Semantic Matching for Migrating Tests Across Similar Interactive Applications

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
Ali Mohebbi

under the supervision of
Prof. Mauro Pezzè and Prof. Valerio Terragni

10 2023

Dissertation Committee

Antonio Carzaniga	Università della Svizzera italiana, Lugano, Switzerland
Fabio Crestani	Università della Svizzera italiana, Lugano, Switzerland
Rui Abreu	University of Porto, Portugal & Meta Platforms, USA
Mattia Fazzini	University of Minnesota, USA

Dissertation accepted on 11 10 2023

Research Advisor

Prof. Mauro Pezzè

Co-Advisor

Prof. Valerio Terragni

PhD Program Director

The PhD program Directors *Walter Binder, Stefan Wolf*

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.

Ali Mohebbi
Lugano, 11 10 2023

Abstract

Interactive mobile apps are extremely popular, and there is a strong demand for short time to market, high quality, and low costs. It is important to automate testing analysis to efficiently and effectively verify these applications, and comply with market requirements. In the last decades, many researchers have proposed several approaches to automatically generate test cases for interactive applications.

Most approaches for testing interactive applications consider structural information, and ignore semantic information. As a result, they are not effective in revealing non-crashing semantic errors and wrong outputs. TEST REUSE is a recent promising research line that aim to generate semantically relevant test cases by migrating test cases across similar applications. TEST REUSE approaches leverages semantic information available in the GUI of applications to migrate test cases from a source application to a target application that shares similar functionalities. TEST REUSE approaches rely on semantic matching to find similar events between source and target applications based on the textual descriptor of GUI widgets. TEST REUSE approaches use semantic matching techniques based on simple considerations, and do not carefully investigate the different design choices and their effects. Understanding the design choices, that is, identifying commonalities and differences in semantic matching techniques in the context of test generation approaches, can help us develop effective migration strategies.

In this thesis, we study the semantic matching of events in approaches that reuse test cases across interactive applications: We study the components of semantic matching of events available in the GUI, and identify the best choices for TEST REUSE among available solutions and those that we propose in this thesis. In the thesis, we (i) define a general context for TEST REUSE approaches that encompasses common components of all TEST REUSE approaches, (ii) introduce a framework to automatically evaluate semantic matching both in-isolation and in the context of test reuse, (iii) conduct a comprehensive empirical study, and (iv) provide important insights about the components of semantic matching and its effect on test reuse.

Acknowledgements

We would like to thank the Swiss National Science Foundation that partially supported this PhD within the SNF project ASTERIx: Automatic System TEsting of inteRactive software applIcations *(SNF 200021_178742)*.

I thank my advisor, Prof. Mauro Pezzè, and my co-advisor, Prof. Valerio Terragni. Their significant contributions have been instrumental in the success of this work. I am grateful for all the help I received from Prof. Leonardo Mariani for his invaluable help and guidance throughout my research. His constructive feedback and compassionate approach played a significant role in shaping my work.

I would also like to extend my sincere thanks to my dear colleague in the STAR group, Rahim Heydarov, and my old friend Dr. Ali Siahkamari. Their support during both the bright and dark hours, were essential to my success.

I would like to express my heartfelt gratitude to my parents for their unwavering support throughout my academic journey. Their love, encouragement, and belief in me have been invaluable in helping me to achieve this milestone. I cannot thank them enough for the sacrifices they have made to make this possible. Their unwavering support has helped me to stay focused and motivated, even when the going got tough.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we discuss the importance of testing and the need for automation. It highlights the importance and role of test reuse, and overviews the main results of the PhD research.

Interactive mobile applications are widely spread in the everyday life. They become increasingly complex, and continuously grow with new functionalities to meet the requirements of a fast changing market [92]. The wide spread of interactive mobile applications largely emphasizes the costs of failures in production, and the fast evolving demand of the market makes it extremely effort demanding to thoroughly test such applications.

Automatically generating test cases can largely reduce the cost of testing, and can make it feasible to thoroughly test applications and reveal faults before expensive impacts in production. The research on automatically generating test cases for interactive mobile applications produced many approaches that we classify in four main categories: random [59], coverage-based [60], model-based [80], and similarity-based approaches [76]. Random approaches randomly interact with the GUI. Model-based approaches build a GUI model that they use to explore the execution space of the application. Coverage-based approaches generate test cases that maximize some structural coverage criteria. Similarity-based approaches migrate test cases across similar applications. All but similarity-based approaches are agnostic to the semantics of the applications, and as such they cannot reveal failures that depend on the semantics of the applications. Semantic-agnostic approaches can generate a large number of test cases, achieve high coverage, and reveal simple crashes. However, semantic-agnostic approaches cannot systematically exercise meaningful scenarios and reveal non-crashing faults such as wrong output. For instance, let us consider an application that identifies the type of the input graphs. Semantic-agnostic approaches inter-

act with the application by creating simple graphs with random nodes connected with random edges. They unlikely create well-formed graphs that can check the correct behavior of the application for all types of graphs.

Applications that address similar requirements and share similar functionalities are a relevant phenomenon in the context of mobile applications. Similarity-based approaches largely reduce the effort to generate semantically relevant test cases, by taking advantage of this phenomenon. Many studies indicate an enormous availability of similar applications: Hu et al. report that 196 (63.4%) of the top 309 non-game mobile apps in the Google Play Store can be classified into 15 groups such that each group share many common functionalities [42]; Ebrahimi et al. classify the 1.8M apps of Apple App Store in 23 categories, and the 2.87M apps of Google Play in 35 distinct categories [31]. The large availability of similar applications makes it extremely promising to automatically share test cases among similar applications.

We classify similarity-based approaches as PATTERN-BASED [67] and TEST REUSE [53, 10, 68] approaches. PATTERN-BASED approaches identify patterns that abstract the knowledge embedded in test cases, and generate test cases by pattern-matching events that comprise the test cases of the source applications with events that comprise the test cases of the target applications. PATTERN-BASED approaches either rely on manually-defined patterns [67] or automatically extract patterns from test cases [65]. Manually defining patterns require a conceivable human effort, while automatically extracting patterns requires to analyze a large number of test cases. Both activities are expensive and produce only a limited amount of patterns. TEST REUSE approaches migrate the knowledge of testing a functionality from the test case of a source application to a target application [53, 10, 68]. The enormous amount of well-tested mobile applications that share similar functionalities can produce a large amount of test cases with no expensive preprocessing.

TEST REUSE approaches semantically match events that comprise the test cases of a well-tested source application to events that comprise test cases for a target application. TEST REUSE approaches consider oracles expressed as assertions as events that may require some extra checks. Developers use different terms for similar semantic information and approaches to migrate test cases must address two key challenges: (i) semantically similar events expressed with syntactically different terms, and (ii) a many-to-many mapping between semantically similar events. TEST REUSE approaches identify semantic similarities among syntactically different items, by semantically matching textual information with word embedding techniques. Also, they identify many-to-many mappings with different selection strategies that steer the exploration toward generating test

cases that exercise the target application realistically [53, 10]. Zhao et al. [109] developed FrUITeR to comparatively assess TEST REUSE approaches. FrUITeR compares approaches monolithically, however, it does not offer a way to compare the contribution of the different components of the approaches.

In this thesis, we address the problem of migrating test cases across semantically similar mobile applications, which is the most promising approach to automatically generate semantically relevant test cases for mobile apps. We define approaches general enough to be applied to general interactive applications. We present the results of experimenting with Android apps. The following characteristics of the ANDROID platform make ANDROID applications an ideal target for studying testing interactive applications [21]: (i) ANDROID has the largest share of the mobile market (ii) Cross-version incompatibilities of ANDROID OS and diversity of devices increase the necessity for automated testing (iii) The open-source nature of the ANDROID platform and its related technologies makes it a more suitable target for academic researchers.

The most relevant characteristics of mobile devices that challenge developments and testing are the heterogeneity of hardware configurations of the devices and the variability of the running conditions [2]. The hardware configurations vary in the types of sensors, displays and CPUs. Moreover, the devices support different mobile networks and communications technologies, resulting in a big variety of configurations that challenge both development and testing.

In this thesis, we (i) discuss the results of an in-depth study of TEST REUSE approaches, (ii) identify the main limitations of current approaches, (iii) define and evaluate solutions to overcome the main limitations of state-of-the-art approaches, and (iv) introduce two frameworks to automatically evaluate the internal components of TEST REUSE.

1.1 Research Hypothesis and Contributions

The main research hypothesis of this thesis is the following:

> *The in-depth study of semantic matching provides insights that facilitate the definition of mature* TEST REUSE *approaches.*

TEST REUSE approaches share many similarities and interesting distinctive factors. To study both the contribution of similarities and the impact of differences, we need to identify the common conceptual components of TEST REUSE approaches, and compare TEST REUSE approaches at a fine granularity level. By

studying TEST REUSE approaches at the granularity of components, we can identify the contributions and limitations at a fine granularity level, and thus identify optimal and sub-optimal combinations of the different components, to define adequate solutions. In this thesis we propose a conceptual architecture that abstracts internal components of TEST REUSE, along with their interactions. We use the ARCHITECTURE to investigate and evaluate the design choices of the current TEST REUSE approaches. The ARCHITECTURE is composed of five components that all TEST REUSE approaches have in common: Corpus of Documents, Word Embedding, Event Descriptor Extractor, Semantic Matching Algorithm, Event Selector.

Different choices for designing and implementing the conceptual components of test reuse approaches may result in different performance of semantic matching and TEST REUSE. In this thesis we compare both the instances of the main components of TEST REUSE available in the current approaches, and alternative instances of components that may improve effectiveness and efficiency of TEST REUSE. In addition, we propose new instances for SEMANTIC MATCHING ALGORITHM and CORPUS OF DOCUMENTS components that outperform existing ones.

The components' instances of TEST REUSE approaches cooperate and affect on each other. In this thesis, we present the results of our study of different combinations of component instances, and discuss the impact of the interactions of different instances of the different conceptual components. A systematic exploration of the possible combinations for TEST REUSE requires an effective infrastructure that automate the process. We propose a framework that integrates different instances of the main components, to automatically evaluate different choices of component instances.

We discuss the results of a thorough empirical evaluation of the components both in-isolation by 8,099 GUI events and in context of the TEST REUSE by 6,000 test migrations. The evaluation of the individual component provides in-depth insights of their contributions and limitations in TEST REUSE. The results of our evaluation indicate both that there exist a lot of space for improving the current semantic matching techniques, and the semantic matching is only one of the key factors for effectively and efficiently migrating test cases across applications.

The TEST REUSE ARCHITECTURE and the frameworks that we introduce enables an evolutionary way of creating TEST REUSE approaches by combining different instances of the components. We offer the TEST REUSE ARCHITECTURE and the frameworks in a replication package to allow interested researchers and practitioners to propose new TEST REUSE approaches, by experimenting with new instances of some components in combination with currently available instances of other components.

Finally, we propose a new generation of TEST REUSE approaches based on best choices of each components, as indicated by the results of our experimental evaluation.

We follow an inductive methodology: we first review the current TEST REUSE approaches, and propose a general ARCHITECTURE that consist of abstract components. We then augments the instances of the different components that current approaches propose with new instances to address limitations of the components proposed so far. Each combination of instances yield a different semantic matching configuration which performs uniquely. We evaluate the configurations in two steps: in isolation and in test reuse context.
In the first step, we synthesis a data set of evens, and study the components of semantic matching. We build the data set base on the test migration scenarios that have been used in the previous studies. Then we build a framework that systematically incorporates different combinations of instances for semantic matching and evaluate their performance with the metrics dedicated for matching of events.
In the second step, we build a framework that integrates the TEST REUSE approaches available in the literature with different semantic matching configurations. We use the framework to migrate scenarios that we considered in the first step with different semantic matching configurations. We evaluate performance of the configurations with metrics dedicated to quality of migrations. At the end of each step we discuss the insights that we gained based on the empirical results and we indicate the best and worst performing instances.

Our study both offers a comparative evaluation of the instances of the different components and indicates the most relevant component for semantic matching and test reuse. Our insights highlight the limitation of the current semantic matching approaches for reusing test cases and indicates the horizon of future studies.
In the thesis, we propose a new approach that we define by assembling the best instance of the different components of test reuse.

1.2 Thesis Structure

This thesis is organized as follows:

- Chapter 2 introduces the core terminology used in the thesis and presents the state of the art in TEST REUSE approaches.
- Chapter 3 introduces the ARCHITECTURE that we propose to abstract TEST

REUSE internal component, and discuss the instances that we considered for each components.

- Chapter 4 presents the framework we propose to evaluate the components both in isolation and in TEST REUSE context.
- Chapter 5 reports the results of evaluating TEST REUSE component in isolation.
- Chapter 6 reports the results of evaluating domain specific word embedding models in isolation.
- Chapter 7 reports the results of our evaluation of the TEST REUSE components in TEST REUSE context.
- Chapter 8 Summarizes the contributions of the thesis and discusses open-research directions.

Chapter 2

Migrating Tests Across Interactive Applications, State of the Art

In this chapter we first introduce the core concepts of testing interactive applications that are relevant in this thesis, and presents the notations that we use in the thesis, to make the thesis self-contained. Then, we overview the approaches to automatically generate test cases for interactive applications.

Testing software systems in general and in particular interactive applications requires to both generate and execute test cases to reveal faults. Capture and replay (C&R) tools automatically generate test cases for interactive applications by recording user interactions and converting them to test cases. C&R tools are easy to use and a tester with limited testing knowledge can generate scripts in a short time. However, C&R tools are fragile with respect to app evolution, in particular GUI changes [77, 102]. Also, they suffer from portability issues: The more accurate the recorded events are (timing, coordinate of events), the more coupled the generated test is to the device characteristics it was recorded on [55]. More importantly, C&R tools require a conceivable manual effort and the effectiveness of C&R tools depends on the effort of the testers [27].

Automated test generation approaches reduce the manual effort required to generate test cases. Automatically generating test cases is still an open challenge, despite the many approaches and tools developed in the last decades. Linares et.al observe than C&R tools are more effective that automatic generators in terms of code coverage if the tester has enough time and information about the AUT [27], since automatic approaches hardly execute branches that are reachable only with complex interactions.

In this chapter, we introduce the main terminology that we use thorough the thesis, and overview the relevant approaches to automatically generate test cases for interactive applications.

2.1 Terminology

A Graphical User Interface of a mobile app, *GUI*, is a forest of hierarchical windows where only an *active window* is available to be used at any time ([73]). Windows include **widgets**, atomic elements that are characterized by attributes (such as *text* and *identifier*). At any time, the active window has a **state S** that encompasses the attribute values of the displayed widgets. The type of the widgets depends on their functionality. Some widgets expose user-actionable events to let users interact with the app ([28]). For example, *Label* widgets provide textual information and users cannot interact with them, while *Button* widgets enable click actions.

An **event** is an atomic interaction on a widget. *GUI events* are human-computer interactions, for instance, click on widgets of type *Button*, or fill widgets of type *EditText*. Following previous studies, we abstract the implemented widget types and group GUI events into *clickable* and *fillable*. Clickable events include simple click, swipe and long click, and are applicable to a wide range of widget types. Some clickable events allow navigating across different pages, like event e_1^s in Figure 2.1. Other events execute some functionality, for instance, event e_4^s saves an expense. The ANDROID context offers different widget types that support click events, such as Button, ListView, Dialog, and ImageButton. Fillable events insert a text into an EditText widget. They provide information that are required for a functionality, for instance, event e_3^s of Figure 2.1 allows to insert the amount of an expense. *Oracle events* check the state of the widgets, for instance, event o_1^s *exist(restaurant)* of Figure 2.1. checks if there is widget in the current state with a text attribute equal to *hello*. An event is a triple $\langle widget, type, input \rangle$, in which widget refers to the GUI element that event will be executed on; type of event could be clickable, fillable or oracle; Input is the string that should be used to fill a fillable widget or needs to be checked in case of the oracle event.

A **GUI test t** is an ordered sequence of events $\langle e_1, ..., e_n \rangle$ on widgets of the active windows. A test execution induces a sequence of state transitions $S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \ldots \xrightarrow{e_n} S_n$ where S_{i-1} and S_i denote the states of the active window before and after the execution of e_i, respectively. A GUI test case can have one or more assertion oracles that check the correctness of the state S_i obtained after the execution of an event e_i ([7]), for example, checking for the absence or presence

of widgets with specific attributes values.

A **GUI model (**TARGET APPLICATION MODEL**)** is a directed graph where nodes correspond to the states of the GUI, and edges are labelled with the events that lead from the state that corresponds to the source to the state that corresponds to the target node of the edge, respectively.

Test Reuse is the process of migrating GUI test cases across apps that share similar functionalities. More formally, given two apps A^s (source) and A^t (target), and a "source" test t^s for A^s, TEST REUSE approaches generate "target" test case t^t that tests A^t as t^s tests A^s. TEST REUSE approaches create t^t by searching A^t for events that are semantically similar to events in t^s. We refer to a pair of source test case and target application $\langle t^s, A^t \rangle$ as a scenario. Each scenario is paired with the ground truth that is defined as the events $e^t_{gt} \in t^t$ that match the events $e^s \in t^s$. The target test case t^t may include *ancillary* events ([109]), that is, events in t^t that do not correspond to any event in t^s, and that are required to reach relevant states in the app. The source test case t^s may include *unmatchable* events, that is, events that do not have any equivalent in the target application. Ancillary and unmatchable events exist because the applications implement functionalities in different ways.

Figure 2.1 shows an example of a migration from a test designed for the source app *Money Tracker* (A) to the target app *EasyBudget* (B). The two test cases verify the same feature, namely adding an item. The example shows events of all types, including clickable, fillable, and oracle, with oval lines that indicate position of interactions. Events in the target test case may either match events in the source test case or be ancillary events needed to move from states in the target app. Event e^t_1 is an example of an ancillary event that leads to the next state, where events e^s_1 and e^t_2 match each other.

