Automatically Testing Interactive Applications

Exploiting interactive applications semantic similarities for automated testing

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Daniele Zuddas

under the supervision of
Mauro Pezzè

July 2019

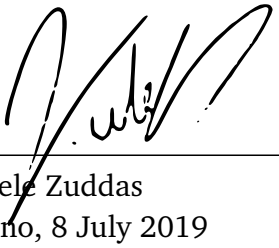
Dissertation Committee

Antonio Carzaniga	Università della Svizzera italiana, Switzerland
Paolo Tonella	Università della Svizzera italiana, Switzerland
Atif Memon	University of Maryland, USA
Tanja E.J. Vos	Technical University of Valencia, Spain
Leonardo Mariani	University of Milano-Bicocca, Italy

Dissertation accepted on 8 July 2019

Research Advisor	PhD Program Director
Mauro Pezzè	**Prof. Walter Binder, Prof. Olaf Schenk**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.



Daniele Zuddas
Lugano, 8 July 2019

Abstract

Interactive applications, such as mobile or web apps, have become essential in our lives and verifying their correctness is now a key issue. Automatic system test case generation can dramatically improve the testing process for these applications and has recently motivated researchers to work on this problem defining a wide range of different approaches. However, most state-of-the-art approaches automatically generate test cases only leveraging the structural characteristics of interactive applications GUIs, paying little attention to their semantic aspects. This led to techniques that, although useful, cannot effectively cover all the semantically meaningful execution scenarios of an interactive application under test and that are not able to distinguish correct behaviors from faulty ones.

In this Ph.D. thesis, we propose to address the limitations of current techniques exploiting the fact that interactive applications often share semantic similarities and implement many functionalities that are not specific to one application only. Our intuition is that these similarities can be leveraged to complement the classical structural information used so far, to define truly effective test case generation approaches. This dissertation presents two novel techniques that exploit this intuition: AUGUSTO and ADAPTDROID.

AUGUSTO exploits a built-in knowledge of the semantics associated with popular and well-known functionalities, such as CRUD operations, to automatically identify these popular functionalities in the application under test and generate effective test cases with automated functional oracles. We demonstrated the effectiveness of AUGUSTO experimenting with different popular functionalities in the context of interactive desktop applications showing that it can reveal faults that cannot be revealed with other state-of-the-art techniques.

ADAPTDROID instead generates semantically meaningful test cases exploiting a radically new approach: obtaining test cases by adapting existing test cases (including oracles) of similar applications instead of generating test cases from scratch. We empirically evaluated this strategy in the context of Android apps and showed that ADAPTDROID is the first fully automatic technique that can effectively adapt test cases, including assertions, across similar interactive applications.

Contents

Contents iii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Research Hypothesis and Contributions . . . 4
- 1.2 Thesis Organization . . . 5

2 Testing Interactive Applications: State of the Art and Open Issues 7
- 2.1 Preliminaries . . . 7
- 2.2 Interactive Applications Testing . . . 8
- 2.3 Automatic Testing of Interactive Applications . . . 10
  - 2.3.1 Random Approaches . . . 10
  - 2.3.2 Model-Based Approaches . . . 11
  - 2.3.3 Coverage-Based Approaches . . . 13
  - 2.3.4 Similarity-Based Approaches . . . 14
- 2.4 Limitations and Open Problems . . . 15

3 Automated Testing of Desktop Applications: an Empirical Study 17
- 3.1 Evaluated Tools . . . 17
- 3.2 Subject Applications . . . 19
- 3.3 Experimental Setup . . . 20
- 3.4 Experimental comparison . . . 21
  - 3.4.1 Fault Revealing Ability . . . 21
  - 3.4.2 Execution Space Sampling Ability . . . 23
  - 3.4.3 Time efficiency . . . 23
- 3.5 Discussion . . . 24
- 3.6 Threats to validity . . . 26

4 Similarities Among Applications: An Opportunity for Testing Interactive Applications 27
4.1 Application Independent Functionalities . . . 29
4.2 Cross-Application Test Case Adaptation . . . 31

5 AUGUSTO: Semantic Testing of Application Independent Functionalities 35
5.1 Motivating Example . . . 35
5.2 Approach . . . 37
5.2.1 AIF Archive . . . 38
5.2.2 Ripping . . . 44
5.2.3 Structural Matching . . . 45
5.2.4 Match Finalizing . . . 49
5.2.5 Reification . . . 50
5.2.6 Testing . . . 51
5.3 Prototype implementation . . . 52
5.4 Evaluation . . . 53
5.4.1 Empirical Setup . . . 54
5.4.2 RQ1 - AIF Detection . . . 55
5.4.3 RQ2 - Effectiveness . . . 57
5.4.4 RQ3 - Comparison . . . 59
5.4.5 Threats to validity . . . 61

6 ADAPTDROID: Semantic Testing via Cross-Application Test Case Adaptation 63
6.1 Preliminaries . . . 63
6.2 Motivating Example . . . 64
6.3 Approach . . . 67
6.3.1 Cross-app Matching of GUI Events . . . 68
6.3.2 Pre-processing . . . 70
6.3.3 Generation of the Initial Population . . . 72
6.3.4 Fitness Calculation . . . 72
6.3.5 Population Evolution . . . 76
6.3.6 Post-Processing . . . 78
6.4 Implementation . . . 78
6.5 Evaluation . . . 79
6.5.1 Empirical Setup . . . 80
6.5.2 RQ1: Effectiveness . . . 82
6.5.3 RQ2: Comparison with Random Search . . . 85

6.5.4 RQ3: Greedy-match Initialization and Fitness-driven Mutations Evaluation 87
6.5.5 Threats to Validity 88

7 Conclusions **91**
7.1 Contributions 93
7.2 Open Research Directions 94

A AIF Models **97**
A.1 AUTH 101
A.2 CRUD 105
A.3 SAVE 110

Bibliography **117**

Figures

3.1 Distribution of detected faults coverage within the time budgets for the different tools 24

4.1 Sign in and Sign up in Jenkins and Ebay 29
4.2 Example of similar note-pad applications in the Google Play Store 31

5.1 AUGUSTO logical architecture 37
5.2 Examples of GUI Pattern model 40
5.3 A simplified version of the GUI of OnlineShopping 44
5.4 A match between the model of the authentication AIF and the GUI of OnlineShopping. Green thick edges are those discovered during the ripping step, while dashed red edges are discovered during the match finalizing step. 46

6.1 ADAPTDROID cross-application test adaptation example 65
6.2 ADAPTDROID logical architecture 67
6.3 Crossover example 76
6.4 Average fitness growth ADAPTDROID vs RANDOM. 87
6.5 Average fitness growth ADAPTDROID vs ADAPTDROID-SIMPLE. . . . 88

A.1 AUTH GUI Pattern model 101
A.2 CRUD GUI Pattern model 105
A.3 SAVE GUI Pattern model 111

Tables

3.1 Testing tools selected for the study 18
3.2 Subject applications 20
3.3 Coverage and Failure revealing capabilities of the tools after 15 hours test sessions 22
3.4 Faults detected by tool 23

5.1 Subject applications 53
5.2 RQ1 - AIF Detection 56
5.3 RQ2 - Effectiveness 58
5.4 RQ3 - Comparison 59

6.1 Running example events descriptors 69
6.2 Evaluation subjects and results 82
6.3 RQ2: Fitness values achieved by ADAPTDROID and RANDOM 85
6.4 RQ3: Fitness values achieved by ADAPTDROID and ADAPTDROID-SIMPLE 87

Chapter 1

Introduction

Software verification is an integral part of modern software development processes and aims to very the correspondence between software *expected* and *actual* behaviors. The most common activity in software verification is testing. Testing assesses the correctness and quality of a software system by verifying whether the results produced by executing it with a finite set of inputs match its expected behavior. IEEE defines testing as

> *"the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain"* [29].

Testing can be carried out at different granularity levels depending on which portion of the application under test (AUT) is targeted: unit testing targets single classes or components, whereas integration and system testing target sets of integrated classes or components and the whole system, respectively [89]. Unit testing aims to reveal faults in the single units, and may miss faults that occur when integrating the numerous AUT software layers or when executing the whole system. For this reason, system testing, that is testing the fully integrated application from the final user viewpoint, is a fundamental type of testing.

System testing requires executing the applications through their interfaces and stimulating all the layers and components involved in the execution. Since the number and complexity of the entities typically involved in a system-level execution could be significant, defining test cases that thoroughly sample and verify the behavior of an application is difficult and expensive. Automating the generation and execution of system test cases can dramatically improve the effectiveness of software verification activities and significantly reduce software development costs. For this reason, automated system test case generation is attracting a growing interest in the research community.

This Ph.D. thesis studies the problem of *automatically generating system test cases* for a popular type of software system, *interactive applications*.

Interactive applications are software systems with a Graphical User Interface (GUI) as the main entry point for interactions with the users. These applications are extremely widespread and are used every day to perform all sorts of tasks, from leisure and travel to banking and insurance. For instance, just the Google Play Store (the official store of Android apps) contains more than 2.5 millions interactive applications. Interactive applications *interact* with the users, meaning that they do not receive an input, process it, and provide an output, but incrementally accept user stimuli (in the form of events on the GUI), reacting to those stimuli by producing new information, and allowing users to further interact with new stimuli not available before. Thus, interactive applications guide the users during the interaction for a more user-friendly experience.

Interactive applications are developed for different platforms, including web, mobile, and desktop. This thesis focuses on the problem of system testing of interactive applications independently from the features that characterise a specific platform, and the proposed approaches are valid for the different platforms.

System test cases for interactive applications are composed of a *GUI interaction* and a *functional oracle*. A GUI interaction is a sequence of user events on the AUT GUI, while a functional oracle is a set of checks on the response of the application shown on the GUI aimed to detect errors. Automating the generation of system test cases for interactive applications faces two main challenges: identifying *meaningful GUI interactions* and defining *automatic functional oracles*.

Meaningful GUI interactions. Given the complexity of contemporary applications' GUIs, the number of possible GUI interaction sequences might be huge. Thus it is important to identify a set of *meaningful* GUI interactions, that is, interactions that represent a realistic usage scenario of the application, leaving out irrelevant combinations of GUI events. For example, a GUI interaction that only navigates through menus and windows is seldom meaningful because it does not execute any of the functionalities implemented in the AUT, whereas a GUI interaction that goes in the "register new user" window, fills out the form with valid data, and clicks "register" is meaningful because it executes an AUT functionality. Irrelevant GUI interactions greatly outnumber meaningful ones, thus generating a reasonable set of *meaningful GUI interactions* that exercise all relevant AUT usage scenarios is a challenging problem.

Automatic functional oracle. Testing aims to detect failures, i.e., wrong behaviors of an application in response to stimuli. While detecting failures that cause system crashes is easy, detecting failures that produce incorrect behaviors/outputs

requires sets of appropriate assertions (called functional oracle) based on some knowledge of the application expected behavior. Because of the non-trivial AUT knowledge required, automatically generating effective functional oracles for non-crashing failures can be challenging [18].

Because of the relevance of interactive applications, the problem of automatically generating system test cases for such applications is increasingly attracting the interest of the research community [45, 48, 70, 77, 80, 115]. State-of-the-art testing strategies share the common idea of analyzing the AUT GUI structure to produce a set of GUI interactions that exercise the elements of the GUI according to some criteria, often based on combinatorial interaction testing, some specific heuristics, or structural information (e.g., data flow relations).

Approaches that focus on structural elements are efficient, but with limited effectiveness due to the lack of knowledge of the AUT semantics. Efficiently identifying all meaningful interaction sequences by exploring the execution space based only on structural information is usually impossible. Moreover, without having any knowledge of the AUT semantics they cannot identify non-crashing failures. Let us consider for instance an interactive application that creates GANTT charts. State-of-the-art techniques based only on the structure of the GUI with no knowledge of the nature of a GANTT chart can generate meaningful test cases that cover some relevant scenarios, e.g., properly allocating activities and resources, only by chance, and cannot check whether the charts produced by the application are correct.

In this thesis, we argue that an automatic testing approach to overcome the limitations discussed above should leverage some information about the AUT *semantics*, to enable to both drive the generation of GUI interactions towards events sequences that are meaningful with respect to the expected usage of the application, and verify the coherence of the AUT actual behavior with respect to its expected behavior. We use the term *semantic testing approach* to refer to automatic system testing approaches that leverage information about the AUT semantics, without relying on structural information only.

Semantic testing approaches proposed so far exploit either specifications (SRS) or models of the software under test [89, 108]. However, specifications and models are rarely available as they are expensive to produce and maintain.

In this thesis we study the problem of defining cost-effective *semantic testing* approaches for interactive applications, that is, we investigate ways to exploit commonly available semantic information to automatically generate system test case that can both execute meaningful interaction and have effective functional oracles, without requiring the presence of artifact expensive to produce and maintain to support our approaches.

1.1 Research Hypothesis and Contributions

The overall research hypothesis of this Ph.D. thesis is:
Cost-effective semantic testing of interactive applications can be achieved exploiting the fact many functionalities are not specific to one application only but are shared among many different interactive applications.

Every year a huge amount of different interactive applications are produced and are made available in the market. In many cases, this myriad of applications share some similarities. We observe that often interactive applications share some common functionalities, for instance, many applications implement the general functionality of authentication. We also observe that several applications target similar goals, and implement similar use cases, as for instance the many applications that manage personal expenses and budget.

The many common functionalities shared among different applications represent an interesting opportunity that can be exploited to produce cost-effective semantics test cases. In this dissertation we advance the state of the art by defining techniques that exploit this unexplored opportunity in two different ways:

Application Independent Functionalities This thesis proposes AUGUSTO (AUtomatic GUi Semantic Testing and Oracles), an approach that exploits functionalities shared across different applications to automatically generate semantically-relevant test inputs and oracles. AUGUSTO relies on the observation that many popular functionalities are implemented in similar ways and respond to the same abstract semantics when they occur in different interactive applications. We call these functionalities application independent functionalities (AIF). Relevant examples of AIF are authentication operations, CRUD (Create, Read, Update, Delete) operations, and search and booking operations. These functionalities are pervasive in software applications and share the same abstract behavior. AUGUSTO exploits AIFs for generating semantically relevant test suites by defining the semantics of popular AIFs once for all according to common knowledge and then using this pre-defined AIF semantics to automatically reveal and test AIFs in different AUTs.

Cross-Application Test Case Adaptation This thesis proposes ADAPTDROID, an approach to adapt system test cases across different applications. ADAPTDROID relies on the observation that when a new application is developed there might already exist other similar applications in the market that have the same overall goal or that, in general, implement the same functionalities.

These similar applications might include executable system test cases that can be adapted and reused the new application that is being developed. ADAPTDROID exploits this opportunity by relying on a search-based algorithm to automatically adapt manually produced system test cases for a similar interactive application (the *donor*) to the application under test (the *receiver*).

1.2 Thesis Organization

The thesis is organized as follows:

- Chapter 2 describes the state of the art of automatic test case generation for interactive applications and discusses strengths and limitations.
- Chapter 3 empirically compares the most mature techniques available for desktop interactive applications, by relying on a benchmark of subject applications, and provides empirical evidence of the limitations of the state-of-the-art approaches.
- Chapter 4 describes the concept of semantic testing that we introduce in this thesis, and discusses how we exploit applications similarities to automatically generate semantic test case.
- Chapter 5 introduces AUGUSTO. It describes how AUGUSTO produces AIF models that it uses to detect and test AIFs in arbitrary applications and presents the results of an empirical evaluation of AUGUSTO.
- Chapter 6 presents ADAPTDROID. It describes how ADAPTDROID uses a concept of semantic similarity to match events across different applications, shows how ADAPTDROID uses a search based algorithm to identify the best adapted test cases across the possible test cases that can be generated for the AUT, and presents the results of an empirical evaluation of ADAPTDROID.
- Chapter 7 summarizes the contributions of the thesis and discusses open research directions.

Chapter 2

Testing Interactive Applications: State of the Art and Open Issues

2.1 Preliminaries

Interactive applications are software systems that accept input from their users mainly through a Graphical User Interface (GUI) [43].

Graphical User Interface (GUI)

A GUI is a forest of hierarchical windows [1] where at any given time only one window can be active and enabled to receive interactions from the user [77].

Windows host **widgets**, that are atomic GUI elements characterized by several properties, such as (i) the *type*, (ii) the displayed *text*, (iii) *xpath*. The type defines the looks of the widget and the interactions that are enabled on it, for instance, click, drag, or text typing. The text property defines a (possibly empty) text that is shown within the widget and that represents the main information for users to understand the semantics of the actions associated with the widget. The XPath is an address that uniquely identifies the widget *w* in the structural hierarchy of the window [97].

At any time, the active window has a **state S**, which is the collection of the displayed widgets states, that is, the values of their properties. Based on their types, widgets offer users events to interact with the GUI, for instance, users can click widgets of type button or of type menu.

[1] The term window is mostly used in the context of desktop applications, while for Android and Web applications, this concept is expressed using the terms *activities* and *pages*, respectively. For simplicity, in this dissertation, we use the generic term *window* for referring also to *activities* and *pages*.

Users interact with interactive applications performing events on the GUI widgets, such as clicking buttons and filling text fields. Applications interpret user events and trigger the computation of proper business logic functions that eventually modify the state of the GUI, for instance, by changing the active window or modifying the state of the currently active window.

Interactive Application Test Case

A system test case t for an interactive application, from now on simply called *test case*, is an ordered sequence of **events** $\mathbf{t} = \langle \mathbf{e_1}, ..., \mathbf{e_n} \rangle$ that operate on the GUI widgets, , typically identifying them through their XPath property. An event is an atomic interaction between users and widgets, for instance clicking on a button or filling a text field with some *input data*.

The execution of a test case induces a sequence of transitions in the *observable* **GUI state** $\mathbf{S_0} \xrightarrow{\mathbf{e_1}} \mathbf{S_1} \xrightarrow{\mathbf{e_2}} \mathbf{S_2} \ldots \xrightarrow{\mathbf{e_n}} \mathbf{S_n}$, where S_{i-1} and S_i denote the states of the active window before and after the execution of event e_i, respectively.

Each test t is associated with **assertions** that check the correctness of the states observed during the execution of t, and that are used as test oracles. An example of assertion is stating that a widget displays a given text. Assertions can be used to verify the correctness of all intermediate states of t execution or to verify only the final state S_n, with different costs and bug-revealing effectiveness [76]. We use O_t to denote the set of assertions associated with a test t.

2.2 Interactive Applications Testing

In the literature, the term *GUI testing* is used to indicate both the system testing of interactive applications, as it is performed stimulating the application under test through its GUI [13], and the activity of solely testing the correctness of the application GUI [42]. To avoid confusion, in this thesis we use the term *interactive applications testing* to refer to the activity of testing an interactive application by interacting with its GUI to asses the correctness of the *whole* application.

Interactive applications testing requires to derive and execute a set of test cases that thoroughly cover the behaviors of the application and that, ideally, can expose all the faults. Unfortunately, the number of possible GUI interactions that can be executed on an interactive application GUI grows exponentially with the number of GUI widgets and windows, thus making it unfeasible to cover it exhaustively.

It is common practice to test interactive applications *manually*, that is, developers rely on their knowledge of the application to design test cases that

properly sample the AUT huge execution space, and then manually execute them on the GUI. To create executable tests developers often use capture and replay tools [55, 95] that can record the manually produced test cases to then be able to re-execute them automatically without additional effort. However, capture and replay tools still require large human effort as the test cases have to be initially defined and then be updated every time the application GUI changes even slightly.

To avoid this human cost, it would be preferable to employ an automatic tool able to autonomously generate sets of effective test cases sparing the developers from the burden of designing and executing large test suites. However, designing tools to automatically generate effective test suites faces two main challenges: identifying *meaningful GUI interaction*, and defining *automatic functional oracles*.

Meaningful GUI interaction. The space of every possible GUI interaction for a non-trivial application is infinitely large. However, many GUI interactions are not relevant for testing, as they do not execute a meaningful and realistic request of one of the AUT functionalities. For instance, a sequence of random clicks on a flight search application GUI would most likely just navigate among different windows without even getting close to perform a search or buying a ticket. Executing such sequences of GUI interactions most likely results in executing a very small part of the application code, leaving the application functionality largely untested. An effective automatic approach has to identify GUI interactions that execute relevant scenarios, that is, scenarios that trigger the execution of the AUT functionalities and execute the AUT business logic methods. If we consider again the example of an application for booking flights, an important and meaningful interaction is accessing the flight search page, filling out the required fields related to the source and destination, and then clicking on the "search" button. But in the space of the possible GUI interactions, this is a quite rare one since there are many more interactions in which the events described are executed only partially or in the wrong order or intertwined with other events that move the application to a page unrelated with the search.

Automatic Functional Oracle. System crashes, that is, states in which the application suddenly closes making it impossible for the user to continue interacting with it, are easy to detect. Semantic errors, that is, states in which the application provides a wrong output or not allow the user to perform a required task as intended, are hardly detectable without some knowledge of the application semantics. These types of errors are very important, and to be caught require appropriate assertions that check the correctness of the information shown in the GUI after the execution of a given GUI interaction [76]. Writing appropriate assertions requires some knowledge of the semantic of the application under test, and therefore it is very challenging to automatically generate useful assertions.

2.3 Automatic Testing of Interactive Applications

Recently, many researchers and practitioners have investigated the challenging problem of automatically generating test cases for interactive applications. So far the research community efforts have mostly focused on the problem of automatically generating GUI interactions, leaving the problem of generating automatic functional oracles largely open.

The techniques proposed for automating the generation of GUI interactions can be roughly divided into four classes: random approaches [46, 110], model-based approaches [70, 77], coverage-based approaches [36, 68], and similarity-based approaches [21, 84].

2.3.1 Random Approaches

Random approaches are arguably the most simple type of test case generation techniques. They produce test cases as sequences of events identified either randomly or using simple heuristics. Probably, the most famous random technique is Monkey [46], available in the official Android toolkit. Monkey tests an android application by executing a stream of completely random events on its GUI and has become the de-facto standard in testing Android apps.

Other random testing techniques use an observe-select-execute cycle, in which the application GUI is observed to detect which events are enabled before randomly selecting the next event to execute [65, 110]. This approach selects events that stimulate the AUT widgets appropriately, thus improving over Monkey that often executes events in GUI states where there are no widgets that accept that type of event (e.g., clicking in a point where there is no button). These techniques use different strategies to detect enabled events. Testar-Random targets desktop applications, and uses the widget types to detect which events can be performed (e.g., a button widget accepts click events and a text field accepts type events). Dynodroid targets Android apps and performs a lightweight analysis of the source code to detect the event listeners registered for each widget (e.g., if a widget registers a onClick listener then it accepts clicks events).

Ermuth et al. recently proposed a technique that enhances random testing by exploiting users usage traces (i.e., GUI interactions performed by users while operating the application). The underlying intuition is to analyze user interactions with the AUT to infer interaction patterns, i.e., short sequences of elementary action events that represent atomic operations from the user viewpoint, such as opening a menu and selecting a menu item. User interaction patterns are used together with elemental events in a random approach to stimulate the GUI of the

AUT more effectively.

Because of their simplicity, these techniques are able to generate and execute test cases very quickly, and therefore they are often used to perform stress testing. Generally speaking, random testing techniques can be quite effective as they can very quickly cover big portions of the AUT interaction space with their high speed [28, 38]. However, because of their lack of guidance, they struggle to cover those complex meaningful interactions that require a long and precise sequence of events. Thus, these approaches often fail to cover entire parts of the application execution space and might leave relevant AUT functionalities completely untested.

2.3.2 Model-Based Approaches

Model-based approaches derive test cases from a model of the AUT GUI according to a given coverage criteria. This model, which we call GUI model, encodes the structure of the AUT GUI and it is typically built by dynamically exploring the GUI with the execution of several GUI interactions. Model-based approaches propose different definitions and encodings of GUI models. In general, GUI models are directed graphs

$$GUIModel : \langle Windows, Edges \rangle$$

where *Windows* is the set of GUI windows, and Edges is the set of transitions among windows. More precisely, windows are defined in terms of the widgets they contain, and edges are defined as

$$Edge : \langle W_{source}, W_{target}, Trigger \rangle$$

where W_{source} and W_{target} are the windows from which the transition originates and to which it arrives, respectively, while $Trigger$ is an event that operates on a widget in W_{source} that causes the current window shown to pass from W_{source} to W_{target}.

Model-based approaches include serial approaches that consists of first build a GUI model and then generate test cases, and iterative approaches that alternate model building and test case generation.

The serial approach was first proposed by Memon et al. who defined the GUI Ripper [77], a technique that first builds an AUT GUI model by traversing the AUT GUI in depth-first order, and then uses the AUT GUI model to derive the EFG model, that is a model of the partial execution order of the events in the GUI [77].

The GUI Ripper seeded a family of approaches, implemented in a tool called Guitar, that uses the Ripper GUI model to generate test cases, by traversing the model using different model abstractions and various GUI coverage metrics based

on combinatorial interaction testing and heuristics [78, 79, 115]. This general approach was initially proposed for desktop applications, and was later applied to other execution platforms [2, 80]. Arlt et al. extended the Guitar approach introducing a model of data flow dependencies between the events extracted with a lightweight static analysis of the GUI event handlers [13]. Arlt et al.'s approach uses this additional model, called EDG, to identify GUI widgets correlated (i.e., that share a data flow dependency) and that therefore should be executed in the same test case to exercise meaningful execution scenarios. Recently, Linares et al. proposed an alternative way of building the GUI model, by using recorded traces of the AUT usage by the developers. Their intuition is that these traces are probably more effective for exploring the AUT GUI than an automatic exploration approach like GUI ripping [64]. Linares et al. also propose a strategy to infer un-common event sequences (i.e., not seen in the execution traces) that are used to identify corner cases.

Approaches that merge model inference and test case generation seamlessly generate test cases and use the information obtained with their execution to update the GUI model. These techniques start by generating random test cases and then move to test case generation strategies based on the information contained in the GUI model. The main difference between the approaches in this class is the strategy they use to generate test cases. AutoBlackTest (ABT) uses Q-Learning to learn how to generate test cases that traverse the GUI of a desktop application triggering the most changes in said GUI [70]. The core intuition of ABT is that a big change in the GUI in response to an event suggests that a relevant computation was performed. ABT generates test cases alternating between events selected at random (80% of the times) with events that maximize the GUI changes (20% of the times) to balance between exploration of the GUI and exploitation of the Q-Learning. Testar-QLearning is a variant of Testar-Random (described in the previous section) that uses Q-Learning [19]. In this case, Q-Learning is used to generate test cases that are able to execute events that were not executed so far, thus generating always different test cases compared to the previous ones. SwiftHand instead uses a continuously built GUI model to generate test cases that can exercise the windows of an Android app while minimizing the number of times the app is restarted [37].

Model-based approaches are limited by the completeness of the GUI model that they build by exploring the AUT GUI. In many applications, some relevant GUI windows may be reachable only by executing complex sequences of events that some exploration strategies may miss, resulting in potentially big portions of unexplored space of GUI interactions. Moreover, current model-based approaches do not distinguish meaningful from irrelevant interactions when producing test

cases, thus resulting in potentially large amounts of irrelevant test cases.

2.3.3 Coverage-Based Approaches

Coverage-Based approaches leverage the AUT source code to drive test case generation and maximize AUT source code coverage, ideally, executing each AUT source code statement.

Several coverage-based approaches steer test case generation leveraging *genetic algorithms*, a classical search-based approach [16] inspired by natural selection that has been used in many software engineering contexts [50, 51, 52]. In a nutshell, a genetic algorithm starts with a random population of solutions for its search problem and iteratively evolves them (using crossover and mutations) selecting the fittest individuals (according to a fitness metric) for reproduction to produce a better population for the next iteration. In the context of testing interactive applications, the search problem is defined as finding the set of test cases with the highest AUT statement coverage. Then, using statement coverage as the fitness metric, genetic algorithms evolve an initial set of random test cases combining/mutating them to finally retain those that achieve the highest coverage [48, 66, 68]. In some cases, these techniques aim to maximize additional goals along with code coverage. For instance, Sapienz maximizes code coverage while also maximizing the number of crashes found and minimizing the length of the test cases.

Other techniques instead use symbolic/concolic execution [33] to generate test cases that aim to achieve 100% code coverage [4, 36, 45]. Symbolic execution is notoriously very expensive as it requires to analyze the AUT source code and it is typically used on rather small command-line software or at the unit level [11]. To reduce the computational cost and make it feasible to apply this analysis for testing interactive applications, Ganov et al. proposed a technique that symbolically executes only GUI event handlers [45], while Cheng et al. use a cloud-based architecture that parallelizes concolic-execution on multiple machines to foster scalability [36].

Azim et al. proposed a coverage-based test case generation strategy based on analyzing the source code and statically computing the activity transition graph of android apps [15]. This approach systematically covers the activity graph by generating appropriate sequences of GUI events or directly triggering intents that can open the targeted activities.

Coverage-based approaches aim to maximize code coverage, thus they are often able to identify meaningful GUI interaction sequences that cover significant portions of the code, and discard non-meaningful interaction sequences that

cover little code. However, coverage-based approaches are limited by the cost of analyzing the source code and often do not scale well to complex applications.

2.3.4 Similarity-Based Approaches

In recent years, a new category of approaches is emerging: the approaches based on the intuition that different applications often share some similarities that can be leveraged for test case generation. Similarity-based approaches are closely related to this dissertation as they share the same intuition of our research hypothesis.

By observing that the GUIs of different applications are often similar, Mao et al. proposed to analyze user interactions across different android applications to identify generally applicable GUI interaction patterns, that can be used as building blocks to generate complex GUI interactions using a state-of-the-art coverage-based approach [69]. In their work, they use crowd testing to collect GUI interactions from multiple applications and they were able to detect several short interaction patterns that improved the effectiveness of the test case generation approach.

Moreira et al. also leveraged the idea of generally applicable user interactions pattern for testing. Moreira et al.'s approach instead of mining these patterns from execution data relies on a set of pre-defined patterns that testers use to model the AUT behaviors [84]. Moreira et al. approach requires an initial step in which a developer maps the predefined patterns to the elements of the AUT GUI to which they can be applied. After that, the approach is able to generate test cases that exploit those patterns using predefined test scenarios.

Some recent research results propose to leverage test cases of existing applications to generate effective test cases for a different AUT. The Rau, Hotzkow, and Zeller's ICSE 2018 poster [91] illustrates an approach that learns how to generate meaningful test cases for an AUT from tests designed for other applications. Behrang et al.'s ICSE 2018 poster proposes to migrate test cases across similar interactive applications [21]. Behrang et al.'s core idea is to reuse the test cases of an application to test a similar one (after suitable syntactical changes). Behrang et al initially proposed a greedy algorithm to adapt test cases across semantically equivalent applications in the context of automatizing the grading of student assignments [22], and are currently working on extending their technique to support also adaptation across similar but not equivalent applications.

