 8, pp. 87–88, 2002.
- [97] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE transactions on software engineering*, vol. 27, no. 12, pp. 1085–1110, 2001.
- [98] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.
- [99] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of unix utilities and services," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1995.
- [100] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.
- [101] Mockaroo, LLC., *Mockaroo realistic data generator*, <http://www.mockaroo.com>, Accessed: 2017-12-11.

- [102] K. Munakata, S. Tokumoto, and T. Uehara, "Model-based test case generation using symbolic execution," in *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, ser. JAMAICA 2013, Lugano, Switzerland: ACM, 2013, pp. 23–28, ISBN: 978-1-4503-2161-7. DOI: 10.1145/2489280.2489282. [Online]. Available: <http://doi.acm.org/10.1145/2489280.2489282>.
- [103] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, Mar. 2014. DOI: 10.1007/s10515-013-0128-9.
- [104] T. R. Niesler and P. C. Woodland, "A variable-length category-based n-gram language model," in *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, IEEE, vol. 1, 1996, pp. 164–167.
- [105] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed Random Testing for Java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07, Montreal, Quebec, Canada: ACM, 2007, pp. 815–816, ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297902. [Online]. Available: <http://doi.acm.org/10.1145/1297846.1297902>.
- [106] P. Papadopoulos and N. Walkinshaw, "Black-box test generation from inferred models," in *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE '15, Florence, Italy: IEEE Press, 2015, pp. 19–24. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2820668.2820674>.
- [107] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, Mar. 2013, pp. 342–351. DOI: 10.1109/ICST.2013.13.
- [108] Y. Pavlov and G. Fraser, "Semi-automatic search-based test generation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012, pp. 777–784.
- [109] S. Poulding and R. Feldt, "Generating structured test data with specific properties using nested monte-carlo search," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14, Vancouver, BC, Canada: ACM, 2014, pp. 1279–1286, ISBN: 978-1-4503-2662-9. DOI: 10.1145/2576768.2598339. [Online]. Available: <http://doi.acm.org/10.1145/2576768.2598339>.
- [110] M. Rahman and J. Gao, "A reusable automated acceptance testing architecture for microservices in behavior-driven development," in *2015 IEEE Symposium on Service-Oriented System Engineering*, Mar. 2015, pp. 321–325. DOI: 10.1109/SOSE.2015.55.
- [111] M. A. Rahman and Y. Wang, "Optimizing intersection-over-union in deep neural networks for image segmentation," in *International symposium on visual computing*, Springer, 2016, pp. 234–244.
- [112] J. Redmon, S. K. Divvala, and R. B. G. and Ali Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *CoRR*, vol. abs/1506.02640, 2015. arXiv: 1506.02640. [Online]. Available: <http://arxiv.org/abs/1506.02640>.
- [113] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," *CoRR*, vol. abs/1612.08242, 2016. arXiv: 1612.08242. [Online]. Available: <http://arxiv.org/abs/1612.08242>.
- [114] —, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

- [115] O. Riganeli, D. Micucci, and L. Mariani, "From source code to test cases: A comprehensive benchmark for resource leak detection in android apps," *Software: Practice and Experience*, vol. 49, no. 3, pp. 540–548, 2019.
- [116] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *International Symposium on Search Based Software Engineering*, Springer, 2015, pp. 93–108.
- [117] J. M. Rojas and G. Fraser, "Code defenders: A mutation testing game," in *Proc. of The 11th International Workshop on Mutation Analysis*, To appear, IEEE, 2016.
- [118] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *Software Testing, Verification and Reliability*, vol. 26, no. 5, pp. 366–401, 2016. DOI: 10.1002/stvr.1601. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1601>. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601>.
- [119] R. Rossi and G. Seghetti, *Method, system and computer program for testing a command line interface of a software product*, US Patent 7,926,038, Apr. 2011.
- [120] U. Rueda, T. E. Vos, F. Almenar, M. Martnez, and A. I. Esparcia-Alcázar, "Testar: From academic prototype towards an industry-ready tool for automated testing at the user interface level," *Actas de las XX Jornadas de Ingeniera del Software y Bases de Datos (JISBD 2015)*, pp. 236–245, 2015.
- [121] O. Russakovsky, L. Li, and L. Fei-Fei, "Best of both worlds: Human-machine collaboration for object annotation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 2121–2131. DOI: 10.1109/CVPR.2015.7298824.
- [122] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," *IEEE Signal Processing Letters*, vol. 24, no. 3, pp. 279–283, Mar. 2017, ISSN: 1070-9908. DOI: 10.1109/LSP.2017.2657381.
- [123] K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller, "Using Dynamic Symbolic Execution to Generate Inputs in Search-based GUI Testing," in *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, ser. SBST '15, Florence, Italy: IEEE Press, 2015, pp. 32–35. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2821339.2821350>.
- [124] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 513–528.
- [125] S. Shamschiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '15, Madrid, Spain: ACM, 2015, pp. 1367–1374, ISBN: 978-1-4503-3472-3. DOI: 10.1145/2739480.2754696. [Online]. Available: <http://doi.acm.org/10.1145/2739480.2754696>.
- [126] M. Shardlow, "An Analysis of Feature Selection Techniques,"
- [127] C. Smidts, C. Mutha, M. Rodriguez, and M. J. Gerber, "Software testing with an operational profile: Op definition," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 39, 2014.
- [128] I. Staretu and C. Moldovan, "Leap motion device used to control a real anthropomorphic gripper," *International Journal of Advanced Robotic Systems*, vol. 13, no. 3, p. 113, 2016.