2.2 Testing Interactive Applications

We classify approaches that test interactive applications into 4 categories: *random, coverage-based, model-based, similarity-based* approaches. In the next section, we introduce each category, discuss the core ideas of representative approaches, and highlight the main limitations of each category.

2.2.1 Random approaches

Random approaches generate GUI test cases by randomly executing GUI events. While these approaches are the most simple ones, some of them included heuris-

Figure 2.1. Test reuse example, the target test cases (B) is obtained by migrating the source test case (A)

tics to avoid irrelevant events.

ANDROID toolkit offers MONKEY [121], a tool for stress testing of Android applications. MONKEY randomly clicks on different positions of the application's window, regardless of its effect on the application.

Machiry et al. define DYNODROID ([59]) an approach that introduces *observe-select-execute* cycles. In each cycle, DYNODROID observes which widgets are available in the current GUI state, selects an event and executes it. DYNODROID select executable events according to two criteria: all widgets should be executed eventually and no widgets should be selected too frequently. In contrast to MONKEY, DYNODROID both excludes useless actions and considers text input events. For example, MONKEY might click on a label widget which is not clickable, and DYNODROID avoids such actions by considering the type of the widgets. DYNODROID is constrained to specific ANDROID SDK since it needs instrumentation of the SDK to identify available event listeners and infers the type of the widgets. Similar to

DYNODROID, TESTAR [100] generates random test cases while considering proper events in each state. TESTAR uses the operating system accessibility API to infer the type of the widgets available in each GUI state.

Eruth et al. introduce the idea of macro events: sequences of low-level GUI events that abstract single logical steps, such as filling and submitting a form [32]. Their approach leverages well-known data mining techniques to automatically extract macro events from execution traces of human users. Their approach combines macro events with random testing to generate more effective test cases than purely random test cases, since they consider realistic sequence that correspond to common paths that users will take.

DEEPGUI [105] extends MONKEY and leverages Deep Reinforcement Learning to identify actionable widgets. DEEPGUI trains its model using screenshots and executed events of GUI states from random crawling of many applications.

The Simplicity of random approaches enables them to quickly generate and execute many test cases [14]. In contrast to low coverage of random testing in other areas [52], random approaches for testing interactive applications can achieve a high coverage. However, they are unaware how much of the application behavior they covered. They both generate many redundant test cases [6] and miss important scenario [67]. Finally, they consider manually specified timeout rather than a stopping criterion that indicates the success of the exploration.

2.2.2 Coverage-based approaches

Coverage-based approaches generate GUI test cases that maximize the coverage of the application under the test. Coverage-based approaches are either search-based or symbolic-execution-based.

Searched-based solutions are used in many software engineering domains [85] such as test prioritization [39], refactoring [71], and software correction [45]. Search-based software engineering (SBSE) reformulates software engineering activities as optimization problems and uses search algorithms to automatically solve the problems [41]. A key element in SBSE is to define the fitness function, which allows to both numerically assess the quality of solutions and guides the search algorithm. Researchers usually use software metrics as fitness function. For instance, they use code coverage to minimize test suites. Researchers widely applied evolutionary algorithms to solve the optimization problems [110]. Evolutionary algorithms first create a population of random solutions (individuals) then they evolved the population. The algorithms iteratively select high quality individuals to act as parents, which generate new solutions by applying crossover and mutation operators. In the context of testing interactive applications, the op-

timisation problem can be defined as finding the set of test cases that maximize statement coverage or multi-objective optimization such as high coverage and high number of faults revealed.

EXSYST is the first approach that uses an evolutionary algorithm to generate test suites for desktop applications [36]. The cross-over operator in the evolutionary algorithm of EXSYST creates two offspring test suites from two parent test suites. The mutation operator works both at test suite level by inserting a test case and at test case level by deleting, changing or adding an event. EXSYST calculates the fitness value for a test suite by aggregating the minimum branch distances [72] of all the branches in the program. The fitness function estimates how close a test suite is to covering all branches of a program. The coverage of the final population is no better than the initial population (randomly generated tests), as EXSYST mainly uses the evolutionary algorithm to minimize the number of test cases.

EVODROID [60] generate test cases that maximize the coverage of unique paths in call graph from the starting node of an app to all its leaf nodes. EVODROID generates the call graph by statically analyzing the program, partitions the call graph nodes to different segments based on their activity and services, and incrementally generates test cases by finding a test case for each segment from the root to leaves.

SAPIENZ uses a multi-objective fitness function with three competing objectives: (i) short sequence of events, (ii) high coverage, (iii) high number of revealed crashes. Other approaches do not simultaneously optimize these competing objectives, whereas SAPIENZ provides a set of optimal tradeoff solutions.

Recently, Dong et al. introduced TIMEMACHINE that extends MONKEY by keeping track of states with high fitness and get back to them when it cannot progress in test generation [29]. TIMEMACHINE takes random actions to reach new states, and restart from one of the most fitted states, when it cannot progress anymore. Differently from other SBSE approaches that consider a population of test suites, TIMEMACHINE evolves a population of app states. TIMEMACHINE defines the fitness function based on how many times the state and its neighbors are visited and trigger the execution of new code.

Symbolic-based approaches adapt either symbolic or concolic execution for GUI testing. BARAD [34] introduces a symbolic abstraction of widgets that enables symbolic analysis of the GUI control flow. Symbolic execution is very expensive due to combinatorial explosion and is typically used in small software systems and at the unit level. BARAD tackles this issue with two strategies: It only considers events with registered event listeners, thus pruning the state space, and introduces symbolic widget to symbolically manipulate GUI widgets and obtain

an executable symbolic version of the GUI.

ACTEVE [4] and GUICAT [19] use concolic execution to generates sequence of events systematically [4]. ACTEVE tackles the combinatorial explosion problem by considering a subsumption condition between event sequences to prune redundant event sequences, while being complete with respect to classic concolic execution. GUICAT [4] propose a cloud based parallel algorithm for mitigating both event sequence explosion and data value explosion.

SYMJS introduces the first symbolic virtual machine for JavaScript and leverages that to generates event sequences for web applications written in JavaScript [50]. SYMJS uses a symbolic and dynamic feedback to direct input space exploration and reduce testing time.

Coverage-based approaches systematically explore the application state space, results in high coverage, and exercise behaviors that would be hard to reach with random techniques. However, there is no guarantee that such high-coverage test suites exercise the application in a realistic way or be effective in revealing faults. These approaches are considerably less scalable and require to instrument the app to measure coverage and possibly execute it symbolically.

2.2.3 Model-based approaches

Model-based approaches generate test cases driven by a GUI model. Models are typically built by dynamically exercising the GUI of the application under test [73]. Model-based approaches either cover the GUI model based on coverage criteria or use the model to make decisions in each state.

GUITAR [80] is the first model-based approach for GUI applications. Researchers deployed GUITAR for desktop [80], mobile [3] and web [74] platforms. The effectiveness of model-based approaches depends on the completeness and quality of the models. Fine-grained models suffer from the state explosion problem, while coarse-grained model do not provide enough information for an effective guidance. ORBIT [104] tackles quality and size issues of GUI models by statically analyzing the application to extract the set of events the GUI application supports, and uses the extracted events to dynamically crawl the application. DroidBot [106] creates a GUI model on-the-fly using method log and process information in addition to GUI state information. DroidBot does not require instrumentation and allows testers to integrate their strategies to exercise the GUI model. APE [37] dynamically adjusts the granularity of the model as needed, instead of operating on a fixed abstraction granularity. APE observes feedback during testing and gradually refines the model while maintaining a balance between the size and precision of the model.

SWIFT [20] uses a machine learning adaptive algorithm (automata learning) that operate on unknown random environments. SWIFT learns a model of the application during testing and uses the learned model to generate events that visit unexplored states of the app. The automata learning algorithm uses the execution of the app on the generated events to refine the model. A key feature of SWIFT is that it avoids restarting the app to reduce the cost.

Many approaches use Q-learning [95] to model behavior of the GUI. Q-learning based approaches determine the reward of an action from the differences among the states reached with the actions. The more two states are different, the higher the reward is, and thus the probability to move in that direction. AUTOBLACKTEST [69] uses Q-learning to generate test cases for desktop applications. It rewards GUI events that activate many changes in the GUI state, and penalizes events that activate marginal changes. A-TEST adapts reinforcement learning to the ANDROID environment [101]. Q-TESTING [81] differentiates states at the granularity of functional scenarios by considering previous states to calculate the reward function. Remembering past states guides Q-TESTING towards unfamiliar functionalities. ARES uses deep learning to infer state similarity and the action-value function, while other approaches use Tabular Reinforcement Learning [89]. Deep Reinforcement Learning is more effective than Tabular when state space is extremely large, like GUI of Android applications.

The effectiveness of model-based approaches is limited to the ability of the model to abstract the behavior of the application, and creating a high quality models is expensive. Similarly to coverage based-approaches, model-based approaches cannot distinguish between semantically relevant and irrelevant tests.

2.2.4 Similarity-based approaches

There are millions of interactive applications and many of them share similar functionalities. The similarity among functionalities offers an opportunity to migrate the knowledge encoded in test cases across applications that share similar functionalities. We classify similarity-based approaches as PATTERN-BASED and TEST REUSE approaches. PATTERN-BASED approaches consider knowledge of testing the functionality as patterns of recurrent events, TEST REUSE approaches reuse test cases of an application for another application. Similarity-based approaches generate test cases that are realistic and semantically relevant. Similarly-based approaches also address the fragility and portability issues of C&R tools: Testers can record test cases of an app and reuse them to test both an evolution of the app and new devices incompatible with the ones used in the recorded testing. Similarly-based approaches can benefit from test cases that C&R tools generate

and that encompass knowledge of testing a functionality: PATTERN-BASED approaches can extract the patterns from the test cases or TEST REUSE approaches use a test case as the source for migration.

PATTERN-BASED approaches abstract the functionalities of the applications as patterns, and match patterns' elements with the events of the application under the test. These approaches either use patterns that are predefined manually or extract patterns from execution traces.

PARADIGM [76] generates GUI test cases for web and mobile applications from user-defined GUI interaction patterns, expressed with a domain-specific language [76, 78]. AUGUSTO [67] introduces the notion of Application Independent Functionalities (AIF) to abstract recurrent patterns that are prevalent among applications of different domains. Previous approaches only used textual information from GUI's DOM, while APPFLOW [42] considers both textual and visual information (by using computer vision) to map elements of predefined patterns to target app events. The effectiveness of PARADIGM, APPFLOW and AUGUSTO depends on the availability of pre-defined patterns, which are difficult and expensive to create manually.

Recent approaches propose different strategies to extract patterns automatically. Linares et al. propose MONKEYLAB: a language GUI model that encode the probability of occurrence of a sentence (words are events in this context) [56]. They train the language model by mining developers' usage of apps and analyzing the APK of android applications. MONKEYLAB can generate test cases that reflect both common and uncommon (corner) cases. POLARIZE [65] extracts *motifs*, a short sequence of recurrent events, from large sets of execution traces of multiple apps, and feed the *motifs* to a genetic algorithm, to generate test cases for other apps, aiming to maximize coverage. Mao et al. [66] propose data mining techniques to extract usage behavior patterns at the granularity of functional scenarios from user traces. AVGUST [108] leverages computer vision and NLP techniques to extract usage patterns from screen recordings of multiple apps. AVGUST is a developer-in-the-loop technique: It ranks the target events based on their similarity to the pattern elements and recommends top events to developers to choose from.

Manually defining patterns for these approaches is expensive, and automatically extracting patterns requires a big data set of execution traces.

TEST REUSE approaches for GUI applications ([109]) automatically migrate GUI test cases (including oracles) across apps that share similar functionalities. Researcher proposed TEST REUSE approaches that migrate test cases of the same

applications cross different platforms. TESTMIG [84] reuses test cases of IOS applications to generate test cases for the ANDROID version of the same applications. TESTMIG uses *tf-idf* to convert text to vectors of real numbers, and uses the cosine similarity formula to compute the similarity of events between the two versions of the same app. Then, it maps events by using probabilistic sequence transduction – a probabilistic model widely used in machine translation. MAPIT migrates test cases in bidirectional way between ANDROID and IOS version of the same application [97]. TRANSDROID [54] migrates test cases of Web apps to their corresponding ANDROID version, with a SEMANTIC MATCHING ALGORITHM similar to CRAFTDROID. Migrating test cases of the same application across different platforms is generally easier than migrating test cases across different apps.

Other TEST REUSE approaches migrate test cases between different applications on the same platform. Rau et al. [86] propose to use semantic matching of GUI elements to migrate test cases across web applications. CRAFTDROID [53] migrates tests across ANDROID applications. GUITESTMIGRATOR [11] migrates test cases between prototype applications with the same specifications. APPTESTMIGRATOR (ATM) [9] extends GUITESTMIGRATOR to migrate test cases of real applications with similar functionality. ADAPTDROID formulates the problem of reusing test cases as a search problem, and explores the space of possible GUI test case with an evolutionary approach. ADAPTDROID uses a fitness function that rewards the tests cases that are most similar to the donor test case [68].

In this thesis we study TEST REUSE approaches, and we focus on the most relevant state-of-the-art TEST REUSE approaches: ATM, CRAFTDROID, and ADAPTDROID that we discuss in detail in the next sections. The three approaches leverage WORD EMBEDDING to semantically match event across applications, and they build a GUI model for guiding the migration process. However, they differ in strategies that they consider for matching of events and guiding the TEST REUSE process.

2.3 Limitations and open problems

Random, coverage-based, and model-based approaches miss important scenarios, even though they might achieve high coverage [107, 21]. For example, it is unlikely that these approaches generate a test case for booking a flight successfully. An approach should enter a valid city name and dates for inputs. It also requires considering certain preconditions related to the booking functionality, such as setting the departure date before the return date. Similarity-based approaches tackle this problem by migrating knowledge from existing test cases or

user executions. PATTERN-BASED approaches require providing patterns, which is expensive. TEST REUSE approach are the most recent and promising category of automatic testing of interactive application and they require more study.

FrUITeR studies TEST REUSE approaches by comparatively evaluating the approaches [109]. FrUITeR defines a set of metrics and automatically evaluates the quality of the generated test cases, thus targeting end-to-end effectiveness of TEST REUSE approaches. Assessing the effectiveness of TEST REUSE approaches is a great step to understands the approaches, however, knowing the factors that lead to the different effectiveness of TEST REUSE provides the essential insights to direct future researches and improve the state-of-the-art.

In this thesis, we study the important distinctive factors of TEST REUSE approaches. We identify the common conceptual components and compare them at a fine granularity level that enables us to identify optimal combination of components and define solutions for their limitations. We propose a conceptual ARCHITECTURE that encompasses components of TEST REUSE and their interactions. We investigate and evaluate the design choices of the current TEST REUSE approaches by using the ARCHITECTURE. We propose a framework that integrates different instances of the main components to systematically explore the possible combinations of the instances. In our evaluation framework we consider instances from different sources: available instances in the current TEST REUSE approaches, alternative instances from other research areas, and new instances that we propose. We recommend the best instances to be combined to create effective TEST REUSE approaches based on our empirical evaluation, and provide practical insights about TEST REUSE components.

The ARCHITECTURE and the frameworks that we introduce indicates new research directions towards test reuse approaches that combine new instances of some components with currently available instances of others. The insights from our experiments offers practical guidelines about TEST REUSE approaches.

Chapter 3

Test Reuse Architecture

In this chapter, we introduce the ARCHITECTURE *that we propose, and discuss the instances that we considered for each component of the architecture. We derived the* ARCHITECTURE *taking advantage of both a thorough inspection of the source code of the state-of-the-art* TEST REUSE *approaches that are publicly available and a complete replica of the experiments reported in the papers. We considered a set of common functionalities between the state-of-the-art* TEST REUSE *approaches and we identified input and output of each component.*

Figure 3.1 overviews the GUI TEST REUSE ARCHITECTURE in a coarse grain level. GUI TEST REUSE combines *semantic matching of GUI events* with *test generation*. Semantic matching of GUI events identifies semantically similar events across source and target apps. Test generation exploits the similarities identified with semantic matching to migrate GUI test cases from the source to the target app. TEST REUSE can be abstracted into two coarse grain components: TEST GENERATOR and SEMANTIC MATCHER. We describe the two components in Section 3.1.

TEST REUSE approaches match semantically similar GUI events across apps. The semantic matching captures the event semantics, while abstracting the implementation details. Indeed, two different apps might implement the same logical action with different widgets (for instance, a button in one case and an image button in another). Intuitively, TEST REUSE approaches aim to generate test cases for the target app that maximize the number of *semantically similar events*, possibly in the order prescribed by the source test.

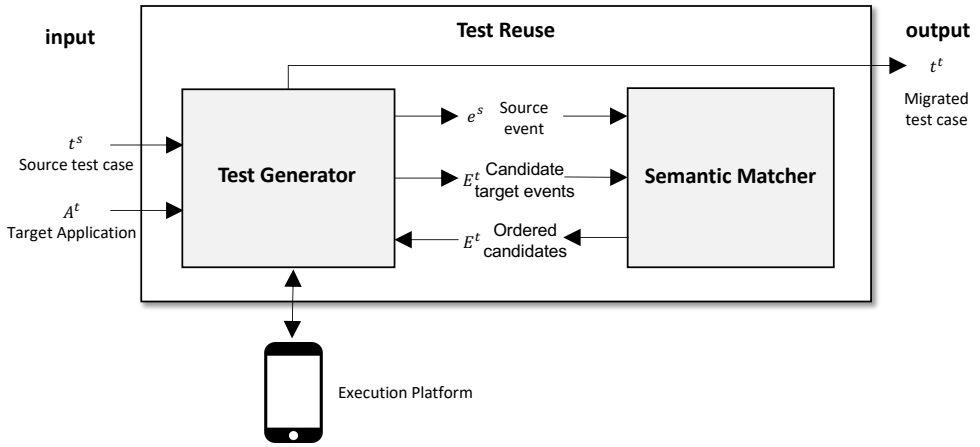


Figure 3.1. TEST REUSE Overview

3.1 Architecture Workflow

In this section, we describe the TEST REUSE ARCHITECTURE that characterizes TEST REUSE across ANDROID applications and its interaction. We discuss the implementation of the different components, and report the choices that we used in our experimental study. We indicate the alternative implementation of each component, and discuss the choices that characterize the current approaches, ATM, CRAFTDROID and ADAPTDROID. The ARCHITECTURE allows us to identify and compare combinations of different choices for each component, and identify SEMFINDER, a new SEMANTIC MATCHING ALGORITHM that supersedes current approaches.