2.4 Limitations and Open Problems

Despite the many approaches proposed to automatically generate test cases for interactive applications, the problem is still largely open and state-of-the-art techniques suffer from two main limitations: *ineffective exploration of the execution space* and *lack of functional oracles*.

Ineffective exploration of the execution space: Automatic test case generator shall produce test cases that exercise all the *semantically meaningful* execution scenarios of the AUT to effectively test interactive applications. This requires generating complex GUI interactions which execute precise, and often long, sequences of events that suitably stimulate the AUT functionalities. Current techniques can cover evenly the events in the GUI, but cannot recognize which combinations of events represent meaningful scenarios, thus they are often unable to cover important GUI interactions that are required to trigger a certain functionality. Let us consider for instance the case of an expense manager application in which a user can set a monthly budget. A meaningful scenario to test would be to set a monthly budget and then add a set of expenses in a certain month that exceeds the budget. Generating a test case able to cover this scenario is extremely unlikely using one of the state-of-the-art approaches as they lack any guidance able to recognize this scenario as relevant.

Lack of functional oracle: Current techniques can detect crashes, but not functional failures. Thus, they miss many relevant failures even when they execute GUI interactions that exercise bugs. If we consider the previous example again, in case the expense manager application erroneously calculated the sum of the expenses and showed to the user that the budget was not exceeded, none of the current techniques would be able to detect that behavior as an erroneous.

Choudhary et al.'s empirical comparison of techniques for testing Android applications [38] provides empirical evidence of these limitations. In their paper they write:

> *"Considering the four criterion of the study, Monkey would clearly be the winner among the existing test input generation tools, since it achieves, on average, the best coverage, it can report the largest number of failures [. . .]"*

Generally speaking, this means that although sophisticated techniques exist, they perform worse than simple random testing. Also, Choudhary et al.'s comparison indicates that current techniques do not cover well the AUT code, as in their study Monkey achieved an average statement coverage of less than 50%. Zeng et al.'s study on automatic testing of Android apps in an industrial settings also

reports similar low coverage results [118]. These low coverage results indicate that current techniques have limited effectiveness in executing (meaningful) GUI interactions that can execute significant portions of the AUT source code.

Choudhary et al. confirm the lack of functional oracles limitation of state-of-the-art approaches as they write:

> *"None of the Android tools can identify failures other than runtime exceptions [. . .]"*

Choudhary et al. also report that the vast majority of runtime exceptions detected are general Java/Android exception and only a few are custom exceptions defined in the application under test. This suggests that current techniques can be effective in revealing crashes related to the underlying Android framework, but are less effective in detecting errors related to the business logic of the tested application.

In a nutshell, these empirical studies provide compelling evidence that the problem of testing interactive applications is still largely open, and current approaches do not exercise well vast portions of the behavior of interactive applications. The results of these studies are specific to Android applications and may not be straightforwardly generalizable to all interactive applications, for instance, due to the different limitations imposed by the size of the screen. In this Ph.D. work, we comparatively evaluated the state-of-the-art automatic test case generators for desktop interactive applications [88] to investigate the generalisability of previous results. Our study confirms the results of the previous studies on Android applications, by showing that test cases produced with automatic test case generators for interactive applications suffer for the same limitations observed in the Android domain. We report the results of the study in the next chapter.

Chapter 3

Automated Testing of Desktop Applications: an Empirical Study

In this chapter, we present the results of our comparative empirical study about the effectiveness of the main state-of-the-art automatic test case generators for desktop interactive applications. We evaluated the main techniques to generate test cases for desktop GUI applications comparing them on a common set of subject applications in terms of:

1. *Fault revealing ability.* We consider the number of faults each testing technique reveals in the subject applications.

2. *Execution space sampling ability.* As a proxy measure for this criteria, we consider the statement and branch coverage achieved by each technique on the subject applications.

3. *Time Efficiency.* We consider the efficiency of each technique with different time constraints.

3.1 Evaluated Tools

We conducted our study searching the state-of-the-art for automatic test case generators for desktop application, implemented in tools usable out of the box. We found several random, model-based, and coverage-based tools for generating test cases for desktop applications, but no similarity-based techniques that target desktop applications. We installed the tools publicly available at the time of writing, and followed the advice of the developers for set up and usage when developers replied to our requests. In the absence of developers' support, we

Table 3.1. Testing tools selected for the study

Tool	Type	Help
Testar-Random [110]	Random	✓
Testar-QLearning [19]	Model-Based	✓
Guitar-EFG [77]	Model-Based	x
Guitar-EIG [115]	Model-Based	x
Guitar-EDG [13]	Model-Based	x
AutoBlackTest [70]	Model-Based	✓

followed the notes in the paper and in the documentation to obtain the best set up, to our knowledge.

Table 3.1 reports the testing tools that we compared in the study. We considered also other testing techniques, but their tools were either not available or we were not able to use them correctly despite receiving help from their developers. The table reports the tool name, type (according to the classification discussed in section 2.3), and whether we received support by their developer to use them appropriately. We considered different variants of some tool (e.g., Testar and GUITAR) as separated tools when implementing largely different techniques. In the remainder of this section, we briefly describe the techniques compared in the study.

TESTAR-RANDOM [110]: a technique implemented in the Testar suite. Testar-Random generates test cases by iteratively detecting the events enabled in the AUT GUI, and randomly selecting and executing one of them. Testar-Random iterates this process until reaching a predefined maximum test case length (100 by default), and then restarts the AUT to start a new test case. Testar-Random keeps generating test cases until a time budget is reached.

TESTAR-QLEARNING [19]: an evolution of Testar-Random. Testar-QLearning selects the events to execute using QLearning. Testar-QLearning builds a model in which each event is associated with a reward that is inversely proportional to the number of times that event was executed. Using this QLearning model, Testar-QLearning generates test cases that maximize the total reward collected, thus each new test case privileges sequences of un-executed events.

GUITAR-EFG [77]: a technique implemented in the Guitar suite. Guitar-EFG uses a model called event flow graph (EFG) that Guitar builds by automatically exploring the GUI with GUI ripping. The EFG models the execution order of the events in the GUI, identifying which events are enabled after the execution of a certain event. Guitar-EFG generates a test suite that maximizes the combinatorial

coverage of a given strength t of the EFG model, that is, it covers all possible event sequences of length t in the EFG model.

GUITAR-EIG [115]: a version of Guitar that relies on a different model, called event interaction graph (EIG). An EIG models only the events that interact with the underlying business logic of the application, filtering out events such as opening/closing windows. Since it retains only business-logic related events, Guitar-EIG allows to generate more relevant test cases than Guitar-EFG.

GUITAR-EDG [13]: a version of Guitar that relies on a different model, called event dependency graph (EDG). Guitar-EDG builds EDG models using a lightweight static analysis of the GUI event handlers that captures dataflow dependencies between GUI events. In a nutshell, the EDG encodes as data-dependent the GUI events that read/write the same data when executed. Guitar-EDG generates test cases that combine only data-dependent events and that thus are correlated.

AUTOBLACKTEST (ABT) [70]: a technique that uses Q-Learning to learn how to generate test cases that can explore in depth the GUI states space of the AUT. ABT builds a model that associates a high reward to the events that cause big changes in the AUT GUI state, following the intuition that a big change in the GUI state relates to a relevant computation performed. ABT then generates test cases by alternating the execution of random events (80% of the times) and events selected according to the QLearning model (20% of the times), that is events that can cause big changes in the GUI state, to alternate between exploring the GUI space and exploiting the QLearning model.

3.2 Subject Applications

As a benchmark, we selected a set of Java desktop GUI applications from different application domains and of different sizes and complexities. In order to be fair, we selected the subject applications from those used in the empirical evaluation of the selected tools [13, 70, 115]. The selection of subjects was constrained also by the fact that some of the selected testing tools work only with Java applications.

Table 3.2 shows the subject applications we selected as our study benchmark. The table reports the name, version, type, and size in thousands of lines of code (column *kLOCs*) of each subject.

Table 3.2. Subject applications

Name	Vers.	Type	kLOCs
Buddi [32]	3.4.0.8	Personal finance management	10.0
UPM [105]	1.6	Password management	3.4
Rachota [90]	2.3	Personal tasks management	10.5
CrossWord [25]	0.3.5	Crosswords building and solving	1.8

3.3 Experimental Setup

We executed each testing tool in Table 3.1 on the subject applications for 15 hours. Since the *Guitar* tools do not set a time budget, we used the combinatorial coverage strength setting to control the execution: Guitar generates a test suite that covers all combinations of events in the model of a given length n and then executes it. With big values of n Guitar spends most of the 15 hours to generate test cases, leaving only a short time for executing them. With small values of n, Guitar might terminate both phases much before the end of the 15 hours time window, without completely exploiting the available time. For these reasons, we run each of the Guitar tools with a setting that allowed the tool to spend the longest time running test cases within the time budget.

We compared the fault revealing ability of the tools, comparing the failures revealed by each compared tool in the subject applications. Testar-Random and Testar-QLearning after testing the AUT report suspicious test cases that might reveal bugs in the AUT. We considered those test cases as *failing test case*. In addition, we inspected the standard error of the subject applications after the execution of each test case generated with the evaluated tools. We consider each test case which execution triggered an uncaught exception in the subject application as a *failing test case*. We inspected all the failing test cases and we classified as a *bug-revealing* every test case that manifests an error in the subject application from the user perspective (thus a test case that triggers an uncaught exception that does not cause any issues to the user while using the application would not be classified as a bug-revealing). Since in some situations multiple test cases triggered the same failure, we manually clustered the failures according to the user observable type of error. We consider each cluster as representative of a single fault in a subject application, and we regard each tool that produces a failing test case in the cluster as having revealed that fault.

We compared the ability of tools to sample the execution space, by comparing statement and branch coverage computed with Cobertura [39].

We compared the efficiency of the tools, by sampling the results of the execu-

tions of the tools (in terms of faults detected and code coverage achieved) after four intervals during the 15 hours runs: 1 hour, 3 hours, 7 hours, and 15 hours. Since, Guitar first generates the test cases and then execute them, sampling their executions in this way would have been unfair (after 1 hour the Guitar tools would probably still be in the test generation phase, thus having executed no tests, detected 0 faults, and achieved 0% coverage). For this reason, we run Guitar four times on each subject application, each time with a different time budget (1 hour, 3 hours, 7 hours, 15 hours), bounding the run time of Guitar as described above.

We run our experiments on two virtual machines, a Windows 10 OS and Ubuntu 16.04 OS machine. We use two different virtual machines because of the different operating systems requirements of the tools. Each virtual machine was configured with 4 cores and 4GB of RAM.

To configure the tools and the subject applications with an optimal initial setup (up to our knowledge), we executed the subject applications with some initial data to increase the possibility of the testing tools to execute interesting executions scenarios. For instance for Buddi, an expense management tool, we set up some expenses and some initial accounts.

We repeated each run of the compared tools three times to mitigate the results randomness.

3.4 Experimental comparison

Table 3.3 summarises the results of our experiments. The table reports average data that we obtained by executing each tool three times on the subject applications for 15 hours. For each tool and each subject application, the table reports the average statement and branch coverage (columns *Statement Cov.* and *Branch Cov.*), the average number of distinct faults detected in each run (column *# Faults*), and the total number of distinct faults detected over the three runs (column *Tot. Faults*).

3.4.1 Fault Revealing Ability

In our study, none of the six testing tools we experimented did ever crash the application under test (i.e., make it closing abruptly). However, in many cases they triggered some exception to be printed in the application standard error. We examined all these exceptions to check whether they were the manifestation of actual faults that affect the user, and we classified those tests as fault revealing. The experiments revealed a total of 5 distinct faults.

Table 3.3. Coverage and Failure revealing capabilities of the tools after 15 hours test sessions

Subject	Tool	Statement Cov.	Branch Cov.	# Faults	Tot. Faults
Buddi	Testar-Random	36.7	22.0	2.0	2
	Testar-QLearning	41.7	25.0	2.0	2
	AutoBlackTest	54.7	36.0	3.0	3
	Guitar-EFG	44.0	22.0	1.0	1
	Guitar-EIG	44.0	23.0	1.0	1
	Guitar-EDG	42.0	21.0	1.0	1
CrossWord	Testar-Random	19.0	11.0	0.0	0
	Testar-QLearning	15.3	5.0	0.0	0
	AutoBlackTest	33.0	15.0	0.0	0
	Guitar-EFG	28.0	10.0	0.0	0
	Guitar-EIG	26.0	7.0	0.0	0
	Guitar-EDG	26.0	7.0	0.0	0
Rachota	Testar-Random	64.0	40.0	0.0	0
	Testar-QLearning	42.7	28.7	0.0	0
	AutoBlackTest	43.3	29.3	0.0	0
	Guitar-EFG	62.0	38.0	0.0	0
	Guitar-EIG	61.0	37.0	0.0	0
	Guitar-EDG	59.0	33.0	0.0	0
UPM	Testar-Random	69.0	45.7	1.3	2
	Testar-QLearning	61.0	32.0	0.0	0
	AutoBlackTest	65.7	40.0	0.0	0
	Guitar-EFG	52.0	24.0	0.0	0
	Guitar-EIG	52.0	24.0	0.0	0
	Guitar-EDG	49.0	24.0	0.0	0

Testar-Random is the testing tools that during the 15 hours run detected the most faults, detecting four faults, two in Buddy and two in UPM. Instead, AutoBlackTest detected three faults, all in Buddy, and the Guitar tools detected only one fault in Buddy. All faults except one were consistently detected in all the runs, as indicated by the same values in columns *# Faults*) and *Tot. Faults*). Only one fault detected by Testar-Random in UPM was not detected in all three runs but was detected only in one of the three runs.

Table 3.4 shows which faults each tool was able to detect. In the table, the three faults in Buddy are labeled B1, B2, and B3, the two faults in UPM are labeled U1 and U2. The table shows that fault B1 was detected by all compared tools, fault B2 was detected by all tools but one, the other three faults were detected by one tool only.

This result provides additional evidence to Choudhary et al.'s statement that suggests that random is the most effective approach in detecting faults.

Table 3.4. Faults detected by tool

	B1	**B2**	**B3**	**U1**	**U2**
Testar-Random	✓	✓		✓	✓
Testar-QLearning	✓	✓			
AutoBlackTest	✓	✓	✓		
Guitar-EFG	✓				
Guitar-EIG	✓				
Guitar-EDG	✓				

3.4.2 Execution Space Sampling Ability

Testar-Random and AutoBlackTest approaches achieved the highest code coverage. AutoBlackTest achieved the highest coverage for two subjects, and a significantly lower coverage than other tools for only one subject. Testar-Random achieved the highest coverage for two subjects as well, but a quite low coverage for the other two subjects. Guitar approaches always achieve good coverage, but never the highest.

Even though AutoBlackTest is a model-based approach, during test case generation it selects most events (80%) randomly. Thus, Testar-Random and AutoBlackTest coverage again further corroborates Choudhary's statement, confirming that Random is one of the best approaches to explore the GUI execution space of the AUT.

3.4.3 Time efficiency

Figure 3.1 shows the average distribution of statement coverage and detected faults over time. Testar-Random approach is the quickest in finding failures, since it detects three out of four failures within an hour, and detects all four failures within the first three hours. AutoBlackTest detects three failures in seven hours. The distributions of the coverage achieved with the different approaches are similar. The Guitar approaches increase code coverage slightly quicker in the first hour but saturate at a lower upper bound than the other approaches: they do not further increase coverage in the 14 hours after the first one. Instead, the other approaches increase code coverage in a longer period. Testar-Random achieves the most noticeable growth: it achieves one of the lowest coverage in the first hour, but the statement coverage keeps growing (albeit not much) for the following hours.

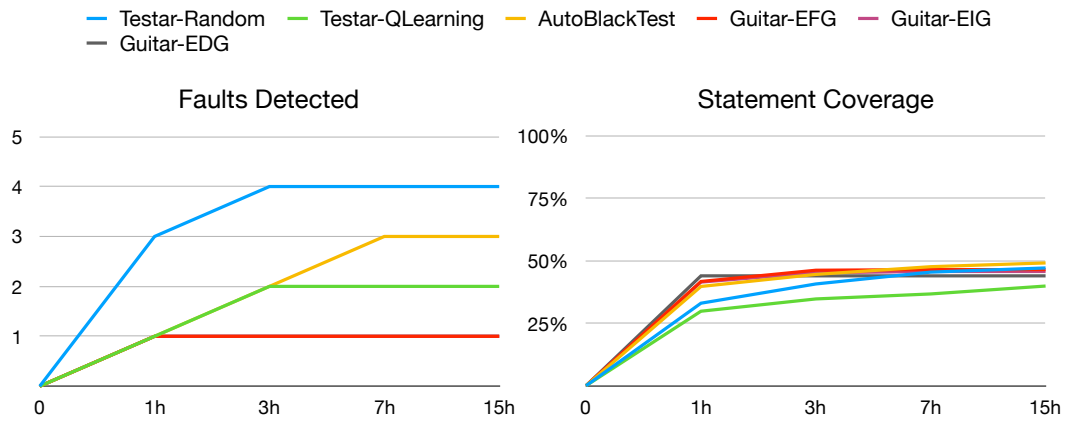


Figure 3.1. Distribution of detected faults coverage within the time budgets for the different tools

3.5 Discussion

Our experiments lead to some interesting considerations:

1. All the techniques compared in the study detected only faults that manifest as uncaught exceptions. Testar-Random and Testar-QLearning, potentially, can also detect faults that manifest as error dialog messages in the AUT. However, in our study these test oracles did not contribute in revealing any fault. Thus, the experiments confirm that the fault revealing ability of current techniques is limited by the limited effectiveness of the test oracles they employ.

2. All techniques covers the code only up to a limited extent. Some techniques achieved a reasonable statement coverage for three subjects (between 50% and 70%), but for CrossWord, none of the techniques achieves more than 33% statement coverage, and in average the studied techniques achieved less than 50% statement coverage for the four subjects. We do not have information about the amount of dead code in the applications, thus we do not have all elements to draw a valid conclusion. However, we believe to be unlikely that all the uncovered code (more than 50%) is actually dead code.

3. Our results confirm Choudhary et al.'s results that indicate the higher effectiveness for random approaches [38]. Our study indicates that Test-Random performed better in terms of detected faults and that both Testar-Random and AutoBlackTest obtained the highest coverage among the experimented

approaches. Testar-Random is a completely random approach, and AutoBlackTest generates test cases by alternating random events (80% of the times) and learned events (20% of the times). This result shows that techniques with significant random strategies, can detect more faults and achieve higher coverage on the long run than other techniques.

4. Our study suggests that model-based techniques that generate test cases after building a GUI model (the Guitar suite), are not very effective in detecting faults and thoroughly covering the execution space. We argue that this depends on the combinatorial nature of the Guitar approach to generate test cases. Long sequences of events may be required to reveal subtle faults and execute large portions of the source code. However, Guitar generates fairly short test cases since generating a test suite that covers all combination of events of higher length is unfeasible because of combinatorial explosions.

5. Testar-Random is the most effective in short time sessions (less than 3 hours). In our experiments with short test sessions, Testar-Random detected the highest number of faults and achieved a good code coverage, albeit other techniques achieved a slightly better code coverage. Guitar techniques can achieve code coverage quite fast, but have low fault revealing ability.

6. Long test sessions (above 7h) do not pay off. In our experiments after 7 hours, all testing techniques we experimented with barely achieved new coverage, and none detected any new fault.

Our empirical study indicates some relevant limitations of the state-of-the-art approaches discussed in Section 2.4 and confirms the empirical results of the studies by Choudhary et al. and Zeng et al. The fault revealing ability of the current techniques appear to be quite limited since none of the techniques can rely on oracles able to detect errors that do not manifest as system crashes or runtime exceptions. Also, the automatically generated test suites do not achieve a fully satisfactory code coverage (in our study statement coverage does not exceed an average of 50%). Our study indicates that these limitations cannot be overcome by simply allocating more time. As Figure 3.1 shows, in the first 3 hours the 6 testing tools detected almost all the faults and barely covered new code and they did achieve new significant results for the following 12 hours.

3.6 Threats to validity

A threat to the internal validity of our study is related to the manual inspection of the generated test cases executions performed by the authors of this study to identify faults. To mitigate this threat, we classified a test case execution as fault-revealing only if all researchers involved in this empirical study agreed that it exposes an error in the subject application.

A threat to the internal validity of our study relates to the setup of the compared approaches. We mitigate this threat by contacting the developers of all the tools and asking them to support us in the usage of their tool.

An external validity threat of our study relates to the generality of the results about the set of approaches and subject applications we experimented with. We mitigate this threat by experimenting with all approaches for which there was an available tool, and contacting developers to properly set up the tools. We mitigate any bias in identifying subject applications, by selecting subjects that belong to a variety of domains, and that were already used in the empirical evaluation of the tools we compared.

Another threat to the external validity of our study derives from the limited number of subjects used in the study and on the low number of faults revealed on those subjects. Even though four subjects and five faults are not enough to generalize our results, if we consider that in our study we referred to publicly distributed applications and real faults only, i.e., faults that were present in subjects distributed versions, our results provide an evidence, albeit limited, of the performance of the studied tools in a realistic environment.

Chapter 4

Similarities Among Applications: An Opportunity for Testing Interactive Applications

As discussed in the previous chapters, state-of-the-art approaches to automatically test interactive applications suffer from two limitations: the *ineffective exploration of the execution space* and the *lack of functional oracles*.

State-of-the-art approaches suffer from these limitations because they rely mostly on structural information obtained either from the GUI or from the source code, largely ignoring the semantics of the application under test (AUT). By considering only structural information, state-of-the-art approaches explore the AUT GUI interaction space aiming to some kind of structural coverage, hardly distinguishing meaningful test cases from irrelevant ones or correct from incorrect behaviors of the AUT.

To overcome these limitations, automatic testing approaches should complement structural information with *semantic information* of the AUT, aiming at what we define a **semantic testing approach**. In principle, a semantic testing approach exploits some semantic information to identify semantically relevant behaviors of the AUT, how they should be executed meaningfully, and how their correctness can be assessed. Even a simple fragment of semantic information such as knowing that two AUT windows are used to perform completely unrelated tasks could be useful: a semantic testing approach could rely on this information to reduce the GUI interaction space avoiding to generate test cases that alternate events between two unrelated windows.

The main challenge that we face to design semantic testing approaches is to identify which semantic information can be leveraged, how to collect it, how to

exploit it, and how to encode it in a way that can be leveraged by an automatic testing approach. Since we aim at cost-effective approaches, semantic information shall be obtainable with low costs, that is, it should be *available in the field* and shall not be produced ad-hoc, like in the case of specification-based testing [72, 89, 108].

In this Ph.D. thesis, we propose to leverage the similarities that exist among different interactive applications to achieve cost-effective semantic testing.

Considering the interactive applications currently available in the market, we can notice that there exists a high level of similarity among different applications. For instance, each app category in the Google Play Store is populated by quite similar apps that share the overall same goal and that mainly differ in the graphical aspects, access permissions, side features, and user experience [63]. In some cases, similarities are very extensive. For example, we found thousands of applications with the exact same goal and almost the same semantics, by simply searching the Google Play Store for applications to track personal expenses with the query "personal expense manager".

In a nutshell, these similarities among applications result in the fact that often the same functionality can be found implemented in different applications. Informally, in this thesis we use the term *functionality* to refer to a semantically coherent and correlated *set of user operations* available to the users through the application GUI that are used to fulfill a specific user goal. For instance, a set of CRUD operations all referring to money transactions in a personal expense manager application is a functionality as it is composed of a set of operations on the GUI (implemented with buttons and other widgets) that are used to fulfill the goal of managing (adding, deleting, editing) money transactions.

The availability of multiple applications that implement the same functionalities is an opportunity that, although mostly overlooked so far (see Chapter 2), has been already exploited in some V&V contexts. For instance, it has been successfully leveraged to support malware detection [47].

In this Ph.D. thesis, we propose to leverage this opportunity to define cost-effective semantic testing approaches that can effectively test these *recurrent* functionalities of interactive applications. More precisely, we defined approaches that can test functionalities that are not specific to one application only more effectively than current state-of-the-art techniques. In this way, a relevant subset of the features present in an application can be thoroughly tested automatically, alleviating the tester from part of the verification effort.

To accomplish this goal, we introduced two new approaches based on the original ideas of: (i) pre-modeling some particularly standard and popular func-

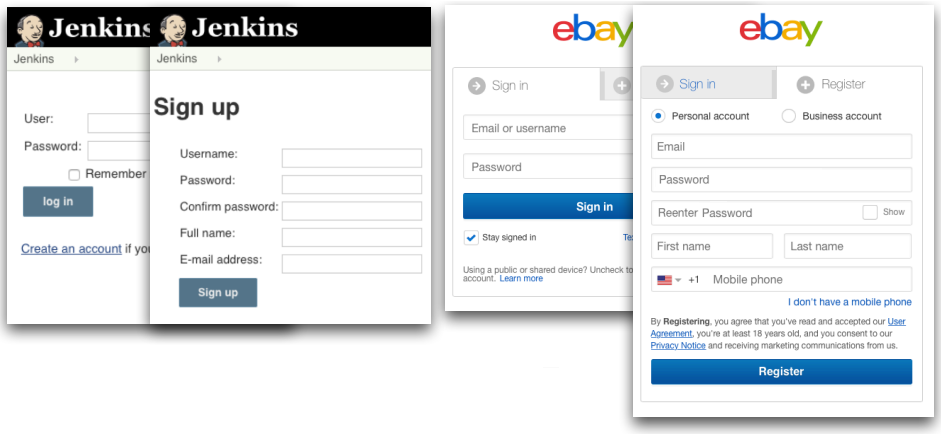


Figure 4.1. Sign in and Sign up in Jenkins and Ebay

tionalities and then use these predefined models to identify and test these functionalities effectively in any AUT, and (ii) re-using test cases (and therefore the testing effort to implement them) across similar applications. We described these two original ideas in the next sections.

4.1 Application Independent Functionalities

Some popular functionalities have reached a high level of standardization due to their diffusion in many applications, meaning that they are implemented in similar ways across different applications GUIs, and have become easily recognizable. This intuition has been investigated in the field of UI design creating catalogs of UI design patterns [103, 109] and has led to the definition of GUI designing tools [23] that allow creating new GUIs by composing recurring GUI patterns.

These recurrent functionalities not only share similarities in their implementation on the GUI, but they also typically share the same semantics, thus offering a consistently similar behavior that can be barely distinguishable across applications, once abstracting away some concrete details. For instance, search and save operations may affect different kinds of entities, but in all cases, they search and save an entity of some type. However, due to their popularity, the semantics of these functionalities is not explicitly provided, since users and developers have clear expectations that derive from their frequent use in many different applications.

In this thesis, we refer to these recurrent functionalities as *application independent functionalities (AIF)* and indicate the shared expectations about their

common behaviors as *common sense knowledge*. These functionalities are pervasively present in software applications, and their behavior remains always the same, despite minor differences.

Application independent functionalities (AIF) satisfy the following properties:

- They are *commonly present* in several applications. Some AIF might be more common in certain domains, for instance, the shopping cart management functionality is very common in the e-commerce domain, whereas others are generally common, such as CRUD functionalities;
- Their semantics is largely *application independent* thus it can be defined abstractly in a way that is independent on the specific interactive application. For example, the general semantics of CRUD functionalities does not depend on the type of the handled object;
- They can be activated from the GUI according to structural *GUI patterns* that users can recognize [103, 109]. For instance, the sign in and sign up functionalities in many applications use similar sets of widgets, although these widgets have different look and feel and placements in the windows. Figure 4.1 shows an example of the look and feel of authentication functionality across GUIs of two different applications.

The *authentication* functionality, composed of sign in, sign up, and sign out operations, is a good example of AIF since: (i) it provides an overall functionality, authentication, which can be found in many applications, (ii) its semantics is well-known and mostly independent on the specific application, and (iii) its presentation in the GUI is predictable and easily recognisable.

There are several other examples of AIFs: the functionality of creating, reading, updating and deleting (CRUD) objects of a type, the functionality of saving the work on a file and then reloading it, the functionality of searching and booking a certain service (car, hotel, flight), the functionality of handling an e-commerce cart. Despite their diffusion, AIFs can easily include faults, even in extremely popular applications, and thus require careful testing. For instance, faults impacting an extensive number of users have been reported for CRUD operations in Jenkins [58] and for authentication operations in DropBox [44].

In the context of testing, AIFs represent a unique opportunity: their semantics can be specified *once for all* (in a machine-readable way) according to common sense knowledge, to be then *automatically adapted and reused* to test the specific AIFs in the applications under test. In order words, if defined and made available once and for all, a semantic testing approach could leverage the semantics of AIFs to achieve effective automatic testing of these functionalities.

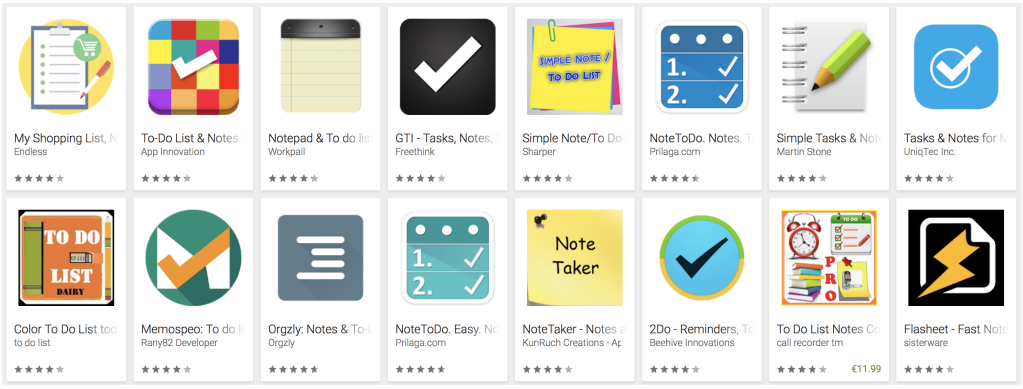


Figure 4.2. Example of similar note-pad applications in the Google Play Store

In this Ph.D. thesis, we exploit the opportunity of AIFs to achieve semantic testing in AUGUSTO (described in Chapter 5), a technique able to autonomously recognize AIFs in an application and use them to automatically generate semantically relevant test cases equipped with functional oracles.

4.2 Cross-Application Test Case Adaptation

Often developers write executable system test cases to test interactive applications [41]. In the case of open source applications, these test cases are publicly available in software hosting platforms, such as GitHub or BitBucket. Recently, Kochhar et. al. presented a study on 627 open-source Android applications showing that 14% of them are released with test cases [60]. Even though the release of executable test cases is not common practice yet, the amount of open-source applications is growing over time, and the amount of openly available tests is expected to grow accordingly.

The presence of many applications that implement semantically similar functionalities and the availability of manually-written test suites lead to an interesting opportunity. When testing an application, developers could rely on the test cases available for a similar application to test their own application. Intuitively, if two applications share the same functionality, the test cases to verify that functionality should be quite similar among the two applications. Thus, it should be possible to adapt the test cases available for an application to obtain test cases for the other application. Performing such adaptation automatically would largely reduce the effort for designing and writing test cases for common test scenarios, leaving developers with the task to write only test cases for functionalities that are not shared among similar applications.