- [129] T. Su, "Fsmdroid: Guided gui testing of android apps," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, IEEE, 2016, pp. 689–691.
- [130] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ACM, 2017, pp. 245–256.
- [131] C. Sun, Z. Zhang, B. Jiang, and W. K. Chan, "Facilitating Monkey Test by Detecting Operable Regions in Rendered GUI of Mobile Game Apps," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Aug. 2016, pp. 298–306. DOI: 10.1109/QRS.2016.41.
- [132] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, IEEE, 2011, pp. 377–386.
- [133] S. H. Tan and A. Roychoudhury, "Relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Florence, Italy: IEEE Press, 2015, pp. 471–482, ISBN: 978-1-4799-1934-5. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2818754.2818813>.
- [134] The jQuery Foundation, *QUnit: A JavaScript Unit Testing framework*. <https://qunitjs.com/>, Accessed: 2019-09-01.
- [135] P. Tonella, R. Tiella, and C. D. Nguyen, "Interpolated n-grams for model based testing," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, Hyderabad, India: ACM, 2014, pp. 562–572, ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568242. [Online]. Available: <http://doi.acm.org/10.1145/2568225.2568242>.
- [136] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 73–81, Mar. 1998, ISSN: 0163-5948. DOI: 10.1145/271775.271792. [Online]. Available: <http://doi.acm.org/10.1145/271775.271792>.
- [137] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, no. 8, pp. 717–727, Aug. 1992, ISSN: 0098-5589. DOI: 10.1109/32.153381.
- [138] J. Voas, L. Morell, and K. Miller, "Predicting where faults can hide from testing," *IEEE Software*, vol. 8, no. 2, pp. 41–48, Mar. 1991, ISSN: 0740-7459. DOI: 10.1109/52.73748.
- [139] J. M. Voas, "Object-oriented software testability," in *Achieving Quality in Software: Proceedings of the third international conference on achieving quality in software, 1996*, S. Bologna and G. Bucci, Eds. Boston, MA: Springer US, 1996, pp. 279–290, ISBN: 978-0-387-34869-8. DOI: 10.1007/978-0-387-34869-8_23. [Online]. Available: https://doi.org/10.1007/978-0-387-34869-8_23.
- [140] T. E. Vos, P. M. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, "Testar: Tool support for test automation at the user interface level," *International Journal of Information System Modeling and Design (IJISMD)*, vol. 6, no. 3, pp. 46–83, 2015.
- [141] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 248–257. DOI: 10.1109/ASE.2008.35.