Figure 3.2 shows the ARCHITECTURE of TEST REUSE at a fine grain level. Given a test case $t^s = \{e^s_0, e^s_1, \cdots e^s_n\}$ from a source application, the TEST GENERATOR explores the target application A^t to find a match for each event in t^s, and it generates a test case t^t for the target application. When the TEST GENERATOR does not find a match of an event because of different implementations of the functionality in $A^s and A^t$, it skips the event. The TEST GENERATOR retrieves a set $E^t = \{e^t_0, e^t_1, \cdots e^t_n\}$ of candidate events from both the current state and TARGET APPLICATION MODEL, and queries the SEMANTIC MATCHER to sort events according to their similarity with the events $e^s \in t^s$. The TARGET APPLICATION MODEL is the GUI model containing GUI states and events which lead to the transition between states. The SEMANTIC MATCHER sorts the candidate events e^t_i according to the similarity score $\langle e^s, e^t_i \rangle$ that it computes by aggregating the scores that it retrieves from the WORD EMBEDDING MODEL for each pair of attributes in the events descriptors ($Score(txt^s, txt^t)$ in Figure 3.2). The TEST REUSE ARCHITEC-

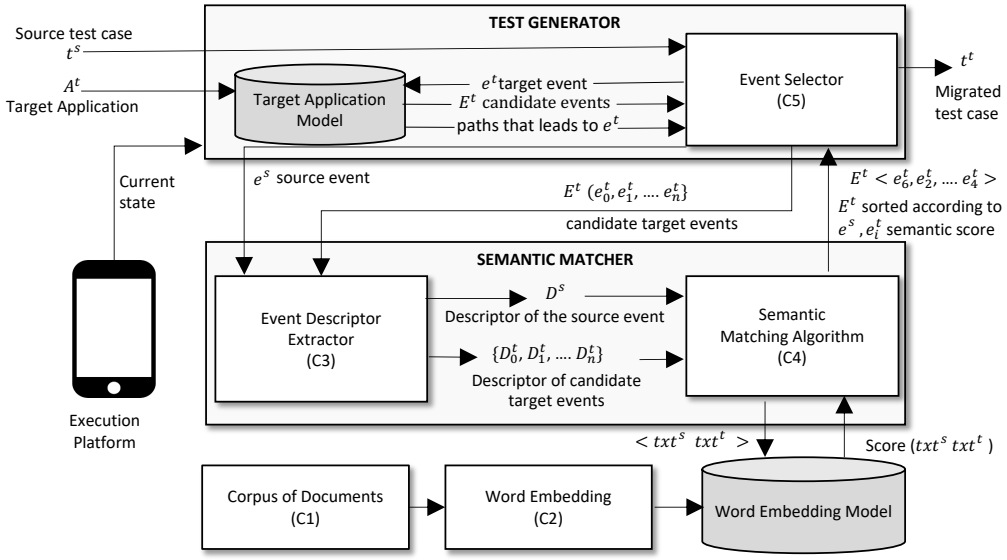


Figure 3.2. TEST REUSE ARCHITECTURE

TURE combines five main components:

(C1) CORPUS OF DOCUMENTS is the set of documents that WORD EMBEDDING uses to build a WORD EMBEDDING MODEL.

(C2) WORD EMBEDDING creates a WORD EMBEDDING MODEL that encodes the semantic space of words and sentences, by mapping words and sentences to vectors that encode the semantic distance of corresponding elements in the CORPUS OF DOCUMENTS.

(C3) EVENT DESCRIPTOR EXTRACTOR extracts the (textual) descriptors D of both the source event e^s and the set of candidate target events $E^t = \{e_0^t, e_1^t, \cdots e_n^t\}$ from the GUI states.

(C4) SEMANTIC MATCHING ALGORITHM returns the set E^t of elements sorted according to the similarity score of the descriptors of the target $\{D_0^t, D_1^t, \cdots D_n^t\}$ and source D^s events.

(C5) EVENT SELECTOR returns a test case $t^t = \langle e_0^t \ldots e_n^t \rangle$ for the target application A^t, where the event e_i^t either matches an event e_i^s in the test case t^s of the source application A^s or complete the sequence of events to obtain an executable test.

We refer to the implementations of TEST REUSE components as instances. A combination of the instances for the SEMANTIC MATCHER's component composes a SEMANTIC MATCHING CONFIGURATION.

3.2 CORPUS OF DOCUMENTS

The quality of a WORD EMBEDDING model depends on the CORPUS OF DOCUMENTS used to train the model. There are two characteristics that the CORPUS OF DOCUMENTS should have to obtain an effective WORD EMBEDDING model. The corpus shall both include as many distinct words as possible, as the model cannot compute similarity scores of words not represented in the vector space (Out-of-Vocabulary issue ([15])), and reflect the way mobile apps commonly use words. Indeed, a word can have a different meaning depending on the context of usage, and WORD EMBEDDING models trained with domain-specific corpora often outperform those trained with general corpora ([51]).

We considered both general and mobile apps specific corpora, to study and quantify the importance of the context of usage. Our study considers four corpora of English documents that are available in our replication package:

Blog Authorship Corpus (BLOGS) ([90]) that consists of 681,288 posts from 19,320 bloggers. This is a well-known corpus often used by the NLP and information science communities ([91, 1]).

User Manuals of ANDROID apps (MANUALS) ([10]) that consists of the user manuals of 500 ANDROID applications. This corpus was built by the authors of ATM ([10]), who used it to train a WORD2VEC WORD EMBEDDING model for running ATM.

Apps Descriptions (GOOGLE-PLAY) that consists of the English descriptions of 900,805 ANDROID apps in the Google Play Store. We constructed this corpus by crawling the list of similar apps of each crawled page. We used as seeds of the crawler the pages of the apps returned by searching random words in the Google Play search bar.

Domain specific corpora (TOPICS): We propose to use a set of word embedding models that are built by the corpus of documents specific to migration scenarios. It is well known to NLP community that specialized word embedding models can affect down stream tasks positively ([51]); In our case, semantic matching is the down stream task. In fact, same words have different meaning in different domains. Following this intuition, we hypothesis that we can further improve the semantic matching by exploiting word embedding models trained on specialized

corpora. We used topic modeling to partition the GOOGLE-PLAY corpus into finer-grained corpora. We build word embedding models for each cluster and use them for semantic matching. In more details, for each migration scenario we use the description of the source application to identify the domain and we used a word embedding model that is built by corpus of the documents with the same domain.

3.3 WORD EMBEDDING

WORD EMBEDDING ([75]) is a class of unsupervised language modeling and feature learning techniques that map words and sentences from a CORPUS OF DOCUMENTS to vectors of real numbers ([99]). A WORD EMBEDDING model assigns each word in the corpus to a unique vector in the space. Words that share common contexts in the corpus are close in the space. TEST REUSE approaches use WORD EMBEDDING models to identify semantically similar, although syntactically different words that independent developers may use to name actions with similar semantics. We experimented with the WORD EMBEDDING techniques that are most commonly used in software engineering ([44]).

WORD2VEC ([75]): one of the most popular WORD EMBEDDING techniques developed in 2013 in Google. It implements a two-layer neural network that is trained to reconstruct linguistic contexts of words.

Global Vectors (GLOVE) ([83]): a probabilistic technique that learns vectors or words from their co-occurrence information (how frequently they appear together in the corpus).

Word Mover's distance (WM) ([47]): a WORD EMBEDDING technique based on the observation that semantic relationships are often preserved in vector operations on WORD2VEC models. For instance, `vector(London) - vector(England) + vector(Germany)` is close to `vector(Berlin)`. WM exploits this property by finding the minimum *traveling distance* between strings ([47]). As such, WM considers distance between strings (one or more words) ([99]) and not only among pairs of words like the distances based on WORD2VEC or GLOVE ([99]). In the context of TEST REUSE this could be useful, because event descriptors often contain multiple words ([10, 53]). WM returns an integer greater than zero, that we normalize from 0 to 1, with a standard normalization $1/(1+\mathrm{WM}(\mathrm{txt}^s, \mathit{txt}^t))$.

FASTTEXT (Fast) ([15]): an extension of WORD2VEC developed in Facebook. While WORD2VEC treats words as the smallest unit to train on, FASTTEXT learns vectors for the n-grams that are found within each word. FASTTEXT computes the vector

of a word as the sum of its n-grams. For example, the word "aquarium" has the n-grams: "aqu/qua/uar/ari/riu/ium". FASTTEXT is designed to alleviate the Out-of-Vocabulary issue ([15]). In fact, even if the word "aquarius" is not present in the corpus, FASTTEXT would embed "aquarius" near to "aquarium" because they share five n-grams.

Bidirectional Encoder Representations from Transformers (BERT) ([26]): a context-sensitive WORD EMBEDDING technique that infers the meaning of words from its surroundings, by training a model on 15% of masked words in sentences. While directional models read the text input sequentially either left-to-right or right-to-left, BERT reads the entire sequence of words at once, thus allowing the model to learn the context of a word in its left and right surrounding.

Neural Network Language Model (NNLM) ([5]): a family of neural network techniques that learn WORD EMBEDDING models jointly with a language model. The model is expressed as a function that captures the distribution of sequences of words in a natural language. The language model estimates the probability of words occurring after a prefix. Thus, NNLM are context-sensitive. In our study, we consider the NNLM technique proposed by Google ([122]).

Universal Sentence Encoder (USE) ([16]): a context-sensitive WORD EMBEDDING technique that Google proposes in two variants, *Transformer*-based and *Deep Averaging Network*-based, that privilege accuracy and consumption of computing resources, respectively. We used the *Deep Averaging Network* variant in our study.

Both ATM and CRAFTDROID rely on models built with WORD2VEC, ADAPTDROID works with Word Movers Distance.

3.4 EVENT DESCRIPTOR EXTRACTOR

The EVENT DESCRIPTOR EXTRACTOR gets the descriptors of the events that SEMANTIC MATCHING ALGORITHM needs to compute the similarity among the source event e^s and candidate target events $\langle e^t_0, e^t_1, \cdots e^t_n, \rangle$. An event descriptor D is a set of textual *attributes* $\{a_1, a_2 \cdots a_m\}$ in the GUI states.

Each attribute is a $\langle type, value \rangle$ pair, for instance $\langle text, press\ ok \rangle$. Our study considers all the attribute types used in ATM, CRAFTDROID, and ADAPTDROID ([53, 10, 68]). All approaches consider both *primitive* and *derived* attributes. *Primitive* attributes are directly associated with the widget of the event

under consideration. *Derived* attributes are attributes that are not directly associated with the widget itself, but to some near widgets ([8]). For instance in the second window of Figure 2.1 (A), the attributes of widget e_2^s are empty, the corresponding field in the window is blank, and we infer the semantics of the widget from the text attribute "Price" of a neighbor widget in the same window.

The *primitive* attributes of a widget *w* are:

text, the visible label associated with *w* (xml attribute `android:text`).

content-description, a textual description of *w* that is not visible in the GUI. It is often used by ANDROID Accessibility APIs as alternate text for describing the widget to visually impaired users (xml attribute `android:contentDescription`).

hint, a textual description of *w* that is used in editable widgets to help the user to fill the correct content (xml attribute `android:hint`).

resource-id, the unique identifier of *w* that developers assign to each widget to reference them in the code (xml attribute `android:id`).

file-name, the name of the file associated with *w*. For example, the name of the image file associated with a widget.

activity-name, the name of the ANDROID activity of the widget *w*.

Both ATM and ADAPTDROID define derived attributes from the spatial positions of the widgets ([10, 68]). CRAFTDROID defines derived attributes from the hierarchical structure of the ANDROID GUI states ([53]), in which widgets have a parent-child-sibling relationship. The *parent* element directly precedes the *child* element in the hierarchy, and *siblings* elements share the same parent.

The derived attributes of a widget *w* are:

parent-text, the *text attribute* of the parent widget of *w*.

sibling-text, the *text attribute* of the sibling widget before *w* in the widget hierarchy.

neighbor-text, the *text attribute* of the widget closest to *w* in the page within a given distance.

Some *attributes* of a widget can be undefined (empty). For example, most widgets lack the *hint* or *content-desc attributes*.

In our experiments we consider the groups of *attributes* of ATM ("A" in Table 3.1), CRAFTDROID ("C" in the table), their intersection (A $\cap$ C) and union (A $\cup$ C). The *attributes* of ADAPTDROID are the same of ATM, except for *hint* that ATM considers, and ADAPTDROID does not, and that is empty in our case studies. These groups allow us to evaluate also the impact of sets attributes that are used by an approach only. For example, we can evaluate the impact of the descriptors

Table 3.1. Groups of event descriptors

attribute category	attribute type	ATM A	ADAPTDROID A	CRAFTDROID C	intersection A ∩ C	union A ∪ C
primitive	text	✓	✓	✓	✓	✓
	resource-id	✓	✓	✓	✓	✓
	content-desc	✓	✓	✓	✓	✓
	hint	✓	✓	✓	✓	✓
	file-name	✓	✓			✓
	activity-name			✓		✓
derived	neighbor-text	✓	✓			✓
	parent-text			✓		✓
	sibling-text			✓		✓

Algorithm 1: Semantic Similarity Calculator

Input: two sentences txt^s and txt^t, a word embedding model $\mathscr{M}$, aggregator $aggr \in$ {avg, sum}
Output: similarity score between txt^s and txt^t

```
function GETSIMSCORE
    ⟨txt^s, txt^t⟩ ← PREPROCESSING(txt^s, txt^t)
    switch M do
        case model at "word" level (WORD2VEC, GLOVE, FASTTEXT) do
            score[][] ← ∅
            for each word wd_1 ∈ txt^s do
                for each word wd_2 ∈ txt^t do
                    score[wd_1][wd_2] ← COSINESIM(M(wd_1), M(wd_2))
            mappedScores ← GETMATCHEDWORDS(score[][])
            return aggr{mappedScores}
        case model at "sentence" level (WMD, BERT, NNLM, USE) do
            return SIM(M( txt^s), M( txt^t))
```

neighbor-text and *file-name*, by comparing the results of groups "A" and "A ∩ C". The two attributes are available in the group "A" and they are absent in group "A ∩ C" and if group "A" performs better than the other group, it means *neighbor-text* and *file-name* attribute have positive impact on the semantic matching.

3.5 SEMANTIC MATCHING ALGORITHM

SEMANTIC MATCHING ALGORITHM returns the set of candidate target events E^t that correspond to a source event e^s, sorted according to the similarity scores between D^s and D^t_i, where D^s is the descriptor of the source event e^s, and D^t_i are the descriptors of the event $e^t_i \in E^t$.

In our study we consider the SEMANTIC MATCHING ALGORITHM of ATM, CRAFTDROID, ADAPTDROID, as well as SEMFINDER, a novel SEMANTIC MATCHING ALGORITHM that we propose. All the algorithms compute the semantic similarity scores among the attribute values of the source and target descriptors using a WORD EMBEDDING model $\mathscr{M}$. All algorithms compute the semantic similarity of values of the textual attributes of the source and target descriptors, txt^s and txt^t, with Algorithm 1 (Function GETSIMSCORE). The function GETSIMSCORE computes a real number that expresses the similarity score between txt^s and txt^t. The, function GETSIMSCORE first pre-processes the strings by removing stop words, lemmatizing the strings, and splitting words originally in camel case notation (Line 2). It then scores the similarity of the pre-preprocessed strings with respect to either a word (WORD2VEC, GLOVE, FASTTEXT) or a sentence (WM, BERT, NNLM, USE) level model $\mathscr{M}$.

Word level models: Function GETSIMSCORE computes the cosine similarity of `vector(`wd_1`)` and `vector(`wd_2`)` for all possible pairs of words of the two strings $\langle wd_1 \in txt^s, wd_2 \in txt^t \rangle$ (Lines 6–8). Then it identifies the best match among the pairs as the pair with the highest cosine similarity, where(Line 9). It finally returns the similarity score computed with the input aggregation function (Line 10). ATM aggregates by summing the scores (*sum*), while CRAFTDROID, ADAPTDROID, and SEMFINDER aggregate by averaging the scores (*avg*). For example, The algorithm 1 considers the following matrix for the two strings, $txt^s =$ "wd_1, wd_2, wd_3" and $txt^t =$ "wd_4, wd_5", in which cells represent semantic similarity score of the corresponding words. The algorithm considers $\langle wd_3, wd_4 \rangle$ as a match since they have the highest score (0.8). The pair of $\langle wd_3, wd_5 \rangle$ has the next highest score (0.6), but they cannot be matched since wd_3 is already matched. The algorithm considers $\langle wd_2, wd_5 \rangle$ as a match since they have the next highest score (0.4). The algorithm ignores wd_1, since no other word is available in txt_t. The algorithm results similarity score of 1.2 and 0.6 for $\langle txt^s, txt^t \rangle$ using *sum* and *average* functions respectively.

$$\begin{array}{c} \\ wd_4 \\ wd_5 \end{array} \begin{array}{c} \begin{array}{ccc} wd_1 & wd_2 & wd_3 \end{array} \\ \begin{pmatrix} 0.15 & 0.2 & 0.8 \\ 0.3 & 0.4 & 0.6 \end{pmatrix} \end{array}$$

Algorithm 2: Semantic Matching Algorithm of ATM

Input: source descriptor D^s, set of target descriptors $\{D_0^t, D_1^t, \cdots D_n^t\}$
Output: sorting of E^t based on the semantic similarity with e^s

```
13 function ATM
14   descScores[] ← ∅
15   label_1^s ← GETFIRSTDEF(D^s[neighbor-text], D^s[resource-id] + D^s[file-name])
16   label_2^s ← GETFIRSTDEF(D^s[text], D^s[content-desc], D^s[hint])
17   for each i from 1 to n do
18     if type(e^s) = type(e_i^t) then
19       label_1^t ← GETFIRSTDEF(D_i^t[neighbor-text], D_i^t[resourse-id] + D^s[file name])
20       label_2^t ← GETFIRSTDEF(D_i^t[text], D_i^t[content-desc] or D_i^t[hint])
21       scores ← ∅
22       for each label^s ∈ {label_1^s, label_2^s} do
23         for each label^t ∈ {label_1^t, label_2^t} do
24           add GETSIMSCORE(label^s, label^t, M, "sum") to scores
25       descScores[D_i^t] ← max{ scores }
26   return E^t sorted by descScore
```

Sentence level models: Function GETSIMSCORE queries the model $\mathscr{M}$ with the strings as a whole. Notably, both ATM and CRAFTDROID use models at word level, we add Lines 11 and 12 to make the algorithm compatible with the sentence level WORD EMBEDDING models that we considered in our study.