More precisely, let us define two interactive applications as *semantically equivalent* if they implement *exactly* the same functionalities. Intuitively, two applications that implement the same functionalities are applications that differ only in the graphical aspects in which they offer the same set of functionalities to the users. Semantically equivalent applications share the same test scenarios and therefore can be tested in similar ways, and we argue that the test cases written for an application could be used to test the execution scenarios of a *semantically equivalent* application, after minor adaptations.

The definition of *semantic equivalence* is quite strict, and applications that are semantically equivalent might be rare, and difficult to identify.

In this thesis, we consider the less strict concept of *semantical similarity*. Informally, we consider two applications semantically similar if they address the same core user needs, that is, they implement the same *core* functionalities. As an example we consider two applications that keep track of expenses as similar even if they share a set of core functionalities, but implement some different functionalities. Undoubtedly, this concept of semantic similarity is quite vague, but it is largely used [47, 63, 75], and even official application stores, such as Google Play Store and Apple Store, implement automatic approaches to suggest apps that are semantically similar to a given app. In general, these approaches cluster semantically similar applications by analyzing their descriptions (in natural language) together with any other information provided by the developers (tags, categorization).

In this thesis, we propose the novel idea of *cross-application test adaptation* among *semantically-similar* applications: Cross-application test adaptation generates semantically meaningful test cases for an AUT by adapting *some* of the test cases of a *semantically similar* application. The test cases generated through adaptation should retain both meaningfulness and functional oracle of the original (manually-written) test case, thus being more effective than those generated by state-of-the-art testing approaches based on structural information only.

We discuss the opportunity of cross-application test adaptation through the example of a software company that develops a new application “newApp” for managing to-do lists. The company can identify thousands of applications semantically similar to “newApp”, by simply searching the Google Play Store with the query “to-do list” . Figure 4.2 shows the first 16 results of the query performed on March 2019. The *Orgzly: Notes & To-Do Lists* application[1] (third applications on the second row in Figure 4.2) contains several test cases in its

[1]https://play.google.com/store/apps/details?id=com.orgzly

open-source GitHub repository[2]. Some of these tests represent common test scenarios that could be adapted for testing similar apps. For example, the test `testCreateAndDeleteBook()` exercises the test scenario that a newly created note disappears from the GUI after being deleted. The developers of "newApp" could (automatically) adapt the existing test cases of *Orgzly: Notes & To-Do Lists* to effectively test their new app.

Automatically adapting test cases across interactive applications is a challenging problem that has not been explored yet, to the best of our knowledge. As mentioned in Chapter 2, only a few preliminary ideas have been proposed, and at the time of writing the problem is still largely open. To address this problem, our intuition is that semantically similar applications implement analogous behaviors which are exercised through widgets that often have semantically similar textual descriptors, for instance, "Add To Do" and "Create Task". As such, test cases could be adapted across similar applications generating GUI interactions that operate on widgets with semantically similar descriptors.

In this Ph.D. thesis, we exploit the opportunity with ADAPTDROID (described in Chapter 6), the first full-fledged technique to automatically adapt test cases and their functional oracle across similar applications.

[2]https://github.com/orgzly/orgzly-android

Chapter 5

AUGUSTO: Semantic Testing of Application Independent Functionalities

In this chapter, we present AUGUSTO (AUtomatic GUi Semantic Testing and Oracles), a semantic testing approach that automatically generates test cases for application independent functionalities (AIFs) in interactive applications. AUGUSTO is based on the intuition that some functionalities are commonly implemented in many applications and, despite having small syntactic differences, they always implement the same semantics, for instance, login, CRUD, and search functionalities. We call these functionalities *application independent functionalities (AIF)*. In this thesis, we investigate how the existence of AIFs can be leveraged to achieve semantic testing, propose a way to define them abstractly and independently from the specific applications, and present AUGUSTO, an automatic approach to discover AIFs, adapt their definition to the AUT, and generate test cases with semantic oracles.

In this section, we introduce the approach through a motivating example, present AUGUSTO, and discuss the results of the empirical evaluation of AUGUSTO.

5.1 Motivating Example

Listing 5.1 shows a fault in the signup functionality that handles the user registration in OnlineShopping, a demo e-commerce application available on git-hub [59].

```
300 private void signup() {
301   if (isValidForm()) {
302     insertIntoDB();
303     JOptionPane.showMessageDialog(SignupPanel, "Please␣Login␣to␣get␣Started
          !", "Congratulations", JOptionPane.DEFAULT_OPTION);
            ...
308     card.show(this.getParent(), "startCard");//Return to Initial Window
309   }else
310     resetForm();
311 }
315 private void insertIntoDB() {
            ...
334   if (resultSet.next()) { //User Already Exists
335     JOptionPane.showMessageDialog(SignupPanel,"Username␣already␣exists");
336     resetForm();
337   }
```

Listing 5.1. Faulty User Registration in OnShop

When a new user registers, the `signup` function is executed (line 300). If the signup form has been correctly filled in, function `isValidForm` returns true (line 301), and function `insertIntoDB` is invoked (line 302). If the username chosen by the user has been already taken by another user, this function correctly shows an error message to the user (line 335). The execution then returns to function `signup` and a message that informs the user that the registration has been completed correctly is *also shown* to the user (line 303). Finally, the application is redirected to the initial window expecting the user to login (line 308). Thus, this buggy code causes the application to behave in a quite confusing way showing both the behavior of a correct and incorrect registration in response to a single user request.

The described fault is quite simple, and a human tester would be able to detect it easily. Nevertheless, it *cannot* be detected with any state-of-the-art technique as it requires a complex GUI interaction of at least 20 events that performs the sign up twice using the same username and that can hardly be generated by current techniques (*ineffective exploration of the execution space*). Moreover, the described fault does not cause the crash of OnlineShopping and therefore could not be detected by current techniques (*lack of functional oracle*).

Using the pre-defined definition of the authentication AIF, AUGUSTO is able to recognize the sign-in sign-out functionality in OnlineShopping and identify the execution scenario in which an existing user is registered again as meaningful. Therefore, AUGUSTO would generate a test case that registers two new users using the same username. When executing this test case in OnlineShopping, AUGUSTO would check for a runtime error message when registering the second user, as

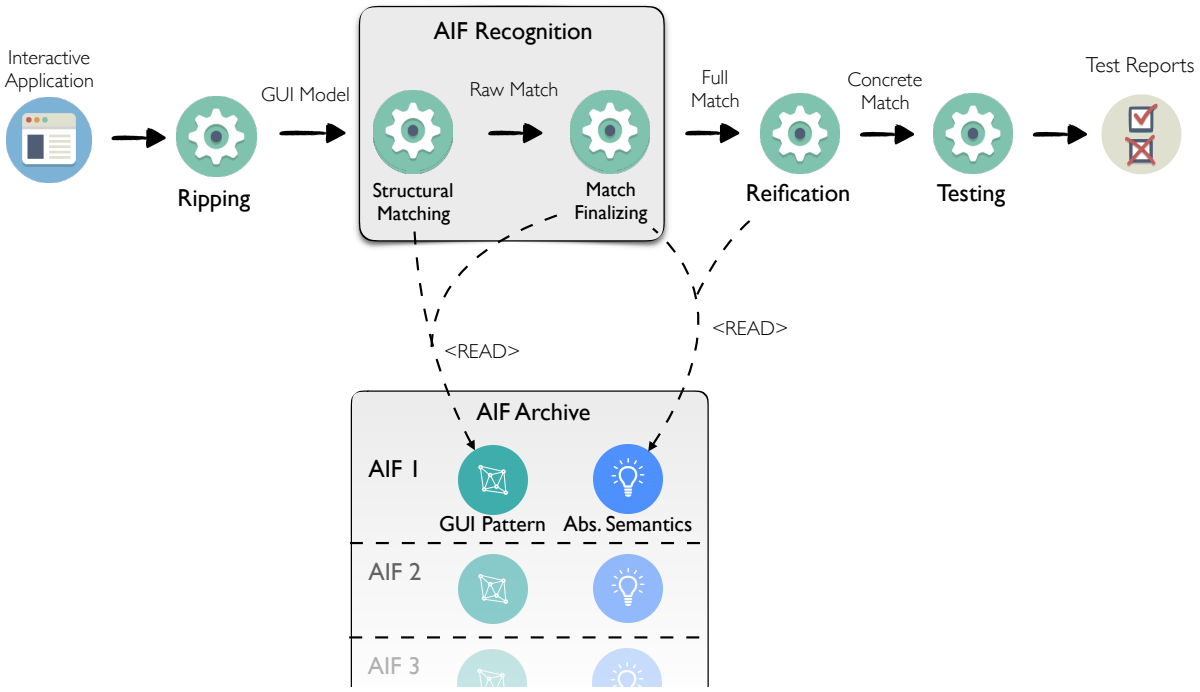


Figure 5.1. AUGUSTO logical architecture

that is the expected behavior of the authentication AIF, therefore exposing the described fault.

5.2 Approach

Figure 5.1 shows the logical architecture of AUGUSTO. The *AIF Archive* is the knowledge repository that contains the set of AIFs supported by AUGUSTO. Each AIF is modeled as a pair <*GUI Pattern*, *Abstract Semantics*>, where the GUI Pattern specifies the set of windows and widgets that may refer to the AIF, and the Abstract Semantics specifies its behavior. The GUI Pattern is used by AUGUSTO to identify a known AIF in the application under test, while the Abstract Semantics model is used to generate meaningful test cases equipped with functional oracles to test it. Both these two types of model allow abstractions and they describe the modelled AIF in the most general way possible, so that the AIF can be recognised and tested despite the peculiarities it might have in the specific AUT.

AUGUSTO works in five steps. The *Ripping* step executes the AUT to dynamically extract a partial model of its GUI, the GUI Model. The *Structural Matching* step exploits the GUI Model to identify the AIFs, by searching for instances of the GUI

Patterns in the GUI model. This step produces a set of raw matches, which can be partial, that is, only a subset of a GUI Pattern might match the GUI Model. AUGUSTO supports partial matches because the GUI Model extracted through ripping might be incomplete. The *Match Finalizing* step generates GUI interactions aiming to complete the partial matches while verifying the consistency between the behaviors specified in the Abstract Semantics model and the behavior of the application. This step produces a set of full matches, which includes every AIFs that have been fully matched in terms of its GUI pattern and its abstract semantics. The *Reification* step further refines the full matches by extracting properties about the concrete behavior of the AUT. For instance, CRUD operations may include a different amount of unique and mandatory fields to create a correct entity. AUGUSTO extracts these properties by stimulating the application with different combinations of inputs. This step produces a set of concrete matches, each being an AIF that occurs in the AUT. The concrete matches are associated with semantic information that takes into consideration the specific characteristics of the AUT. Finally, the *Testing* step generates and executes test cases using the semantic information associated to the AIF both to derive interesting scenarios to test and to check the correctness of the results produced by the application.

5.2.1 AIF Archive

The AIF archive is the repository that contains the manually modelled AIFs that AUGUSTO can detect and test.

Conceptually, we model each AIF as a graph with conditional edges. In this graph, the nodes are *abstract windows* and conditional edges are *abstract edges*. An *abstract window* defines a set of GUI elements that are part of the AIF and that are shown to the user in the same window in any arbitrary application. An *abstract edge* defines a transition among two abstract windows and its annotated with a precondition that defines when the transition can be executed and a postcondition that defines the effect of the execution of the transition.

More formally, an AIF model is a pair

$$AIF : \langle AbstractWindows, AbstractEdges \rangle$$

where *AbstractWindows* is the set of abstract windows and *AbstractEdges* is the set of abstract edges defined as

$$AbstractEdge : \langle AW_{source}, AW_{target}, Trigger, Precondition, Postcondition \rangle$$

where AW_{source} and AW_{target} are the source and the destination abstract windows, respectively, and $Trigger$ is an event that operates on a abstract widget in AW_{source} that fires the transition.

The nodes and edges of the graph (abstract windows) define the appearance of the AIF in the GUI of an arbitrary interactive application, while the edges pre and post conditions define the behavior of the AIF. In AUGUSTO, these two concepts are defined in two different types of models: the GUI Pattern model and the Abstract Semantics model.

GUI Pattern Model

The GUI pattern models specify the general appearance of the AIF and how it occurs in the GUI of interactive applications. AUGUSTO uses the GUI pattern models to automatically recognize whether the AUT implements the AIF. The current UI modeling languages such as IFML [30] model the concrete UI of a specific application, and are not designed to model abstract portions of UIs that are general, flexible and that can fit multiple applications. Thus we decided to define an ad-hoc language for the GUI pattern model to specify how AIFs occur in GUIs as sets of *abstract windows* that contain *abstract widgets* and are connected through *abstract edges*.

An *abstract window* identifies a window or a portion of a window in the application, and is defined as a set of *abstract widgets* that are required to be present in the window. Abstract widgets refer to widgets in the GUI, and are not specified using a concrete type (e.g., JButton, JTextField), but they are abstracted in 3 classes according to the type of user event that can be performed on them: (i) *action*, which are widgets that can be clicked, for instance buttons, (ii) *input*, which are widgets that can be used to enter data, for instance text fields, and (iii) *selectable*, which are widgets that can be selected, for instance lists or tables. Abstract widgets are annotated with both regular expressions and cardinality. Regular expressions specify the labels associated with the widgets and cardinality expresses the quantity of that particular widget that can be in a window. Cardinality can be *one* (exactly 1), *some* (1 or more), *lone* (1 or 0) or *any* (0 or more).

Figure 5.2 shows a simplified GUI pattern for an authentication functionality specified in xml format. The `window` xml elements define the abstract windows that compose the AIF. In the example of the figure, the `login` window corresponds to the presence of a window that includes an input field for the username, an input field for the password, an action widget to login, and an optional action widget for registering. The definitions are flexible as they are not bound to specific GUI

```
<window id="loginform" card=one>
      <action_widget id="signup" card=lone>
            <label>^(register|signup|sign up).*$</label>
      </action_widget>
      <action_widget id="login" card=one>
            <label>^(login|enter|sign in).*$</label>
      </action_widget>
      <input_widget id="pass" card=one>
            <label>^(pass|password).*</label>
      </input_widget>
      <input_widget id="user" card=one>
            <label>^(user|username|email).*</label>
      </input_widget>
</window>
<window id="signupform" card=one>
      <action_widget id=“save" card=one>
            <label>^(ok|save|record|signup|sign up)</label>
      </action_widget>
      <input_widget id="signupuser" card=one>
            <label>^(user|username|email).*</label>
      </input_widget>
      <input_widget id="signuppass" card=one>
            <label>^(?!re-enter|repeat)(pass|password).*</label>
      </input_widget>
      <input_widget id="signuppass2" card=lone>
            <label>^(repeat|re-enter|confirm).*</label>
      </input_widget>
      <input_widget id="otherfields" card=any>
            <label>.*</label>
      </input_widget>
</window>
<window id="loggedpage" card=some>
      <action_widget id="logout" card=one>
            <label>^(logout|exit|sign out|signout).*$</label>
      </action_widget>
</window>
<edge id="ae1" type=uncond from=signup to=signupform/>
<edge id="ae2" type=uncond from=logout to=loginform/>
<edge id="ae3" type=cond from=save to=loginform;loggedpage/>
<edge id=“ae4" type=cond from=login to=loggedpage/>
```

Figure 5.2. Examples of GUI Pattern model

widgets, for instance buttons, but refer to general classes of widgets, for instance action widgets, and cardinalities allow further abstractions. For example, the cardinality of the `otherfields` fields in the `signupform` abstract window allows the abstract window to match a registration form with an arbitrary number of fields.

An *Abstract edge* connects an action widget of an abstract window to another abstract window to indicate a possible execution flow. *Unconditional* abstract edges indicate that the target window is always reached when interacting with the source action widget, for instance clicking on a navigation menu. *Conditional*

abstract edges indicate that the target window is reached only if certain preconditions are satisfied, for instance successfully submitting a form [93]. The example of Figure 5.2 includes two conditional and two unconditional edges. Uncertainty is represented as a set of target windows. For example, the edge associated with the *save* action widget indicates that after registering the execution may reach either the login form or a window in which the user is logged in.

Abstract windows are logical windows, thus the same concrete window of an application may host multiple abstract windows, for instance, the login and registration abstract windows might be found in the same concrete window. Windows may have a cardinality to indicate that they are not required to be present in the target application. This might be useful for example in cases like confirmation windows which might or might not be shown in an application.

In short, the example GUI Pattern model shown in Figure 5.2 specifies that an authentication functionality is composed of three logical windows, one containing the login form, one containing the signup form, and one containing the logout action widget. The pattern specifies that the signup form is composed of three required input fields (username, password, and repeat password) and an unspecified number of other input fields. When clicking on the register action widget on the login form, the application *always* goes to the signup form. When the login action widget is clicked, the application conditionally goes to a page containing the logout widget depending on some conditions (the login form is filled correctly). Also, when clicking the save action widget in the registration form (if the form is filled correctly) the application might go to either the login form or to a page containing the logout action widget. This representation, although very simple, defines the general way in which the authentication functionality is implemented in the GUI of most applications.

The GUI pattern model is defined similarly to the GUI model described in Section 2.3.2 (with the exception that widgets, windows, and edges are abstract), thus can be represented as a tuple:

$$GUIPatternModel : \langle AbstractWindows, AbstractEdges \rangle$$

where an abstract edge is defined as

$$Edge : \langle AW_{source}, AW_{target}, ATrigger \rangle$$

and AW_{source} and AW_{target} are the source and target abstract windows, respectively, and $ATrigger$ is an event that operates on a abstract action widget in AW_{source}.

Abstract Semantics Model

```
/* GUI elements definition */
sig loginform, signupform, loggedpage extends Window{}
sig login, signup, register, logout extends Action_widget{}
sig user, pass, pass2,..., otherfields extends Input_widget{}
one sig Curr_win { /* Current window */
  is_in: Window one -> Time,
}
/* Functionality internal state elements */
sig Usr {
  username: one Value,
  password: one Value
}
sig Users{
  list: Usr set -> Time
}
/* Semantic Property */
one sig Required{
  fields: set otherfields
}
pred preconditions [w: Widget, t: Time] {
  w in register ⟹ not user.content.t=none ∧ not pass.content.t=none ∧
    (∀ us: Users.list.t | user.content.t≠us.username) ∧
    pass.content.t=pass2 ∧ (∀ iw: Required.fields | not iw.content.t=none)
}
pred postconditions [w: Widget, t,t': Time] {
  w in register ⟹ one us: Users | us.username=user.content.t ∧
    us.password=pass.content.t ∧ Users.list.t'=Users.list.t+us ∧
    (Curr_win.is_in.t'=loginform ∨ Curr_win.is_in.t'=loggedpage)
}
```

Listing 5.2. Examples of Abstract Semantics model

While the GUI Pattern model specifies the structure of a functionality in terms of logical windows and possible windows transitions, the *Abstract Semantics* model specifies the behavior of a functionality, and it is used by AUGUSTO to generate test cases and oracles. In a nutshell, the Abstract Semantics model specifies the effect on the application of the interactions with the widgets defined in the AIF GUI Pattern in terms of

- the condition necessary to successfully execute an operation (precondition),
- the window that is shown after the execution of an event (transition),
- the state of the application after the execution of the event (postcondition).

To produce a model that we can leverage to generate test cases, we *Abstract Semantics* model of an AIF requires to be specified formally with a language that

(1) allows to specify logical predicates, (2) is declarative, (3) can be analyzed automatically, (4) has appropriate analysis tools. Logical predicates allow us to express concepts like preconditions and postcondition, a declarative language allows us to define the behavior of the AIF by specifying only the logic of the operations without describing their control flow, the availability of tools for automatic analysis allows us to leverage the models to generate test cases automatically.

Among the different formal specification languages available in the state of the art, we selected Alloy [57] to specify the abstract semantics model as it satisfies all the required properties described above: (i) Alloy is a formal specification language based on set theory that allows to express predicates in first order logic. (ii) Alloy is declarative and can precisely specify the behavior of almost any software system. (iii) Alloy can be analyzed automatically and has an efficient tool, the Alloy Analyzer, able to analyze an Alloy model and simulate the execution of the operations defined within the model.

The Abstract Semantics model is specified in Alloy [57], a formal specification language based on set theory that allows to precisely specify the behavior of almost any software system. We decided to use Alloy to benefit from both the simplicity and expressiveness of the language and the efficiency of the Alloy Analyzer, an automatic tool able to analyze an Alloy model and simulate the execution of the operations defined within the model.

Listing 5.2 shows an excerpt of the Abstract Semantics model of the authentication functionality. The abstract Semantics model shall define every structural element (windows, action widgets, input widgets, and selectable widgets) that is defined in the GUI Pattern model of the functionality (lines 2–4). The widgets defined in the GUI pattern are annotated with a tag (not shown in the example) whose value is the identifier of the corresponding widget in the Alloy model. In this way, after mapping a GUI Pattern to the concrete GUI of the application, every event on a widget can be associated with its semantics. The Abstract Semantics model defines the state variables that are necessary to define the behavior of the functionality (lines 5–15). In the figure, the model defines the current window (lines 5–7) and the list of registered users (lines 9–15). Finally, the model defines the preconditions (line 19) and the postconditions (line 23) of the functionalities operations. The example figure shows pre and postcondition only for the registration operation. The precondition requires the username (`user`) and the password (`pass`) to be not empty, the repeated password (`pass2`) to be the same than the password, all the required fields (`Required.fields`) to be not empty, and the username to be unique. The postcondition adds a new user to the set of registered users and changes the current window to the window named `loginForm`. For simplicity, we omitted some of the checks in the precondition,

such as the individual validity checks on the input fields.

The behaviors of an AIF might not always be definable in a completely application independent way, since it may depend on some *semantic properties* specific to the application under test. The Abstract Semantics model shall define these semantic properties abstractly, thus enabling their inference during the Reification step. AUGUSTO supports the automatic inference of semantic properties when they are specified as a property that affects one or more items that should be fully defined at a later stage. In the model in Listing 5.2, the item `Required` expresses the concept of some fields in the registration form to be *required* to be filled in to submit the form, and it is an example of a property that is indicated in advance as affecting set of fields (lines 16–18) and that will be refined based on the interaction with the actual application. We discuss the properties that our technique support and the strategy to infer them in Section 5.2.5.

5.2.2 Ripping

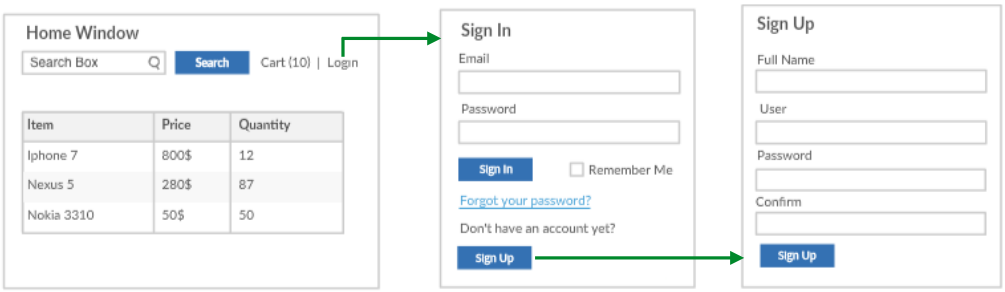


Figure 5.3. A simplified version of the GUI of OnlineShopping

To identify possible known AIFs in an application under test, its GUI must be analyzed and encoded in a suitable form to allow for the searching of GUI Patterns. The *Ripping* step dynamically analyzes the GUI of the AUT in input to produce a data structure that we call *GUI Model* and that is defined as described in Section 2.3.2.

AUGUSTO builds the GUI model following the GUI ripping technique defined by Memon and colleagues [77]. The GUI Ripping explores the GUI of an application with a depth-first strategy that clicks on all the widgets in a window and that recursively continues performing clicks in any newly opened window.

For each widget seen in the AUT GUI, AUGUSTO additionally calculates a **descriptor**, that is a string that *encodes* the semantics of the GUI widget, and saves it in the GUI model together with the other properties of the widget. The

way the widget descriptor is calculated depends on the type of the widget. For widgets that are usually described by means of text displayed on the widgets (such as buttons or menu items) AUGUSTO uses their text property as the descriptor. For widgets which contain images (such as image buttons), AUGUSTO uses the file name of the image they show (after splitting the words in the file name if they are identified by the use of camel case notation or underscores). Instead, for widgets that normally do not contain a text that describes them (such as text fields) AUGUSTO uses as descriptor the text contained in a label widgets placed nearby, according to the strategy proposed by Becce and colleagues [20] (in a nutshell, a label placed on the left or on top of a text field normally describes it). For widgets that only show text but do not allow any type of event (such as label widget) AUGUSTO uses their widget ID (if available) as the descriptor because these widgets are normally used to present data, thus their text might change.

Figure 5.3 illustrates an excerpt of a GUI model obtained by ripping a OnlineShopping. In the "Sign Up" window we can notice that each text field has a label on top that describes it. AUGUSTO uses those labels as descriptors for the form fields widgets. The ripping may not discover all the edges and windows and therefore the extracted GUI model might be incomplete. In particular, ripping might not be able to traverse some conditional edges (window transitions executable only under certain conditions), because it might fail in satisfying the precondition of the functionality associated with the edge [93]. For example, there is no edge for the `Sign In` button in the `Sign In` window of the example GUI model shown in Figure 5.3 because the edge is traversed only after a successful login. AUGUSTO addresses this incompleteness when recognizing AIFs in the next steps of the process.

5.2.3 Structural Matching

The *Structural Matching* step searches for occurrences of AIFs in the AUT. To do so, AUGUSTO identifies portions of the AUT GUI model that match the structure defined in the GUI patterns associated with the AIFs in the AIF archive. A match is a *subgraph* of the GUI Model (i.e., a subset of its windows and edges) in which all the windows and edges match abstract windows and edges of the AIF GUI pattern model. Since the GUI model extracted through ripping might be incomplete (the ripping cannot explore conditional edges), the structural matching considers only the unconditional abstract edges defined in the GUI pattern, and therefore we call the matches identified in this phase as *raw*. In practice, AUGUSTO finds a raw match if it recognizes all the windows reachable by navigating the unconditional abstract edges of the GUI pattern in the GUI model and if it finds all the abstract

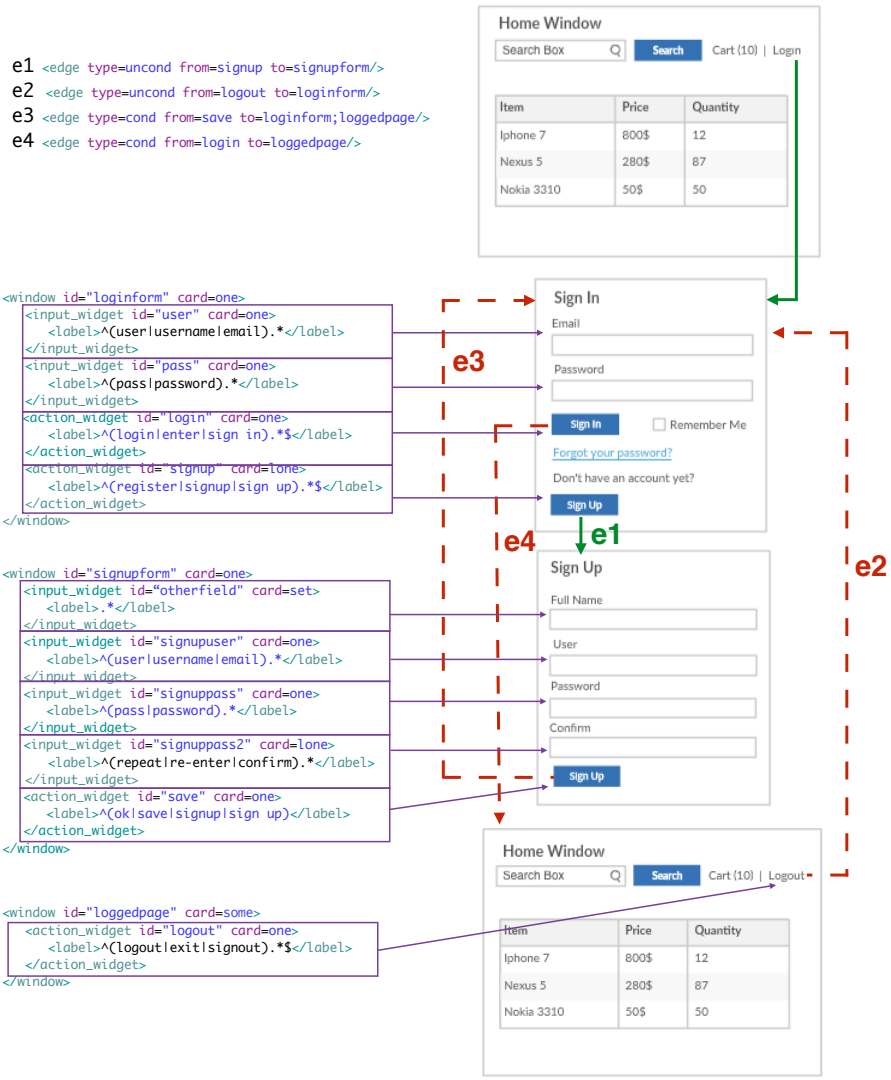


Figure 5.4. A match between the model of the authentication AIF and the GUI of OnlineShopping. Green thick edges are those discovered during the ripping step, while dashed red edges are discovered during the match finalizing step.

unconditional edges that originate from those abstract windows. The conditional edges, if present in the pattern, are searched in the next step.

Considering the GUI pattern model reported in Figure 5.2, a raw match must contain windows that match abstract windows with ids *loginform* and *signupform* (they can be reached navigating unconditional edges only) and edges that match abstract edge with id *ae1* (it is unconditional and originates from the before mentioned abstract windows).

More rigorously, a widget wid matches an abstract widget $awid$ (denoted as $wid \sim_{wid} awid$) if wid has a type compatible with $awid$ type, thus if $awid$ is of type *action* wid must be a type of widget that can be clicked and so on, and if wid descriptor matches the regular expression defined in $awid$ label.

A window w matches an abstract window aw (denoted as $w \sim_w aw$) if there exists a set of matching widgets $wids \subset w$ for each abstract widget $awid \in aw$ such that $\forall wid \in wids : wid \sim_{wid} awid$. The matching between a window and an abstract window considers the cardinality of the widgets, thus the size of set $wids$ must be compatible with $awid$ cardinality (as defined in the GUI pattern model). Thus, if aw cardinality is *one* $wids$ must have size one, if aw cardinality is *some* $wids$ must have size greater than zero, and so on.

An edge e in the GUI model matches an abstract edge ae in the GUI pattern model (denoted $e \sim_e ae$) if its source and target windows match ae source and target abstract windows and if its trigger event operates on a widget that matches the abstract action widget that is the trigger of abstract edge ae.