- [142] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical Testing of Software Based on a Usage Model," *Softw. Pract. Exper.*, vol. 25, no. 1, pp. 97–108, Jan. 1995, ISSN: 0038-0644. DOI: 10.1002/spe.4380250106. [Online]. Available: <http://dx.doi.org/10.1002/spe.4380250106>.
- [143] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and software technology*, vol. 43, no. 14, pp. 841–854, 2001.
- [144] E. J. Weyuker, "Testing component-based software: A cautionary tale," *IEEE software*, vol. 15, no. 5, pp. 54–59, 1998.
- [145] T. D. White, G. Fraser, and G. J. Brown, "Modelling Hand Gestures to Test Leap Motion Controlled Applications," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Apr. 2018, pp. 204–213. DOI: 10.1109/ICSTW.2018.00051.
- [146] T. D. White, G. Fraser, and G. J. Brown, "Improving random gui testing with image-based widget detection," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: ACM, 2019, pp. 307–317, ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330551. [Online]. Available: <http://doi.acm.org/10.1145/3293882.3330551>.
- [147] J. A. Whittaker and M. G. Thomason, "A Markov Chain Model for Statistical Software Testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, Oct. 1994, ISSN: 0098-5589. DOI: 10.1109/32.328991.
- [148] J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 1, pp. 93–106, 1993.
- [149] D. Wigdor and D. Wixon, *Brave NUI World: Designing Natural User Interfaces for Touch and Gesture*. Elsevier, 2011.
- [150] S. Wood, K. Reidy, N. Bell, K. Feeney, and H. Meredith, *The emerging role of Microsoft Kinect in physiotherapy rehabilitation for stroke patients*, https://www.physio-pedia.com/The_emerging_role_of_Microsoft_Kinect_in_physiotherapy_rehabilitation_for_stroke_patients, Accessed: 2017-10-12.
- [151] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Second International Workshop on Automation of Software Test (AST '07)*, May 2007, pp. 5–5. DOI: 10.1109/AST.2007.15.
- [152] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," in *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '09, Victoria, BC, Canada: ACM, 2009, pp. 183–192, ISBN: 978-1-60558-745-5. DOI: 10.1145/1622176.1622213. [Online]. Available: <http://doi.acm.org/10.1145/1622176.1622213>.
- [153] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: ACM, 2016, pp. 987–992, ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2983958. [Online]. Available: <http://doi.acm.org/10.1145/2950290.2983958>.
- [154] J. Zhang, M. Huang, X. Jin, and X. Li, "A real-time chinese traffic sign detection algorithm based on modified yolov2," *Algorithms*, vol. 10, no. 4, p. 127, 2017.

- [155] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *2010 IEEE International Conference on Software Maintenance*, IEEE, 2010, pp. 1–10.
- [156] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997, ISSN: 0360-0300. DOI: 10.1145/267580.267590. [Online]. Available: <http://doi.acm.org/10.1145/267580.267590>.
- [157] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach," *IEEE transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.
- [158] J. Zukowski, *The definitive guide to Java Swing*. Apress, 2006.

A Participant task sheet

Here is a participant task sheet as described in Chapter 4. Every application and every task was randomised for each user, to reduce the learning effect of interacting with certain applications before others.

GUI Interaction Experiment

Participant 1

Application 1

Minesweeper

No information is required for this application.

Please run Minesweeper.sh from the user directory.

3 Minute Warm Up

Application 1: Minesweeper

Tasks:

- [] - Start a game of JMine
- [] - Flag a mine location
- [] - Close the window and play a Difficult game of JMine
- [] - Find a mine in a corner
- [] - Win a game
- [] - Close the window and play a Medium game of JMine
- [] - Close the window and play an easy game of JMine
- [] - Flag a location where there is not a mine
- [] - Find a mine

Application 2

JabRef

No information is required for this application.

Please run JabRef.sh from the user directory.

3 Minute Warm Up

Application 2: JabRef

Tasks:

- [] - Mark an entry
- [] - Find an entry using the "Search" functionality
- [] - Generate using ArXiv ID (e.g., arXiv:1501.00001)
- [] - Open the preferences (don't change anything!)
- [] - Add a book
- [] - Add an article
- [] - Clean up entries in the current library (add some if they do not exist)
- [] - Add an entry from a web search
- [] - Add a duplicate entry and remove using "Find Duplicates" function.
- [] - Add a string to the current library
- [] - Rank an entry 5 stars
- [] - Create a new Bibtex library
- [] - Unmark an entry
- [] - Add a new group and at least one article to the group
- [] - Add an InProceedings
- [] - Save the library in /home/thomas/jabref

Application 3

Dietetics

Information

- Translation: peso corporeo - body weight
- Translation: altezza - height
- Translation: eta - age

Please run Dietetics.sh from the user directory.