We now describe the four algorithms and their key differences.

Semantic Matching of ATM ([10]) Lines 13 to 26 of Algorithm 2 encode the SEMANTIC MATCHING ALGORITHM of ATM. The algorithm collects two textual representations of the source event: $label_1^s$ (Line 15) and $label_2^s$ (Line 16). The algorithm initializes $label_1^s$ to the first *defined* attribute among $\langle$*neighbor-text*, *resource-id* + *file-name*$\rangle$ in D^s. If all of such attributes are undefined, the algorithm initializes $label_1^s$ to the empty string. ATM extracts the *neighbor-text* attribute only for filling events, while for clicking events it considers the attribute as undefined. The algorithm initializes $label_2^s$ to the first *defined* attribute among $\langle$*text*, *content-desc*, *hint*$\rangle$ in D^s (Line 16). For each event $e_i^t \in E^t$ that has the same type of e_s (either both filling or both clicking events), the algorithm collects $label_1^t$ and $label_2^t$ in the same way it collects $label_1^s$ and $label_2^s$, respectively. Then, the algorithm invokes Function GETSIMSCORE (Algorithm 1) for each combination of $\langle label^s \in \{label_1^s, label_2^s\}, label^t \in \{label_1^t, label_2^t\}\rangle$, using "sum" as aggregation function. The algorithm assigns the highest returned value to the score of the current target event ($score[D_i^t]$ Line 25), and sorts E^t based on the final scores

Algorithm 3: Semantic Matching Algorithm of CRAFTDROID

Input: source descriptor D^s, set of target descriptors $\{D_0^t, D_1^t, \cdots D_n^t\}$
Output: sorting of E^t based on the semantic similarity with e^s

```
27 function CRAFTDROID
28     descScores[] ← ∅
29     for each i from 1 to n do
30         if type(e^s) = type(e_i^t) then
31             scores ← ∅
32             for each a_i ∈{ text ∪ hint, resource-id, content-desc, actitivty-name, parent-text,
                 sibiling-text} do
33                 add GETSIMSCORE(D^s[a_i], D_i^t[a_i], M, "avg") to scores
34
35             descScores[D_i^t] ← avg{ scores }
36     return E^t sorted by descScore
```

(Line 26).

Semantic Matching of CRAFTDROID ([53]) Lines 27 to 36 of Algorithm 3 encode the SEMANTIC MATCHING ALGORITHM of CRAFTDROID. For each target event e_i^t of the same type of e^s (either both filling or both clicking events), CRAFTDROID gets the similarity scores of their descriptor attributes (Line 32) and adds them to List *scores*. The algorithm only compares corresponding attributes. For example, it compares *resource-id* of the source descriptor only to *resource-id* of the target descriptor. CRAFTDROID computes the final score of the current target descriptor as the average of List *scores* (Line 35).

Semantic Matching of ADAPTDROID ([68]) Lines 37 to 50 of Algorithm 4 encode the SEMANTIC MATCHING ALGORITHM of ADAPTDROID. For each event $e_i^t \in E^t$ that has the same type of e_s (either both filling or both clicking events), ADAPTDROID builds two strings txt^s and txt^t, by concatenating the values of the attributes of D^s (separated by a white space) and D^t as follows: (step i) It adds $\langle$*text*, *neighbor*$\rangle$ in D^s to txt^s (Line 41); (step ii) If txt^s is still empty, it adds $\langle$*file-name*$\rangle$ to the txt^s (Line 45); (step iii) If either txt^s remains empty or there is an associated file (*file-name* not empty), it adds $\langle$*resource-id*, *content-desc*$\rangle$ to the txt^s (Line 47); (step iv) It adds $\langle$*activity-name*, *hint*, *sibling-text*, *parent-text*$\rangle$ (Line 48). It computes the score of the current target descriptor by averaging the similarity scores between txt^s and txt^t (Line 49). ADAPTDROID SEMANTIC MATCHING ALGORITHM balances the effect of too many attributes that may lead to a noisy text and too few attributes that may lead to insufficient semantic information, by adding attributes in a specific order, only if necessary. For example, a non-empty

Algorithm 4: Semantic Matching Algorithm of ADAPTDROID

Input: source descriptor D^s, set of target descriptors $\{D_0^t, D_1^t, \cdots D_n^t\}$
Output: sorting of E^t based on the semantic similarity with e^s

```
37 function AdaptDroid
38     descScores[] ← ∅
39     for each i from 1 to n do
40         if type(e^s) = type(e_i^t) then
41             ⟨txt^s, txt^t⟩ ← ⟨∅, ∅⟩
42             for each j ∈{s, t} do
43                 txt^j ← D^j[text] ∪ D^j[neighbor-text]
44                 if txt^j = ∅ then
45                     txt^j ← D^j[file-name]
46                 if txt^j = ∅ or D^j[file-name]! = ∅ then
47                     txt^j ← D^j[resource-id] ∪ D^j[content-desc]
48                 txt^j ← D^j[activity-name] ∪ D^j[hint] ∪ D^j[parent-text] ∪ D^j[sibling-text]
49             descScores[D_i^t] ← getSimScore(txt^s, txt^t, M, "avg")
50     return E^t sorted by descScore
```

Algorithm 5: Semantic Matching Algorithms of SEMFINDER

Input: source descriptor D^s, set of target descriptors $\{D_0^t, D_1^t, \cdots D_n^t\}$
Output: sorting of E^t based on the semantic similarity with e^s

```
51 function SemFinder
52     descScores[] ← ∅
53     for each i from 1 to n do
54         if type(e^s) = type(e_i^t) then
55             ⟨txt^s, txt^t⟩ ← ⟨∅, ∅⟩
56             for each a_i ∈{ text, resource-id, content-desc, hint, file-name, neighbour-text} do
57                 txt^s ← txt^s ∪ D^s[a_i]
58                 txt^t ← txt^t ∪ D^t[a_i]
59             descScores[D_i^t] ← getSimScore(txt^s, txt^t, M, "avg")
60     return E^t sorted by descScore
```

⟨*file-name*⟩ attribute usually indicates the presence of an *ImageButton*, and consequently the need of additional textual information to get an accurate semantics.

Semantic Matching of SEMFINDER Lines 51 to 60 of Algorithm 5 encode the SEMANTIC MATCHING ALGORITHM SEMFINDER that we propose in this PhD work. For each event $e_i^t \in E^t$ that shares the type of e_s (either both filling or both clicking events), SEMFINDER builds two strings txt^s and txt^t. It (i) builds txt^s by concate-

nating all the values of the attributes of D^s (separated with a space), (ii) builds txt^t with the values in D_i^t, (iii) prunes words repeated in the same string, (iv) gets the similarity score between txt^s and txt^t and uses *average* in case of word level WORD EMBEDDING model, (v) assigns the result to the final score of the current target descriptor (Line 59). The SEMFINDERS intuition is that even though some attributes could be sometime more important than others, strict prioritization could result in information loss. While collecting the values of a fixed subset of attributes and concatenating them without prioritization could be a safer option. A relevant word that is important for matching may be the value of different attributes in the two applications. Both strictly prioritizing and comparing subsets of attributes may miss the semantic relation of the word in the two apps. Sentence based approaches that concatenate words benefit from the full capacity of sentence level embedding that considers contextual information from neighbors words in a sentence. The EVENT DESCRIPTOR EXTRACTOR of SEMFINDER is the only responsible to select a fixed subset of attributes.

ATM, CRAFTDROID, ADAPTDROID, and SEMFINDER share the general framework, and differ in three main aspects: the type of attributes they consider, as we discuss in Section 3.4, the way they aggregate the similarity scores of multiple pairs of attributes, and the way they aggregate the similarity scores of word-level models.

Attributes of source and target descriptors The algorithms compare attributes by considering either specific combinations of types of source and target attributes or a priority between attributes. CRAFTDROID selects attributes by type only, and compares only attributes with the same type. ATM selects attributes by both type and priority, and compares attributes of a subset of combinations of types, and prioritizes attributes according to their order, by considering the first not-empty attribute. ADAPTDROID selects attributes by priority only, according to the types of the attributes. SEMFINDER compares the set of attributes as a whole. For example, let us consider a source event e^s, with descriptor $D^s = [$*text*: "address"] that indicates an attribute of type *text*, and a target event e_1^t, with a descriptor $D_1^t = [$*neighbor-text*: "search", *resource-id*: "location"] that reports two attributes. CRAFTDROID does not compare any pairs of attributes, since the types are different. ATM compares the pair "*address*"–"*search*" since it selects the first non-empty attribute of the target event based on the predefined order. ADAPTDROID also compares the pair "*address*"–"*search*" that it selects according to the higher priority of type "*neightbor-text*" than "*resource-id*". SEMFINDER compares the pair "*address*"–"*search location*".
Table 3.2 gives an example of how SEMANTIC MATCHING ALGORITHMS calculate

the similarity score of two target events, e_1^t and e_2^t, and the source event e^s using a pre-trained WORD2VEC model. In case of other WORD EMBEDDING models the process would be the same, but the numbers varies. In the example, e^s and e_1^t are semantically the correct match. Source and target events have different type of attributes, thus CRAFTDROID do not compare any pairs of attributes and results score of 0 for both target events. ATM only considers combination of $label_2^s$ and $label_1^t$ based on the available attributes. Other pairs of labels contains one or two empty labels and receive score of 0. For example, empty values of *neighbor-text, resource-id* and *file-name* attributes compose an empty value for $label_1^s$ and any combination of $label_1^s$ will result 0 score. In the last step, ATM uses *max* aggregation function and chooses the only non-zero combination of labels as the final score. ADAPTDROID creates txt^s by the only available attribute of e^s and it selects the *neighbor-text* attribute for the target events since the attribute has a higher priority over *resource-id* attribute. SEMFINDER considers all the available attribute to create txt^s and txt^t. ADAPTDROID and SEMFINDER consider only one string for each source and target event pair (txt^s, txt^t) and that makes aggregation function irrelevant. SEMFINDER is the only SEMANTIC MATCHING ALGORITHM that chooses the correct match in the example.

Table 3.2. Example of attribute selection by each SEMANTIC MATCHING ALGORITHM

Source Event [a]	Algorithm	Target Event [b]	Pairs for similarity score	Aggregtion	Score [c]
e^s	CRAFTDROID	e_1^t	No pairs	average	0
		e_2^t	No pairs	average	0
	ATM	e_1^t	$\langle label_2^s$:address, $label_1^t$:search$\rangle$= 0.13, other pairs get score of 0	max	0.13
		e_2^t	$\langle label_2^s$:address, $label_1^t$:number$\rangle$= 0.18, other pairs get score of 0	max	0.18
	ADAPTDROID	e_1^t	$\langle txt^s$:address, txt^t:search$\rangle$= 0.13	NA	0.13
		e_2^t	$\langle txt^s$:address, txt^t:number$\rangle$= 0.18	NA	0.18
	SEMFINDER	e_1^t	$\langle txt^s$:address, txt^s:search location$\rangle$= 0.22 s	NA	0.22
		e_2^t	$\langle txt^s$:address, txt^t:number one$\rangle$ = 0.18	NA	0.18

[a] The descriptor of e^s is $D^s = [text : \text{address}]$
[b] The descriptor of e_1^t and e_2^t are $D_1^t = [neighbor$: search, id: location] and $D_2^t = [neighbor$: number, id: one], respectively
[c] SEMANTIC MATCHING ALGORITHMS used WORD2VEC pre-trained model to calculate similarity scores

Similarity score aggregation of attribute types ATM and CRAFTDROID aggregate similarity score of attributes, while ADAPTDROID and SEMFINDER do not. ATM aggregates the similarity score of multiple pairs of attribute types by using *max*imum (Line 25 of Algorithm 2), while CRAFTDROID aggregates them by *average* (Line 34 of Algorithm 3). ATM aggregation by maximum misses pairs of attributes that may be relevant but with low score, while CRAFTDROID s aggregation by average does not. For example, let us consider a source event e^s, with

descriptor D^s= [*resource-id*: "button", *text*: "save"], and two target events e_1^t and e_2^t with descriptors D_1^t =[*resource-id*: "button", *text*: "save"] and D_2^t=[*resource-id*: "button", *text*: "exit"], respectively. Maximum assigns the same similarity score of 1 to both target events. Average assigns similarity scores of 1 and 0.56 to e_1^t and e_2^t using a pre-trained WORD2VEC model, respectively. SEMFINDER and ADAPTDROID consider attributes combined in a string, and do not combine the similarity scores of multiple pairs of attribute types.

Similarity scores aggregation of word embedding models ATM aggregates the similarity scores of words in a string by summing the similarity scores of words (Line 24 of Algorithm 2). CRAFTDROID aggregates the scores of words by *average* (Line 33 and 49 of Algorithm 3). ADAPTDROID works directly at sentence level with Word Mover. We use *average* to aggregate the similarity scores of words when we combine the ADAPTDROID and SEMFINDER SEMANTIC MATCHING ALGORITHM with word level WORD EMBEDDING. *Sum* privileges (assign high score to) strings with many words, and may assign higher score to two attributes with many unrelated words than to two attributes with fewer highly related (semantically similar) words, as the semantic score of two words in the model is always positive. For example, let us consider a source event e^s with descriptor D^s = [*text* : "new todo task"], and two candidate events e_1^t and e_2^t with descriptors D_1^s = [*text:* "add todo"] and D_2^t = [*text*: "add todo reminder"], respectively. A pre-trained WORD2VEC model scores the pairs of words in the attributes as $\langle$todo, todo$\rangle = 1$, $\langle$new, add$\rangle = 0.28$, $\langle$ task, reminder$\rangle = 0.14$. CRAFTDROID aggregates the scores by average: $score(e_1^t) = 0.64$ and $score(e_2^t) = 0.47$, while ATM aggregates by sum $score(e_1^t) = 1.28$ and $score(e_2^t) = 1.4$.

3.6 EVENT SELECTOR

The EVENT SELECTOR builds a test case t^t for the target application A^t by chaining the candidate events that the SEMANTIC MATCHING ALGORITHM suggests as semantic matches of the events in the source test case t^s. The EVENT SELECTOR incrementally processes the events e_i^s of the source test case t^s. It retrieves a set E^t of candidate events that correspond to the current event e_i^s from both the current state of the target application A^t and the TARGET APPLICATION MODEL.

ATM retrieves the candidate events from the current state of the app under test, and considers the events in the TARGET APPLICATION MODEL only if the SEMANTIC MATCHER does not find any event in the current state with a semantic similarity score above a constant threshold. CRAFTDROID retrieves the candidate events from both the current state and the TARGET APPLICATION MODEL. ADAPT-

DROID retrieves events from the current state, and selects the candidate events as the events with a semantic similarity score above a constant threshold. If ATM and CRAFTDROID do not find events in the current state, the EVENT SELECTOR looks for an event that belongs to a state different from the current state, and they selects a sequence of events that head to the candidate event, from the TARGET APPLICATION MODEL (*ancillary events* in FrUITeR ([109]), *leading events* in CRAFTDROID ([53])).

Once identified a matching event e^t for e_i^s, EVENT SELECTOR adds the event (and the leading events, if any) to the test case for the target application t^t, and computes the next current state of the target application A^t by executing the added events. Both ATM and CRAFTDROID consider backtracking: if the TEST GENERATOR cannot proceed, it rolls back to previously selected events and continues with different choice of events. ATM and CRAFTDROID consider backtracking differently. ATM backtracks when it cannot find a matching event for asource event e_i^s. CRAFTDROID backtracks when it cannot find a path to a target candidate that it has selected from the Target Application Model. When ATM and CRAFTDROID do not find a matching event, they skip e_i^s and proceed with e_{i+1}^s, while ADAPTDROID randomly selects an event as next match. ADAPTDROID improves the migrated test cases t^t with a genetic algorithm that uses the test cases as the initial population and the TARGET APPLICATION MODEL to repair infeasible tests that the crossover operations generate during the evolution. In our study, we considered the implementations of EVENT SELECTOR of both CRAFTDROID and ATM. We excluded the EVENT SELECTOR of ADAPTDROID from our study as its genetic algorithm is too computationally expensive, which would have drastically limited the scale of our experiments. Nevertheless, CRAFTDROID and ATM are two state-of-the-art approaches, which are representative of TEST REUSE for ANDROID applications.

ATM EVENT SELECTOR: Lines 62 to 83 of Algorithm 6 encode the ATM EVENT SELECTOR algorithm that initializes a TARGET APPLICATION MODEL (Line 63), and iteratively looks for next events that match the input source events, until either a matching event is found or a timeout expires (call Function FINDNEXTEVENT at Line 69).