Finally, a raw match is a tuple

$$\langle MatchedWindows, MatchedEdges \rangle$$

where

- $MatchedWindows$ and $MatchedEdges$ are a subset of the windows and edges in the GUI model,
- for each abstract window aw in the GUI pattern model (reachable only through unconditional abstract edges) exists a set of windows $ws \subset MatchedWindows$ such that $\forall w \in ws : w \sim_w aw$ and the cardinality of set ws is consistent with the cardinality of abstract window aw (as defined in the GUI pattern model),
- for each abstract unconditional edge ae in the GUI pattern model which source abstract window is matched by a window $w \in MatchedWindows$ exists a edge $e \in MatchedEdges : e \sim_e ae$.

Figure 5.4 graphically illustrates the structure of a raw match. The figure shows the GUI model with the abstract windows in xml format next to the corresponding matching windows (the upper window does not match any abstract window). The upper three windows are the windows discovered through the ripping. The second and third windows (from above) match the *loginform* and *signupform* abstract windows, which are the abstract windows reachable by navigating the unconditional edges of the AUTH GUI pattern model. Thus, those two windows

together with edge *e1* generate a raw match between the GUI model and the AUTH pattern.

Once a raw match is identified, AUGUSTO records the matchings among windows, widgets and edges in the GUI model with their counterpart in the pattern model. AUGUSTO uses these matchings during the execution of the test cases to map the events generated from the abstract semantics model to concrete events on the AUT GUI. The recorded matching are shown in Figure 5.4 with purple arrows.

```
1 /* GUI elements definition */
2 sig aw1 extends register{}
3 sig iw1 extends user{}
4 sig iw2 extends pass{}
5 sig iw3 extends pass2{}
6 sig iw4 extends otherfields{}
```

Listing 5.3. Lines added to the Abstract Semantics model

The matchings are reflected in the semantics model that the Alloy Analyzer uses to generate tests for the specific AIF in the AUT. Each widget in the GUI model that matches an abstract widget in the pattern is added to the semantics model. Listing 5.3 shows the lines added to the semantic model to reflect the matchings of the elements of the sign up window in our example AUT. The widget with id *aw1* matches the action widget *register* in the GUI pattern, as expressed in the alloy model with a syntax similar to object extensions in OOP. The last line shows the matching of field "Full Name" of the form, that has id *iw4* in the GUI model, with the *otherfields* input widget of the GUI pattern model. These lines in the model inform the Alloy Analyzer of the elements in the model that have been matched and that can be used during the generation of the test cases. For example, since the last line in Listing 5.3 specifies that one widget matches the *otherfields* abstract widget, the Alloy Analyzer generates tests for an authentication AIF that includes only one extra field in the form aside the classical username, password, and confirm password.

The problem of identifying GUI patterns in the GUI model is an instance of the subgraph isomorphism problem, which is proven to be NP-complete [40]. However, since the number of distinct windows in an application is not large, typically not greater than a hundred, the problem can be solved in a few seconds as confirmed in our empirical experience.

5.2.4 Match Finalizing

The *Match Finalizing* step completes the raw matches, that is, each raw match is either discarded or extended to a full match by including the conditional edges. To do so, AUGUSTO generates GUI interactions that can explore the parts of the AUT GUI that are not reached during the ripping phase and therefore completes the GUI model, by generating GUI interactions using the (partial) information contained in the raw match.

More precisely, for each conditional abstract edge in the AIF GUI pattern model of a raw match, AUGUSTO generates a probing GUI interaction that might execute that edge in the AUT. A probing GUI interaction is a test case that terminates with the execution of the conditional edge in the AUT when its precondition is satisfied. The probing GUI interaction is executed on the AUT, and if the expected transition is observed, the GUI model is updated. Additionally, if a new window is discovered, the window is explored using the ripping approach before being added to the GUI model.

AUGUSTO generates the probing GUI interactions by instructing the Alloy Analyzer to generate a sequence of events that covers a certain operation or condition of the Alloy model. The Alloy Analyzer requires in input the abstract semantics model, a condition that must be covered, and the maximum length of the interaction sequence that must be produced. AUGUSTO asks the Alloy Analyzer to generate sequences of length up to a given boundary that execute the patterns conditional edges.

When executing a GUI interaction that requires input values, such as filling a text field, AUGUSTO uses an archive of input values organized according to their type (e.g., emails are distinguished from dates) and divided between valid and invalid values. The archive includes predefined values for most common data types, but it can be extended with values specific for an AUT.

To generate probing GUI interaction for the specific (candidate) AIF in the AUT, AUGUSTO updates the AIF abstract semantics model using the information contained in the raw match, thus concretizing some aspects of the AIF semantics which have been observed in the AUT. For instance, the abstract semantics model of the authentication does not specify the number of additional fields (apart from the typical username, password, and confirm password) in the signup form. Since the raw match identified that in `onlineShopping` there is only one additional field, that information is plugged in the Alloy model.

After generating probing GUI interaction for each conditional edge in the AIF model, AUGUSTO recalculates the match between the AIF GUI pattern model and the AUT GUI model, searching also for conditional edges. If a match is found, we

call it a complete match as it contains all the structural elements defined in the GUI pattern model.

In the case of the sample raw match of the AUTH pattern with the `onlineShopping` application, AUGUSTO starts by generating a probing GUI interaction to execute the unconditional edge with id *ae3* (that is a sequence of events that fill the signup form and then click on "Sign Up"). When executing that interaction in `onlineShopping` GUI, AUGUSTO confirms that the AUT has the expected transition and confirms the edge. Then, AUGUSTO proceeds to confirm edge with id *ae4*. To confirm that edge AUGUSTO generates a GUI interaction that first signs up and then signs in. When executing that interaction, AUGUSTO finds a new window that matches the abstract window *loggedpage*, thus transforming the raw match into a complete match. The resulting complete match is shown in Figure 5.4.

5.2.5 Reification

The *Reification* step adapts a full match to the specific semantics of the application, by focusing on the *semantic properties* defined in the Abstract Semantics model. The Abstract Semantics model encodes the semantic properties in a general way, leaving unspecified parts that are automatically adapted to the specific characteristics of the AUT. For instance, the property that requires some fields to be non-empty is defined in Figure 5.2 as being associated with a set of input widgets, but the exact set of widgets is left unspecified. The Reification step adapts the semantic properties to the behavior observed for the AUT.

As for the *match finalizing* step, AUGUSTO generates and executes several probing GUI interactions on the AUT to observe its behaviors and infer semantic properties. In a nutshell, AUGUSTO starts by generating a probing GUI interaction that executes an operation affected by a semantic property, that is the operation precondition predicates over the semantic property. Considering our running example, since the *required field* semantic property is used in the precondition of the *Sign Up*, AUGUSTO will generate probing GUI interactions that execute the *Sign Up*. For example, a probing GUI interaction may try to execute the *Sign Up* operation present in the *Sign Up* window of Figure 5.4 with a non-empty *Full Name*, being *Full Name* the only field that needs to be determined as required or not. In fact fields *username*, *password* and *repeated password* are known to be required (see Figure 5.2).

When executing the GUI interaction, AUGUSTO observes the AUT behavior and detects whether the AUT behaves as expected, i.e., the AUT performs the window transition specified in the pattern model. Supposing that when executing the probing GUI interaction in the AUT the registration is performed correctly

(thus online shopping transitions to the login page), AUGUSTO uses the Alloy Analyzer constraint solver to make a guess consistent with the collected evidence. For instance, AUGUSTO may guess that the field *Full Name* is mandatory, but up to now it may also guess the opposite. AUGUSTO automatically includes the guess in the Alloy model by adding some fields to the set of fields affected by the property –in this example it adds *Full Name* to *Required.fields*– and tries to generate a new probing GUI interaction which executes the same conditional edge without satisfying its precondition. In this case, AUGUSTO generates a GUI interaction that leaves the *Full Name* field empty. The execution of this interaction can either confirm or refute the guess. If the interaction refutes the guess, i.e., the registration was successful also in the second case, AUGUSTO makes a new guess based on the new evidence. This process iterates, alternating among interactions that satisfy the precondition and interactions that do not, until either there is only one possible guess consistent with all the collected observations or a timeout is reached. In both cases, AUGUSTO incorporates the guess in the model. In the example, the first guess is correct and it is confirmed by an interaction that fails to sign up with an empty `Full Name`.

This process is quite general and can discover several classes of semantic properties. The current version of AUGUSTO supports any semantic property that can be expressed as a property associated with a (possibly empty) set of elements of the GUI, for instance, the property that an input field in a form is either *required* or *unique*.

5.2.6 Testing

The *testing* phase generates test cases that stimulate the discovered AIFs within semantically relevant usage scenarios. AUGUSTO generates a test suite that satisfies the following criteria:

- *Conditional edge coverage*: This criterion requires sampling the AIFs in every execution context: for each condition associated with a conditional edge of the model, and for each combination of truth values computed according MC/DC [53], there must exist a test case that exercises that combination. We selected MC/DC as condition coverage criterion because it offers a good compromise between cost and completeness. For instance, the precondition shown in Figure 5.2 at line 20 is a conjunction of four boolean basic condition. Therefore, AUGUSTO generates five test cases (MC/DC coverage can be satisfied with n+1 test cases) that stimulate that precondition and that satisfy MC/DC coverage.

- *Pairwise edge coverage*. This criterion requires combining the execution of multiple edges to test combinations of operations. For each pair of edges in the AIF match, there must be a test case that exercises the pair. Considering our example, this criteria imposes to generate test cases in which both sign in and sign out are executed subsequently.

AUGUSTO generates test cases that satisfy these criteria using the Alloy Analyzer, in the same way as it generates the probing GUI interactions of the previous steps. The maximum test cases length is an input parameter of AUGUSTO, as per the previous two phases.

AUGUSTO generates a functional oracle for each test case by mapping the post-conditions, which define the window that must be displayed after the execution of an event and its content, into assertions that are checked after the execution of each event. Thus, AUGUSTO generates oracles that predicate on the state of the GUI and not on the AUT internal state. AUGUSTO examines the correctness of the AUT internal state indirectly by executing subsequent sequences of events. For example, in the case of the authentication functionality, AUGUSTO generates a complex test case that performs a sign up and then signs in the newly created user. After executing the signup procedure, AUGUSTO oracle verifies that the application displays the expected window. Displaying the expected window transition does not guarantee that the user is correctly registered. The subsequent part of the test case signs in that newly created user, and thus checks the correctness of the signin, by verifying that the user lands on a window with a log out button, and indirectly verifies also the internal correctness of the signup procedure.

If we consider again our running example, to cover the conditional edge about the registration operation with MC/DC (see line 22 of Figure 5.2), AUGUSTO generates a non-trivial test case that first registers a new user and then registers again a user with the same username of the already existing user. The test case includes a functional oracle that checks that the current window is still the window with the registration form after an error message has been possibly displayed. The execution of the test leads to a failure that is detected with the oracle, because the onlineShopping application shows an error message but behaves as if the registration has been completed successfully, which violates the generated oracle.

5.3 Prototype implementation

To evaluate the effectiveness of the approach described in the previous section, we developed a prototype of AUGUSTO. The prototype targets Java desktop interactive applications and uses IBM Rational Functional Tester [55] as the

underlying technology for interacting with the GUI of the AUT. The prototype consist of about 15k Java 8 lines of code.

After executing each GUI interaction, the prototype cleans the internal state of the AUT with a script manually defined for each AUT.

We populate the AIF archive with an initial set of three AIFs:

1. AUTH, the typical authentication functionality that allows users to sign up, sign in and sign out from the applications to access private contents;

2. CRUD, the common functionality of adding, removing, updating and deleting objects of a type. To give an example, in a budgeting application, the functionality of adding/removing expenses and bank accounts would be two instances of the CRUD AIF;

3. SAVE, the typical functionality of desktop application that allows to save and load data in and from files.

We modeled the AIFs according to a *common sense* knowledge of these functionalities by the author of this thesis and of the other colleagues involved in AUGUSTO work. Appendix 7.2 reports the complete models of these AIFs. The prototype and its implementation are freely available under MIT licensing and it can be found at *http://github.com/danydunk/Augusto*.

5.4 Evaluation

Table 5.1. Subject applications

Name	Vers.	Type	kLOCs
Buddi [32]	3.4.0.8	Personal finance management	10.0
UPM [105]	1.6	Password management	3.4
Rachota [90]	2.3	Personal tasks management	10.5
Spark [98]	2.7.5	Messaging	2.0
TimeSlotTracker [24]	1.3.1	Personal tasks management	3.5
PDF-sam [107]	0.7	PDF merging/splitting	3.1
OnlineShopping [59]	1.0	E-commerce	1.5
CrossWord [25]	0.3.5	Crosswords building and solving	1.8

We experimentally evaluated AUGUSTO approach by addressing three research questions:

(RQ1) How effective is AUGUSTO in *detecting* application independent functionalities?

This research question investigates the capability of AUGUSTO to automatically detect the presence of the modeled AIFs in the tested applications.

(RQ2) How effective is AUGUSTO in *testing* application independent functionalities?

This research question investigates AUGUSTO's ability to automatically generate test cases to find and report faults in the detected AIFs.

(RQ3) How does AUGUSTO *compare* to state of the art testing techniques in testing AIFs?

This research question investigates if testing the AIFs present in an application with AUGUSTO delivers better results than testing the same functionalities with other approaches, thus motivating the adoption of AUGUSTO in addition to existing techniques. To answer this question we compared AUGUSTO with all other available testing techniques for desktop interactive applications, namely AutoBlackTest (ABT), Testar-Random, Testar-QLearning, Guitar-EFG, Guitar-EDG, and Guitar-EIG.

5.4.1 Empirical Setup

For our empirical study, we selected as subjects eight interactive applications from different application domains, six of which were already used in previous studies [13, 70, 71]. Table 5.1 provides essential information about the selected applications. Since a database is required to enable all the functionalities in Buddi and UPM, we configured an initial db with custom data for Buddi and an empty db for UPM. For each application in the study, we manually defined a state-cleaning script that deletes the files created by the application AUT during the execution of the interaction.

The testing techniques compared in RQ3 required the same configurations, that is, a pool of input values that can be used during the testing activity and the definitions of some configuration parameters. For all the techniques, we populated the pool of inputs value with the same valid and invalid values, defined coherently with the nature of the data processed by the subject applications.

In our evaluation, we used the best configuration possible for each tool, based on our knowledge of the techniques. In AUGUSTO, we used a test case length of 15 events for all applications with the exception of OnlineShopping that has been tested with a test case length of 22 events. We set to 30 minutes the maximum amount of time for the reification step. In ABT we used episodes of 30 events (note

that since each episode can start from any state of the system, the resulting test cases can have an arbitrary length) and the ϵ—greedy policy with $\epsilon = 0.8$, as used in ABT original paper [70]. For Testar-Random and Testar-QLearning we used test cases of size 100 which is the default configuration. In all the experiments ABT, Testar-Random, and Testar-QLearning have been executed for the same time than AUGUSTO. Finally, for the techniques of the GUITAR family, we generated the test cases using 3-wise coverage for test case generation, which guarantees GUITAR to be executed for a longer time (in some case significantly longer) than AUGUSTO, thus favoring GUITAR over AUGUSTO.

Since all other testing tools are not limited to AIFs, simply running the tools on the full applications would produce incomparable data for RQ3. We know by construction that the other tools can test applications more broadly than AUGUSTO and AUGUSTO cannot achieve any result of competing tools with non-AIFs. The purpose of RQ3 is to investigate if the opposite is also true, that is, if AUGUSTO obtaines better results than competing approaches when testing AIFs. Only for the purpose of RQ3, to make this comparison as fair as possible and have the competing techniques spending all the time testing AIFs only, as AUGUSTO does, we modified the subject applications disabling every functionality that is not an AIF. The result is that all tools spent the whole time budget testing the same set of functionalities.

We run our experiments on two virtual machines, one with Windows 10 OS and one with Ubuntu 16.04 OS. We had to use two different virtual machines because of the different operative systems requirements of the tools. AUGUSTO, AutoBlackTest, Testar-Random, and Testar-QLearning run in the Windows virtual machine, while Guitar-EFG, Guitar-EIG, and Guitar-EDG run in the Ubuntu virtual machine. Each virtual machine was configured with 4 cores and 4GB of RAM.

To mitigate the randomness in the results, we repeated all the experiments three times and reported average values.

5.4.2 RQ1 - AIF Detection

To answer RQ1, we studied the completeness and precision of the algorithm for detecting AIFs. We first identified the AIFs present in the subject applications by inspecting every window of every application, and looking for instances of the three defined AIFs (CRUD, AUTH, SAVE). We identified a total of 18 occurrences across the applications. An AIF occurrence is the occurrence of the set of operations specified in the AIF. For example, an instance of a CRUD includes operations to create, read, update and delete the entities of a kind. The applications and their AIFs are reported in the *AUT* and *AIF* columns of Table 5.2, respectively. Each AIF

Table 5.2. RQ1 - AIF Detection

AUT	AIF	ID	Match	Structure	Sem. Properties	
					Compl.	FP
UPM	CRUD	1	yes	precise	100%	0
	SAVE	2	yes	precise	n/a	n/a
Spark	AUTH	3	(yes)	precise	100%	0
Rachota	CRUD	4	yes	precise	100%	0
		5	yes	precise	100%	0.7
		6	no	-	-	-
OShopping	AUTH	7	yes	precise	100%	1.0
Buddi	CRUD	8	yes	lack delete button	100%	0.7
		9	yes	precise	100%	0
		10	yes	precise	100%	0
		11	(yes)	precise	50%	3.7
		12	yes	precise	100%	0
	SAVE	13	yes	lack replace file window	n/a	n/a
PDFsam	CRUD	14	(yes)	precise	100%	0
TTracker	CRUD	15	yes	precise	100%	0
	CRUD	16	no	-	-	-
	CRUD	17	no	-	-	-
CrossWord	SAVE	18	no	-	-	-

is associated with an identifier (column *ID*).

We then executed AUGUSTO on the applications and checked the discovered matches. We indicate the result of this check in column *Match*: *yes* indicates the presence of a concrete match that can be used for generating test cases, *no* indicates that no match is found, and *(yes)* indicates the presence of a match that AUGUSTO could find only with some manual intervention. Out of 18 cases, AUGUSTO missed only 4 AIFs. For TTracker the missed matches are caused by the limitation of the *ripping* phase that was not able to discover the GUI portions that contain the AIFs. The missed AIF in Rachota was caused by two CRUD AIFs sharing some windows, a case not supported by AUGUSTO. AUGUSTO never identified a non-AIF functionality as an AIF, that is, it never produced false positives during AIF detection.

AUGUSTO required manual intervention to deal with cases not supported by the prototype in 3 of the 14 identified AIFs. In the case of Buddi (case 11), we manually excluded a Combo Box producing behaviors that are not supported by our technique. To address cases 3 and 14 we extended the definition of two GUI Patterns to accept labels that are not typically used for the operations of CRUD and AUTH. For instance, we set the label `accounts` as a valid alternative of `sign`

`up/register` in AUTH. Although these are small interventions, they prevented the fully automatic execution of the approach in three cases.

We also evaluated the accuracy of the discovered matches in terms of the widgets included in the AIF match: Column *Structure* indicates if the match includes *all and only* the widgets that we manually identified as related to the AIF. The value *precise* indicates a perfect match, that is, no missing neither unrelated widgets associated with the AIF. Note that in 12 out of 14 cases AUGUSTO produced a perfect match. In case 8 AUGUSTO missed only an element, reported in the table, due to particular implementation choices in the application, and in case 13 AUGUSTO missed a window because of a bug in the application (the bug was then reported in the testing phase). In no case AUGUSTO associated unrelated widgets to the AIF, that is, AUGUSTO never confused the additional elements present in a window with the ones that refer to the identified AIF.

We also evaluated the ability of AUGUSTO to identify semantic properties, in this case, to identify the required and unique fields for CRUD and AUTH AIFs. We evaluated this aspect by considering completeness, defined as the average percentage of required and unique fields identified correctly by AUGUSTO (column *Compl.*), and false positives, defined as the average number of fields wrongly associated with a required or unique property (column *FP*). We report the value *n/a* when the AIF does not include any semantic property to be discovered.

The results obtained with semantic properties show that AUGUSTO is quite effective both in terms of completeness, only in one case some fields have not been associated with the corresponding property, and rate of false positives, only in four cases there are false positives. Note that completeness and the number of false positives associated with semantic properties could be improved by allocating more time to the reification phase.

In a nutshell, AUGUSTO identified the AIFs present in the subject applications in 78% of the cases (in 3 cases requiring manual intervention) and produced highly accurate matches, including 86% perfect matches, and identified the vast majority of the semantic properties present in the application.

5.4.3 RQ2 - Effectiveness

The effectiveness of testing techniques is typically assessed by considering code coverage and fault revealing ability. Since AUGUSTO does not target the whole application, code coverage is not an informative metric. Thus, to answer RQ2 we evaluated AUGUSTO by considering only its fault revealing ability. We measure the number of faults revealed in the subject applications. Note that we

Table 5.3. RQ2 - Effectiveness

AUT	AIF	ID	Avg TC	Avg Fail	Avg FA	Avg Fault	#Fault (Exc)
UPM	CRUD	1	17.0	6.7	0.4	2.0	3 (1)
	Save	2	75.5	1.0	0.7	0.4	1 (1)
Spark	Auth	3	33.7	6.7	6.8	0	0 (0)
Rachota	CRUD	4	8.3	0.7	0.6	0	0 (0)
		5	76.0	7.3	7.3	0	0 (0)
OShopping	Auth	7	17.0	4.5	4.0	0.3	1 (0)
Buddi	CRUD	8	17.0	5.5	5.5	0	0 (0)
		9	18.0	2.7	2.7	0	0 (0)
		10	18.7	0	0	0	0 (0)
		11	22.8	12.7	6.3	1.0	1 (0)
		12	19.2	0	0	0	0 (0)
	Save	13	50.7	12.4	0	1.0	1 (0)
PDFsam	CRUD	14	9.4	0	0	0	0 (0)
TTracker	CRUD	15	11.7	0	0	0	0 (0)
						Overall	7 (2)

did not inject faults in the subject applications thus we can only asses the number of faults revealed while we cannot evaluate the completeness of the revealed faults. To assess the effectiveness of AUGUSTO, we manually inspected all the test cases that were reported as *failing* by AUGUSTO functional oracle, and manually classified them as *true* or *false positives* depending on whether they show the manifestation of an error or not.

Table 5.3 reports for each AIF identified by AUGUSTO, the average number of generated test cases (column *Avg TC*), the average number of test cases that fail because of the violation of a functional oracle (column *Avg Fail*), the average number of false alarms produced, that is, the number of failing test cases that do not expose any fault in the program (column *Avg FA*), the average number of faults detected per AIF in a run (column *Avg Fault*), and the total number of faults detected in the three runs (column *#Fault*). Column *#Faults* also indicates the number of faults that cause the AUT to print an exception on the log. In our experimentation AUGUSTO did not detect any failure that causes the application to crash.

The average number of test cases generated by AUGUSTO varies a lot, ranging from 8.3 to 76.0. This big variability that is observed even for AIFs of the same kind in the same application (a good example is the number of test cases for the CRUDs in Rachota), depends on the specific structural match, concrete semantics and semantic properties that are extracted. This shows how AUGUSTO, can flexibly adapt these definitions to the specific case, by generating a number of test cases

Table 5.4. RQ3 - Comparison

AUT	Hours	Augusto	ABT		TestarRand		TestarQL		GuitarEFG		GuitarEIG		GuitarEDG	
			Rep	Cov	Rep	Cov	Rep	Cov	Rep	Cov	Rep	Cov	Rep	Cov
UPM	3.0	4	2	1	2	1	1	1	1	1	1	1	1	1
Spark	2.0	0	0	0	0	0	0	0	0	0	0	0	0	0
Rachota	2.5	0	0	0	0	0	0	0	0	0	0	0	0	0
OShopping	8.0	1	0	0	0	0	0	0	0	0	0	0	0	0
Buddi	11.0	2	0	0	0	0	0	0	0	0	0	0	0	0
PDFsam	1.5	0	0	0	0	0	0	0	0	0	0	0	0	0
TTracker	1.5	0	0	0	0	0	0	0	0	0	0	0	0	0
Overall Reported		7	2		2		1		1		1		1	

that depends on the complexity of the tested functionality.

AUGUSTO produces some false alarms as reported in the table. This is due to two main reasons: acceptable mismatches between the semantics model and the concrete behavior of the application, and imprecise semantics properties inference. Both these sources of imprecision cause the generation of imprecise functional oracles. In several cases sets of failures refer to the same cause (for instance, a single imprecise property may cause the failure of multiple test cases) and identifying the cause of the failure for one test can be used to drastically reduce the inspection time of the other tests failing for the same reason.

In the experiments, AUGUSTO revealed a total of 7 faults, with only two faults causing the program to log an exception. This result shows that the automatic functional oracle included in the test cases is an essential element for revealing failures.

AUGUSTO revealed some interesting faults, such as the one described in chapter 5.2 of this thesis. Another interesting fault was detected in UPM: When editing the identifier of an account, if the change is undone and the account is saved, the operation fails with an error message stating that the identifier already exists, even though the identifier is the current identifier of the edited account.

In a nutshell, AUGUSTO generated test cases for the AIFs present in several applications and revealed multiple faults, including several ones that cannot be detected with standard implicit oracles.

5.4.4 RQ3 - Comparison

We compare the number of faults detected by AUGUSTO with ABT, Testar-Random, Testar-QLearning, Guitar-EFG, Guitar-EIG, and Guitar-EDG, by manually inspecting the test cases that each tool reported as *failing* (typically crashes). We then inspected the standard error of the application after the execution of each

test case, and considered as failing each test case that causes the AUT to print an exception. We manually inspected all the failing test cases and we classified as real failures (as done for AUGUSTO) the test cases that trigger an error in the subject application from the user perspective.

Table 5.4 shows the results obtained by AUGUSTO, ABT, Testar-Random, Testar-QLearning, Guitar-EFG, Guitar-EIG, and Guitar-EDG when testing AIFs. Column *AUT* indicates the subject application. Column *Hours* reports the average time in hours spent by AUGUSTO to test the application. ABT, Testar-Random, and Testar-QLearning have been executed for the same amount of time, while Guitar-EFG, Guitar-EIG, Guitar-EDG have been configurated to be executed *at least* for that time. Column AUGUSTO indicates the number of faults detected by AUGUSTO over all the three runs. For the other tools, the table distinguishes between reported (column *Rep*) and covered faults (column *Cov*). A reported fault is a fault which manifest as either a crash or as an exception printed on the AUT log (remember that the other tools do not include a functional oracle) and that we manually classified as a true failure. A covered fault is a fault that has been activated by a generated test case, but no failure has been reported due to the lack of an oracle. Since we do not have knowledge of all the faults in the AUTs and manually inspecting all the test cases generated by the competing tools was unfeasible, we consider as covered faults only the faults which were detected by AUGUSTO. Thus, if one of the tools covered a fault that does not manifest with a crash or with an exception we had no way of detecting it.

All the faults reported and covered by the competing techniques are a subset of the faults reported by AUGUSTO, confirming the higher effectiveness of semantics approaches when testing AIFs. AUGUSTO has been able to test interesting cases and interesting combinations of events revealing 7 faults, while for 4 of these faults the other techniques have not been even able to produce the sequence that covers the faulty case. Moreover, even when the other techniques manage to cover the fault, there is a good chance that the fault is not reported due to lack of non-trivial oracles. In our evaluation, all other techniques reported 2 crashing faults and covered but did not report another fault.

Finally, 4 of the subjects used in this evaluation (Buddi, Rachota, UPM, CrossWord) were used also in the empirical study presented in section 3. In that study, all other state-of-the-art techniques were able to detect a total of 5 faults. Thus, for these 4 applications, when testing AIFs AUGUSTO managed to report more faults (7) than all other techniques that tested the whole applications (i.e., without being restricted to AIFs only). This result corroborates our intuition that semantic testing approaches are more effective than traditional structural approaches when testing interactive applications.

In a nutshell, AUGUSTO has been able to both sample the execution space of the AIFs more effectively than competing approaches and report failures that could not be reported by any competing approach, at the cost of some false alarms. AUGUSTO proved to be an effective complement to current *general purpose* testing techniques for interactive applications and showed to be able to achieve the benefits of *automatic semantic testing* for at least some portions of the application under test.

5.4.5 Threats to validity

In this section, we discuss the main threats to the validity of the experiments. A threat to internal validity is the generality of the AIFs models that we used in our evaluation. To mitigate the risk of defining models that fit the applications used in the evaluation but not others, we defined the AIF archive before selecting the subject applications.

Another threat to internal validity is related to the manual activities performed by the researchers involved in the work to classify the failing test cases reported by AUGUSTO as *faulty* or *false alarm*, and to modify the subject applications for RQ3. For the first threat, to reduce any bias, only the failing test cases for which all the researchers involved agree that they expose a fault were classified as faulty. For the second threat, after modifying the applications we verified that the AIFs continue working the same including the presence of the faulty behaviors.

The external validity threats of our study relate to the generality of the results with respect to the set of AIFs and set of applications that we used. Although we cannot make claims about the generalizability of the results to other AIFs, the AIFs that we used were all successfully matched and have been all useful to reveal faults. We thus expect AUGUSTO to be able to effectively exploit other AIFs too.

In terms of subject applications, to mitigate any issue with generalizability, we selected applications that belong to a variety of domains, most of which were already used in other studies, which facilitates comparison, and experimented with a relatively high number of AIFs per application.

Chapter 6

ADAPTDROID: Semantic Testing via Cross-Application Test Case Adaptation

In this chapter, we present ADAPTDROID, a technique to adapt test cases across similar interactive applications. ADAPTDROID is based on the observation that there exist many similar applications that offer the same functionalities. ADAPTDROID leverages this opportunity to reduce the cost of testing by automatically adapting existing manually-written test cases of a certain interactive application to test another similar application.

In a nutshell, we define the goal of ADAPTDROID in the following way:

Given two similar applications A_D (donor), A_R (recipient) and a "donor" test case t_D for A_D, ADAPTDROID *aims to generate a test t_R that tests A_R in an analogous way with respect to t_D.*

In the remainder of this section, we discuss some preliminaries of ADAPTDROID approach, present a motivating example, describe AUGUSTO approach, and discuss the empirical evaluation we carried out to evaluate it.

6.1 Preliminaries

To simplify the problem of cross-application adaptation, we make some assumptions on the events and assertions that compose t_D and t_R: we assume test cases to contain events of two types

- **click**(w): clicking on a widget w (e.g., button);

- **fill**(w, input): writing the string *input* in a widget w (e.g., text field).

Also, we assume test cases to contain assertions only at the end of the test case (thus, applied after the execution of all t events).