3 Minute Warm Up

Application 3: Dietetics

Tasks:

- [] - View the notes on BMI (translation: Note sul BMI)
- [] - View information on the BMI formula (translation: Informazioni forumule)
- [] - Classify someone as "sovrappeso"
- [] - Classify someone as "sottopeso"
- [] - Calculate BMI for a created person
- [] - View the program info
- [] - Classify someone as "con un obesita di primo livello"

Application 4

Simple Calculator

No information is required for this application.

Please run Simple_Calculator.sh from the user directory.

3 Minute Warm Up

Application 4: Simple Calculator

Tasks:

- [] - Perform a calculation using multiplication
- [] - View the about page
- [] - Chain together 4 unique calculations without clearing the screen.
- [] - Perform a calculation using multiplication of a negative number.
- [] - Perform a calculation using addition
- [] - clear the contents of the screen (do a calculation then clear if it is already clear)
- [] - Perform a calculation using non-integer numbers
- [] - Perform a calculation using subtraction
- [] - Perform a calculation using division
- [] - Calculate the square root of 9801

Application 5

UPM

Information

- To perform the tasks, a new database needs to be created first.

Please run UPM.sh from the user directory.

3 Minute Warm Up

Application 5: UPM

Tasks:

- [] - Copy the password of an account (add one if none exist)
- [] - Create a new database
- [] - Add an account with a generated password
- [] - Edit an existing account (add one if none exist)
- [] - View an existing account
- [] - Add an account with a manual password
- [] - Add an account to the database
- [] - Export a database
- [] - Put a password on a database
- [] - Copy the username of an account (add one if none exist)
- [] - View the "About" page

Application 6

blackjack

No information is required for this application.

Please run blackjack.sh from the user directory.

3 Minute Warm Up

Application 6: blackjack

Tasks:

- [] - Bet 200
- [] - Enter a game of Black Jack
- [] - Get a picture card
- [] - Lose a round
- [] - Bet 50
- [] - Get blackjack (picture card and an ace)
- [] - Lose a round
- [] - Win a round
- [] - Bet 500
- [] - Bet 1000
- [] - Win a round
- [] - Bet 100
- [] - Bet 20

Finally:

- [] - Bet everything you have (go "all in")

Application 7

ordrumbox

No information is required for this application.

Please run ordrumbox.sh from the user directory.

3 Minute Warm Up

Application 7: ordrumbox

Tasks:

- [] - Create a piano track and include it in the drum beat
- [] - Play a song
- [] - Change the volume
- [] - Add a filter to a track
- [] - Create a drum beat
- [] - Rename an instrument
- [] - Change the gain
- [] - Change the pitch
- [] - Decrease the tempo
- [] - Change the frequency
- [] - Increase the tempo
- [] - Save a song beat
- [] - Create a new song

Application 8

SQuiz

No information is required for this application.

Please run SQuiz.sh from the user directory.

3 Minute Warm Up

Application 8: SQuiz

Tasks:

- [] - Start a new quiz (translation: Nuova Partita)
- [] - Place on the score board
- [] - Get a question wrong
- [] - View the about page
- [] - View the rankings page (translation: classifica)
- [] - Get a question right
- [] - Get a question either right or wrong
- [] - View the statistics page

Application 9

Java Fabled Lands

No information is required for this application.

Please run Java_Fabled_Lands.sh from the user directory.

3 Minute Warm Up

Application 9: Java Fabled Lands

Tasks:

- [] - View the current code words
- [] - Enter combat
- [] - Save the game
- [] - View the original rules
- [] - Buy or sell an item at a market
- [] - View the quick rules
- [] - Loot an item
- [] - Load a new game
- [] - Load a hardcore game
- [] - View the "about" page
- [] - Write a note
- [] - View the ship's manifest

Application 10

bellmanzadeh

No information is required for this application.

Please run bellmanzadeh.sh from the user directory.

3 Minute Warm Up

Application 10: bellmanzadeh

Tasks:

- [] - Set an alternative description and values for all current variables (create variables if they do not exist).
- [] - Add an objective function (create variables if they do not exist)
- [] - Add a Boolean type variable
- [] - Add a constraint (create variables if they do not exist)
- [] - Add an Integer type variable
- [] - Add a Float type variable