Function FINDNEXTEVENT selects the next event that corresponds to e_i^s (Line 1) by (1) looking for a matching event in the current state (Lines 5-12), (2) looking for a matching event in the TARGET APPLICATION MODEL (WTG in ATM terminology), if it does not find a matching event in the current state and the source event is a GUI event (and not an oracle event) (Lines 13-20), (3) randomly selecting an event in the current state, if it does not find a matching event in the TARGET

APPLICATION MODEL (Lines 22-24), (4) moving back to the former page, after a maximum number of randomly selected events (Lines 25-27).

It computes the next current state by executing the events that FINDNEXTEVENT selects (Line 70), adds the events to a buffer (Line 70), and updates both the TARGET APPLICATION MODEL (Line 71) and the target test case t^t (Lines 73-76). It updates the target test case t^t by adding the events (Line 76) after simple syntactic checks if the events are oracle events (Line 74). If the EVENT SELECTOR cannot find a matching event within a timeout, it backtracks to the state of the previously matched event, and restarts from an alternative event.

Algorithm 6: ATM EVENT SELECTOR Algorithm

Input: source test case t^s, set of source events $\{e_1^s, e_2^s, \dots e_n^s\}$
Output: migrated test case t^t, set of target events $\{e_1^t, e_2^t, \dots e_m^t\}$

```
62 function ATMEVENTSELECTOR
63     WTG ← static GUI model of target app
64     for each i from 1 to n do
65         executedEvents ← ∅
66         matched ← false
67         randomEventCounter ← 0
68         while ! timeout do
69             e_i^t, matched ← FINDNEXTEVENT(e_i^s)
70             execute(e_i^t)
71             add e_i^t to executedEvents
72             update(WTG)
73             if matched = true then
74                 if type(e_i^s) = oracle and !extraChecks(e_i^s, e_i^t) then
75                     break
76                 add executedEvents to t^t
77                 break
78         if matched = false and type(e_s) = GUI and hasAlternative(e_{i-1}^t) then
79             i ← i - 1
80             e_i^t ← getFirstAltrenative(e_i^t)
81             execute t^t
82
83     return t^t
```

CRAFTDROID EVENT SELECTOR: Lines 110 to 129 of Algorithm 8 encode the CRAFTDROID EVENT SELECTOR algorithm. CRAFTDROID initializes the TARGET APPLICATION MODEL (UITG in CRAFTDROID terminology) (Line 111), and iteratively generates test cases that correspond to the input source test case t^s (Lines 112-

Algorithm 7: Find Next Event Algorithm

Input: A source event e^s
Output: A target event e^t, A boolean that indicates if the e^t is a match for e^s

```
1  function FINDNEXTEVENT
2      e^t ← ∅
3      matched ← false
4      randomEventCounter ← 0
5      currentState ← current GUI state of the target application
6      candidates ← getCandidates(currentState, e^s)
7      sortedCandidates ← SEMANTIC MATCHER (candidates, e^s)
8      sortedCandidates ← getAboveThreshold(sortedCandidates)
9      if sortedCandidates ≠ ∅ then
10         e^t ← firstItem(sortedCandidates)
11         setAltrenatives(e^t, sortedCandidates − e^t )
12         matched ← true
13     if e^t = ∅ and type(e^s) = GUI then
14         candidates ← getCandidates(WTG, e^s)
15         sortedCandidates ← SEMANTIC MATCHER (candidates, e^s)
16         sortedCandidates ← getAboveThreshold(sortedCandidates)
17         if sortedCandidates ≠ ∅ then
18             temp ← firstItem(sortedCandidates)
19             path ← shortestPathTo(WTG, temp)
20             e^t ← fistItem(path)
21     if e^t = ∅ then
22         if randomEventCounter ≤ RANDOM_EVENT_THRESHOLD then
23             e^t ← selectRandomEvent(currentState)
24             randomEventCounter ← randomEventCounter + 1
25         else if type(e^s) = GUI then
26             e^t ← back
27             randomEventCounter ← 0
28     return e^t, matched
```

129), aiming to maximize a test similarity score, computed as the average similarity score of the events in the test. It terminates either when the test similarity score does not improve (Line 129) or when a timeout expires.

It generates the tests by scanning the events e^s_i in the input source test case t^s, and looking for events that match the current event in t^s. It selects all GUI events in both the current state and the TARGET APPLICATION MODEL (Line 116) and all oracle events in the current state only (Line 119), and sorts them by semantic similarity with respect to the current event e^s_i in the source test cases t^s (Line 120).

It computes the leading events for the candidate events (at the top of the *sortedCandidate* list), that is, the sequences of events that lead to a state in which the event is executable (call to GETLEADINGEVENTS at Line 122), and adds the non-empty leading event sequence to the target test case t^t (Line 126), after few simple syntactic checks for oracle events (Line 124).

Function GETLEADINGEVENTS retrieves all paths in the TARGET APPLICATION MODEL that lead to the event e^t (Line 132), and incrementally executes them starting from the shortest ones, till it finds an executable sequence of leading events to return (Lines 133-137).

Algorithm 8: CraftDroid EVENT SELECTOR Algorithm

Input: source test case t^s, set of source events $\{e_1^s, e_2^s, \dots e_n^s\}$
Output: migrated test case t^t, set of target events $\{e_1^t, e_2^t, \dots e_m^t\}$

```
110 function CRAFTDROIDEVENTSELECTOR
111     UITG ← static GUI model of target app
112     while true do
113         t^t ← ∅
114         for each i from 1 to n do
115             currentState ← current GUI state of the target application
116             if type(e_i^s) = GUI then
117                 candidates ← getCandidates([currentState, UITG], e_i^s)
118             else
119                 candidates ← getCandidates(currentState, e_i^s)
120             sortedCandidates ← SemantiMatching(candidates, e_i^s)
121             for each e^t in sortedCandidates do
122                 leadingEvents ← GETLEADINGEVENTS(e^t)
123                 if leadingEvents ≠ ∅ then
124                     if type(e_i^s) = oracle and ! extraChecks(e_i^s) then
125                         continue
126                     add leadingEvents to t^t
127                     break
128         if delta-fitness(t^t) < threshold or timeout then
129             return t^t
```

Input: A target event e^t
Output: A set of target events $E^{t'}$ that leads to a GUI state in which e^t can be excuted

```
130 function GETLEADINGEVENTS
131     pathes ←allPathesTo(UITG, e^t)
132     sortedPathes ← sort pathes by length ascendingly
133     for each path in sortedPathes do
134         isValid ← execute(path)
135         update(UITG)
136         if isValid then
137             return path + e^t
138     return ∅
```

Chapter 4

Evaluation Frameworks

In this section we present the framework we defined to evaluate the different components both in isolation and in combination, to thoroughly compare the state-of-the-art approaches, identify the impact of the different components, and devise an ideal migration framework.

4.1 Research Questions

We evaluate semantic matching both in isolation and in the context of TEST REUSE. The evaluation in isolation (research questions RQ1, RQ2, RQ3, RQ4) investigates the effectiveness of semantic matching for a broad set of configurations in controlled setup, without referring to a specific TEST REUSE approach. We evaluate semantic matching in isolation by focusing on the ability of semantic matching to match events from a data set that contains events observed during test reuse, We evaluate semantic matching independently from the migration of the test case, with evaluation metrics specific to matching of events. The evaluation in the context of TEST REUSE (research questions RQ5, RQ6, RQ7) investigates the effectiveness and impact of semantic matching when used with the test generation process. We empirically evaluated semantic matching in isolation and in the context of TEST REUSE with pairs of source test case and target application $\langle t^s, A^t \rangle$ with 95 and 89 from the test migration scenarios provided by ATM and CRAFTDROID for 30 apps, respectively.

Semantic matching in isolation

RQ1 Baseline Comparison: *Do semantic approaches based on word embedding outperform syntactic and random approaches?*

RQ2 Component Effectiveness in Isolation: *What are the most effective instances of each component on the semantic matching of events?*

RQ3 Component Impact Analysis in Isolation: *Which component types have the greatest impact on the semantic matching of events?*

RQ4 Effectiveness of domain specific word embedding models: *Do word embedding models trained by specialized corpora outperform models trained on more general corpora?*

RQ1 validates the usefulness of semantic matching for TEST REUSE by comparing the effectiveness of semantic approaches to both syntactic (EDIT DISTANCE SIMILARITY and JACCARD SIMILARITY) and random approaches. RQ2 studies the implementations of different components of semantic matching approaches, and identifies the implementations that perform best. RQ3 studies the impact of the component types on the effectiveness of semantic matching, and identifies the most impactful ones. RQ4 validates our hypothesis that clustering a general corpus of document into specialized corpora can improve effectiveness of word embedding models built by the specialized corpora in comparison to the more general corpus.

We created a data set, which consists of source events that we extracted from the subjects of ATM and CRAFTDROID studies, to answer RQ1-3. The data set consists of 337 queries that are pairs of a source event and target candidates such as $\langle e^s, E^t \rangle$.

We create domain specific models and used them as instances of the WORD EMBEDDING components to answer RQ4. In this way we obtain new SEMANTIC MATCHING CONFIGURATIONS that we evaluate as any other SEMANTIC MATCHING CONFIGURATIONS that we consider for RQ1-3. We compare the domain specific models with a new baseline as well as corpora introduced in section 3.3

We propose an evaluation framework named SEMANTIC MATCHING EVALUATOR to experiment with 337 different configurations to answer RQ1, RQ2, RQ3 in chapter 5. We will address RQ4 in chapter 6 since it requires special setup and we consider 240 configurations that use word embedding models that we trained with corpora of documents available in this thesis.

Semantic matching in the context of test reuse

RQ5 Impact of Semantic Matching in the Context of Test Reuse: *Does the effectiveness of semantic matching impact on test reuse?*

RQ6 Component Effectiveness in the Context of Test Reuse: *What are the most effective instances of each component for test reuse?*

RQ7 Component Impact Analysis in the Context of Test Reuse: *Which component types have the most relevant impact on test reuse?*

RQ5 studies the correlation between semantic matching and TEST REUSE, and identifies the combinations of configurations that achieve the most effective TEST REUSE. RQ6 studies the impact of the implementations of different components on TEST REUSE, and identifies the implementations with the best impact on TEST REUSE. RQ7 studies the impact of the component types on TEST REUSE, and identifies the critical component types. We report the results of experimenting semantic matching with two state-of-the-art TEST REUSE approaches: ATM and CRAFTDROID.

We migrate the scenarios with different SEMANTIC MATCHING CONFIGURATIONS and evaluate the generated test cases individually with the general metrics proposed in the literature [109]. We evaluate the test cases with respect to the ideal migration. We propose TEST MIGRATION EVALUATOR, an evaluation framework to migrate and assess the quality of the migrated test cases, to answer RQ5, RQ6, RQ7.

4.2 SEMANTIC MATCHING EVALUATOR

We evaluated semantic matching in isolation (RQ1, RQ2, RQ3, RQ4), by systematically evaluating all possible configurations against every individual query produced with any of the scenarios from subject applications.

We evaluated the queries with the SEMANTIC MATCHING EVALUATOR, a prototype framework that we developed in PYTHON to evaluate semantic matching queries. Our SEMANTIC MATCHING EVALUATOR is a general framework that can be configured with different choices of components' instances. We integrated instances of components in SEMANTIC MATCHING EVALUATOR as follows:

Corpus of Documents: We used four corpora that we introduce in section 3.2 to train WORD EMBEDDING models. Pre-processing of training sets impacts effectiveness of the WORD EMBEDDING models positively ([24, 48]) and we used the same operations that we considered for Algorithm 1 including: removing stop words, lemmatizing the strings, and splitting words originally in camel case notation.

Word Embedding: We built WORD EMBEDDING models with WORD2VEC, WMD, GLOVE and FASTTEXT techniques. We used default parameters of standard PYTHON

libraries for the training of the models. We did not build models with BERT, USE and NNLM, because these techniques require a non-trivial parameter tuning that goes beyond the scope of this work. We also considered seven pre-trained WORD EMBEDDING models trained for each of the techniques introduced in section 3.3 (WORD2VEC, WDM, GLOVE, FASTTEXT, BERT, USE, NNLM) that are provided by the authors of such techniques, and that are obtained with not-publicly-available corpora of documents (such as, different versions of Google News and Twitter datasets). As such, we were not able to consider such corpora as individual components, like we did for MANUALS, BLOGS, and GOOGLE-PLAY.

Semantic Matching Algorithm: We implemented all the four algorithms introduced in section 3.5 in PYTHON. After reviewing publicly available source code of ATM [1] , CRAFTDROID [2] and ADAPTDROID, we observed their SEMANTIC MATCHING ALGORITHMS are internal algorithms of the TEST REUSE components and can be hardly executed in isolation. We re-implemented ATM and CRAFTDROID algorithms by referring to their original JAVA implementation and we reused the original PYTHON code of CRAFTDROID as much as possible.

Event Descriptor Extractor: Our implementations of the EVENT DESCRIPTOR EXTRACTOR instances relies on an EXECUTOR-PLUGIN that we implemented to execute source and target test cases and retrieve events from the GUI state at runtime using the framework APPIUM. APPIUM framework facilitates interactions between testing environments and the ANDROID platform. We used the retrieved events to create a set of 337 queries. Our implementations of the EVENT DESCRIPTOR EXTRACTOR instances extract the values of the nine widget attributes in Table 3.1 from the events in the queries. We implemented our own extractors, rather than rely on the implementations of ATM or CRAFTDROID, to have a common tool to collect all the descriptors. Our event extractor retrieves all types of click and fill events that ATM and CRAFTDROID use in their experiments. We create a set of queries by executing both the source test cases and the ground truth test cases, and capturing the relevant attributes of the events. We built a query by considering a source event from a test case t^s and all actionable events that have been observed during the execution of the ground truth, when migrating the test case t^s to the target application t^t.

We denote descriptors and algorithms with the suffixes "_*d*" and _*a*, respectively. For instance, ATM_d denotes the descriptor set and ATM_a the algorithm

[1] Farnaz Behrang and Alessandro Orso. ATM implementation. `https://sites.google.com/view/apptestmigrator`. 2019.

[2] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. Craftdroid implementation. `https://github.com/seal-hub/CraftDroid`. 2019.

of ATM, respectively. We modified the SEMANTIC MATCHING ALGORITHMS to work with sets of descriptors different from the ones used in the original algorithms, by either pruning the attributes that do not belong to the set from the algorithm or appending the new attributes at the end of the *text* attribute in the algorithm. For instance, we combine the CRAFTDROID_d set with ATM_a, by appending *activity-name*, *parent-text* and *sibling-text* to the attribute *text* at Lines 16 and 20 of Algorithm 2; We combine the "intersection" set with CRAFTDROID_a, by removing *activity-name*, *parent-text* and *sibling-text* from the set of attribute types at Line 32 of Algorithm 3 By appending the attributes at the end of the *text* attribute, we comply with both ATM and CRAFTDROID: ATM prioritizes attributes by position, with the highest priority to *text*, and CRAFTDROID handles *text* jointly with the *hint* attribute (line 32 of Algorithm 2).

4.2.1 Evaluation Metrics

Our experiment for semantic matching in isolation comprises queries that score events in the target app according to their similarity with respect to events in the source test case. A query q sorts a set of input events E^t of the target app, according their similarity score with respect to an event e^s in the source test case, and returns a sorted list: $\langle e^s, E^t \rangle \xrightarrow{q} (E^t_{sorted})$. We rank each query q_i according to the position of the correct event e^t_{gt} that is the event that the query should return according to the ground truth. The $rank_i$ of a query q_i is the position of e^t_{gt} in the list sorted according to the similarity score of q_i. We rank events with the same score as the average of their positions in the list.

We consider queries that we extract from subject applications and we measured the effectiveness of SEMANTIC MATCHING CONFIGURATIONS with two metrics that we compute on the returned ranks: MRR, the Mean Reciprocal Rank ([58]); TOP1, the ratio of queries in which the rank of the correct answer is one.

The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer: 1 for first place, 1/2 for second place, 1/3 for third place and so on. The mean reciprocal rank is the average of the reciprocal ranks of our 337 queries Q.

$$\textbf{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \in (0; 1]$$

MRR is a standard statistical measure for evaluating any process that produces a list of possible responses to a query q, sorted by their probability of correctness. MRR is suitable in our context because it focuses on a single correct answer (e^t_{gt}), while other metrics like Mean Average Precision (MAP) and

Normalized Discounted Cumulative Gain (NDCG) focus on multiple correct answers ([58]).

TOP1 is the ratio of queries in which the ground truth (e^t_{gt}) is in the first position of the returned list. TOP1 is less informative than MRR, because it ignores the position of e^t_{gt} when it is not top in the list. However, TOP1 is significative in our context, since most TEST REUSE approaches choose the first event in the list.

$$\textbf{TOP1} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\left\{\begin{array}{ll} 1 & \text{if } \text{rank}_i = 1 \\ 0 & \text{otherwise} \end{array}\right\} \in [0;1]$$

Table 4.1. Example of semantic matching queries

Query	Source Event	Target[a] Events	Similarity Score	Correct Answer
q_1	e^s_1	e^t_2	0.8	✓
		e^t_1	0.6	
		e^t_3	0.5	
q_2	e^s_2	e^t_6	0.8	
		e^t_4	0.6	
		e^t_5	0.55	✓

[a] Target events are ordered by similarity score of e^s_i and e^t_j

Table 4.1 shows an example of two queries q_1 and q_2 with their similarity score and the ground truth. We calculate MRR and TOP1 of the example as follows:

MRR: The similarity scores of target events of q_1 ranks the correct match first and we assign reciprocal rank of 1 to q_1. The similarity scores of target events of q_2 ranks the correct match third and we assign reciprocal rank of $1/3$ to q_2. MRR averages reciprocal rank of the queries, that is $Mean(1, 1/3) = 0.6$.

TOP1: The ratio of the queries in which the correct answer is in the first position is 1 to two, that is $\frac{1}{2} = 0.5$.