Moreover, we assume assertions to be of two types:

- **exists(txt)** checks if among all the widgets currently shown in the GUI it exists a widget whose text is equal to *txt*. More formally, *exists(txt)* returns `true` (the test passes) if $\exists w \in S : text(w) =$ txt, `false` otherwise. This assertion types can be used to verify that a certain element appears or disappears after the execution of a certain sequence of events.

- **hasText(w, txt)** checks if a certain widget exists and has the expected text. More formally, *hasText(w, txt)* returns `true` (the test passes) if $w \in S \wedge text(w) =$ txt, `false` otherwise. Widget has text assertions can be used to verify the correctness of a data shown in certain widgets, such as the total of a sum.

The notation `not(`a`)` denotes the negation of an assertion a, e.g., `not(exists('AB'))` checks for the non existence of a widget with text `'AB'`.

We do not consider complex events (e.g., multi-touch gestures, drag-and-drop) nor assertions that verify peculiar properties of widgets (e.g., its color or position). However, notice that the considered event/assertion types described above can cover most of the test scenario of interactive applications. E.g., exists('test') can be used to verify if the correct window-transition is performed since 'test' would not be visible in case of a wrong transition.

6.2 Motivating Example

This section illustrates the problem of cross-application test adaptation through an example.

We discuss how to adapt a test case of *Splendo*, an Android app to manage tasks lists [10], to test *Bills Reminder*, an Android app to manage bills [3]. The test case of *Splendo adds a new task to the task list, and verifies that the task disappears once marked as done*. The test case adapted for *Bills Reminder* adds a new bill to the bill list and verifies that the bill disappears once marked as paid. Figure 6.1 shows the donor and adapted test cases of our example. Test cases can be adapted across these two apps since, although in different domains, they share the logical operations of creating a new element (a task in *Splendo*, a bill in *Bills Reminder*) and marking it as completed (*done* in *Splendo*, *paid* in *Bill Reminder*).

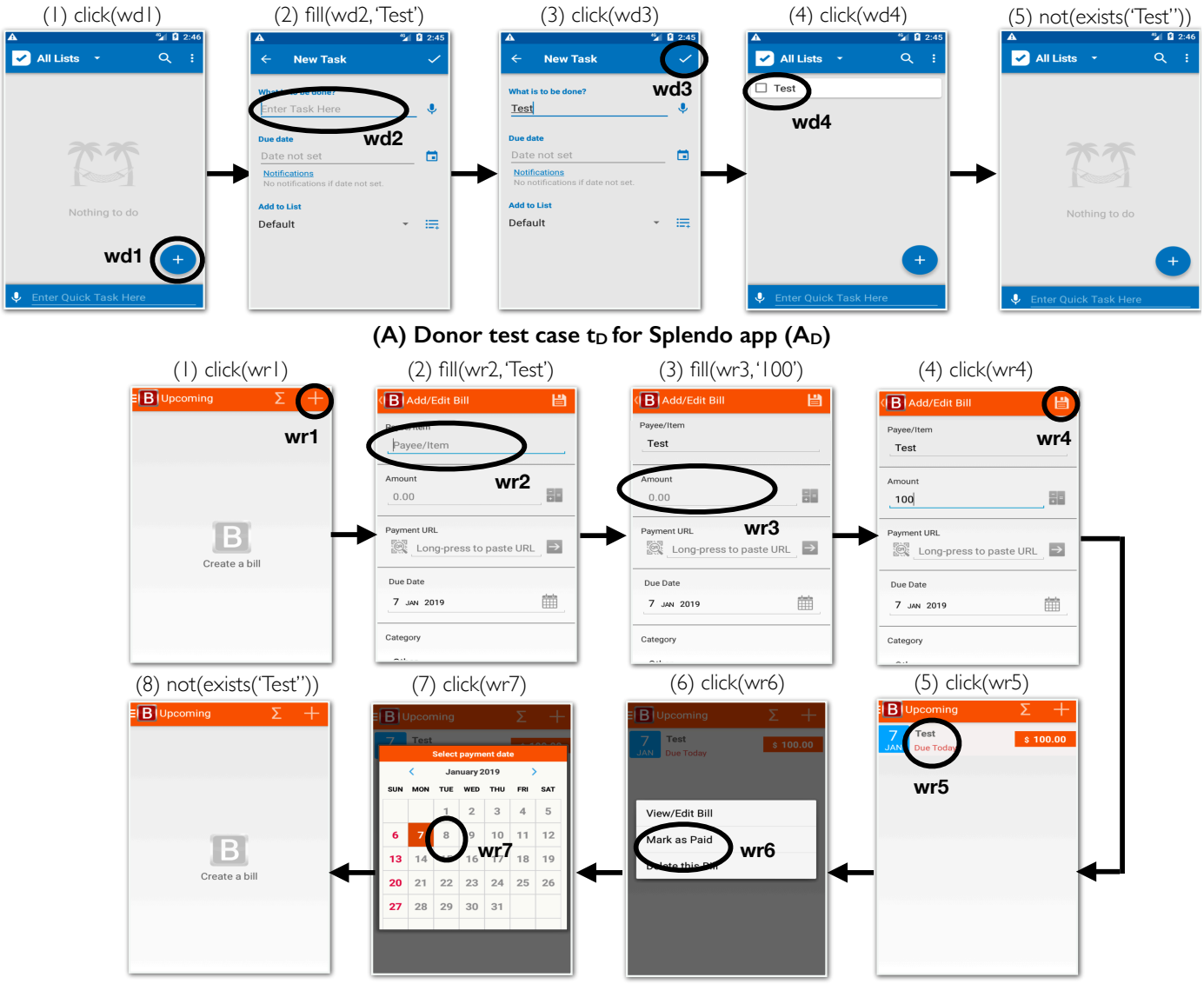


Figure 6.1. ADAPTDROID cross-application test adaptation example

It is not trivial to generate a test case for *Bill Reminder* with the meaningful sequence of events, appropriate data for the bill, and oracle showed in Figure 6.1B. The difficulty in doing so derives from the fact that the events to execute must be selected from a huge number of possible event combinations. On the contrary, the donor test case substantially simplifies the generation of that test as it can directly and explicitly guide the test case generation process through the right sequence of steps. In addition, test case adaptation offers a unique opportunity: automatically obtaining an oracle by adapting the oracle in the donor test.

The information in the donor test case are extremely useful for generating a test case for the recipient app, but adapting the donor test case to the recipient app still presents several challenges that we exemplify in the example of Figure 6.1. The example highlights the three main challenges of automatically adapting test cases across interactive applications:

Huge set of possible test cases. The space of the possible test cases for the recipient application grows exponentially with the number of widgets in the recipient GUI. Since test case adaptation must select the best adaptation of the donor test among all the possible tests for the recipient application, it requires an effective strategy that can distinguish the relevant and the irrelevant operations that can be performed to replicate the donor test case.

Different GUI widgets. The donor test case may exercise GUI widgets that are logically equivalent but very different from widgets to be exercised in the recipient test case. For instance, the widget to input the task name to be added in *Splendo* is identified with the label *"What is to be done?"*, while the corresponding widget in *Bills Reminder* is identified with the label *"Payee/Item"*. Also, in *Splendo* a task is saved by clicking on the tick mark button, while *Bills Reminder* requires to click an image button showing a floppy disk.

No one-to-one GUI event matching. The adapted test case might be composed of a different number of events compared to the donor test case. For instance, the donor test case shown in Figure 6.1(A) is composed of five events, while the corresponding recipient test case shown in Figure 6.1(B) is composed of eight events. In our example, creating a bill in *Bills Reminder* requires more events than creating a task in *Splendo*, and marking a bill as paid in *Bills Reminder* requires a date, while marking a task as done in *Splendo* does not. Finally, the GUI of the target application might be organized in a way that it imposes to execute some events in a different order than the donor test case.

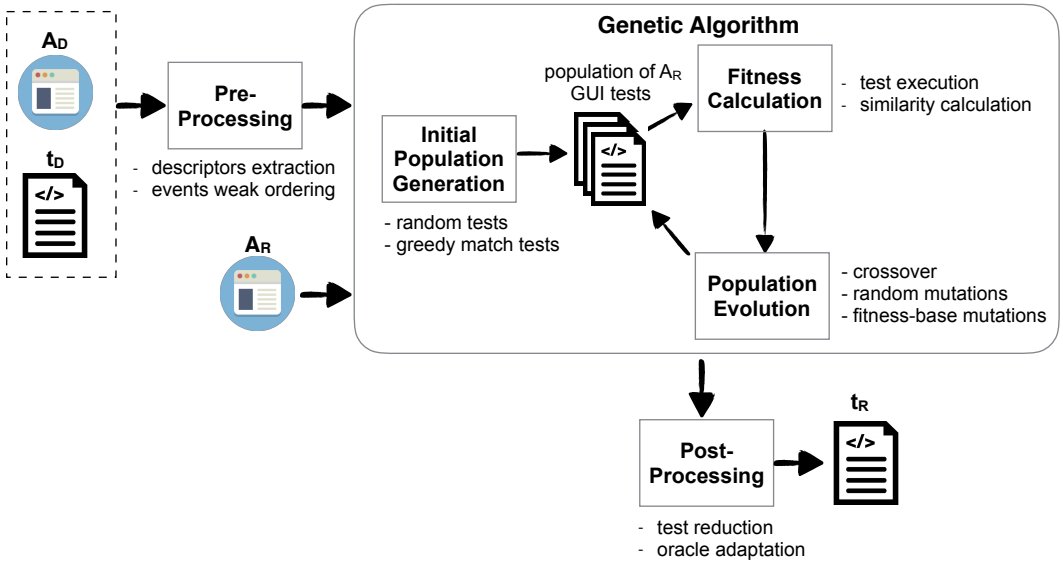


Figure 6.2. ADAPTDROID logical architecture

6.3 Approach

This section presents ADAPTDROID, an automatic technique for generating test cases through adaptation. Figure 6.2 shows the ADAPTDROID process, which takes as input a donor application A_D, a donor test t_D, and a recipient application A_R, and generates an adapted test case t_r that exercises A_R analogously to how t_D exercises A_D. Note that ADAPTDROID is entirely black box, that is, it does not require the source code of the donor and recipient apps but only their binaries.

The test adaptation process is obtained from the execution of five phases: pre-processing, initial population generation, fitness calculation, population evolution, and post-processing. The population evolution and fitness calculation phases may be executed multiple times, as illustrated in Figure 6.2.

The *pre-processing* phase executes the donor test t_D on the donor application A_D to extract information relevant to the test adaptation process, such as the identifiers associated with the widgets executed by the donor test case. The *Initial Population Generation*, *Fitness Calculation* and *Population Evolution* phases implement a genetic algorithm [50, 51, 52] that creates and then evolves a population of candidate tests guided by a fitness function that steers the evolution toward a t_R that is as similar as possible to the donor test case.

ADAPTDROID exploits a specific version of each of these three phases to achieve this goal, as suggested by the labels present next to each phase in Figure 6.2. Evolution and fitness evaluation iterate until either a perfect adapted test case

is found (i.e., fitness equals to one) or a time-budget expires. Finally, before returning the adapted test case to the developer, the *post-processing* phase shortens the test by removing irrelevant events and add adapted assertion extracted from the donor test case, if present.

To pair in a meaningful way events executed in different applications, ADAPTDROID uses a notion of semantic matching, which exploits the descriptors associated with the widgets executed to perform the events.

The usage of a genetic algorithm equipped with a proper set of evolution operators allows ADAPTDROID to address the challenge posed by execution space size (*Huge set of possible test cases*). The definition of a flexible fitness function that can capture the different nature of t_D and t_R addresses the challenge about the diversity of the structure and organization of the windows in the donor and receiving apps (*No one-to-one GUI event matching*). Finally, the definition of a matching strategy that takes into account the semantic of the paired events addressed the challenge about differences of the individual widgets (*Different GUI widgets*).

In the next sections, we first explain the concept of *cross-app event matching* that ADAPTDROID employs in the approach, then we present each of ADAPTDROID steps.

6.3.1 Cross-app Matching of GUI Events

To adapt test cases across applications, ADAPTDROID pairs t_D and t_R events, according to their *semantics similarity*, that is, the *logical* similarity between operations in the donor and the recipient applications, while *abstracting from syntactic differences*, that is, the two apps may implement semantically similar operation by interacting with completely syntactically different widgets. Thus ADAPTDROID measures the similarity between t_D and t_R events by considering exclusively semantic information.

ADAPTDROID computes the *semantic matching* between an event e_i in a donor test case t_D and an event e_j in a recipient test case t_R ($e_i \sim e_j$) based on the events (i) *type*, (ii) *descriptor*, and (iii) *textual input* of the events.

The **type** of an event can be either *click* or *fill* (see Section 6.1).

The **descriptor** of an event is a semantically relevant string extracted from the widget that is used to emit the event. In particular, given an event that operates on a widget w, the descriptor of the event is the *descriptor* widget w, where the widget descriptor is calculated using the same strategy used by AUGUSTO and presented in Section 5.2.2. Table 6.1 shows the descriptors extracted for each event in the donor and recipient test cases shown in Figure 6.1.

Table 6.1. Running example events descriptors

t_D	events	descriptors	t_R	events	descriptors
e_{D1}	`click(wd1)`	bs_add_task	e_{R1}	`click(wr1)`	action_add
e_{D2}	`fill(wd2,'Test')`	What is to be done?	e_{R2}	`fill(wr2,'Test')`	Payee/Item
e_{D3}	`click(wd3)`	action_save_task	e_{R3}	`fill(wr3,'100')`	Amount
e_{D4}	`click(wd4)`	Test	e_{R4}	`click(wr4)`	action_save
			e_{R5}	`click(wr5)`	Test
			e_{R6}	`click(wr6)`	Mark as Paid
			e_{R7}	`click(wr7)`	8

The **textual input** is simply the input data used in events that require inputs. For instance, a fill event `fill(w,txt)` is associated with the input `txt`. If the event is not associated with any input, this information is empty.

ADAPTDROID computes the similarity between events, by comparing their attributes *semantically*. To this end, we defined a function for **comparing strings semantically** that compares two strings and returns a Boolean value that indicates if they represent the same concept. More formally, let IsSemSim(`txt1`, `txt2`) be such a Boolean function that returns true if the two input strings `txt1` and `txt2` are semantically similar, and false otherwise. Each input string of the function might contain several words, thus `txt1`="add account" is a valid input for function IsSemSim.

ADAPTDROID implements function IsSemSim using the Word Mover's Distance (WMD) [61], which is a well-known technique to compute the semantic distance between two sets of words. WMD is based on *Word2vec* [82], a vector-based *word embedding* [104] where words with similar semantics are located closely in the vector space [82]. We decided to use WMD because it is the only string semantic distance in the state of the art that handles strings that contain several words.

Given two strings, WMD returns a number between 0 to 1 that expresses how close these two strings are in the vector space. If the number returned by WMD[1] is greater than a given threshold τ then IsSemSim(`txt1`, `txt2`) = true, false otherwise.

Notice that despite the fact that WMD returns a number, we decided to define IsSemSim as a Boolean function that identifies if two strings represent two concepts that are semantically close. We decided to opt for a Boolean function because the distances calculated by WMD are not accurate enough to assume that the highest similarity is always the best one. For example, it is not possible to

[1] Stop-words removal and lemmatization [67] are applied to facilitate WMD calculation and make it more resilient to differences such as verb conjugation.

safely consider two strings with WMD 0.55 (e.g., strings “same” and “different”) to be more (semantically) similar than two string with WMD 0.50 (e.g., strings “same” and “similar”).

We finally define the **semantic matching of events**. Given a donor test case t_D, a recipient test case t_R, an event $e_i \in t_D$ of type $type_i$, with descriptor d_i and textual input ti_i, and an event $e_j \in t_R$ of type $type_j$, with descriptor d_j and textual input ti_j, we say that e_i semantically matches e_j, expressed as $\mathbf{e_i \sim e_j}$, if one of the following cases holds.

Matching click events: this is the case of events that are both clicks and both execute a similar functionality. Formally, $\mathbf{e_i \sim e_j}$ iff $type_i = type_j = click(w) \wedge IsSemSim(d_i, d_j)$.

Matching fill events: this is the case of events that are both fill operations and both execute a similar functionality with the same inputs. Formally, $\mathbf{e_i \sim e_j}$ iff $type_i = type_j = fill(w,txt) \wedge IsSemSim(d_i, d_j) \wedge ti_i = ti_j$.

Matching fill-to-click events: this is the case of a fill operation that can be mapped to an equivalent click operation (e.g., entering the value 1 on a calculator app can be mapped to clicking the button with the label 1 in another calculator app). We do not allow the opposite, that is, mapping click events to fill events, otherwise ADAPTDROID could easily (and incorrectly) obtain a mapping by entering the values of the labels in the buttons clicked by the donor test case in the input fields available in the recipient application. Formally, $\mathbf{e_i \sim e_j}$ iff $type_i = fill(w,txt) \wedge type_j = click(w) \wedge IsSemSim(ti_i, d_j)$.

If we consider the events in our running example (see Table 6.1), ADAPTDROID matches the events in t_D with those in t_R as follows: $e_{D1} \sim e_{R1}$, $e_{D3} \sim e_{R1}$, $e_{D3} \sim e_{R4}$, $e_{D4} \sim e_{R5}$.

6.3.2 Pre-processing

The *Pre-processing* phase collects information about t_D, information that ADAPTDROID uses later during the test case adaptation process. ADAPTDROID traces two main aspects: the descriptors of t_D events, and the sets of events that can be likely executed in a different order.

Event sequence: ADAPTDROID executes t_D in A_D and for each event extracts its descriptor.

Event ordering: The order of the events as executed in the donor test is not necessarily the only possible one. In several cases, there are groups of events that can be executed in a different order without affecting the results (e.g., the fill events necessary to fill in a form). Considering this and that, as mentioned,

Algorithm 1: Events clustered based on ordering

input : $t_D = \langle e_1, ..., e_n \rangle$, states $S = \langle S_0, ..., S_n \rangle$
output: clusters (ordered list of event clusters)

```
1  function CLUSTEREVENTS(E, S)
2      clusters ← {}
3      c ← {e_1}
4      for i from 2 to n-1 do
5          if EVENTISENABLEDINSTATE(e_{i-1},S_{i+1}) ∧ EVENTISENABLEDINSTATE(e_i,S_{i-1})
             then
6              c ← c ∪{e_i}
7          else
8              append c to clusters
9              c ← {e_i}
10     append c to clusters
11     return clusters
```

A_R might require to execute the events in a different order, it is important for ADAPTDROID to know if there exist alternative execution orders of the events of t_D. Knowing these alternative execution orders, ADAPTDROID can ease the adaptation process by not imposing that t_R must follow the same exact event ordering of t_D, but allowing it to follow any of t_D equivalent orderings.

ADAPTDROID identifies the possible equivalent event orderings by checking if pairs of consecutively executed events can be executed in the opposite order (i.e., both events are enabled in their source states). The consecutive events that satisfy this condition are part of the same set of events cluster that can be arbitrarily reordered. Listing 1 shows the pseudocode of the equivalent event ordering analysis algorithm. The analysis of t_D in Figure 6.1 produces all clusters each with a single event, thus indicating that the t_D events execution order is the only possible one. Although the algorithm checks pairs of consecutive events, it can identify clusters of events that can be executed in a different order. For instance, assuming that e_1, e_2, and e_3 can be executed in any order, the algorithm would firstly identify that e_1 and e_2 can be executed in the opposite order, and then it identifies that e_2 and e_3 can also be executed in the opposite order, thus including all three events in the same cluster.

To facilitate the definition of the successive phases of the approach, we introduce the Boolean operator $<_{clusters}$, such that $e_i <_{clusters} e_j$ iff $\exists c_i, c_j \in clusters : e_i \in c_i \wedge e_j \in c_j \wedge i < j$. In a nutshell, this operator returns true if e_i is in a previous event cluster than e_j, and thus it must be executed before.

6.3.3 Generation of the Initial Population

The first step of a genetic algorithm is to generate the initial population of N candidate solutions, where N is an input parameter of the approach ($\mathcal{P}_0$) [16]. A candidate solution for ADAPTDROID is a test case t_R for the recipient application A_R. ADAPTDROID populates $\mathcal{P}_0$ with both randomly generated test cases (to guarantee diversity in the population) and test cases that are similar to t_D, generated in a greedy fashion (to have"good" genetic material for evolution).

Random Test Cases: ADAPTDROID generates a random test for the initial population by first randomly selecting the test case length between 1 and a maximum test case length (given as input). Then, it looks for the events available on the A_R initial GUI state, randomly selects an event, executes it, and repeats this process until generating a test case of the selected length. When ADAPTDROID executes a *fill* event, it randomly selects an input string from a pool of input values composed of the strings used in t_D *fill* events, and a set of randomly generated strings.

Greedy Match Test: a greedy matching test is a test case that greedily executes events in A_R that match those of t_D. ADAPTDROID generates greedy match test cases by opening the target application A_R and checking if it can execute an event that matches t_D events in the current state. If it does not identify any event that matches t_D events, AUGUSTO randomly selects a new event and executes it, and continues for a number of steps equal to the length of t_D.

6.3.4 Fitness Calculation

The ADAPTDROID fitness calculation phase executes the test cases in $\mathcal{P}_i$, and associates them a fitness score between 0 and 1 that characterizes their similarity to the test case t_D.

Population Execution

Each test case in $\mathcal{P}_i$ is firstly executed on A_R. To execute a test case, ADAPTDROID opens A_R and executes in sequence each event in the test. Note that, as a result of the population evolution phase it can happen that some test cases are not completely executable, meaning that not all the events can be performed (for instance because a widget that has to be clicked is no longer visible). More specifically, the crossovers and mutations can add unfeasible events in the test cases. In such a case, ADAPTDROID skips the execution of the unfeasible events and removes them from the test and then proceeds to execute the following events.

Note also that the test cases in $\mathcal{P}_i$ do not include assertions as the assertions are added only during the post-processing phase.

During the execution of the test cases in the population, ADAPTDROID observes builds and updates a GUI model of A_R. The model is defined as described in Section 2.3.2 and has exactly the same structure of the GUI model employed by AUGUSTO (see Section 5.2.2). ADAPTDROID uses A_R*GUI Model* during the population evolution phase to appropriately combine and mutate the test cases.

Fitness Score

ADAPTDROID computes a score on how much a test case $t \in \mathcal{P}_i$ is similar to t_D considering two aspects: the event similarity and assertion applicability. The *events similarity* captures the similarities between events in t_R and t_D. The *assertion applicability* captures whether the assertions in the donor test case are compatible with the states reached by test t_R.

Given a donor test case t_D, the resulting formula for the fitness score is

$$\text{FITNESS-SCORE}(t_R) = \frac{|\, M^\star \,| + |\, O_D^\star \,|}{|\, t_D \,| + |\, O_D \,|} + \epsilon$$

where $M^\star$ represents the events in t_R that have been properly matched with t_D, O_D represents the assertions present in t_D, and $O_D^\star$ represents the assertions in O_D that have been properly injected in t_R. The operator || indicates the set cardinality operator. Finally, ϵ an arbitrarily small value that is added to the fitness score to reward the test cases with the higher potential to improve in the future iterations. In the following, we describe in details the computation of each element of the fitness function.

Events similarity. This quantity captures how similar the events in t_R are to those in t_D. Considering our flexible semantic matching strategy presented in Section 6.3.1, the events in t_D can be matched to the events in t_R in multiple ways, for instance, a given click event in t_D might match with multiple click events in t_R (if they have similar descriptors). When counting the number of events in t_D that have been properly matched in t_R, there could be multiple ways of pairing the events, and thus multiple results possible.

More formally, let $\mathcal{M}$ denote the set of all possible mappings between the events in t_D and t_R. A mapping M is a subset of the events in t_D that have at least one match in t_R, that is, $M \subseteq t_D : \forall e_x \in M \; \exists e_y \in t_R \wedge e_x \sim e_y$. To have a meaningful mapping and thus a meaningful fitness score, ADAPTDROID imposes some constraints on the mappings in $\mathcal{M}$ and filters out those that do not satisfy

them. A mapping $M \subseteq t_D$ is a *valid* mapping if and only if all of these criteria are satisfied:

Unique events matching. An event in t_R can be matched with one event in t_D at maximum.

Event ordering. The ordering of the events as extracted in the pre-processing phase from the donor test case must be respected. More formally, M is valid if $\forall e_i, e_j \in M \subseteq t_D$ with $e_i <_{clusters} e_j \; \exists e_a, e_b \in t_R : e_i \sim e_a \wedge e_j \sim e_b$, where $a < b$.

Consistent matching. A valid mapping must be consistent, meaning that two events in the donor test case associated with the same event descriptor must be matched to consistent recipient events. This constraint avoids mapping two equivalent events in t_D (such as clicking twice on the same button) to different widgets in A_R. More formally, M is valid if $\forall e_i, e_j \in M \subseteq t_D : d_{e_i} = d_{e_j}, \exists e_a, e_b \in t_R : e_i \sim e_a \wedge e_j \sim e_b \wedge d_{ea} = d_{eb}$.

The *event similarity* component of the fitness function is computed identifying the valid mapping $M^\star$ with the highest cardinality (number of elements), that is, $M^\star \in \mathcal{M}$ satisfies $\nexists M \in \mathcal{M}$ such that $\mid M \mid > \mid M^\star \mid$. The fitness score considers $\mid M^\star \mid$ at the numerator and $\mid t_D \mid$ at the denominator to account for the subset of the events that have been successfully matched in t_R.

If consider our running example, we can notice that the possible events mapping $M : \; e_{D2} \sim e_{R1}, \; e_{D4} \sim e_{R1}$ would be filtered by the unique event matching constraint, while mapping $M^\star = \; e_{D2} \sim e_{R1}, \; e_{D4} \sim e_{R4}, \; e_{D6} \sim e_{R5}$ is the valid mapping with the highest cardinality.

Assertion applicability. The assertion applicability captures the fact that a good adaptation of the donor test case must reach a state in A_R compatible with the assertions O_D available in t_D. In other words, after the execution of the adapted test events, it should be possible to see a widget which is compatible with the widget the original assertion predicates upon. ADAPTDROID quantifies this aspect in the fitness score as the proportion of assertions in t_D that can be checked in t_R.

An assertion is checkable if it exists a state traversed by the execution of t_R after the execution of the last event in the mapping $M^\star$ that includes the elements that should be verified by the assertion.

Formally, an assertion $o \in O_D$ (not using a negation) can be applied to a state S_i visited by t_R, if there exist a widget $w \in S_i$: ISSEMSIM(d_w, d_o), where d_o is the assertion descriptor, which we define as follows. If $o =$`exists(txt)`, $d_o =$`txt`, otherwise if $o =$`hasText(w,txt)`, $d_o = d_w$. We also define function ISAPPLICABLE$(o, M^\star)$ that returns 1 if o is applicable in a state traversed by t_R after the execution of the last event in the mapping, 0 otherwise.

In case o is defined as a negation, such as `not(exists(txt))`, we apply a slightly different semantics. In fact, it is easy to find a state that does not contain a certain widget, indeed most of the states traversed by an execution satisfy this condition. In this case, we require t_R to first cover the positive part of the assertion, `exists(txt)` in the example, and then cover the negated assertion in the same window. In this way, we make sure that t_R satisfies the negation explicitly moving from a state in a window that does not satisfy it to a state in the same window that satisfies it.

Formally, given $o \in O_D$ (using a negation), ISAPPLICABLE$(o, M^\star)$ returns 1 if the positive version of o is applicable to a state S_i traversed during t_R execution, its negated version is applicable to a state S_j traversed after the last event in the mapping $M^\star$, and S_i is traversed before S_j and are in the same window; 0 otherwise.

In our example, t_D assertion verifies that no widget with text "Test" exists. The assertion is applicable to the last state visited by t_R because it does not contain a widget with the descriptor "Test", but such widget appears in that window in state 4.

The quantity $O_D^\star$ that is part of the fitness score counts the number of assertions that are applicable in the adapted test case, that is, $O_D^\star = \sum_{a \in O_D}$ ISAPPLICABLE$(a, M^\star)$. The contribution to the fitness score is the ratio between $\mid O_D^\star \mid$, the applicable assertions, and $\mid O_D \mid$, the assertions available in the donor test case.

The value of the fitness score without considering ϵ thus ranges between 0 and 1, with 1 representing an adapted test case that reproduced all the events in the donor test case, including the assertions. To privilege the test cases that have more chance to improve in the future, we increase by a small value $\epsilon > 0$ the fitness score of the test cases that traversed states that can be clearly exploited to improve the tests in the future (the fitness score is, in any case, bounded to 1). These states are the ones where the events that have not been matched so far could have been executed. For instance, the state includes a button that is enabled and that matches with a click event in the donor test case that has not been matched yet. In this way, when two test cases perform the same in terms of matching events and applicable assertions, the genetic algorithm privileges the test cases with the greater potential. The specific value of $epsilon$ is not important as long as it is less than the increase in fitness given by a matched event or by an applicable assertion, thus $epsilon < \frac{1}{\mid t_D \mid + \mid O_D \mid}$.

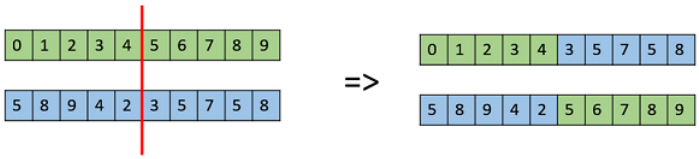


Figure 6.3. Crossover example

6.3.5 Population Evolution

The population evolution phase combines and modifies the best elements in the current population $\mathcal{P}_i$ to generate a new population $\mathcal{P}_{i+1}$ with fitter test cases (i.e., higher fitness score). The evolution process follows a classic genetic approach [113], which work in four consecutive steps: elitism, selection, crossover, and mutation.

Elitism

Before starting the evolution phase, ADAPTDROID adds the best E test cases seen so far into $\mathcal{P}_i$, thus obtaining a population of $N+E$ tests, where N is the fixed size of the population. This process is called *elitism* [113] and it is a standard way used in genetic algorithms to ensure that the best solutions are not lost during the evolution process [113]. ADAPTDROID then uses the $N+E$ tests in the population to generate $\mathcal{P}_{i+1}$.

Selection

ADAPTDROID selects $N/2$ pairs of test cases from $\mathcal{P}_i$ as candidates for crossover. ADAPTDROID implements the standard selection approach called *roulette wheel* [113], which gives to each test a probability of being selected proportional to its fitness. Thus, a test case with a higher fitness is likely to be selected several times, while a test with low fitness might not be selected at all [113].

Crossover

Each pair of test cases selected is combined producing a new test case with probability C, or the tests are left unmodified and passed to the next phase with probability $1-C$. ADAPTDROID implements a single-point cut crossover. Given a selected pair t_{R1} and t_{R2}, ADAPTDROID selects two random cut points that split t_{R1} and t_{R2}, in two segments. Then, it creates two new test cases, one concatenating the first segment of t_{R1} and the second segment of t_{R2}, and the other concatenating

the first segment of t_{R2} and the second segment of t_{R1}. Figure 6.3 visually depicts the crossover of two test cases.

The crossover could create an unfeasible test case if the execution of the first segment terminates in a window that is different from the one that is expected by the first event in the second segment. To make the test feasible, ADAPTDROID tries to automatically insert between the two segments a sequence of events that repairs the test making it runnable. ADAPTDROID identifies such sequence by querying the *GUI Model* of A_R that represents which events can trigger a window transition. ADAPTDROID incrementally builds this model observing the window transitions triggered by the execution of the test cases.

Mutation

When P_{i+1} reaches a size of N elements, the crossover step terminates and the algorithm modifies test cases in P_{i+1} by mutating them. As for the generation of the initial population, ADAPTDROID aims both at introducing random genetic diversity and converging to a (sub)optimal solution faster. As such, ADAPTDROID uses two types of mutations: random and fitness-driven.

Random mutations, as the name suggests, randomly modify test cases. Each test case in P_{i+1} has a probability RM of being mutated by one of the three random mutations supported by ADAPTDROID: (i) adding an event in a random position; (ii) removing a randomly selected event; and (iii) adding multiple fill events in a window containing text fields. The rationale of the last mutation type is that in case of forms with several fields, it might require a huge number of generations to fill all those fields with the random add mutation. Thus, this mutation speeds up the evolution by filling all the text fields in a single mutation.

Fitness-driven mutations mutate a test case applying heuristics with the goal of improving the fitness score of the modified test case. Each test case in P_{i+1} has a probability FM of being mutated using one of these two mutations:

- *Fitness-Driven Remove*. It removes one of the events in t_R that does not match (according to $\sim$) any of event in t_D. This mutation helps removing spurious events that do not contribute to increasing the value of the fitness.

- *Fitness-Driven Add*. It selects an event $e_x \in t_D$ that is not matched by any of the events in t_R, and adds a new event e_j in t_R such that $e_x \sim e_j$. The added event is selected among the events which are available during the execution of t_R.

6.3.6 Post-Processing

The search for an adapted test case keeps evolving and evaluating populations of test cases until a predefined number of evolutions have been performed or when a test case with fitness 1.0 is found. When the search terminates, the test case with the highest fitness is post-processed and returned to the user. The post-processing phase performs two operations: reduces the test case length by eliminating irrelevant events, and applies (if possible) the donor assertions to it.

To reduce the test case length, ADAPTDROID removes one by one the events that are not part of the mapping used to calculate the fitness score. After one event is removed, ADAPTDROID executes the test and recalculates its fitness. If the fitness decreases, the event is added back because the removed event, even though did not directly contribute to the fitness value, enabled other relevant events to be executed. If we consider the example in Figure 6.1, events 2, 3, 6, and 7 of the adapted test case were not matched with any of the events in the donor test case. ADAPTDROID tries to remove these events one by one, but removing each of them would cause a decrease in the fitness (e.g., if event 3 is removed the form cannot be submitted and the "Test" item cannot be created), thus the test case is left unchanged.

Finally, if the fitness function evaluated t_D assertions as applicable to t_R, ADAPTDROID adds them unchanged at the end of the test case. Thus, in our example the original assertion `not(exists(txt))` is applied to the A_R state after the execution of all the events in t_R.

6.4 Implementation

To evaluate the effectiveness of the ADAPTDROID approach, we developed a prototype that targets Android applications. The prototype is developed in using Java 8, it comprises about 10k lines of code, and it relies on the Appium [8] framework as the underlying technology to interact with the Android applications GUIs.

Appium framework allows to read an Android application GUI and get information on all the widgets shown in the screen at a given moment but does not allow to read the widgets that are outside the screen boundaries. In those cases, it is required to scroll the screen to read and access the hidden widgets. ADAPTDROID prototype during the evolution phase might randomly add scroll events in order to access all widgets in the GUI. We did not discuss this technical detail in the approach to keep the description simpler.

ADAPTDROID prototype executes the android AUT using Android 5.1 emulators and supports the parallel use of multiple emulators to optimize the test case execution step. To guarantee that each test case is executed starting from the same app state, ADAPTDROID prototype deletes and re-installs the AUT in the emulator before each test case execution.

To calculate the semantics distance between string ADAPTDROID uses a pre-trained Word2Vec model with more than 3 million words obtained from crawling Google News [111].

The prototype and its implementation are freely available under MIT licensing and it can be found at *http://github.com/danydunk/AdaptDroid*.

6.5 Evaluation

We experimentally evaluated ADAPTDROID with a prototype implementation that we used to adapt eight test cases from four different donor apps to generate test cases for 24 different recipient apps. Our evaluation addresses three research questions:

(RQ1) Can ADAPTDROID effectively *adapt test cases* across different interactive applications?

This research question investigates the capability of ADAPTDROID to automatically produce a test case for the AUT which has the same semantics of a source test written for a different application.

(RQ2) Does ADAPTDROID *explore the search space* more effectively than random exploration?

This research question investigates whether the test case produced by ADAPTDROID could be generated also employing a simple random search and whether the fitness function presented in Section 6.3.4 is actually fundamental to generate good solutions.

(RQ3) What is the contribution of the *greedy match initial test cases and fitness-driven mutations* on ADAPTDROID approach effectiveness?

This research question investigates whether the population initialization with greedy-match test cases and the fitness-driven mutation employed by ADAPTDROID are actually helpful in converging to a good solution quicker than the default random initialization and mutations employed by standard search-based approaches.

6.5.1 Empirical Setup

We selected a total of 32 Android apps from the Google Play Store by referring to the following popular types of mobile applications: “Expense Tracking”, “To-do List”, “Note Keeping”, and “Online Shopping”. We considered these 4 types of applications because they represent applications with recurrent functionalities and therefore we believe they are amenable to test adaptation. We used eight apps as donor (A_D) and 24 apps as recipients (A_R).

We selected the candidate donor applications by (i) querying the Google Play Store with the name of each category in the search bar, and (ii) selecting the first two apps that are either free or freemium and do not require login credentials at start-up.

We selected three A_R for each donor app A_D, by (i) querying the Google Play Store page of each A_D for *similar* apps, (ii) selecting the first three suggested similar apps that are were not selected as A_D, and have the same characteristics of donor apps described above. This process resulted in a total of 24 pairs $<A_D, A_R>$ of donors and recipient applications, selected objectively.

Donor test cases. We asked four independent testers to design a total of eight test cases (one for each donor app) for the eight donor applications, and to evaluate the total 24 test cases that ADAPTDROID generated for the corresponding recipient applications. We decided to experiment with donor test cases developed by independent testers and not on test cases released with the apps, to be able to ask for an independent assessment of the adaptations that ADAPTDROID generates for the recipient apps. We believe the best person to assess a test adaptation to be the creator of the original test case, which would be hard to contact in the case of test cases released with open-source apps. The independent testers are Ph.D. students majoring in software engineering, not related to the research project, and not aware of the goal of the study when designing the tests.

We assigned to each tester two randomly selected A_D of different categories[2] and asked them to design a test case for each assigned apps.

The first five columns of Table 6.2 provide essential information about the evaluation subjects. They show the donor apps and their category (Column *Donor App* (A_D)), the testers who designed the test (Column *Tester*), the length of the donor test cases in terms of number of events (Column $|t_D|$), the recipient apps(Columns *Recipient App* (A_R)), and a progressive id that identify each pair $<A_D, A_R>$ (column *ID*).

We asked each tester to evaluate whether their test cases could be adapted to the corresponding A_R. Each tester evaluated six pairs $\langle A_D, A_R \rangle$ on a scale "Yes",

[2] In this way, each tester had to design two conceptually different tests.

"No", and "Partially", where "Partially" means that some parts of the donor test cannot be replicated in the target application (Column *Adaptable?*). We also asked the testers to manually adapt the donor test cases for the recipient apps in the cases they estimated as fully ("Yes") or partially ("Partially") adaptable. The testers estimated as either fully ("Yes") or partially ("Partially") adaptable 18 and 3 pairs of test cases, respectively (75% and 12.5%), and considered as non-adaptable ("No") only 3 pairs of test cases (12.5%), thus confirming our intuition that tests are often adaptable across similar applications.

Testers T2 and T3 designed donor test cases that exercise functionalities of the donor apps not present in the corresponding recipient application, and thus could not be adapted. We asked them to design a new donor test cases referring to functionalities present also in the recipient applications. Tester T2 succeeded (we mark the corresponding cells in Column *Adaptable?* with a *), while tester T3 was not able to do so as the donor app Pocket Universe did not have any functionality that was found also in the recipient apps.

Running ADAPTDROID. We ran ADAPTDROID 21 times (we excluded the three cases in which the donor test was not adaptable), giving in input each pair of A_D and A_R and the corresponding manually-written t_D. We ran ADAPTDROID with a budget of 100 generations (i.e., iterations of the genetic algorithm) with the following configurations: We use a population size (N) of 100 elements, we set the bound on the maximum test case length to 25 and we set the size of E (# test cases for the elitism) to 10. ADAPTDROID generates the initial population with 10% of greedy matching and 90% random test cases. We set the crossover probability (C) at 0.4, and the probability of both random (RM) and fitness-driven mutations (FM) to 0.35. We set to 0.65 the threshold τ for the WMD string semantic similarity. We selected these configuration values by performing some trial runs.

Regarding execution time, on average ADAPTDROID completed 100 generations in 24 hours. Most of ADAPTDROID execution time is spent executing the generated test cases on the emulator (100 generations containing 100 test cases amounts to 10000 test cases to execute). This time could be significantly reduced by using more parallel emulators to execute the test cases or using real devices instead of emulators.

Note that for the adaptation with ID 21, the Appium framework had compatibility issues with the recipient application A_R that prevented ADAPTDROID to generate tests for A_R. Since we had already asked tester T_4 to explore A_R and manually adapt t_D for the A_R, we did not ask him/her to re-do the work for another A_R.

Table 6.2. Evaluation subjects and results

Tester	Donor App (A_D)	$\lvert t_D \rvert$	ID	Recipient App (A_R)	Adaptable?	Q_T	$\lvert t_R \rvert$	# spurious	# missing	Q_S	O. Adapted?
T1	Expense Manager [26] (Expense Tracking)	15	1	KPmoney [6]	Partially	3	13	6	0	1,00	No
			2	Monefy [14]	Yes	4	10	1	0	1,00	Yes
			3	Money [92]	Yes	4	15	0	0	1,00	Wrongly
	Mirte Notebook [83] (Note Keeping)	10	4	Xnotepad [54]	Partially	1	7	4	2	0,50	No
			5	Color Notes [9]	Yes	2	15	9	2	0,75	Wrongly
			6	Kepp Mynotes [112]	Yes	1	2	1	5	0,14	No
T2	Markushi Manager [74] (Expense Tracking)	16	7	Spending Tracker [81]	Yes*	2	23	1	8	0,73	Wrongly
			8	Smart Expendit [35]	Yes*	0	18	-	-	0.00	-
			9	Gastos Diarios [34]	Partially*	0	6	-	-	0.00	-
	Bitslate Notebook [27] (Note Keeping)	13	10	Notes [100]	Yes	3	8	0	2	0,80	No
			11	Noteme [102]	Yes	4	10	0	2	0,83	Yes
			12	Notepad [5]	Yes	1	12	10	11	0,15	Yes
T3	Pocket Universe [106] (To-dolist)	11	13	Seven Habits [7]	No	-	-	-	-	-	-
			14	Ob Planner [31]	No	-	-	-	-	-	-
			15	Simple Checklist [101]	No	-	-	-	-	-	-
	Aliexpress [1] (Online Shopping)	16	16	Banggood [17]	Yes	2	11	4	1	0,88	No
			17	Light in the box [62]	Yes	1	7	4	5	0,38	No
			18	Shein [49]	Yes	0	9	-	-	0.00	No
T4	Zalando [116] (Online Shopping)	6	19	Zara [117]	Yes	3	8	0	0	1,00	No
			20	Romwe [94]	Yes	3	5	0	1	0,83	No
			21	Yoox [114]	Yes	-	-	-	-	-	-
	Splendo [10] (To-dolist)	10	22	To Do List [86]	Yes	3	9	4	6	0,45	Yes
			23	Tasks [85]	Yes	1	8	6	5	0,29	No
			24	Tick Tick [56]	Yes	1	15	11	8	0,33	Yes

6.5.2 RQ1: Effectiveness

After collecting the output of ADAPTDROID, we asked the testers to evaluate the effectiveness of ADAPTDROID in adapting their tests. We showed the adapted tests t_R generated by ADAPTDROID for the assigned pairs $\langle A_D, A_R \rangle$ to each tester. For each t_R, we asked the tester to judge the quality of t_R on a scale from 0 to 4, where 0 indicates that t_R is not related at all to the original test semantics, and 4 indicates that t_R is an adaptation as good as the one they manually produced. Column Q_T in Table 6.2 shows the *tester quality evaluation* for each adaptation case.

If the tester gave a score different from zero, we asked him/her to further evaluate the test generated by ADAPTDROID by marking the *spurious events* (those that do not contribute in the test adaptation) and to give us the number of *missing events* to have a perfect adaptation. Note that this process was facilitated by the fact that the testers have already implemented the adapted test case manually. Column $|t_R|$ shows the length (in terms of the number of events) of the test cases generated by ADAPTDROID. Columns *#Spurious* and *#Missing* show the number of events in t_R the tester marked as spurious and missing, respectively. Using this information we computed Q_S, a *structural* quality indicator that encompasses the completeness of the matched events. We calculate Q_S as follows:

$$Q_S = 1 - \frac{\#missing}{\mid t_R \mid - \#spurious + \#missing}$$

In eight cases out of 20 (40%) testers evaluated ADAPTDROID adaptations as high quality ($Q_T \geq 3$), three of which were considered perfect adaptations. In three cases (15%), the adapted test generated by ADAPTDROID were evaluated with a medium quality ($Q_T = 2$) and in nine cases (45%) they received a low quality evaluation ($Q_T \leq 1$). Overall, ADAPTDROID adaptations received an average score of 1.95. As Table 6.2 shows, the testers often reported several spurious and missing events. In particular, indicator Q_S, which shows the ratio of matched events, has an average value of 0.53, indicating that overall ADAPTDROID was able to execute 53% of the *true* event matches identified by the testers. There is a good correlation between the two quality indicators Q_T and Q_S (Pearson coefficient [87] is = 0.89), which confirms that adapting a large portion of the test is important for the testers. However, it is interesting to notice that a low value of Q_S is not *always* associated with a low value of Q_T. Consider for instance the case with ID 22. In that case, ADAPTDROID failed to execute half of the matching events, but the tester gave those events a low importance and thus still gave a high Q_T score to the adaptation.

We can notice that in the 3 cases where the testers identified that the original test case could be adapted only partially (ID 1, 4, 9), ADAPTDROID adaptations had a good quality in one case (ID 1) and low quality in the other two cases (ID 4 and 9). These results seem to suggest that the cases in which the donor test is only partially adaptable are more challenging (as expected) and that impacts ADAPTDROID adaptations quality. However, ADAPTDROID approach showed that in some cases it *can* achieve a good result also in these cases.

The last Column in Table 6.2 (*O. Adapted?*) reports whether ADAPTDROID was not able to adapt the assertions ("No"), adapted the assertions and the tester evaluated them as correct ("Yes") or incorrect ("Wrong"). ADAPTDROID adapted the original assertions in eight cases (40%), and in three cases the adaptation was incorrect. The reason of the wrong oracle adaptations is due to the fact that ADAPTDROID (if possible) simply adds the original assertion without performing any adaptation on it, thus it is not resilient to the possible difference in which data are presented across the different GUIs. For instance, in the adaptation with ID 3, the assertion in the donor test checks if a widgets with descriptor "expenses" has text "100", but the corresponding widget in the recipient app showed had text "-100", which is semantically equivalent (100 expenses = -100 total balance) but syntactically different and therefore the oracle was deemed as incorrect.

Notice the adapting the assertions of the donor tests was not always possible. In the three cases in which the donor test was only partially adaptable, the testers identified their assertions as not adaptable (that contributed to their evaluation of the donor tests being only partially adaptable). Thus, if we exclude these cases,

ADAPTDROID adapted the original assertions in eight cases out of 17 (47%), five of which were correct (29%).

To better evaluate the performance of ADAPTDROID, we manually evaluated the cases in which it performed poorly. We noticed three main issues that prevented ADAPTDROID to be effective:

1) Significant differences between t_D and t_R. In some cases, A_R and A_D had significant differences that required complex adaptations that go beyond the capabilities of ADAPTDROID. For instance, in adaptation with ID 18, t_D searches in the *Aliexpress* [1] Online Shopping application for a USB drive and adds it to the shopping basket. Since *Shein* [49] is an Online Shopping application that sells clothing, T3 adapted t_D searching for a T-shirt instead. As expected, ADAPTDROID was not able to perform this adaptation as it tried to search for a USB drive resulting in an empty search result. As another example, in adaptations with ID 23 and 24 t_D adds a task under the pre-existing *work* tasks list. However, the recipient applications do not have a predefined *work* task list, thus t_R should create one. Being task list creation not part of t_D, ADAPTDROID was not able to do so, thus producing a poor adaptation.

2) Missed Event Matches. ADAPTDROID relies on matching events based on their descriptors extracted from the GUI. However, events matching was not always effective due to (i) the unsoundness of WMD in matching event descriptors; and (ii) the limited semantics information of the event descriptors. To give an example, some of the considered apps had image buttons with file names like "fabButton.png" which is hardly helpful to describe the semantic of the widget and identify similar events across applications.

3) Incidental Event Matches. In our experiments, we noticed that ADAPTDROID often wrongly matched widgets that were actually not semantically equivalent. In most cases, this problem was caused by the use of the same word but in different contexts. For instance, WMD identifies as semantically correlated the strings "add configuration" and "add task" because the two strings both contain the word "add".

The first issue described is one of the intrinsic challenges of the test case adaptation problem. ADAPTDROID approach is able to mitigate this problem when the differences are of a smaller entity, like those described in the running example used to present the technique, but it is not able to overcome it in case of significant difference like those described above.

These other two problems discussed instead, boil down to the overall same issue: the event matches computed for the fitness function calculation can often be imprecise due to the limitations of the WMD or limited descriptiveness of the

Table 6.3. RQ2: Fitness values achieved by ADAPTDROID and RANDOM

	ADAPTDROID		RANDOM			ADAPTDROID		RANDOM	
ID	**H-Score**	**Gen.**	**H-Score**	**Gen**	**ID**	**H-Score**	**Gen**	**H-Score**	**Gen**
1	0,64	33	0,30	79	11	0,41	86	0,23	6
2	0,47	59	0,23	3	12	0,11	1	0.11	1
3	0,47	95	0,30	2	16	0,50	43	0,30	11
4	0,78	91	0,36	8	17	0,50	75	0,23	23
5	0,24	4	0,21	40	18	0,30	92	0,17	62
6	0,39	1	0,33	79	19	0,42	19	0,38	32
7	0,74	59	0,72	7	20	0,54	5	0,50	1
8	0,41	25	0,32	3	22	0,63	59	0,38	46
9	0,50	17	0,37	19	23	0,46	32	0,34	59
10	0,45	15	0,33	61	24	0,46	21	0,29	7

event descriptors. For this reason, in some cases the computed fitness values mislead the search-based adaptation process causing low quality results. In our analysis we noticed that this overall issue affected in a way or another basically all the 20 adaptations instances in our study. However, in some cases ADAPTDROID search based approach was able to overcome these issues, in some cases learning to avoid a given (incidentally) matching event cause executing it did not allow to match other events, or learning sequences of events that even though did not match the donor events, contributed to arrive to perform other matching events.

Considering the complexity of the problem, the results of our study are both positive and promising. ADAPTDROID often produced test adaptations which were deemed of good quality by the testers and that in nearly half of the cases included oracles, which significantly increase the failure discovery ability of the test cases compared to tests generated without adaptation. Although we reported data for every adaptation task we studied, it is possible to configure ADAPTDROID to report only adapted test cases that reach a minimum fitness score, thus significantly improving the quality of the generated output for the testers who are not annoyed with poorly adapted test cases. For instance, with a 0.45 threshold, ADAPTDROID filters out 5 of the poor test cases, and reports only 4 poor test cases out of a total of 15 adapted test cases (26%). However, our evaluation showed that more work is needed to improve the approach. ADAPTDROID widget matching based on WMD showed to be imprecise and that limited the overall results of the approach, especially in its ability to adapt the oracle.

6.5.3 RQ2: Comparison with Random Search

RQ2 aims to verify whether ADAPTDROID adaptations could be generated with random search. To answer RQ2 we created a variant of ADAPTDROID that we call

RANDOM with the following modifications: (i) we removed the roulette-wheel selection in the crossover phase and we replaced it with random selection; (ii) we set that all the test cases in the initial population are generated randomly; (iii) we set the probability of fitness-driven mutations to 0; (iv) we disabled elitism. Thus, RANDOM carries population initialization and evolution completely random. We opted to use a random variant of ADAPTDROID rather than an existing random test generator to perform a meaningful evaluation. If we compare ADAPTDROID with an existing random test generator we cannot ensure that the differences are due to the search strategy and not due to the two tools differences in events and inputs to generate tests.

We ran RANDOM with the same 100 generations budget used to run ADAPTDROID and we computed the fitness score of each randomly generated tests. We compare ADAPTDROID and RANDOM in terms of the highest fitness score they achieved. The rationale of this choice is to verify whether it is possible to generate at random test cases of similar quality (i.e., similar fitness value) compared to those produced by ADAPTDROID. Table 6.3 shows for ADAPTDROID and RANDOM the highest fitness score observed after 100 generations (Column "H-Score") and the number of the generation in which the H-Score was first observed (Column "Gen."). As the results show, RANDOM consistently achieves a lower fitness value compared to ADAPTDROID, and only in a few cases they achieve similar or equal values. In average ADAPTDROID achieved a fitness of 0.48, while RANDOM had an average fitness of 0.32. To further validate this result, we applied a two-tailed statistical t-test [12, 99] (after verifying that the distributions of fitness values achieved by ADAPTDROID and RANDOM follow a normal distribution with the Shapiro-Wilk test [96]). The resulting $pvalue$ is 0.002, thus confirming that the fitness values achieved by ADAPTDROID are higher than those achieved by RANDOM (higher average fitness) and that this result is statistically significant ($pvalue < 0.05$). Moreover, it is interesting to notice that in the few cases in which ADAPTDROID and RANDOM have similar values they both achieve low fitness.

Figure 6.4 shows the average "H-Score" per generation of the 20 adaptations. The figure shows that ADAPTDROID steadily increases while the fitness of Random achieves saturation much faster. This is also reflected by the fact that ADAPTDROID often observes the highest fitness after more generations than RANDOM (Column "Gen." Table 6.3).

Figure 6.4 indicates that the search-based approach is essential to produce test cases with high fitness scores. ADAPTDROID starts by generating test cases that greedily execute events that can match the donor test events, thus it obtains a higher fitness than RANDOM already in the first generation. However, Figure 6.4 shows that this simple greedy strategy is not enough as the fitness value keeps

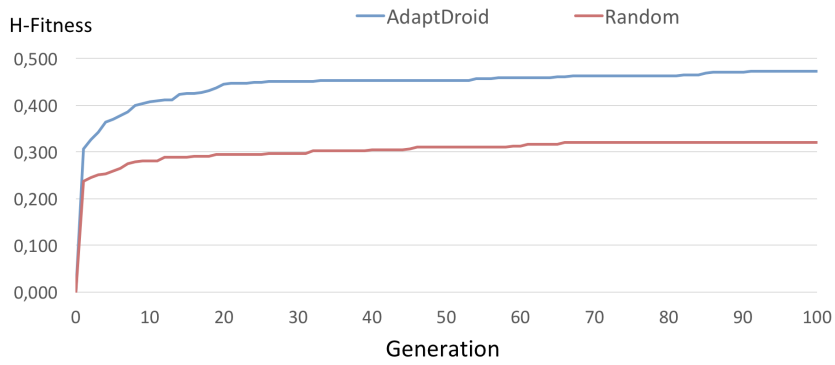


Figure 6.4. Average fitness growth ADAPTDROID vs RANDOM.

Table 6.4. RQ3: Fitness values achieved by ADAPTDROID and ADAPTDROID-SIMPLE

	ADAPTDROID		ADAPTDROID-SIMPLE			ADAPTDROID		ADAPTDROID-SIMPLE	
ID	H-Score	Gen.	H-Score	Gen	ID	H-Score	Gen	H-Score	Gen
1	0,64	33	0,57	50	11	0,41	86	0,44	41
2	0,47	59	0,43	98	12	0,11	1	0.11	1
3	0,47	95	0,40	82	16	0,50	43	0,64	36
4	0,78	91	0,77	80	17	0,50	75	0,44	72
5	0,24	4	0,27	9	18	0,30	92	0,30	64
6	0,39	1	0,33	11	19	0,42	19	0,42	22
7	0,74	59	0,74	36	20	0,54	5	0,54	11
8	0,41	25	0,41	16	22	0,63	59	0,55	26
9	0,50	17	0,50	22	23	0,46	32	0,38	10
10	0,45	15	0,31	10	24	0,46	21	0,53	93

increasing significantly after the first generation.

In summary, the experimental results indicate that ADAPTDROID search-based approach is effective in exploring the search space, and confirm our hypothesis that the test cases generated by ADAPTDROID can hardly be generated at random. Moreover, these results suggest that the search-based approach leveraged by ADAPTDROID is fundamental to obtain tests with high fitness values.

6.5.4 RQ3: Greedy-match Initialization and Fitness-driven Mutations Evaluation

RQ3 aims to assess the contribution of ADAPTDROID *smart* initialization and mutations on the overall approach effectiveness. To answer this research question we created a variant of ADAPTDROID that we call ADAPTDROID-SIMPLE with the following modifications: (i) we set that all the test cases in the initial population are generated randomly; (ii) we set the probability of fitness-driven mutations to

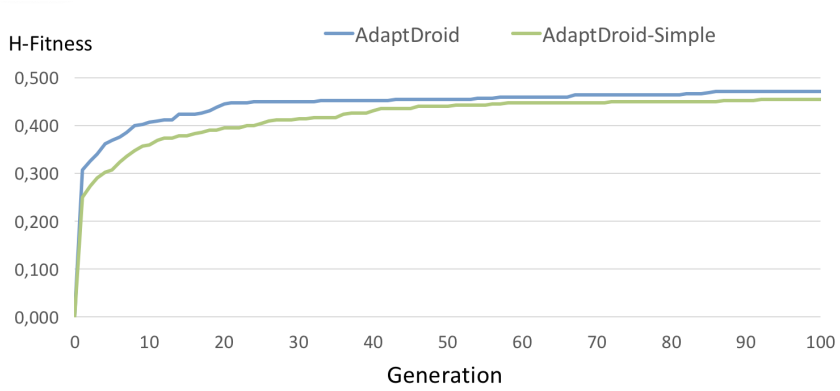


Figure 6.5. Average fitness growth ADAPTDROID vs ADAPTDROID-SIMPLE.

0.

We ran ADAPTDROID-SIMPLE with the same 100 generations budget used to run ADAPTDROID. Table 6.4 reports for ADAPTDROID and ADAPTDROID-SIMPLE the highest fitness score observed after 100 generations (Column "H-Score") and the number of the generation in which the H-Score was first observed (Column "Gen."). As the results show, in most of the cases ADAPTDROID achieves slighter higher values of fitness value than ADAPTDROID-SIMPLE, while in four cases it is ADAPTDROID-SIMPLE to achieve higher values. In average ADAPTDROID achieved a fitness of 0.48, while ADAPTDROID-SIMPLE had a average fitness of 0.45, thus showing a negligible difference in terms of the best solution that can be reached over 100 generations.

Figure 6.5 shows the average "H-Score" per generation of the 20 adaptations. The figure shows that even though the two approach reach almost identical fitness values, ADAPTDROID reaches the highest fitness value quicker, thus requiring tens of generations less to converge to the final solution.

These results show that although the greedy-match tests initialization and the fitness-driven mutation have limited impact on the quality (in terms of fitness value) of the final solution found, our results indicate that the *smart* initialization and mutations employed by ADAPTDROID can allow the approach to perform the search more effectively and converge quicker to the final solution.

6.5.5 Threats to Validity

A threat to external validity derives from the limited set of categories of applications that we used to select the donor applications, and that questions the generalisability of our results to other categories of applications. We selected the

donor applications from four categories that we believed to be amenable for test case adaptation to be able to evaluate the effectiveness of the technique in the cases when adapting the donor test is indeed possible.

Another threat to external validity derives from the choice of the specific donor and recipient applications. We avoid biases in the selection of the applications by choosing the donors from the first search results of Google Play Store and by choosing the recipients from the similar apps suggested by the Google Play Store.

A threat to internal validity derives from the selection of the testers of our study. The four testers are all experienced in testing but they are not the developers of the selected applications. Although their test cases might be different from those that the original developers of the app would write, we let the testers take confidence with the apps before designing the test cases, mitigating the issue of working with non-representative test cases.

Another threat of internal validity derives from the statistical significance of the results of our study. Since ADAPTDROID has a random component in it, multiple runs may yield different results. However, since the evaluation of the results involved human participants, who are able to evaluate a small number of tests only, we necessarily had to limit the number executions used in the evaluation.

A final threat to the internal validity derives from the setting of ADAPTDROID configuration parameters for the empirical study. The selection of different probabilities for crossover, greedy-match initialization, random mutations, fitness-driven mutations might result in the approach having a significantly different performance. We mitigated this issue performing some trial runs and selecting the best configuration parameters according to the results of those runs and our knowledge of the technique.

Chapter 7

Conclusions

This thesis addresses the problem of automatically generating system test cases for interactive applications. This problem has been investigated both academically and industrially, aiming to reduce the overall cost of verification and validation activities for the development of interactive applications. So far, most of the approaches proposed to address this problem mainly explore the GUI of the application under test guided by structural information to generate test cases.

In this thesis, we study in depth this problem and its state of the art. To study the strengths and limitations of the current techniques, we present an empirical comparison that we conducted among the most mature test case generation techniques for desktop interactive applications. This study provides empirical evidence showing that the effectiveness of state-of-the-art techniques is still fairly limited as they often fail to generate semantically meaningful test cases that can trigger important faults and do not include functional oracles in the test cases they produce.

This thesis pinpoints the lack of information about the AUT semantics as the main factor limiting the effectiveness of the existing techniques and to advance the state of the art proposes a radically new idea: *leveraging the recurrence of functionalities among different interactive applications*. The fact that often interactive applications share similarities and often implement the same functionalities can be exploited as a way of obtaining semantic information that can be used to effectively generate meaningful test cases equipped with functional oracle. In this dissertation we elaborated this unexplored opportunity and identified two ways in which it can be leveraged for semantic testing: (i) the semantics of some popular and recognizable functionalities in interactive applications, called application independent, can be predefined once and for all and then be used to automatically generate effective test cases for these functionalities in any AUT; (ii) the manually

written test cases used to test certain functionalities in existing applications can be used, after being properly adapted, to test the same functionalities in the AUT. To exploit these two ideas, we propose two novel techniques: AUGUSTO and ADAPTDROID.