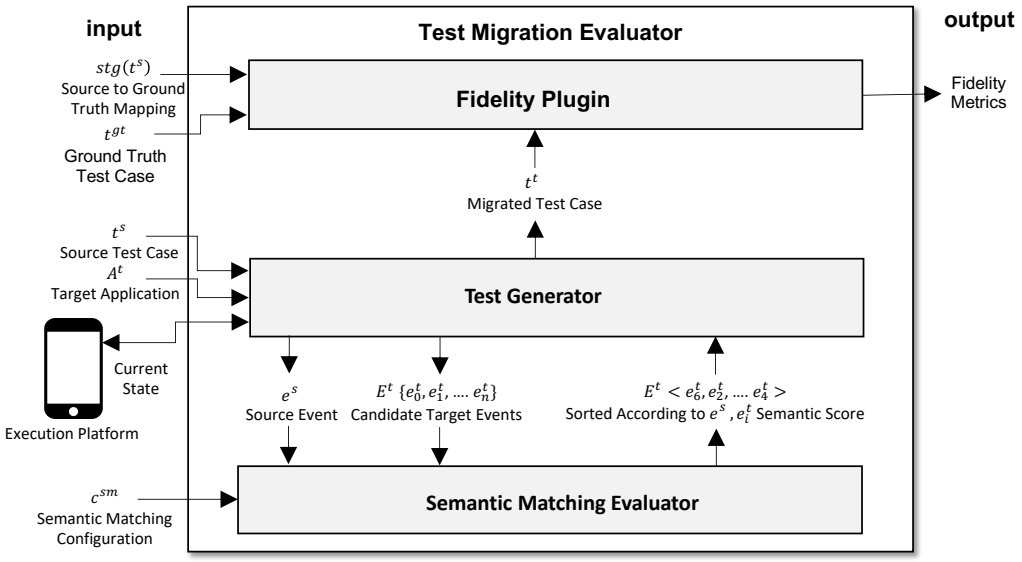


Figure 4.1. TEST MIGRATION EVALUATOR

4.3 TEST MIGRATION EVALUATOR

We assess semantic matching in the context of TEST REUSE, by comparing the quality of the test cases that ATM and CRAFTDROID GENERATORS migrate using different SEMANTIC MATCHING CONFIGURATIONS.

We evaluated the test reuse with TEST MIGRATION EVALUATOR, a prototype framework that integrates the SEMANTIC MATCHING EVALUATOR with both the ATM and CRAFTDROID TEST GENERATOR, as well as with a FIDELITY PLUG-IN that measures the fidelity of the source and migrated test cases with respect to the ground truth.

Figure 4.1 overviews the TEST MIGRATION EVALUATOR workflow. We used a modified version of SEMANTIC MATCHING EVALUATOR that evaluates one query for a given SEMANTIC MATCHING CONFIGURATION instead of evaluating a set of configurations for a given set of queries. The SEMANTIC MATCHING EVALUATOR receives a query $\langle e^s, E^t \rangle$ and a SEMANTIC MATCHING CONFIGURATION c^{sm}, uses the instances specified in the c^{sm} for the semantic matching components, and returns E^t sorted according to similarity score of events in E^t and e^s. The TEST GENERATOR component in TEST MIGRATION EVALUATOR operates as we described in section 3.1: It receives a test case t^s and a target application A^t as input, and migrates each event in the t^s based on results of queries that it sends to the SEMANTIC MATCHER. In the framework, the SEMANTIC MATCHING EVALUATOR plays the role of a universal SEMANTIC MATCHER that answers queries with different

SEMANTIC MATCHING CONFIGURATIONS. The FIDELITY PLUG-IN receives three inputs to calculate fidelity metrics: i) the migrated test case t^t ii) mapping of source test case events to ground truth events $stg(t^s)$ iii) the ground truth t^{tg}.

In our implementation of TEST MIGRATION EVALUATOR we interface both ATM and CRAFTDROID GENERATORS with SEMANTIC MATCHING EVALUATOR. In this way, we can combine different choices of TEST GENERATORS with SEMANTIC MATCHERS and build new TEST REUSE approaches. For instance, our TEST MIGRATION EVALUATOR can evaluate the CRAFTDROID generation approach with SEMFINDER SEMANTIC MATCHING ALGORITHM, thus extending the original CRAFTDROID approach.

4.3.1 FIDELITY PLUG-IN

FIDELITY PLUG-IN evaluates the quality of the migrated test cases as the fidelity of two associations: the *source-to-ground-truth* and the *ground-truth-to-migrated* associations. The *source-to-ground-truth* association maps source test cases to ground truth test cases. It indicates the sequence of events in the source test case for the source app that shall be migrated to obtain an optimal test case for the target app. The *ground-truth-to-generated* association maps ground truth test cases to their corresponding migrated test cases. It indicates the events of the ground truth that are correctly mapped to the target app.

The *source-to-ground-truth* association is defined manually once for all (Section 4.4). Given a source test case $t^s = \langle e_0^s, e_1^s, \cdots e_k^s \rangle$ and a ground truth test case $t^{gt} = \langle e_0^{gt}, e_1^{gt}, \cdots e_m^{gt} \rangle$, we refer to this association as $stg : t^s \rightarrow t^{gt}$[3], such that, $stg(e_i^s) = e_j^{gt}$.

The FIDELITY PLUG-IN automatically builds the *ground-truth-to-generated* association $m : t^{gt} \rightarrow t^t$ as follows: Each event in the ground truth test case is associated with the event in the target test case that shares the values of the identifier attributes, following their order of occurrence in the grounds truth test case. More formally, let $t^{gt} = \langle e_0^{gt}, e_1^{gt}, \cdots e_m^{gt} \rangle$ and $t^t = \langle e_0^t, e_1^t, \cdots e_n^t \rangle$ be test cases and $a(e_i^j)$ be the attributes of event e_i^j, the *ground-truth-to-generated* association is a partial function $m : t^{gt} \rightarrow t^t$ that associates events in the ground truth test case to events in the target test case according to the following rule:

$$m(e_i^{tg}) = e_j^t \text{ iff } a(e_i^{tg}) = a(e_j^t) \wedge \forall w < j, \exists k < i \mid a(e_i^{tg}) \neq a(e_w^t) \vee m(e_k^{tg}) = e_w^t$$

Let us define, $m(t^{gt}) = \bigcup_i m(e_i^{gt})$, that is, the set of events in the target test case that are associated with any event in the ground truth test case.

[3] For sake of notation, we represent the set of events in a test case $\{e_1, \ldots, e_n\}$ with the same symbol used to represent the test case t.

We measure the fidelity of the associations with the F1-SCORE fidelity metric that we compute from *Precision, Recall* as in FrUITeR ([109]):

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$
$$F1\text{-}score = \frac{2 \times Recall \times Precision}{Recall + Precision}$$

where we define true positives (TP), false positives (FP), and false negatives (FN) for a source test case t^s, a target test case t^t, and a ground truth t^{gt}, and the related *stg* and *m* associations, as follows:

TP: the cardinality of *m* association given $stg(t^s)$ as the input, that is the number of events that can be mapped from the source test case to the target test case through *stg* and *m* associations:

$$TP = \left| m(stg(t^s)) \right|$$

FP: the cardinality of the difference between the migrated test and the *ground-truth-to-migrated* association, that is, the number of events in the migrated test case that do not exist in the ground truth:

$$FP = \left| t^t \setminus m(t^{gt}) \right|$$

FN: the cardinality of the difference between the *source-to-ground-truth* and *ground-truth-to-migrated* associations given the *source-to-ground-truth* as the input, that is, the number of events in the source test case that have an equivalent event in the ground truth, but their equivalent does not exist in the migrated test case.

$$FN = \left| stg(t^s) \setminus m(stg(t^s)) \right|$$

Figure 4.2 shows an example of associations between source, ground truth and migrated test cases. We calculate fidelity of the example migration as follows:

TP: The only source event that can be mapped to the target test case is e_1^s and that results $TP = 1$.

FP: The set of $\langle e_3^t, e_4^t, e_6^t \rangle$ in composes the events in the migrated test case t^t that do not exist in the ground truth and they result $FP = 3$.

FN: The set of $\langle e_2^s, e_4^s \rangle$ composes the events in the source test case t^s that have an equivalent in the ground truth, but their equivalent does not exist in the migrated test case. They result $FN = 2$.

$$Precision = \frac{1}{1+3} = 0.25$$

$$Recall = \frac{1}{1+2} = 0.33$$

$$\textit{F1-score} = \frac{2 \times 0.33 \times 0.25}{0.33 + 0.25} = 0.28$$

In general, high F1-score witness good performance. In our evaluation, we compare the values of F1 score in different setups and configurations, to characterize the setups and configurations. We do not refer to absolute thresholds.

FrUITeR computes the fidelity of the mappings between the source and migrated test cases by instrumenting the TEST REUSE approaches. We compute the fidelity between source and ground-truth test cases and between ground-truth and migrated test case to avoid the instrumentation overhead.

We implemented our plug-in in Python, instead of extending FrUITeR ([109]), because (i) FrUITeR requires transforming the test cases into a canonical format, and the transformers are available for Java test cases only, while CRAFTDROID test cases are in *JSON* format, and (ii) FrUITeR identifies events by *resource-id* or *XPath* that do not uniquely identify events across the migration process: *resource-id* can be shared across multiple widgets, and the migration process generates different *XPath* for the same events.

4.4 Subjects

For our experiments, we considered all publicly available test migration scenarios of both ATM and CRAFTDROID: 248 scenarios, from 42 ANDROID apps. We considered the 147 scenarios of the 30 ANDROID apps that we could compile and execute. We could not experiment with 12 apps, since some of ATM apps are available in new versions that do not compile anymore with the ATM scenarios, and some CRAFTDROID apps both require communication with a server and are available with new API or Security protocol not compatible with the CRAFTDROID scenarios. We pruned 51 redundant scenarios, that is, scenarios that occur

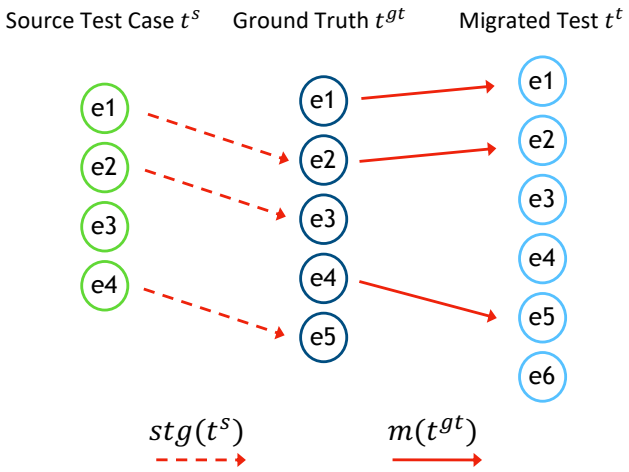


Figure 4.2. Example of *source-to-ground-truth* $stg(s^t)$ and *ground-truth-to-migrated* $m(t^{gt})$ associations

in other scenarios, and experimented with the 95 unique and compatible scenarios. We addressed RQ1, RQ2, RQ3 and RQ4 with experiments on all the 95 scenarios. We addressed RQ5, RQ6 and RQ7 with experiments on 89 scenarios for 29 out of 30 ANDROID apps, since the FirefoxFocus app includes a key widget with an unconventional type incompatible with APPIUM, the test automation tool that we used in the experiments with the TEST REUSE tools. We experimented CRAFTDROID on all scenarios, and ATM on ATM scenarios only, because ATM instruments the source code of the apps, and the instrumentation logic designed in the tool works only for the scenarios in the original study. Instrumenting different apps results in compilation errors.

Thus, we identify two sets of scenarios:

- *Shared Scenarios*: 27 scenarios that both ATM and CRAFTDROID can process. We refer to these scenarios to comparatively evaluate semantic matching in the context of the two approaches applied to the same scenarios.

- *All Scenarios*: All 89 scenarios. We refer to these scenarios to investigate semantic matching with a wide range of applications.

Table 4.2 summarizes the scenarios we use in our experiments.

Table 4.2. Subjects of our experiment

subject from	category	test case description	app name	# of DL[a]
ATM	Expense Tracker	Add an expense entry to expense list	EasyBudget [117]	100k
			Expenses [127]	1K
			Daily Budget [126]	50K
			Open Money [141]	1K
	Note Taking	Add a note and save	Swiftnotes [113]	-
			Writely Pro [132]	-
			Pocket Note [134]	-
	Shopping List	Add a shopping item	Shop.List1 [114]	-
			Shop.List2 [140]	100K
			Shop.List3 [136]	5K
			OI Shop. List [131]	1M
CRAFTDROID	Browser	Go to an URL, go to another URL go back to the first URL	Lightning [115]	10K
			Privacy [138]	1K
			FOSS [120]	-
			FirefoxFocus [129]	5M
	To-Do List	Add a todo task and save Remove the recent task	Minimal [135]	-
			Clear List [119]	-
			Todo List [137]	-
			Simply Do [125]	-
			Shop. List [124]	-
	Shopping	1) Sign up 2) Sign in	Rainbow [133]	0.5M
			Yelp [142]	50M
	Mail Client	1) Search for an email 2) Send an email	Mail.ru [128]	50M
			myMail [130]	10M
			AnyMail [118]	10M
	Tip Calculator	Add a bill with information of the tip then calculate the share of tip per person	TipCalculator [116]	
			TipCalc [116]	500
			Simple Tip [139]	1K
			TipCalc.Plus [143]	500
			FreeTipCalc. [123]	1K

[a] Number of downloads

Chapter 5

Semantic Matching in Isolation

In this chapter we report the results of our evaluation of the components of test reuse in isolation.

5.1 Experimental Setup

We thoroughly explored every possible SEMANTIC MATCHING CONFIGURATIONS by combining the instances introduced in Chapter 3 to address RQ1, RQ2, and RQ3. Figure 5.1 shows all the combinations of instances, which yields 337 SEMANTIC MATCHING CONFIGURATIONS. We experimented with 19 WORD EMBEDDING models available in SEMANTIC MATCHING EVALUATOR (components C1, C2), 12 of which obtained by training four WORD EMBEDDING techniques with three corpora of documents and 7 pre-trained models. We added two syntactic techniques to the SEMANTIC MATCHING EVALUATOR to compare syntactic techniques with word embedding models which consider semantic of words. We combined the 19 embedding models and the two syntactic techniques with four EVENT DESCRIPTOR EXTRACTOR (component C3) and four SEMANTIC MATCHING ALGORITHMS (component C4). We also experimented with the random baseline.

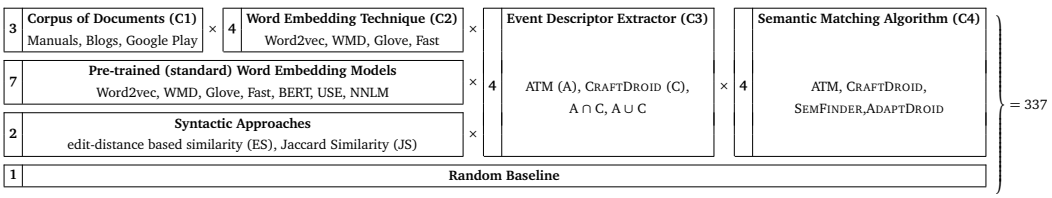


Figure 5.1. The 337 configurations of components' instances considered in our study

We used both the embedding and syntactic techniques similarly: we invoke

the technique with the two texts to obtain similarity score between [0,1], which indicates similarity of the two texts.

Word embedding techniques calculate the position of the two input texts in the vector space as A and B vectors, and return the cosine similarity of the two vectors calculated as follows:

$$S_c(A,B) = \cos(\theta) = \frac{A.B}{\|A\|\|B\|} = \sum_{i=1}^{n} \frac{A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2 . \sum_{i=1}^{n} B_i^2}}$$

We considered two canonical syntactic techniques that compute the syntactic similarity of words/sentences: EDIT DISTANCE SIMILARITY, and JACCARD SIMILARITY. EDIT DISTANCE SIMILARITY computes the distance of two words wd_1 and wd_2 as

$$ES(wd_1, wd_2) = \frac{max(|wd_1|, |wd_2|) - LD(wd_1, wd_2)}{max(|wd_1|, |wd_2|)} \in [0;1]$$

where LD(wd_1 wd_2) is the "Levenshtein distance ([49]) of wd_1 and wd_2, that is, the minimum number of operations (deletion, insertion and substitution) required to transform wd_1 into wd_2 and vice versa. EDIT DISTANCE SIMILARITY returns 1 if the words are identical. For example, EDIT DISTANCE SIMILARITY of two words wd_1 ="first" and wd_2 = "last" is calculated as follows:

$$ES(\text{first, last}) = \frac{max(5,4) - 3}{max(5,4)} = 0.4$$

EDIT DISTANCE SIMILARITY operates at word level, and thus replaces the query of the WORD EMBEDDING model at line 8 of Algorithm 1.

JACCARD SIMILARITY computes the similarity of two sentences txt_1 and txt_2 as the number of elements that belong to both strings over the number of elements that occur in either or both strings:

$$\text{JS}(\text{txt}_1, \text{txt}_2) = \frac{|\text{txt}_1 \cap \text{txt}_2|}{|\text{txt}_1 \cup \text{txt}_2|} \in [0;1]$$

JACCARD SIMILARITY returns 1 when txt_1 and txt_2 have all identical words, regardless of their position in the sentences. JACCARD SIMILARITY operates at sentence level, and thus replaces the interrogation of the WORD EMBEDDING model at line 12 of Algorithm 1. For example, JACCARD SIMILARITY of two strings $\langle txt_1 = \text{"first name"}, txt_2 = \text{"last name"} \rangle$ is calculated as follows:

$$\text{JS("first name", "last name")} = \frac{|\{\text{name}\}|}{|\{\text{first, last, name}\}|} = 0.33$$

If we use EDIT DISTANCE SIMILARITY and *average* aggregation function (EDIT DISTANCE SIMILARITY works at word level) to score $\langle txt_1 = \text{"first name"}, txt_2 = \text{"last name"}\rangle$; then we calculate the similarity score matrix as follows in which cells indicate EDIT DISTANCE SIMILARITY of corresponding words in rows and columns.

$$\begin{array}{c} \\ \text{last} \\ \text{name} \end{array} \begin{array}{c} \begin{array}{cc} \text{first} & \text{name} \end{array} \\ \begin{pmatrix} 0.4 & 0.25 \\ 0.0 & 1 \end{pmatrix} \end{array}$$

Similarity scores lead to matching of ⟨name, name⟩ and ⟨first, last⟩. The similarity score of $\langle txt_1, txt_2\rangle$ is equal to $average(1, 0.4) = 0.7$

The random baseline assigns a random score between 0 and 1 to each pair of events. We repeated the experiments 100 times, to cope with the stochastic nature of the random baseline, and report the median.