AUGUSTO exploits the opportunity of AIF to generate effective test cases, and we evaluated it on seven desktop interactive applications and compared it to six state-of-the-art approaches. AUGUSTO was able to reveal five more bugs than the competing techniques, showing that it is more effective than state-of-the-art techniques in finding bugs when testing AIFs, thus confirming our research hypothesis that indeed interactive applications recurring functionalities can be exploited to achieve effective testing.

ADAPTDROID exploits the possibility of reusing manually written test cases across similar applications. We evaluated ADAPTDROID in a study with four human subjects adapting 24 test cases across similar Android apps. In our study ADAPTDROID adaptations were evaluated as high quality by the human subjects in 40% of the cases, thus showing that, albeit still limited, the approach is able to automatically adapt test cases across applications.

The two approaches presented in this thesis are different and complementary to each other. Ideally, both approaches test functionalities which can be found on many interactive applications. AUGUSTO requires the initial effort of modeling the functionalities and to be applicable requires functionalities to be implemented in a recognizable way across different applications. In these cases, AUGUSTO demonstrated to be able to generate thorough bug-revealing test suites. Instead, ADAPTDROID does not require any initial effort (test cases can be mined from open source applications) and allows the tested functionalities to differ more across different applications as only the elements involved in the original test case (and not the whole functionality) have to be recognized. However, the testing effectiveness of ADAPTDROID depends on the number and quality of the donor test cases used as input to the adaptation process. Thus, ADAPTDROID is able to detect faults in the AUT only when an appropriate donor test case that can discover that fault is available for adaptation. In general, both AUGUSTO and ADAPTDROID can be used to generate an effective initial test suite able to meaningfully test many functionalities of an interactive application under test, allowing the developers to focus most of their testing efforts on the peculiar aspects of their application.

7.1 Contributions

The main contributions of this thesis to the problem of testing interactive applications are an empirical study that compares the main state-of-the-art techniques for desktop applications and the definition of two novel approaches. In details, this thesis contributes to advancing the state of the art in automatic test case generation for interactive applications by proposing:

An empirical study that evaluates the test case generation state of the art for desktop applications. We presented the results of a study in which we compare the main state-of-the-art techniques for testing desktop application by applying them on a set of common subject applications. The results of the study provide new compelling evidence showing that current techniques still have limited effectiveness in thoroughly covering the execution space of the AUT and in detecting faults. The results of this study were presented at the 2018 International Workshop on User Interface Test Automation and Testing Techniques for Event Based Software [88].

A novel approach to generate test cases for Application Independent Functionalities. We propose AUGUSTO, the first automated technique that can generate meaningful test cases equipped with functional oracles. AUGUSTO differs from the other approaches in the literature because it targets application independent functionalities (AIF), encoded with high-level models of their expected semantics, to guide the generation of test cases towards meaningful executions of the AUT, and to add fault revealing assertions. We empirically evaluated the effectiveness of the approach modeling three AIFs and running the approach on seven applications. AUGUSTO was able to detect seven failures in the subject applications, five of which required a functional oracle to be detected. We also compare AUGUSTO to the main state-of-the-art techniques showing its superior fault-revealing ability when testing AIFs. AUGUSTO has been presented at the 2016 International Conference on Software Testing, Verification and Validation doctoral symposium [119] and at the 2018 International Conference on Software Engineering [73].

A novel approach to generate effective test cases by adapting and reusing tests manually written for similar applications. We propose ADAPTDROID, an automatic approach able to generate meaningful test cases equipped with oracles by adapting existing test cases of applications similar to the AUT instead of generating test cases from scratch. ADAPTDROID uses a novel concept of event semantic matching across applications based on word embeddings to evaluate the similarity of a given test with respect to the donor one and then relies on a search-based algorithm to find the most similar test case for the AUT. We evaluated

ADAPTDROID adapting a set of eight manually written test cases on 24 similar applications and then asking the authors of the test cases to evaluate ADAPTDROID adaptations. The evaluation showed promising results as ADAPTDROID adaptations were evaluated as good in 40% of the cases. ADAPTDROID approach and evaluation has been submitted to the 2019 Symposium on the Foundations of Software Engineering and at the moment of writing is under review [120].

7.2 Open Research Directions

The results of this thesis open new research directions towards the automatic generation of effective test case for interactive applications.

Automatic modeling of AIFs. AUGUSTO approach showed to be very effective given some manually pre-defined AIF models. In this thesis, modeling AIFs in a way that is abstracted enough to be usable across many different applications was done manually. Automatizing this process, for instance, mining applications in software repositories and detecting similarities, could greatly increase AUGUSTO applicability and create the space for a wide range of techniques that exploit AIF existence for different tasks and not only testing.

Cross-platform test case adaptation. ADAPTDROID was the first approach that showed that test case adaptation across similar applications is feasible and it demonstrates it with an empirical study on Android applications. Test case adaptation could be pushed even further, adapting test cases across different platform. For instance, adapting a test case for an Android app to a similar IOS application. Thus, enhancing ADAPTDROID to support different platforms allowing cross-platform adaptation is a possible research direction that would extend the applicability and effectiveness of test case adaptation.

Appendices

Appendix A

AIF Models

This section reports the full AIF model that were used for AUGUSTO evaluation (see Chapter 5.4).

The following listing presents the Abstract Semantics model preamble, i.e., the initial part of the model that is shared by the Abstract Semantics of all AIFs.

```
1  open util/ordering [Time] as T
2  open util/ternary
3  open util/relation
4  ----------------------Utils-----------------------
5  sig Time { }
6  abstract sig Operation { }
7  sig Click extends Operation {
8        clicked: one Actionwidget
9  }
10 sig Fill extends Operation {
11       filled: one Inputwidget,
12       with: lone Value
13 }
14 sig Select extends Operation {
15       wid: one Selectablewidget,
16       which: one Object
17 }
18 one sig Track {
19       op: Operation lone -> Time
20 }
21 pred transition [t, t': Time] {
22       (one aw: Actionwidget, c: Click | click [aw, t, t', c]) or
23       (one iw: Inputwidget, v: Value, f: Fill| fill [iw, t, t', v, f]) or
24       (one iw: Inputwidget, f: Fill| fill [iw, t, t', none, f]) or
25       (one sw: Selectablewidget, s: Select, o: Object | select [sw, t, t',
             o, s])
```

```
26 }
27 pred System {
28       init [T/first]
29       ∀t: Time - T/last | transition [t, T/next[t]]
30 }
31 ----------------Generic GUI Structure ----------------
32 abstract sig Window {
33       aws: set Actionwidget,
34       iws: set Inputwidget,
35       sws: set Selectablewidget
36 }
37 abstract sig Actionwidget {
38       goes: set Window
39 }
40 sig Value { }
41 one sig Optionvalue0 extends Value{ }
42 one sig Optionvalue1 extends Value{ }
43 one sig Optionvalue2 extends Value{ }
44 one sig Optionvalue3 extends Value{ }
45 one sig Optionvalue4 extends Value{ }
46 sig Tobecleaned extends Value{ }
47 abstract sig Inputwidget {
48       content: Value lone -> Time,
49       val: set Value
50 }
51 sig Object {
52       appeared: one Time
53 }
54 abstract sig Selectablewidget {
55       list: Object set -> Time,
56       selected: Object lone ->Time
57 }
58 fact {
59       \#Tobecleaned =1 ⟹ not(Tobecleaned in Fill.with)
60       ∀iw: Inputwidget | iw in Window.iws
61       ∀aw: Actionwidget | aw in Window.aws
62       ∀sw: Selectablewidget | sw in Window.sws
63 }
64 fact noredundant{
65       no t: Time | \#Track.op.t =1 and Track.op.t in Fill and Track.op.t.
             with =Track.op.t.filled.content.(T/prev[t])

66       no t: Time | \#Track.op.t =1 and Track.op.t in Click and Track.op.(T
             /prev[t]) in Click and Track.op.t.clicked =Track.op.(T/prev[t]).
             clicked
```

```
67        no t: Time | \#Track.op.t =1 and Track.op.t in Select and Track.op.(
              T/prev[t]) in Select and Track.op.(T/prev[t]).wid =Track.op.t.wid

68        no t: Time | \#Track.op.t =1 and Track.op.t in Fill and Track.op.(T/
              prev[t]) in Fill and Track.op.t.filled =Track.op.(T/prev[t]).
              filled

69  }
70  ----------------Generic GUI Semantics --------------
71  one sig Currentwindow {
72        isin: Window one -> Time
73  }
74  pred click [aw: Actionwidget, t, t': Time, c: Click] {
75        --- precondition ---
76        aw in Currentwindow.isin.t.aws
77        clickpre [aw, t]
78        --- effect ---
79        (clicksemantics [aw, t] and clicksuccesspost [aw, t, t']) or
80        (not clicksemantics [aw, t] and Currentwindow.isin.t' =Currentwindow.
              isin.t and clickfailpost [aw, t, t'])

81        --- operation is tracked ---
82        c.clicked =aw and Track.op.t' =c
83  }
84  pred fill [iw: Inputwidget, t, t': Time, v: Value, f: Fill] {
85        --- precondition ---
86        (v =none) ⟹ not(iw.content.t =Tobecleaned)
87        iw in Currentwindow.isin.t.iws
88        fillpre [iw, t, v]
89        --- effect ---
90        (fillsemantics [iw, t, v] and iw.content.t' =v and fillsuccesspost [
              iw, t, t', v]) or

91        (not fillsemantics [iw, t, v] and iw.content.t' =iw.content.t and
              fillfailpost [iw, t, t', v])

92        --- general postcondition ---
93        Currentwindow.isin.t' =Currentwindow.isin.t
94        ∀iww: (Inputwidget - iw) | iww.content.t' =iww.content.t
95        ∀sw: Selectablewidget | sw.selected.t' =sw.selected.t and sw.list.t'
              =sw.list.t

96        --- operation is tracked ---
97        f.filled =iw and f.with =v and Track.op.t' =f
98  }
```

```
99  pred select [sw: Selectablewidget, t, t': Time, o: Object, s: Select] {
100       --- precondition ---
101       sw in Currentwindow.isin.t.sws
102       o in sw.list.t
103       selectpre [sw, t, o]
104       --- effect ---
105       (selectsemantics [sw, t, o] and sw.selected.t' =o and
              selectsuccesspost [sw, t, t', o]) or

106       (not selectsemantics [sw, t, o] and sw.selected.t' =sw.selected.t and
               selectfailpost [sw, t, t', o])

107       --- general postcondition ---
108       Currentwindow.isin.t' =Currentwindow.isin.t
109       ∀sww: (Selectablewidget - sw) | sww.selected.t' =sww.selected.t and
              sww.list.t' =sww.list.t

110       sw.list.t' =sw.list.t
111       --- operation is tracked ---
112       s.wid =sw and s.which =o and Track.op.t' =s
113 }
```

Listing A.1. Abstract Semantics model preamble

A.1 AUTH

This section reports the GUI Pattern model and Abstract Semantics model of the *authentication* AIF.

```
<pattern alloy="AUTH.als" name="AUTH">
   <window id="pre" dynamic="false" card="lone" alloy="Pre">
       <action_widget id="paw10" card="one" alloy="Go">
           <label>.*(login|sign in).*$</label>
       </action_widget>
   </window>
   <window id="initial" dynamic="false" card="one" alloy="Initial">
       <action_widget id="paw1" card="one" alloy="Login">
           <label>^(login|enter|go).*$</label>
       </action_widget>
       <action_widget id="paw2" card="one" alloy="Signup">
           <label>^(register|signup|sign up|accounts).*$</label>
       </action_widget>
       <input_widget id="piw1" card="one" alloy="User">
           <label>^(user|username|email).*</label>
       </input_widget>
       <input_widget id="piw2" card="one" alloy="Password">
           <label>^(pass|password).*</label>
       </input_widget>
   </window>
   <window id="signup" card="one" dynamic="false" alloy="Signup">
       <action_widget id="paw3" card="one" alloy="Ok">
           <label>^(ok|save|record|signup|sign up|create account)</label>
       </action_widget>
       <action_widget id="paw4" card="one" alloy="Cancel">
           <label>^(cancel|clear|back|close)</label>
       </action_widget>
       <input_widget id="piw3" card="one" alloy="User_save">
           <label>^(user|username|email).*</label>
       </input_widget>
       <input_widget id="piw4" card="one" alloy="Password_save">
           <label>^(?!re-enter|repeat|confirm)(pass|password).*</label>
       </input_widget>
       <input_widget id="piw5" card="one" alloy="Re_password">
           <label>^(repeat|re-enter|confirm).*</label>
       </input_widget>
       <input_widget id="piw6" card="set" alloy="Field">
           <label>.*</label>
       </input_widget>
   </window>
   <window id="logged" card="one" dynamic="true" alloy="Logged">
       <action_widget id="paw5" card="one" alloy="Logout">
           <label>^.*(logout|exit|sign out|signout|log out)$</label>
       </action_widget>
   </window>
   <edge type="static"><from>paw10</from><to>initial</to></edge>
   <edge type="dynamic"><from>paw1</from><to>logged</to></edge>
   <edge type="static"><from>paw2</from><to>signup</to></edge>
   <edge type="static"><from>paw4</from><to>initial</to></edge>
   <edge type="dynamic"><from>paw3</from><to>pre</to><to>initial</to></edge>
   <edge type="static"><from>paw5</from><to>initial</to><to>pre</to></edge>
</pattern>
```

Figure A.1. AUTH GUI Pattern model

```
1  --------------------Initial State--------------
2  pred init [t: Time] {
3        no Track.op.t
4        Currentwindow.isin.t =aws.Login
5        \#List.elements.t =0
6  }
7  ---------------Generic AUTH Structure ----------
8  abstract sig Go, Login, Signup, Ok, Cancel, Logout extends Actionwidget { }

9  abstract sig User, Password, Usersave, Passwordsave, Repassword, Field
       extends Inputwidget { }


11 fact {
12 \#Tobecleaned=1
13       not(User in Propertyrequired.requireds)
14       not(Password in Propertyrequired.requireds)
15       not(Usersave in Propertyrequired.requireds)
16       not(Passwordsave in Propertyrequired.requireds)
17       not(Repassword in Propertyrequired.requireds)
18 }
19 ---------------Generic AUTH Semantics----------
20 one sig Propertyrequired{
21       requireds: set Inputwidget
22 }
23 sig Objectinlist extends Object{
24       vs: Value lone -> Inputwidget
25 }
26 one sig List {
27       elements: Objectinlist set -> Time
28 }
29 pred fillsemantics [iw: Inputwidget, t: Time, v: Value] { }
30 pred fillsuccesspost [iw: Inputwidget, t, t': Time, v: Value] {
31             List.elements.t' =List.elements.t
32 }
33 pred fillfailpost [iw: Inputwidget, t, t': Time, v: Value] {
34             List.elements.t' =List.elements.t
35 }
36 pred fillpre[iw: Inputwidget, t: Time, v: Value] {
37       \#iw.content.(T/first) =1 ⟹ not(v =none)
38 }
39 pred selectsemantics [sw: Selectablewidget, t: Time, o: Object] { }
40 pred selectsuccesspost [sw: Selectablewidget, t, t': Time, o: Object] { }

41 pred selectfailpost [sw: Selectablewidget, t, t': Time, o: Object] { }
42 pred selectpre[sw: Selectablewidget, t: Time, o: Object] { }
```

```
44 pred clicksemantics [aw: Actionwidget, t: Time] {
45       (aw in Login) ⟹ filledlogintest [t] and existingtest [t]
46       (aw in Ok) ⟹ filledrequiredtest[t] and uniquefieldstest [t] and
             samepasstest [t]

47 }
48 pred clicksuccesspost [aw: Actionwidget, t, t': Time] {
49       Currentwindow.isin.t' =aw.goes
50       (aw in Ok) ⟹ add [t, t'] else List.elements.t' =List.elements.t
51       (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
52 }
53 pred clickfailpost [aw: Actionwidget, t, t': Time] {
54       (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
55       List.elements.t' =List.elements.t
56 }
57 pred clickpre[aw: Actionwidget, t: Time] { }
58 pred add [t, t': Time] {
59       one o: Objectinlist |∀ iw: (Usersave + Passwordsave + Field) | not(o
              in List.elements.t) and o.appeared =t' and o.vs.iw =iw.content.
             t and List.elements.t' =List.elements.t+o

60 }
61 pred filledlogintest [t: Time] {
62       ∀iw: (User+Password)| \#iw.content.t =1 and not(iw.content.t=
             Tobecleaned)

63 }
64 pred existingtest [t: Time] {
65       one o: List.elements.t | Password.content.t =o.vs.Passwordsave and
             User.content.t =o.vs.Usersave

66 }
67 pred samepasstest [t: Time] {
68       Passwordsave.content.t =Repassword.content.t
69 }
70 pred filledrequiredtest [t: Time] {
71       \#Usersave.content.t =1 and not(Usersave.content.t=Tobecleaned)
72       \#Passwordsave.content.t =1 and not(Passwordsave.content.t=
             Tobecleaned)

73       \#Repassword.content.t =1 and not(Repassword.content.t=Tobecleaned)

74       ∀iw: Field| (iw in Propertyrequired.requireds) ⟹ \#iw.content.t =1
              and not(iw.content.t=Tobecleaned)
```

```
75 }
76 pred uniquefieldstest [t: Time] {
77       ∀o: List.elements.t | (\#o.vs.Usersave= 1 ⟹ Usersave.content.t ≠o.
             vs.Usersave)

78 }
```

Listing A.2. AUTH Abstract Semantics model

A.2 CRUD

This section reports the GUI Pattern model and Abstract Semantics model of the *CRUD* AIF.

```
<pattern alloy="CRUD.als" name="CRUD">
    <window id="initial" dynamic="false" card="one" alloy="Initial">
        <action_widget id="paw1" card="set" alloy="Create_trigger">
            <label>^(?!window - )(.*( new | add | create ).*|^(new (?!-)|add |create ).*|^(new|add|create)$)</label>
        </action_widget>
        <action_widget id="paw3" card="set" alloy="Update_trigger">
            <label>^(?!window - )(.*( edit | update | modify ).*|^(edit (?!-)|update |modify ).*|^(edit|update|modify)$)</label>
        </action_widget>
        <action_widget id="paw4" card="set" alloy="Delete_trigger">
            <label>^(?!window - )(.*( delete | remove ).*|^(delete |remove ).*|^(delete|remove)$)</label>
        </action_widget>
        <selectable_widget id="psw1" card="one" alloy="">
            <label>.*</label>
        </selectable_widget>
    </window>
    <window id="form" card="one" dynamic="false" alloy="Form">
        <action_widget id="paw5" card="one" alloy="Ok">
            <label>^(ok|save|record)</label>
        </action_widget>
        <action_widget id="paw6" card="one" alloy="Cancel">
            <label>^(cancel|clear)</label>
        </action_widget>
        <input_widget id="piw1" card="some" alloy="">
            <label>.*</label>
        </input_widget>
    </window>
    <edge type="static"><from>paw1</from><to>form</to></edge>
    <edge type="static"><from>paw6</from><to>initial</to></edge>
    <edge type="dynamic"><from>paw5</from><to>initial</to></edge>
    <edge type="dynamic"><from>paw3</from><to>form</to></edge>
    <edge type="dynamic"><from>paw4</from><to>initial</to></edge>
</pattern>
```

Figure A.2. CRUD GUI Pattern model

```
1  ---------------------Initial State---------------
2  pred init [t: Time] {
3        no Selectablewidget.list.t
4        no Track.op.t
5        no Selectablewidget.selected.t
6        Currentwindow.isin.t =sws.Selectablewidget
7        \#Createtrigger =0 ⟹ Currentcrudop.operation.t =CREATE else \#
             Currentcrudop.operation.t =0

8        \#Tobecleaned =1
9        ∀iw: Inputwidget | \#iw.content.(T/first) > 0
10 }
11 ---------------Generic CRUD Structure ----------
12 abstract sig Ok, Cancel extends Actionwidget { }
13 abstract sig Createtrigger extends Actionwidget { }
14 abstract sig Updatetrigger extends Actionwidget { }
```

```
15 abstract sig Deletetrigger extends Actionwidget { }
16 fact {
17       \#Ok  2#Selectablewidget = 1all iw:  Inputwidget |
              not(iw.content.(T/first) = Tobecleaned)
              = not(iw in Propertysemantic.requireds)
18 }
19 ---------------Generic CRUD Semantics----------
20 abstract sig Crudop {}
21 one sig CREATE, UPDATE extends Crudop {}
22 one sig Currentcrudop {
23       operation: Crudop lone -> Time
24 }
25 one sig Propertysemantic{
26       uniques: set Inputwidget,
27       requireds: set Inputwidget
28 }
29 sig Objectinlist extends Object{
30       vs: Value lone ->Inputwidget
31 }
32 pred fillsemantics [iw: Inputwidget, t: Time, v: Value] { }
33 pred fillsuccesspost [iw: Inputwidget, t, t': Time, v: Value] {
34       Currentcrudop.operation.t' =Currentcrudop.operation.t
35 }
36 pred fillfailpost [iw: Inputwidget, t, t': Time, v: Value] {
37       Currentcrudop.operation.t' =Currentcrudop.operation.t
38 }
39 pred fillpre[iw: Inputwidget, t: Time, v: Value] {
40       (\#iw.content.(T/first) =1 and not(iw.content.(T/first) =Tobecleaned
              )) ⟹ not(v =none)

41 }
42 pred selectsemantics [sw: Selectablewidget, t: Time, o: Object] { }
43 pred selectfailpost [sw: Selectablewidget, t, t': Time, o: Object] {
44       Currentcrudop.operation.t' =Currentcrudop.operation.t
45       ∀iw: Inputwidget | iw.content.t' =iw.content.t
46 }
47 pred selectsuccesspost [sw: Selectablewidget, t, t': Time, o: Object] {
48       \#Createtrigger =0 ⟹ (Currentcrudop.operation.t' =UPDATE and
              loadform[o, t']) else (\#Currentcrudop.operation.t' =0 and ∀iw:
              Inputwidget | iw.content.t' =iw.content.t)

49 }
50 pred selectpre[sw: Selectablewidget, t: Time, o: Object] {
51       o in sw.list.t
52 }
53 pred clicksemantics [aw: Actionwidget, t: Time] {
```

```
54      (aw in Ok and Currentcrudop.operation.t in CREATE) ⟹
            filledrequiredtest [t] and uniquetest [t]

55      (aw in Ok and Currentcrudop.operation.t in UPDATE) ⟹
            filledrequiredtest [t] and uniqueforupdatetest [t]

56      (aw in Deletetrigger) ⟹ 2=(1+1)
57  }
58  pred clicksuccesspost [aw: Actionwidget, t, t': Time] {
59      Currentwindow.isin.t' =aw.goes
60      (aw in Createtrigger) ⟹ (Currentcrudop.operation.t' =CREATE and
            Selectablewidget.list.t' =Selectablewidget.list.t and (∀ iw:
            Inputwidget | iw.content.t' =iw.content.(T/first)) and \#
            Selectablewidget.selected.t' =0)

61      (aw in Updatetrigger) ⟹ (Currentcrudop.operation.t' =UPDATE and
            Selectablewidget.list.t' =Selectablewidget.list.t and loadform[
            Selectablewidget.selected.t, t'] and Selectablewidget.selected.t'
             =Selectablewidget.selected.t)

62      (aw in Deletetrigger) ⟹ (\#Selectablewidget.selected.t' =0 and
            delete [t, t'])

63      (aw in Deletetrigger and \#Createtrigger =0) ⟹ (Currentcrudop.
            operation.t' =CREATE and (∀ iw: Inputwidget | iw.content.t' =iw.
            content.(T/first)))

64      (aw in Deletetrigger and \#Createtrigger > 0) ⟹ (\#Currentcrudop.
            operation.t' =0 and (∀ iw: Inputwidget | iw.content.t' =iw.
            content.(T/first)))

65      (aw in Cancel and \#Createtrigger > 0) ⟹ (\#Currentcrudop.operation
            .t' =0 and Selectablewidget.list.t' =Selectablewidget.list.t and
             \#Selectablewidget.selected.t' =0)

66      (aw in Cancel and \#Createtrigger =0) ⟹ (Currentcrudop.operation.t'
             =CREATE and Selectablewidget.list.t' =Selectablewidget.list.t
            and \#Selectablewidget.selected.t' =0 and (∀ iw: Inputwidget |
            iw.content.t' =iw.content.(T/first)))


68      (aw in Ok and Currentcrudop.operation.t in CREATE) ⟹ (\#
            Selectablewidget.selected.t' =0 and add [t, t'])

69      (aw in Ok and Currentcrudop.operation.t in UPDATE) ⟹ (\#
            Selectablewidget.selected.t' =0 and update [t, t'])
```

```
70      (aw in Ok and \#Createtrigger > 0) ⟹ (\#Currentcrudop.operation.t'
            =0)

71      (aw in Ok and \#Createtrigger =0) ⟹ (Currentcrudop.operation.t' =
            CREATE and (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/
            first)) and \#Selectablewidget.selected.t' =0)

72  }
73  pred clickfailpost [aw: Actionwidget, t, t': Time] {
74      Selectablewidget.list.t' =Selectablewidget.list.t
75      (∀ iw: Inputwidget | iw.content.t' =iw.content.t)
76      Selectablewidget.selected.t' =Selectablewidget.selected.t
77      Currentcrudop.operation.t' =Currentcrudop.operation.t
78  }
79  pred clickpre[aw: Actionwidget, t: Time] {
80      (aw in Updatetrigger) ⟹ \#Selectablewidget.selected.t =1
81      (aw in Deletetrigger) ⟹ \#Selectablewidget.selected.t =1
82  }
83  pred add [t, t': Time] {
84      one o: Objectinlist |∀ iw: Inputwidget | not(o in Selectablewidget.
            list.t) and o.appeared =t' and (iw.content.t =Tobecleaned ⟹ \#
            o.vs.iw =0 else o.vs.iw =iw.content.t) and Selectablewidget.list.
            t' =Selectablewidget.list.t+o

85  }
86  pred filledrequiredtest [t: Time] {
87      ∀iw: Propertysemantic.requireds| not(iw.content.t =Tobecleaned) and
            not( \\#iw.content.t =0)

88  }
89  pred uniquetest [t: Time] {
90      ∀iw: Propertysemantic.uniques | ∀o: Selectablewidget.list.t | (\#(o.
            vs.iw)=1) ⟹ iw.content.t ≠o.vs.iw

91  }
92  pred uniqueforupdatetest [t: Time] {
93      ∀iw: Propertysemantic.uniques | ∀o: (Selectablewidget.list.t-
            Selectablewidget.selected.t) | (\#(o.vs.iw)=1) ⟹ iw.content.t
            ≠o.vs.iw

94  }
95  pred loadform [o: Object, t': Time] {
96      ∀iw: Inputwidget | iw.content.t' =o.vs.iw
97  }
98  pred update [t, t': Time] {
```

```
99        one o: Object | ∀iw: Inputwidget | not(o in Selectablewidget.list.t)
              and o.appeared =Selectablewidget.selected.t.appeared and (iw.
              content.t =Tobecleaned ⟹ \#o.vs.iw =0 else o.vs.iw =iw.content
              .t) and Selectablewidget.list.t' =(Selectablewidget.list.t -
              Selectablewidget.selected.t)+o

100   }
101   pred delete [t, t': Time] {
102       Selectablewidget.list.t' =Selectablewidget.list.t - Selectablewidget.
              selected.t

103   }
```

Listing A.3. CRUD Abstract Semantics model

A.3 SAVE

This section reports the GUI Pattern model and Abstract Semantics model of the *save* AIF.

```
<pattern alloy="SAVE.als" name="SAVE">
    <window id="initial" dynamic="false" card="one" alloy="Initial">
        <action_widget id="paw1" card="one" alloy="New">
            <label>^.*- new.*</label>
        </action_widget>
        <action_widget id="paw2" card="one" alloy="Open">
            <label>^.*- open.*</label>
        </action_widget>
        <action_widget id="paw3" card="lone" alloy="Save">
            <label>^.*- save(?! all).*</label>
        </action_widget>
        <action_widget id="paw4" card="lone" alloy="Saveas">
            <label>^.*- save as.*</label>
        </action_widget>
    </window>
    <window id="saving" card="one" dynamic="false" alloy="Saving">
        <action_widget id="paw6" card="one" alloy="Saves">
            <label>save</label>
        </action_widget>
        <action_widget id="paw7" card="one" alloy="Cancelsave">
            <label>(cancel|back)</label>
        </action_widget>
        <input_widget id="piw1" card="one" alloy="Filename">
            <label>.*name.*</label>
        </input_widget>
    </window>
    <window id="opening" card="one" dynamic="false" alloy="Opening">
        <action_widget id="paw8" card="one" alloy="Openo">
            <label>open</label>
        </action_widget>
        <action_widget id="paw9" card="one" alloy="Cancelopen">
            <label>(cancel|back)</label>
        </action_widget>
        <selectable_widget id="psw1" card="one" alloy="Opening_list">
            <label>.*</label>
        </selectable_widget>
    </window>
    <window id="encrypt" card="lone" dynamic="true" alloy="Encrypt">
        <action_widget id="paw10" card="one" alloy="Encryptb">
            <label>^(ok|encrypt)</label>
        </action_widget>
        <action_widget id="paw11" card="one" alloy="Backe">
            <label>^(no|cancel)</label>
        </action_widget>
        <input_widget id="piw2" card="one" alloy="Password">
            <label>password.*</label>
        </input_widget>
        <input_widget id="piw3" card="one" alloy="Repassword">
            <label>.*(repeat password|confirm password|confirmation).*</label>
        </input_widget>
    </window>
```

```
    <window id="decrypt" card="lone" dynamic="true" alloy="Decrypt">
        <action_widget id="paw12" card="one" alloy="Decryptb">
            <label>^(ok|decrypt)</label>
        </action_widget>
        <action_widget id="paw13" card="one" alloy="Backd">
            <label>^(no|cancel)</label>
        </action_widget>
        <input_widget id="piw4" card="one" alloy="Depassword">
            <label>password.*</label>
        </input_widget>
    </window>
    <window id="choice" card="lone" dynamic="true" alloy="Choice">
        <action_widget id="paw14" card="one" alloy="Yes">
            <label>^(ok|yes|encrypt)</label>
        </action_widget>
        <action_widget id="paw15" card="one" alloy="No">
            <label>^(no|cancel)</label>
        </action_widget>
    </window>
    <window id="replace" card="lone" dynamic="true" alloy="Replacedialog">
        <title>.*(replace|exists|overwrite).*</title>
        <action_widget id="paw16" card="one" alloy="Replace">
            <label>.*(replace|yes|ok).*</label>
        </action_widget>
        <action_widget id="paw17" card="one" alloy="Noreplace">
            <label>^(no|cancel)</label>
        </action_widget>
    </window>
    <edge type="static"><from>paw1</from><to>initial</to><to>saving</to></edge>
    <edge type="static"><from>paw2</from><to>opening</to></edge>
    <edge type="dynamic"><from>paw3</from><to>saving</to></edge>
    <edge type="static"><from>paw4</from><to>saving</to></edge>
    <edge type="dynamic"><from>paw6</from><to>initial</to><to>replace</to><to>choice</to><to>encrypt</to></edge>
    <edge type="static"><from>paw7</from><to>initial</to></edge>
    <edge type="dynamic"><from>paw8</from><to>initial</to><to>decrypt</to></edge>
    <edge type="static"><from>paw9</from><to>initial</to></edge>
    <edge type="dynamic"><from>paw10</from><to>initial</to></edge>
    <edge type="static"><from>paw11</from><to>saving</to><to>initial</to></edge>
    <edge type="dynamic"><from>paw12</from><to>initial</to></edge>
    <edge type="static"><from>paw13</from><to>opening</to><to>initial</to></edge>
    <edge type="static"><from>paw14</from><to>encrypt</to></edge><edge type="static"><from>paw15</from><to>initial</to></edge>
    <edge type="static"><from>paw16</from><to>initial</to><to>choice</to><to>encrypt</to></edge>
    <edge type="static"><from>paw17</from><to>initial</to><to>saving</to></edge>
</pattern>
```

SAVE GUI Pattern model

```
1  --------------------Initial State---------------
2  pred init [t: Time] {
3        no Track.op.t
4        Currentwindow.isin.t =aws.New
5        \#Openinglist.list.t =0
6        \#(Auxiliary.saved.t) =1
7        no Selectablewidget.selected.t
8        \#Tobecleaned =1
9        ∀iw: Inputwidget | \#iw.content.(T/first) > 0
10 }
11 ---------------Generic SAVE Structure ----------
12 abstract sig New, Open, Save, Saveas, Saves, Cancelsave, Openo, Cancelopen,
       Encryptb, Backe, Decryptb, Backd, Yes, No, Replace, Noreplace extends
      Actionwidget { }

13 abstract sig Filename, Password, Repassword, Depassword extends Inputwidget
       { }

14 abstract sig Openinglist extends Selectablewidget { }

16 ---------------Generic SAVE Semantics----------
17 one sig Auxiliary{
18       pwd: Object lone -> Value,
19       haspwd: one Object,
20       names: Object one -> Value,
21       saved: Value lone -> Time
22 }
23 pred fillsemantics [iw: Inputwidget, t: Time, v: Value] {

25 }
26 pred fillsuccesspost [iw: Inputwidget, t, t': Time, v: Value] {
27       Openinglist.list.t' =Openinglist.list.t
28       (Auxiliary.saved.t') = (Auxiliary.saved.t)
29 }
30 pred fillfailpost [iw: Inputwidget, t, t': Time, v: Value] {
31       Openinglist.list.t' =Openinglist.list.t
32       (Auxiliary.saved.t') = (Auxiliary.saved.t)
33 }
34 pred fillpre [iw: Inputwidget, t: Time, v: Value] { }
35 pred selectsemantics [sw: Selectablewidget, t: Time, o: Object] {

37 }
38 pred selectsuccesspost [sw: Selectablewidget, t, t': Time, o: Object] {
39       Openinglist.list.t' =Openinglist.list.t
40       (Auxiliary.saved.t') = (Auxiliary.saved.t)
41       (∀ iw: Inputwidget | iw.content.t' =iw.content.t)
```

```
}
pred selectfailpost [sw: Selectablewidget, t, t': Time, o: Object] {
      Openinglist.list.t' =Openinglist.list.t
      (Auxiliary.saved.t') = (Auxiliary.saved.t)
      (∀ iw: Inputwidget | iw.content.t' =iw.content.t)
}
pred selectpre [sw: Selectablewidget, t: Time, o: Object] { }
pred clicksemantics [aw: Actionwidget, t: Time] {
      (aw in Save) ⟹ \#(Auxiliary.saved.t) =0
      (aw in Openo) ⟹ \#Openinglist.selected.t =1
      (aw in Saves) ⟹ \#Filename.content.t =1 and not(Filename.content.t
          =Tobecleaned)

      (aw in Encryptb) ⟹ Password.content.t =Repassword.content.t
      (aw in Decryptb) ⟹ Depassword.content.t =(Openinglist.selected.t).(
          Auxiliary.pwd)

}
pred clicksuccesspost [aw: Actionwidget, t, t': Time] {
      (aw in New and \#aw.goes =0) ⟹ new[t,t']
      (aw in New and \#aw.goes > 0) ⟹ same[t,t'] and Currentwindow.isin.t
          ' =aw.goes

      (aw in Open) ⟹ same[t,t'] and Currentwindow.isin.t' =aw.goes
      (aw in Save) ⟹ same[t,t'] and Currentwindow.isin.t' =aw.goes
      (aw in Saveas) ⟹ same[t,t'] and Currentwindow.isin.t' =aw.goes
      (aw in Saves and exisit[t, Filename.content.t]) ⟹ ((\#Replace =1)
          ⟹ (Currentwindow.isin.t' =aws.Replace and same[t,t']) else ((\#
          Encryptb =1 or \#Yes =1) ⟹ (same[t,t'] and (\#Yes =1 ⟹
          Currentwindow.isin.t' =aws.Yes else Currentwindow.isin.t' =aws.
          Encryptb)) else (savenp[t,t', Filename.content.t])))

      (aw in Saves and not(exisit[t, Filename.content.t])) ⟹ ((\#Encryptb
          =1 or \#Yes =1) ⟹ (same[t,t'] and (\#Yes =1 ⟹
          Currentwindow.isin.t' =aws.Yes else Currentwindow.isin.t' =aws.
          Encryptb)) else (savenp[t,t', Filename.content.t]))

      (aw in Cancelsave) ⟹ returned[t, t']
      (aw in Openo) ⟹ ((Openinglist.selected.t) in (Auxiliary.haspwd))
          ⟹ (Currentwindow.isin.t' =aws.Decryptb and same[t,t']) else (
          openo[t,t'])

      (aw in Cancelopen) ⟹ returned[t, t']
      (aw in Encryptb) ⟹ save[t,t', Password.content.t, Filename.content.
          t]
```

```
68      (aw in Backe) ⟹ (\#Yes =1) ⟹ savenp[t,t', Filename.content.t]
            else((aw.goes in aws.New) ⟹ returned[t,t'] else (same2[t,t']
            and Currentwindow.isin.t' =aw.goes))

69      (aw in Decryptb) ⟹ openo[t, t']
70      (aw in Backd) ⟹ ((aw.goes in aws.New) ⟹ returned[t,t'] else (
            same2[t,t'] and Currentwindow.isin.t' =aw.goes))

71      (aw in Yes) ⟹ same[t,t'] and Currentwindow.isin.t' =aw.goes
72      (aw in No) ⟹ savenp[t,t', Filename.content.t]
73      (aw in Replace) ⟹ ((\#Encryptb =1 or \#Yes =1) ⟹ (same[t,t'] and
             (\#Yes =1 ⟹ Currentwindow.isin.t' =aws.Yes else Currentwindow.
            isin.t' =aws.Encryptb)) else (savenp[t,t', Filename.content.t]))

74      (aw in Noreplace) ⟹ ((aw.goes in aws.New) ⟹ returned[t,t'] else (
            same2[t,t'] and Currentwindow.isin.t' =aw.goes))

75  }
76  pred clickfailpost [aw: Actionwidget, t, t': Time] {
77      (aw in (Encryptb+Decryptb)) ⟹ (∀ iw: Inputwidget | iw.content.t' =
            iw.content.(T/first)) else (∀ iw: Inputwidget | iw.content.t' =
            iw.content.t)

78      Openinglist.list.t' =Openinglist.list.t
79      Openinglist.selected.t' =Openinglist.selected.t
80      (Auxiliary.saved.t') = (Auxiliary.saved.t)
81  }
82  pred clickpre [aw: Actionwidget, t': Time] { }
83  pred save [t, t': Time, password, filename: Value] {
84      (filename in (Openinglist.list.t).(Auxiliary.names)) ⟹ (one o:
            Object | o in Auxiliary.haspwd and not(o in Openinglist.list.t)
            and o.appeared =Auxiliary.names.filename.appeared and (password
            =Tobecleaned ⟹ \#o.(Auxiliary.pwd)=0 else o.(Auxiliary.pwd) =
            password) and o.(Auxiliary.names) =filename and Openinglist.list.
            t' =(Openinglist.list.t - (Auxiliary.names).filename)+o) else (
            one o: Object | o in Auxiliary.haspwd and not(o in Openinglist.
            list.t) and o.(Auxiliary.pwd) =password and o.(Auxiliary.names)
            =filename and o.appeared =t' and Openinglist.list.t' =
            Openinglist.list.t + o)

85      \#(Auxiliary.saved.t') =1
86      Currentwindow.isin.t' =aws.New
87      \#Openinglist.selected.t' =0
88    (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
89  }
90  pred savenp [t, t': Time, filename: Value] {
```

```
91        (filename in (Openinglist.list.t).(Auxiliary.names)) ⟹ (one o:
              Object | not(o in Auxiliary.haspwd) and not(o in Openinglist.list.
              t) and o.appeared =Auxiliary.names.filename.appeared and o.(
              Auxiliary.pwd) =none and o.(Auxiliary.names) =filename and
              Openinglist.list.t' =(Openinglist.list.t - (Auxiliary.names).
              filename)+o) else (one o: Object | not(o in Auxiliary.haspwd) and
               not(o in Openinglist.list.t) and o.(Auxiliary.pwd) =none and o.(
              Auxiliary.names) =filename and o.appeared =t' and Openinglist.
              list.t' =Openinglist.list.t + o)

92        \#(Auxiliary.saved.t') =1
93        Currentwindow.isin.t' =aws.New
94        \#Openinglist.selected.t' =0
95        (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
96  }
97  pred new [t, t': Time] {
98        \#(Auxiliary.saved.t') =0
99        (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
100       Openinglist.list.t' =Openinglist.list.t
101       \#Openinglist.selected.t' =0
102       Currentwindow.isin.t' =aws.New
103 }
104 pred openo [t, t': Time] {
105       \#(Auxiliary.saved.t') =1
106       (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
107       Openinglist.list.t' =Openinglist.list.t
108       Currentwindow.isin.t' =aws.New
109       \#Openinglist.selected.t' =0
110 }
111 pred exisit [t: Time, name: Value] {
112       name in (Openinglist.list.t).(Auxiliary.names)
113 }
114 pred returned [t, t': Time] {
115       (Auxiliary.saved.t') = (Auxiliary.saved.t)
116       (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
117       \#Openinglist.selected.t' =0
118       Openinglist.list.t' =Openinglist.list.t
119       Currentwindow.isin.t' =aws.New
120 }

122 pred same [t, t': Time] {
123       (Auxiliary.saved.t') = (Auxiliary.saved.t)
124       (∀ iw: Inputwidget | iw.content.t' =iw.content.t)
125       Openinglist.selected.t' =Openinglist.selected.t
126       Openinglist.list.t' =Openinglist.list.t
127 }
```

```
128 pred same2 [t, t': Time] {
129       (Auxiliary.saved.t') = (Auxiliary.saved.t)
130       (∀ iw: Inputwidget | iw.content.t' =iw.content.(T/first))
131       Openinglist.selected.t' =Openinglist.selected.t
132       Openinglist.list.t' =Openinglist.list.t
133 }
```

SAVE Abstract Semantics model

Bibliography

[1] Alibaba [Accessed: 2018-10-30]. Aliexpress, https://play.google.com/store/apps/details?id=com.alibaba.aliexpresshd.

[2] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S. and Memon, A. M. [2012]. Using gui ripping for automated testing of Android applications, *Proceedings of the International Conference on Automated Software Engineering*, ASE '12, ACM, pp. 258–261.

[3] Amazier [Accessed: 2018-10-30]. Bills reminder, https://play.google.com/store/apps/details?id=com.amazier.apps.billsreminder.

[4] Anand, S., Naik, M., Harrold, M. J. and Yang, H. [2012]. Automated concolic testing of smartphone apps, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, ACM, pp. 59:1–59:11.

[5] Andrey [Accessed: 2018-10-30]. Notepad, https://play.google.com/store/apps/details?id=ru.andrey.notepad.

[6] AndroMoney [Accessed: 2018-10-30]. Andromoney, https://play.google.com/store/apps/details?id=com.kpmoney.android.

[7] andtek [Accessed: 2018-10-30]. Seven habits, https://play.google.com/store/apps/details?id=com.andtek.sevenhabits.apk.

[8] *Appium* [Accessed: 2018-10-30]. https://github.com/appium.

[9] Apps, D. [Accessed: 2018-10-30a]. Color notes, https://play.google.com/store/apps/details?id=com.socialnmobile.dictapps.notepad.color.note.

[10] Apps, S. [Accessed: 2018-10-30b]. Pdfsam, https://play.google.com/store/apps/details?id=com.splendapps.splendo.

[11] Aquino, A., Denaro, G. and Pezzè, M. [2017]. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions, *Proceedings of the International Conference on Software Engineering*, ICSE '17, IEEE Computer Society, pp. 427–437.

[12] Arcuri, A. and Briand, L. [2014]. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, *Software Testing, Verification and Reliability* **24**(3): 219–250.

[13] Arlt, S., Podelski, A., Bertolini, C., Schaf, M., Banerjee, I. and Memon, A. M. [2012]. Lightweight static analysis for gui testing, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '12, IEEE Computer Society, pp. 301–310.

[14] AS, A. [Accessed: 2018-10-30]. Monefy, `https://play.google.com/store/apps/details?id=com.monefy.app.lite`.

[15] Azim, T. and Neamtiu, I. [2013]. Targeted and depth-first exploration for systematic testing of android apps, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, ACM.

[16] Back, T. [1996]. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*, Oxford university press.

[17] Banggood [Accessed: 2018-10-30]. Banggood, `https://play.google.com/store/apps/details?id=com.banggood.client`.

[18] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. [2015]. The oracle problem in software testing: A survey, *IEEE Transactions on Software Engineering* **41**(5): 507–525.

[19] Bauersfeld, S. and Vos, T. E. [2014]. User interface level testing with testar; what about more sophisticated action specification and selection?, *Seminar Series on Advanced Techniques and Tools for Software Evolution*, SATToSE '14, Springer, pp. 60–78.

[20] Becce, G., Mariani, L., Riganelli, O. and Santoro, M. [2012]. Extracting widget descriptions from guis, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, FASE '12, Springer, pp. 347–361.

[21] Behrang, F. and Orso, A. [2018a]. Automated test migration for mobile apps, *Proceedings of the International Conference on Software Engineering*, ICSE Poster '18, ACM, pp. 384–385.

[22] Behrang, F. and Orso, A. [2018b]. Test migration for efficient large-scale assessment of mobile app coding assignments, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '18, ACM, pp. 164–175.

[23] Bennett, R. [Accessed: 2018-10-30a]. Patternry, http://patternry.com/patterns/.

[24] Bennett, R. [Accessed: 2018-10-30b]. Timetracker, https://sourceforge.net/projects/ttracker/.

[25] Benwestgarth [Accessed: 2018-10-30]. Crossword sage, https://sourceforge.net/projects/crosswordsage/.

[26] Bishinews [Accessed: 2018-10-30]. Expense manager, https://play.google.com/store/apps/details?id=com.expensemanager.

[27] Bitslate [Accessed: 2018-10-30]. Notebook, https://play.google.com/store/apps/details?id=com.bitslate.notebook.

[28] Böhme, M. and Soumya, P. [2014]. On the efficiency of automated testing, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, ACM, pp. 71–80.

[29] Bourque, P. and Fairley, R. E. (eds) [2014]. *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society.

[30] Brambilla, M. and Fraternali, P. [2014]. *Interaction flow modeling language: Model-driven UI engineering of web and mobile apps with IFML*, Morgan Kaufmann.

[31] BrightTODO [Accessed: 2018-10-30]. Bright todo, https://play.google.com/store/apps/details?id=com.obplanner.

[32] *Buddi* [Accessed: 2018-10-30]. http://buddi.digitalcave.ca.

[33] Cadar, C. and Sen, K. [2013]. Symbolic execution for software testing: Three decades later, *Communications of the ACM* **56**(2): 82–90.

[34] Carvajal, M. [Accessed: 2018-10-30]. Gastos diarios, https://play.google.com/store/apps/details?id=mic.app.gastosdiarios_clasico.

[35] CASorin [Accessed: 2018-10-30]. Smart expenditure, https://play.google.com/store/apps/details?id=com.smartexpenditure.

[36] Cheng, L., Chang, J., Yang, Z. and Wang, C. [2016]. Guicat: Gui testing as a service, *Proceedings of the International Conference on Automated Software Engineering*, ASE '16, ACM, pp. 858–863.

[37] Choi, W., Necula, G. and Sen, K. [2013]. Guided gui testing of android apps with minimal restart and approximate learning, *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, ACM, pp. 623–640.

[38] Choudhary, S. R., Gorla, A. and Orso, A. [2015]. Automated test input generation for Android: Are we there yet?, *Proceedings of the International Conference on Automated Software Engineering*, ASE '16, IEEE Computer Society, pp. 429–440.

[39] Cobertura [Accessed: 2018-10-30]. Cobertura, http://cobertura.github.io/cobertura/.

[40] Cook, S. A. [1971]. The complexity of theorem-proving procedures, *Proceedings of the Annual ACM Symposium on Theory of Computing*, STOC '71, ACM, pp. 151–158.

[41] Coppola, R., Morisio, M. and Torchiano, M. [2017]. Scripted gui testing of android apps: A study on diffusion, evolution and fragility, *Proceedings of the International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE '17, ACM, pp. 22–32.

[42] Dinh, D. T., Hung, P. N. and Duy, T. N. [2018]. A method for automated user interface testing of windows-based applications, *Proceedings of the International Symposium on Information and Communication Technology*, SoICT 2018, ACM, pp. 337–343.

[43] Dix, A. [2009]. Human-computer interaction, *Encyclopedia of database systems*, Springer, pp. 1327–1331.

[44] Dropbox [Accessed: 2018-10-30]. Yesterday's authentication bug, https://blogs.dropbox.com/dropbox/2011/06/yesterdays-authentication-bug/.

[45] Ganov, S., Killmar, C., Khurshid, S. and Perry, D. E. [2009]. Event listener analysis and symbolic execution for testing gui applications, *Formal Methods and Software Engineering*, Springer, pp. 69–87.

[46] Google [Accessed: 2017-08-12]. Monkey runner, http://developer.android.com/tools/help/monkey.html.

[47] Gorla, A., Tavecchia, I., Gross, F. and Zeller, A. [2014]. Checking app behavior against app descriptions, *Proceedings of the International Conference on Software Engineering*, ICSE 2014, ACM, pp. 1025–1035.

[48] Gross, F., Fraser, G. and Zeller, A. [2012]. Search-based system testing: high coverage, no false alarms, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '12, ACM, pp. 67–77.

[49] Group, S. [Accessed: 2018-10-30]. Shein, https://play.google.com/store/apps/details?id=com.zzkko.

[50] Harman, M. [2007]. The current state and future of search based software engineering, *Proceedings of Future of Software Engineering*, FOSE '07, IEEE Computer Society, pp. 342–357.

[51] Harman, M. and Jones, B. F. [2001]. Search-based software engineering, *Information and Software Technology* **43**(14): 833–839.

[52] Harman, M., Mansouri, S. A. and Zhang, Y. [2012]. Search-based software engineering: Trends, techniques and applications, *ACM Computing Surveys* **45**(1): 11.

[53] Hayhurst, K. J. and Veerhusen, D. S. [2001]. A practical approach to modified condition/decision coverage, *20th DASC. 20th Digital Avionics Systems Conference*, NASA Langley Technical Report Server, pp. 1B2/1–1B2/10 vol.1.

[54] Hlcsdev [Accessed: 2018-10-30]. Bloc notes, https://play.google.com/store/apps/details?id=com.hlcsdev.x.notepad.

[55] *IBM Rational Functional Tester* [Accessed: 2018-10-30]. http://www-03.ibm.com/software/products/en/functional.

[56] Inc., A. [Accessed: 2018-10-30]. Ticktick, https://play.google.com/store/apps/details?id=com.ticktick.task.

[57] Jackson, D. [2002]. Alloy: a lightweight object modelling notation, *ACM Transactions on Software Engineering and Methodology* **11**(2): 256–290.

[58] Jenkins [Accessed: 2018-10-30]. Issue 25012, https://issues.jenkins-ci.org/browse/JENKINS-25012?jql=issuetype.

[59] Joriwal, H. [Accessed: 2018-10-30]. Onlineshopping, https://github.com/himalayjor/OnlineShoppingGUI/tree/master/OnlineShopping.

[60] Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T. and Lo, D. [2015]. Understanding the test automation culture of app developers, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '15, IEEE Computer Society, pp. 1–10.

[61] Kusner, M. J., Sun, Y., Kolkin, N. I. and Weinberger, K. Q. [2015]. From word embeddings to document distances, *Proceedings of the International Conference on International Conference on Machine Learning*, ICML'15, pp. 957–966.

[62] LightInTheBox [Accessed: 2018-10-30]. Lightinthebox, https://play.google.com/store/apps/details?id=com.lightinthebox.android.

[63] Linares-Vásquez, M., Holtzhauer, A. and Poshyvanyk, D. [2016]. On automatically detecting similar android apps, *Proceedings of the International Conference on Program Comprehension*, ICPC '14, IEEE Computer Society, pp. 1–10.

[64] Linares-Vásquez, M., White, M., Bernal-Cárdenas, C., Moran, K. and Poshyvanyk, D. [2015]. Mining android app usages for generating actionable gui-based execution scenarios, *Proceedings of the Working Conference on Mining Software Repositories*, MSR '17, IEEE Computer Society, pp. 111–122.

[65] Machiry, A., Tahiliani, R. and Naik, M. [2013]. Dynodroid: An input generation system for android apps, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '13, ACM, pp. 224–234.

[66] Mahmood, R., Mirzaei, N. and Malek, S. [2014]. Evodroid: Segmented evolutionary testing of android apps, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, ACM, pp. 599–609.

[67] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S. J. and McClosky, D. [2014]. The Stanford CoreNLP natural language processing toolkit, *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, ACL '14, Association for Computational Linguistics, pp. 55–60.

[68] Mao, K., Harman, M. and Jia, Y. [2016]. Sapienz: multi-objective automated testing for Android applications, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '16, ACM, pp. 94–105.

[69] Mao, K., Harman, M. and Jia, Y. [2017]. Crowd intelligence enhances automated mobile testing, *Proceedings of the International Conference on Automated Software Engineering*, ASE '17, IEEE Computer Society, pp. 16–26.

[70] Mariani, L., Pezzè, M., Riganelli, O. and Santoro, M. [2012]. Autoblacktest: Automatic black-box testing of interactive applications, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, IEEE Computer Society, pp. 81–90.

[71] Mariani, L., Pezzè, M., Riganelli, O. and Santoro, M. [2014]. Automatic testing of GUI-based applications, *Software Testing, Verification and Reliability* **24**(5): 341–366.

[72] Mariani, L., Pezzè, M. and Zuddas, D. [2015]. Recent advances in automatic black-box testing, *Advances in Computers*, Elsevier.

[73] Mariani, L., Pezzè, M. and Zuddas, D. [2018]. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles, *Proceedings of the International Conference on Software Engineering*, ICSE '18, pp. 280–290.

[74] Markushi [Accessed: 2018-10-30]. Expensemanager, `https://play.google.com/store/apps/details?id=at.markushi.expensemanager`.

[75] McMillan, C., Grechanik, M. and Poshyvanyk, D. [2012]. Detecting similar software applications, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 364–374.

[76] Memon, A., Banerjee, I. and Nagarajan, A. [2003a]. What test oracle should i use for effective gui testing?, *Proceedings of the International Conference on Automated Software Engineering*, ASE '03, IEEE Computer Society, pp. 164–173.

[77] Memon, A. M., Banerjee, I. and Nagarajan, A. [2003b]. GUI ripping: Reverse engineering of graphical user interfaces for testing, *Proceedings of The Working Conference on Reverse Engineering*, WCRE '03, IEEE Computer Society, pp. 260–269.

[78] Memon, A. M., Banerjee, I., Nguyen, B. and Robbins, B. [2013]. The first decade of gui ripping: Extensions, applications, and broader impacts, *Proceedings of The Working Conference on Reverse Engineering*, WCRE '13, IEEE Computer Society, pp. 11–20.

[79] Memon, A. M. and Xie, Q. [2005]. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software, *IEEE Transactions on Software Engineering* **31**(10): 884–896.

[80] Mesbah, A., Bozdag, E. and van Deursen, A. [2008]. Crawling ajax by inferring user interface state changes, *Proceedings of the International Conference on Web Engineering*, ICWE '08, ACM, pp. 122–134.

[81] Mhriley [Accessed: 2018-10-30]. Spending tracker, `https://play.google.com/store/apps/details?id=com.mhriley.spendingtracker`.

[82] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. [2013]. Distributed representations of words and phrases and their compositionality, *Proceedings of the International Conference on Neural Information Processing Systems*, NIPS '13, pp. 3111–3119.

[83] Mirte [Accessed: 2018-10-30]. Notebook, `https://play.google.com/store/apps/details?id=com.mirte.notebook.apk`.

[84] Moreira, R. M., Paiva, A. C. and Memon, A. [2013]. A pattern-based approach for gui modeling and testing, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '13, IEEE Computer Society, pp. 288–297.

[85] Nottage, S. [Accessed: 2018-10-30]. Tasks, `https://play.google.com/store/apps/details?id=com.tasks.android`.

[86] Ovdovchenko, M. [Accessed: 2018-10-30]. Todo list, `https://play.google.com/store/apps/details?id=com.mykhailovdovchenko.todolist`.

[87] Pearson, E. S. [1931]. The test of significance for the correlation coefficient, *Journal of the American Statistical Association* **26**(174): 128–134.

[88] Pezzè, M., Rondena, P. and Zuddas, D. [2018]. Automatic gui testing of desktop applications: an empirical assessment of the state of the art, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA companion '18, ACM.

[89] Pezzè, M. and Young, M. [2007]. *Software Testing and Analysis: Process, Principles and Techniques*, Wiley.

[90] *Rachota* [Accessed: 2018-10-30]. http://rachota.sourceforge.net/en/index.html.

[91] Rau, A., Hotzkow, J. and Zeller, A. [2018]. Efficient gui test generation by learning from tests of other apps, *Proceedings of the International Conference on Software Engineering*, ICSE Poster '18, ACM, pp. 370–371.

[92] Realbyteapps [Accessed: 2018-10-30]. Money manager expense, https://play.google.com/store/apps/details?id=com.realbyteapps.moneymanagerfree.

[93] Ricca, F. and Tonella, P. [2001]. Analysis and testing of web applications, *Proceedings of the International Conference on Software Engineering*, ICSE '01, IEEE Computer Society, pp. 25–34.

[94] Romwe [Accessed: 2018-10-30]. Romwe, https://play.google.com/store/apps/details?id=com.romwe.

[95] *Selenium* [Accessed: 2018-10-30]. https://www.seleniumhq.org.

[96] Shapiro, S. S. and Wilk, M. B. [1965]. An analysis of variance test for normality (complete samples), *Biometrika* **52**(3/4): 591–611.

[97] Song, F., Xu, Z. and Xu, F. [2017]. An xpath-based approach to reusing test scripts for android applications, *Web Information Systems and Applications Conference (WISA), 2017 14th*, WISA '17, IEEE Computer Society, pp. 143–148.

[98] *Spark* [Accessed: 2018-10-30]. https://igniterealtime.org/projects/spark.

[99] Student [1908]. The probable error of a mean, *Biometrika* pp. 1–25.

[100] Studio, D. H. D. [Accessed: 2018-10-30]. Bloc note, https://play.google.com/store/apps/details?id=com.studio.tools.one.a.notes.

[101] Systems, J. [Accessed: 2018-10-30]. Simplest checklist, https://play.google.com/store/apps/details?id=jakiganicsystems.simplestchecklist.

[102] Taxaly [Accessed: 2018-10-30]. Fast notepad, https://play.google.com/store/apps/details?id=com.taxaly.noteme.v2.

[103] Tidwell, J. [2010]. *Designing interfaces: Patterns for effective interaction design*, "O'Reilly Media, Inc.".

[104] Turian, J., Ratinov, L. and Bengio, Y. [2010]. Word representations: a simple and general method for semi-supervised learning, *Proceedings of the 48th annual meeting of the association for computational linguistics*, Association for Computational Linguistics, pp. 384–394.

[105] *Universal Password Manager* [Accessed: 2018-10-30]. http://upm.sourceforge.net/index.html.

[106] Universe, P. [Accessed: 2018-10-30]. Ike to do list, https://play.google.com/store/apps/details?id=com.pocketuniverse.ike.

[107] Vacondio, A. [Accessed: 2018-10-30]. Pdfsam, https://sourceforge.net/projects/pdfsam/.

[108] Van Lamsweerde, A. [2009]. *Requirements engineering: from system goals to UML models to software specifications*, Wiley.

[109] van Welie, M. [Accessed: 2018-10-30]. Pattern library, http://www.welie.com/patterns/index.php.

[110] Vos, T. E., Kruse, P. M., Condori-Fernández, N., Bauersfeld, S. and Wegener, J. [2015]. Testar: Tool support for test automation at the user interface level, *International Journal of Information System Modeling and Design* **6**(3): 46–83.

[111] *W2vec pre-trained model* [Accessed: 2018-10-30]. https://code.google.com/archive/p/word2vec/.

[112] Whiteglow [Accessed: 2018-10-30]. Keep my notes, https://play.google.com/store/apps/details?id=org.whiteglow.keepmynotes.

[113] Whitley, D. [1994]. A genetic algorithm tutorial, *Statistics and computing* **4**(2): 65–85.

[114] Yoox [Accessed: 2018-10-30]. Yoox, https://play.google.com/store/apps/details?id=com.yoox.

[115] Yuan, X., Cohen, M. B. and Memon, A. M. [2011]. Gui interaction testing: Incorporating event context, *IEEE Transactions on Software Engineering* **37**(4): 559–574.

[116] Zalando [Accessed: 2018-10-30]. Zalando, https://play.google.com/store/apps/details?id=de.zalando.mobile.

[117] Zara [Accessed: 2018-10-30]. Zara, https://play.google.com/store/apps/details?id=com.inditex.zara.

[118] Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W. and Xie, T. [2016]. Automated test input generation for android: Are we really there yet in an industrial case?, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '16, ACM, pp. 987–992.

[119] Zuddas, D. [2016]. Semantic testing of interactive applications, *Companion proceedings of the International Conference on Software Testing, Verification and Validation*, IEEE Computer Society, pp. 391–392.

[120] Zuddas, D., Terragni, V., Mariani, L. and Pezzè, M. [2018]. Semantic gui testing via cross-application test adaptation, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '19, ACM, p. under review.