We experimented with 95 unique scenarios and we obtained the ground truth for CRAFTDROID scenarios from the original CRAFTDROID paper, and we manually defined the ground truth for the ATM scenarios. The target test case t^t may include ancillary events ([109]), that is, events in t^t that do not correspond to any event in t^s, and that are required to reach relevant states in the app. Since this set of research questions deal with semantic matching in isolation, we do not consider ancillary events for these questions. Some events occur multiple times in test cases for the same app. For example, events in a dialog box containing "Ok" and "Cancel" buttons may occur multiple times. We prune redundant events, that is events that share all nine event descriptors with other events, and we obtained 337 unique queries for evaluating semantic matching. For example, if an "OK" event appears in different screens with the same descriptor, we consider it redundant, and we only consider one occurrence.

We define the set of candidate target events $E^t = \{e^t_0, e^t_1, \cdots e^t_n\}$ for each $e^s \in t^s$ as the set of events that are actionable in all the GUI states that the ground truth t^{gt} visits. More formally, $E^t = \{e^t \ : \ \exists\ S \ \in \ \mathbb{S}, e^t \text{ is actionable in } \ S\}$, where $\mathbb{S}$ is the sequence of state transitions obtained by executing t^{gt}. We pruned redundant events from E^t which occur when different states share the same window. The cardinality of E^t ranges from 5 to 80, with an average of 24.03 and median of 19 events. Our definition of E^t leads to semantic matching queries that are coherent with TEST REUSE, which matches events across applications by considering also target events in the TARGET APPLICATION MODEL that span multiple windows ([10, 53]).

Figure 5.2 shows the process of creating queries from t^s and t^{gt}. First EXECUTOR-

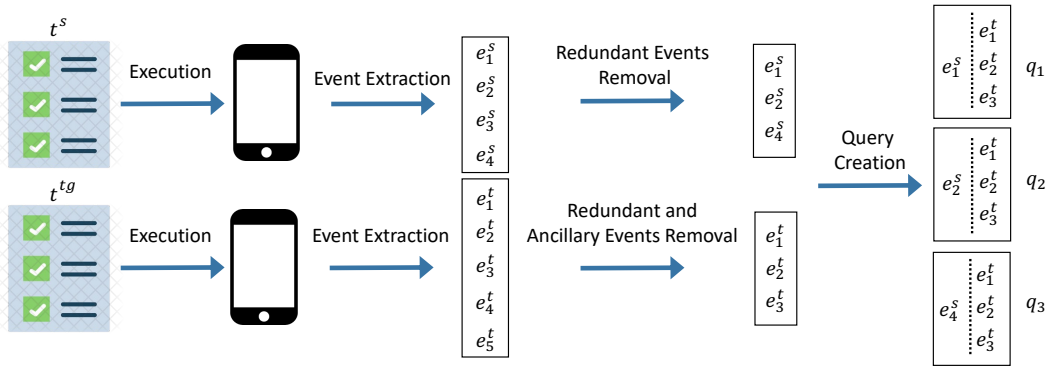


Figure 5.2. Creating semantic matching queries

PLUGIN executes the test cases and extract events. We transform t^s and t^{gt} to $t^{s'}$ and $t^{gt'}$ by removing redundant events from t^s and t^{gt}, and ancillary events from t^{gt}. For each event $e_i^s \in t^{s'}$ we consider a query in which target candidates are all events in $t^{gt'}$.

5.2 Experimental Results

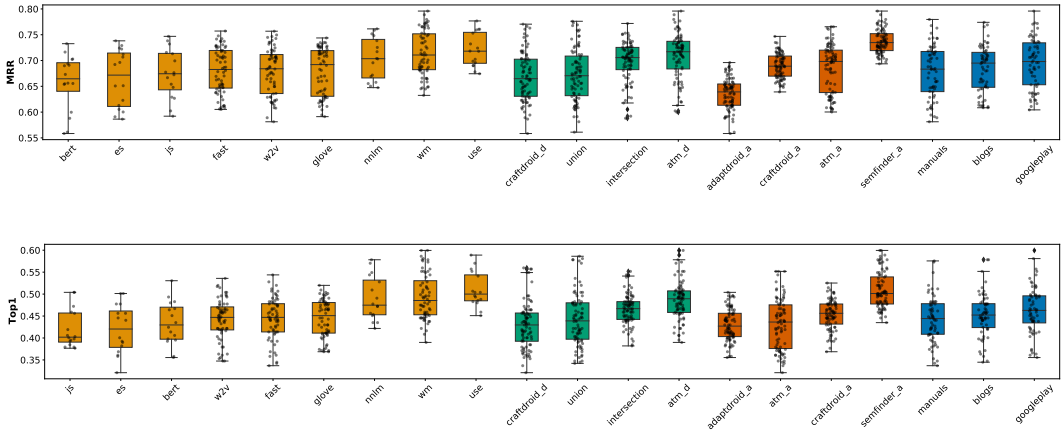


Figure 5.3. Distribution of MRR (top) and TOP1 (bottom) for instances

We ran our 337 queries for each of the 337 configurations which answer queries uniquely and we analysed 113,569 query answers in total.

Table 5.2 shows statistical description of MRR and TOP1. The best configuration with respect to the both metrics is [GOOGLE-PLAY (C1), WM (C2), ATM_d (C3), SEMFINDER (C4)] and the worst is random. MRR values are higher than

Table 5.1. Distributions of the 337 combinations sorted by MRR and TOP1

type	instance	MRR [1:3] [a]	MRR [1:16]	MRR [1:33]	TOP1 [1:3]	TOP1 [1:16]	TOP1 [1:33]
C1	blogs	0%	19%	12%	0%	**15%**	12%
	manuals	**33%**	19%	12%	0%	**15%**	12%
	googleplay	**33%**	**31%**	**27%**	**33%**	**15%**	**19%**
C2	w2v	0%	0%	6%	0%	0%	0%
	glove	0%	0%	0%	0%	0%	0%
	wm	**100%**	**94%**	**56%**	**100%**	**69%**	**62%**
	fast	0%	0%	6%	0%	0%	4%
	bert	0%	0%	0%	0%	0%	0%
	nnlm	0%	0%	12%	0%	15%	15%
	use	0%	6%	15%	0%	15%	19%
	js	0%	0%	3%	0%	0%	0%
	es	0%	0%	0%	0%	0%	0%
C3	ATM_d	**100%**	**50%**	**36%**	**67%**	**46%**	**42%**
	CRAFTDROID_d	0%	19%	19%	0%	8%	19%
	intersection	0%	6%	18%	0%	0%	15%
	union	0%	25%	27%	33%	**46%**	23%
C4	ATM_a	0%	19%	15%	0%	6%	15%
	ADAPTDROID_a	0%	0%	0%	0%	0%	0%
	CRAFTDROID_a	0%	0%	0%	0%	0%	0%
	SEMFINDER	**100%**	**81%**	**85%**	**100%**	**93%**	**81%**

[a] The columns indicate the percentage of queries that locate the correct answer in positions [1:3] (1% percentile), [1:16] (5% percentile), [1:33] (10% percentile) of the list of 337 configurations sorted by MRR or TOP1

TOP1 (0.17 higher on average) since TOP1 only considers a positive score if the correct answer ranks top, however, MRR always considers a positive score for the correct answer, relative to its rank.

Figure 5.3 shows the distributions of MRR and TOP1 by instance. For example, the box plot of SEMFINDER on the right of Figure 5.3 shows the distribution of the MRR values of all the 84 configurations with SEMFINDER as the SEMANTIC MATCHING ALGORITHM. The box plots of the same component type are sorted by median.

The instances of the component types are unevenly distributed among the configurations. For example, WM is present in 64 configurations, while USE only in 16. This is because for WM we considered the pre-trained standard model and three models built from the three corpora of documents, while for USE we only considered the pre-trained model.

Table 5.1 shows the distributions of the various component instances for three

Table 5.2. Statistical description of MRR and TOP1 in 337 SEMANTIC MATCHING CONFIGURATIONS

	Min	Q1	Q2	Q3	Max	Average
MRR	0.201	0.649	0.693	0.724	0.795	0.685
TOP1	0.065	0.465	0.510	0.508	0.671	0.518

percentiles 1% (top 3 entries [1:3]), 5% (top 16 entries [1:16]), and 10% (top 33 entries [1:33]). The values in the cells indicate the percentage of configurations that use a given instance of a component (row). For instance, every configuration that both MRR and TOP1 rank in the top three positions ([1:3]) uses WM (100% in the [1:3] cells of row C2.wm).

We tested the pairs of instances in Table 5.1 for *statistical significance* using Mann-Whitney U test ([64]). We rejected the null hypothesis that the two distributions are the same with *p-value* <= 0.05. The null hypothesis invalidated 23 out of 51 pairs of instances from MRR metrics, and 18 for TOP1 metrics. Appendix .1 reports the results of the Mann-Whitney U test.

RQ1: Baseline Comparison.

Both MRR and TOP1 ranking indicate that all configurations perform significantly better than the random baseline. Both metrics rank random last in the sorted list, with MRR and TOP1 values of 0.201 and 0.065, respectively, much lower than the configurations with the second lowest values, 0.595 and 0.359, respectively.

Table 5.1 indicates that syntactic based similarity metrics (EDIT DISTANCE SIMILARITY and JACCARD SIMILARITY) perform worse than WORD EMBEDDING models. Indeed, only one of the 32 configurations with either JACCARD SIMILARITY or EDIT DISTANCE SIMILARITY appear in the top 10% configurations sorted by either MRR or TOP1 values. The distribution of MRR and TOP1 of Figure 5.3 confirms EDIT DISTANCE SIMILARITY and JACCARD SIMILARITY in the leftmost side of the distribution, sorted in increasing order.

The experimental results allow us to answer positively to RQ1: All SEMANTIC MATCHING CONFIGURATIONS perform significantly better than the random baseline.

The experimental results confirm the hypothesis: The semantic approaches that use WORD EMBEDDING perform significantly better than the syntactic baseline, with a big gap between the random baseline and the less performant SEMANTIC MATCHING CONFIGURATION according to both MMR and Top1.

RQ2: Component Effectiveness in Isolation.

The results reported in Table 5.1 and Figure 5.3 indicate that the most effective instances of the four components evaluated in isolation are: GOOGLE-PLAY, WM, ATM_d, and SEMFINDER for each of the four components of Figure 5.1. Below, we discuss the evidence from experimental data in details.

Corpus of Documents (Component C1) Table 5.1 indicates that googleplay is the CORPUS OF DOCUMENTS that occurs more often in the top ranked combinations, according to both MRR and TOP1 for all percentiles. The distribution in Figure 5.3 confirms the result with the distribution of GOOGLE-PLAY in the rightmost position, but only provide statistical significance of the ⟨GOOGLE-PLAY , MANUALS⟩ pair.

The sum of the values of the columns for the three corpora of documents (C1) is less than 100%, since the configurations include pre-trained models, some of which with high MRR and TOP1 ranking.

We studied the impact of the Out Of Vocabulary (OOV) issue that occurs when the query involves words that do not belong to the considered corpus. We collected the OOV issues for the 48 configurations with WORD2VEC as WORD EMBEDDING technique, and GOOGLE-PLAY, MANUALS, and BLOGS as corpora of documents, and compared the cumulative number of OOV for the three clusters that use googleplay, manuals, and blogs, respectively. The cumulative 25,119, 364,049 and 208,075 OOV for GOOGLE-PLAY, MANUALS, and BLOGS indicate that GOOGLE-PLAY suffers significantly less than MANUALS and BLOGS from OOV.

Word Embedding (Component C2) Figure 5.3 indicates that sentence level WORD EMBEDDING techniques, WM and USE, are the best techniques according to both MRR and TOP1. The difference between WM and USE and other techniques (FASTTEXT, WORD2VEC, GLOVE, BERT, JS, ES) is statistically significant for both MRR and TOP1. Table 5.1 indicates that three out of four sentence level techniques (WM, USE, and NNLM) dominate all other techniques. The inspection of GUI textual attributes indicates that many of them are expressed with multiple words, and this explains the better performance of sentence level over word level techniques.

EVENT DESCRIPTOR EXTRACTOR **(Component C3)** Figure 5.3 indicates that ATM_d and *intersection* perform better than *union* and CRAFTDROID_d, as event de-

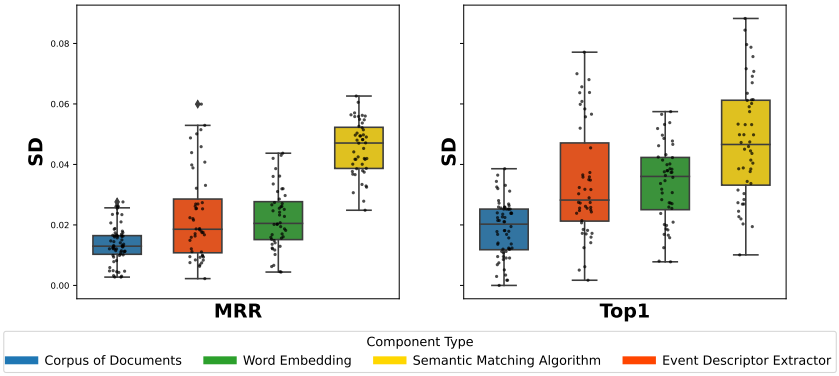


Figure 5.4. Impact analysis of the components

scriptor selectors, and the difference is statistically significant. Table 5.1 confirms the dominance of ATM_d also in terms of occurrences in top ranked positions according to both MRR and TOP1 for all percentiles.

A deep analysis of the results reveals an unbalanced distribution of the attribute types in our subjects: 8,099 source and target events define the *activity-name* attribute, 7,837 the *resource-id* attribute, 4,532 the *text* attribute, 957 the *neighbor-text* attribute, 837 the *content-desc* attribute, 600 the *parent-text* attribute, 554 the *file-name* attribute, 165 the *sibiling-text* attribute, and no events defines the *hint* attribute. The poor performance of *union* and CRAFTDROID_d may depend on the high frequency of the *activity-name* attribute, which is defined for each event. Unrelated events in the target app that share the activity name of the source event may yield a similarity score higher than the correct match (e^t_{gt}), and this impact on the final score.

Semantic Matching Algorithm (Component C4) Figure 5.3 indicates that median of SEMFINDER is higher than the median of ATM_a, CRAFTDROID_a, and ADAPTDROID_a; SEMFINDER outperforms other algorithms always with statistical significance. Table 5.1 confirms that SEMFINDER is the semantic matching algorithm that occurs more often in the top ranked combinations, according to both MRR and TOP1 for all percentiles.

The analysis of the performance of the algorithms indicates ADAPTDROID_a as the best performing approach: the configurations with ADAPTDROID_a complete all 337 queries in 170 seconds in average, the configurations with SEMFINDER in 255 seconds, the configurations with CRAFTDROID_a in 393 seconds, and the configurations with ATM in 600 seconds. This suggests that combining attribute values into a single sentence can reduce the runtime while improving the results of semantic matching, as long as there is no prioritization of attributes.

The experimental results indicate that the most effective instances for the SEMANTIC MATCHER components are: SEMFINDER, ATM_d, GOOGLE-PLAY, and WM.

RQ3: Component Impact Analysis in Isolation.

The results of the impact analysis reported in Figure 5.4 indicate that Semantic Matching Algorithm (C4) is the configuration with the highest impact, followed by WORD EMBEDDING Technique (C2), EVENT DESCRIPTOR EXTRACTOR (C3), and CORPUS OF DOCUMENTS (C1).

We studied the impact of the component types with a "local" sensitivity analysis ([23]) that varies the instance of one component type at a time while holding the others fixed ([40]). We clustered the 337 configurations of the four component types, by varing an instance of a component while fixing all instances of the other three components. For example, if we consider C2 and exclude the random baseline, we have nine possible instances (7 word embedding and two syntactic instances). Every time we fix the values for components C1, C3, C4, we define a new cluster with 9 configurations (in which only C2 varies). We compute the *standard deviation* (SD) of the MRR values of these nine configurations. This SD value represents the impact of C2 in the cluster (if the choice of C2 has high impact, the SD value is high, otherwise it is low) ([40]).

SD is a measure of the amount of variation or dispersion of a set of values. A low SD indicates that the values tend to be close to the mean of the set, while a high SD indicates that the values are spread out over a wider range. This means that if a component has a high impact on the semantic matching, the SDs values of each cluster must be high.

We repeated this process for C2 48 times, that is, for every possible combination of the values of components C1, C3, and C4, obtaining $3 \times 4 \times 4 = 48$ SDs that globally capture the impact of C2 on the semantic matching. We ran this analysis for all four component types.

We computed the SDs for both the MRR and the TOP1 values. Figure 5.4 shows the distributions of the SDs values for category type, and sorts the components left to right by impact.

The experimental results indicate that the Semantic Matching Algorithm (Component C4) is the configuration with the highest impact, followed by EVENT DESCRIPTOR EXTRACTOR (Component C3), WORD EMBEDDING Technique (Component C2), and CORPUS OF DOCUMENTS (Component C1).

Chapter 6

Domain Specific Word Embedding Models

In this chapter, we report the results of our evaluation of domain specific word embedding models in isolation by leveraging our SEMANTIC MATCHING EVALUATOR *framework.*

In Section 5.2 we observed that the GOOGLE-PLAY corpus, which is specific to the mobile applications domain, leads to better results than general corpora when dealing with semantic matching in isolation. The results reported in Section 5.2 confirm the common understanding that word embedding techniques trained on domain specific corpora perform better than techniques trained on general corpora, on downstream tasks [51]. We can create domain specific models by either fine tuning a model that has been trained on a general corpus [12] or training a model from scratch on a corpus specific to the domain [38]. GU et al. concluded that training a model from scratch is more effective [38]. Following the best practices, we also build the domain specific corpora from scratch. We investigated whether models trained on GOOGLE-PLAY corpus reflect the same word usage that mobile apps commonly adopt. Indeed, a word can have different meanings depending on the context of usage (polysemy). Mobile applications refer to many unrelated domains that use the same words differently. For example, applications of categories “Fitness & Health” and “Food & Drink” use the word “bar“ differently.

We addressed RQ4 with a study on the impact of WORD EMBEDDING models trained on specialized corpora that contain only semantically related app descriptions on the semantic matching of TEST REUSE approaches. We experimented with various configurations and algorithms of the most common topic modeling approaches. We leveraged topic modeling to partition GOOGLE-PLAY corpus into

domain specific clusters and we create WORD EMBEDDING models for each cluster. We extended our SEMANTIC MATCHING EVALUATOR to automatically select the specialized word embedding models that corresponds to the most semantically related partition based on the Google Play descriptions of the source app. We experimented with 48 SEMANTIC MATCHING CONFIGURATIONS that include domain specific WORD EMBEDDING models and 192 baselines configurations.

6.1 Experimental Setup

We partitioned the app descriptions of the GOOGLE-PLAY corpus into semantically coherent clusters by means of *topic modeling* ([30, 13, 98]), commonly used to classify apps into meaningful categories ([112, 111, 93, 94]).

A topic model is a statistical model for discovering the abstract "topics" that occur in a collection of documents. In the GOOGLE-PLAY corpus, a document is the English description of an app in the Google Play Store. There are three key design choices to customize a topic modeling approach to a specific problem: (i) the topic modeling algorithm, (ii) the target number of topics, and (iii) preprocessing of the corpus. We investigated different combinations of these design choices to select the best approach.

Topic Modeling Algorithms We experimented with three of the most commonly used topic modeling algorithms:

Latent semantic analysis (LSA) ([30]) is a mathematical method, based on a distributional hypothesis that takes into account how frequently words appear in a document and in the whole corpus.

Latent Dirichlet Allocation (LDA) ([13]) is a probabilistic method that assumes the distributions of both topics in a document and words in topics are Dirichlet distributions.

Hierarchical Dirichlet process (HDP) ([98]) is an extension of LDA. HDP uses statistical inference to learn the number of topics based on the corpus.

Target Number of Topics LSA and LDA take the number of topics as an input. We assumed the number of topics should be in the same range as the categories that Google Play Store considers (32 non-game categories). Thus, we experimented with a number of topics that ranges from 2 to 102.

Pre-processing In the section 4.2 we explained the canonical pre-processing steps that we performed on the GOOGLE-PLAY corpus. We also considered additional pre-processing steps that are often crucial to obtain a meaningful topic

modeling ([63]): Vocabulary and document pruning. Vocabulary pruning removes words that have either a very low or a very high frequency in the corpus since such words, also called domain-specific stop words ([63]), produce noise and result in low-quality topic models. Document pruning removes documents that are either too short or too long, which might lead to a low-quality topic model ([33]). Short documents might not contain enough information to characterize a topic ([57]), whereas long documents usually cover multiple topics.

To identify the strategy that works best for the corpus at hand, we experimented with different pre-processing strategies for vocabulary and document pruning. Indeed, the best strategy can only be determined empirically, because the effectiveness of such strategies depends on the intrinsic characteristics of the corpus ([25, 61]). We experimented with the following popular strategies:

Vocabulary pruning strategies:

S1 It defines lower and upper bounds based on the word frequency. It prunes words that either occur in more than X_{up}% documents or in less than X_{low}% documents (default values $X_{up} = 15\%$ and $X_{low} = 0.5\%$ ([62, 33])).

S2 It defines lower and upper bounds by assuming a Gaussian distribution of words frequency. It prunes words with a frequency that belongs to the tails of the Gaussian plot: it prunes the first and last X% of the distribution (default value $X = 5\%$).

Document pruning strategies:

S3 It prunes documents that have more than X_{up} and less than X_{low} words (default values $X_{up} = 1000$ and $X_{low} = 50$ ([33])).

S4 It sorts the documents based on their size and prunes the top and bottom X% (default value $X = 5\%$).

S5 It defines lower and upper bounds by assuming that documents size has a Gaussian distribution. Similar to vocabulary pruning, it prunes the first and last X% of the distribution (default value $X = 5\%$).

Automated Identification of the Best Model A topic model computes the *word probability distribution* over topics, which is represented as words sorted in descending order with respect to their contribution to the topic. A good topic model is interpretable, that is, the words with the highest probability in the probability distribution are semantically coherent ([17]). We automatically evaluated

the topic coherence, by computing the topic *coherence value* (cv) metric ([87]), which uses co-occurrence of words to quantify the semantic coherence of topics ([79]). We relied on *cv* as it is recognized to be the best quantitative metric that captures the coherence of topic models ([87]). In fact, *cv* outperforms existing metrics with respect to the correlation to human judgments ([87]). This metric captures the coherence of a model as a value between 0 and 1, representing highly coherent models with high values.

We experiment on a random sample of 50,000 documents in GOOGLE-PLAY ($\sim$5.5%), to efficiently identify the best configuration for topic modeling ([62]). To reduce the number of configurations to evaluate, we incrementally considered the number of topics with step 10 from 2 to 102, for each combination of pre-processing strategy and algorithm (LDA and LSA). After identifying the best range for each configuration, we tried all numbers close to that range with a radius of 10 numbers, to see if we can find a number of topics that leads to a better result (higher cv). For each pruning strategy, we explored different parameters values in addition to default ones and selected the value that yields the better performance.

The configuration with the best performance among the ones that we explored ($cv = 0.64$) uses the LDA topic modeling algorithm with a target of 27 topics, and applies strategies S1 for vocabulary pruning using the default values (words that either occur in more than 15% or less than 0.5% of documents), followed by S5 for document pruning using the tuned value of $X = 15\%$ (document that has size of the first and last 15% of the Gaussian distribution).

The results of these experiments gave us three important insights: (i) The order in which the vocabulary and document pruning are performed affects the results, and we generally get better results if we apply vocabulary pruning first; (ii) The LDA algorithm performs better than HDP and LSA; and (iii) There are some domain-specific common words (such as "Application" and "Google") that appear as the most contributing words for some of the topics of the best models. Such common words reduce the quality of the models.

Best Model Validation We recomputed the coherence values of the three models that achieved the highest values on the random sample by referring to the whole GOOGLE-PLAY corpus. We confirmed that the selected topic model configuration achieves the best performance indeed. We observed an even higher coherence value $cv = 0.71$ ($cv = 0.64$ on the 5.5% random sample of GOOGLE-PLAY).

We enriched the vocabulary by adding a manually created list of domain-specific common words based on the topics in the sampled dataset, to prune some domain-specific common words (insight (iii)). We build a new topic model with

the same configuration, and obtained a model of improved quality, $cv = 0.73$.

We confirmed the high quality of the selected model by manually inspecting the obtained model according to a standard protocol used in previous works ([96, 18, 103, 43]). We checked the following conditions: (i) The 15 most probable words in the word probability distribution of each topic are semantically coherent ([96, 18]); (ii) The most probable words in the word probability distribution do not contain common words ([103, 43]). We observed that there are no more than three common words in the top 15 contributors.

6.1.1 SEMANTIC MATCHING EVALUATOR Setup

The best topic model yields 27 clusters of the GOOGLE-PLAY apps. In the rest of the paper, we use **TOPICS** to refer to such a partition of GOOGLE-PLAY. The size of each cluster varies from 8,859 to 41,936 documents. We trained 27 domain-specific word embedding models, one for each cluster. We experimented with various hyper-parameters and architecture for building the word embedding models. None of hyper-parameters and architectures outperformed the default values available in related python packages that implemented the techniques, thus we use the default values. We used our SEMANTIC MATCHING EVALUATOR to both evaluate the word embedding models in the semantic matching in isolation and compare them with the ones obtained by GOOGLE-PLAY and other baseline corpora.

We considered four word embedding techniques that we chose for training the models in section 4.2: (i) WORD2VEC (ii) Word Mover's distance (WM) (iii) GLOVE (iv) FASTTEXT.

We considered four baseline corpora ordered from the most general to the most domain-specific: (i) BLOGS (ii) MANUALS (iii) GOOGLE-PLAY (iv) CATEGORIES. We introduced the first three corpora in 3.2. CATEGORIES is a partition of GOOGLE-PLAY according to the categories of the Google Play Store. We added the CATEGORIES corpus to our experiments as it represents a baseline for a more specialized domain-specific corpus.

In this experiment, we used the same experimental setup as semantic matching in isolation 5, except for the set of SEMANTIC MATCHING CONFIGURATIONS. We considered configurations that use TOPICS and CATEGORIES in addition to GOOGLE-PLAY, MANUALS, and BLOGS. Each corpus creates 48 configurations and in total we experimented with 240 configurations.

While BLOGS, MANUALS, and GOOGLE-PLAY corpora lead to a single word embedding model (which can be used for any pair of source and target apps), CATEGORIES and TOPICS lead to multiple word embedding models. Given an arbitrary

Table 6.1. Distribution of MRR and TOP1 values of the configurations of the semantic matching grouped by corpus of documents

corpus of documents	MRR				TOP1				rank of AVG	
	AVG	Min	Median	Max	AVG	Min	Median	Max	MRR	TOP1
BLOGS	0.6991	0.6084	0.7039	0.7740	0.5147	0.3857	0.5163	0.6468	4	4
MANUALS	0.6976	0.6052	0.7042	0.7796	0.5043	0.3768	0.5148	0.6439	5	7
GOOGLE-PLAY	0.7101	0.6165	0.7134	0.7958	0.5273	0.3976	0.5222	0.6706	2	2
CATEGORIES	0.6959	0.6015	0.6999	0.7791	0.5147	0.3798	0.5014	0.6587	6*	5***
TOPICS	0.6907	0.5865	0.7034	0.7834	0.5035	0.3620	0.5059	0.6587	9	8
h_categories_edit	0.6943	0.5934	0.6991	0.7791	0.5116	0.3768	0.4955	0.6587	7	6*
h_googleplay_edit	0.7088	0.6150	0.7133	0.7958	0.5247	0.3946	0.5207	0.6706	3***	3***
h_topics_edit	0.6908	0.5851	0.7017	0.7834	0.5031	0.3620	0.4940	0.6587	8	9
comb_topics_google-play	0.7393	0.6411	0.7438	0.8135	0.5610	0.4273	0.5608	0.6944	1***	1***

The table reports the paired t-test [88] p-value computed for TOPICS and each of the other configurations
* p-values < 0.05
** p-values < 0.01
*** p-values < 0.001

pair of source and target apps, we select the most appropriate model as follows: For CATEGORIES, we simply select the model associated with the category of the source app as specified in the Google Play Store; For TOPICS, we query our topic model from the Google Play description of the source app, to find the cluster semantically closest to the source app. We then retrieve the word embedding model trained on this cluster and use it for semantic matching the pairs of GUI events from the source and the target apps. We are investigating TEST REUSE among applications with similar functionalities, thus, we only considered the source app, as we assume that the source and target applications belong to the same cluster.

6.2 Experimental Results

We group the 240 configurations of semantic matching according to the corpus of documents they use. TOPICS refer to all the 48 SEMANTIC MATCHING CONFIGURATIONS that use the 27 clusters of documents obtained with topic modeling to create the word embedding models. The first five rows of Tables 6.1 report *avg*, *mean*, *median*, and *max* of each group with respect to the metrics MRR and TOP1. The last two columns show the performance of each group by reporting the ranking based on the average MRR and TOP1.

The results show that the configurations that use TOPICS to train word embedding models perform worse than the ones that use the other corpora, for both TOP1 and MRR (rows from 1 to 5). This negative result does not confirm our initial intuition.

6.2.1 Out-of-Vocabulary Issue

A possible explanation of the negative result could be the Out-of-Vocabulary issue (OOV) ([15]), which occurs when querying a word embedding model with words that are not in the corpus. Indeed, the partition of GOOGLE-PLAY leads to small clusters of documents, each of which might contain only a subset of all the unique words in GOOGLE-PLAY. If the OOV issue occurs while semantic matching of two GUI events, such events will not match.

To investigate if the negative result is due to the OOV issue, we produced a hierarchy of models that avoids the OOV issue by design. If a text query $q = \langle txt_1, txt_2 \rangle$ involves a word that does not belong to the current word embedding model, we propagate the query in the model hierarchy. If all models return OOV, we use EDIT DISTANCE SIMILARITY, which is not based on word embedding. Figure 6.1 shows the structure of the hierarchy. We considered the following hierarchy: topics, category, google-play, and edit distance. In total, we created three hierarchies of models. We refer to a hierarchical model by the first and last levels of the hierarchy. For example, *h_topics_edit* means that we first query the word embedding models trained on TOPICS. If a query manifests an OOV issue, we propagate the query to the model trained on the whole corpus, GOOGLE-PLAY. If the query still manifests an OOV issue, we compute the similarity score with EDIT DISTANCE SIMILARITY. We integrated the hierarchical models into SEMANTIC MATCHING EVALUATOR to investigate the impact of OOV issue.

The three bottom but one rows of Table 6.1 show the results. The TOP1 and MRR values of the hierarchical models with TOPICS as the first level are not significantly better than the ones of TOPICS only. This indicates that the OOV issue is not responsible for the poor results. Interestingly, *h_googleplay_edit* and *h_categories_edit* perform worse than the GOOGLE-PLAY and CATEGORIES. This suggests that the OOV issue may sometimes be beneficial by avoiding spurious matching of events.

6.2.2 Complementary Study

None of the 337 SEMANTIC MATCHING CONFIGURATIONS achieve a perfect semantic matching for all queries (the values of Columns "Max" in Table 6.1 are always < 1.0). Thus, it is important to understand if the configurations that use TOPICS and GOOGLE-PLAY perform poorly for different queries, that is, to see to what extent the word embedding models trained with TOPICS and GOOGLE-PLAY are complementary. In other words, although the configurations with TOPICS perform worse than those with GOOGLE-PLAY, it might be that TOPICS configurations

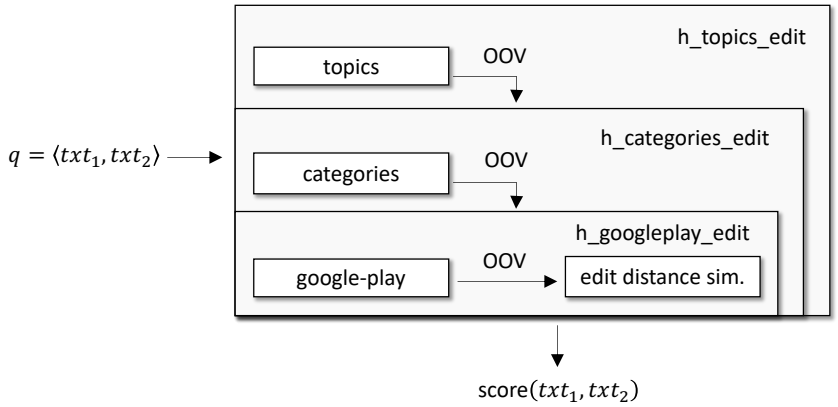


Figure 6.1. Hierarchical word embedding models

achieve good results on queries for which GOOGLE-PLAY configurations perform poorly, and vice versa.

We studied the complementarity of TOPICS and GOOGLE-PLAY by creating an artificial semantic matching configuration: *comb_topics_google-play*. This configuration considers the best ranking of the ground truth e^t_{gt} of either TOPICS and GOOGLE-PLAY for each query. Note that such a configuration cannot be constructed in a real scenario because the ground truth e^t_{gt} would be unknown.

The results shown in the bottom row of Table 6.1 suggest that there exists a moderate level of complementarity between TOPICS and GOOGLE-PLAY. The average MRR and TOP1 of *comb_topics_google-play* are higher than the ones of GOOGLE-PLAY and TOPICS with statistical significance. The average MRR of *comb_topics_google-play* is 0.0292 and 0.0486 higher than the average MRR of GOOGLE-PLAY and TOPICS, respectively. The average TOP1 of *comb_topics_google-play* is 0.0337 and 0.0575 higher than the average TOP1 of GOOGLE-PLAY and TOPICS, respectively.

The results indicate highly-specialized domain-specific corpora do not improve the effectiveness of the semantic matching of GUI events.

Chapter 7

Test Reuse

In this chapter we report the results of our evaluation of the TEST REUSE *components in* TEST REUSE *context.*

We evaluated semantic matching in the context of TEST REUSE (RQ5, RQ6, RQ7) by TEST MIGRATION EVALUATOR with selected configurations against 89 test migration scenarios. We experimented with all the scenarios that can be derived from the 8 categories in Table 4.2.

7.1 Experimental Setup

Migrating GUI test cases is time consuming due to the cost of executing GUI events on actual apps. It takes 15 minutes on average to migrate an ATM test case on our server, and 30 minutes to migrate a CRAFTDROID test case, with an overall cost of over 1,000 days of computation time to investigate the 337 configurations of our experimental setup. We designed a feasible evaluation context, by sampling the SEMANTIC MATCHING CONFIGURATIONS according to a process that considers enough configurations to study every possible instance of every component: (i) We ordered the configurations based on the MRR values that we computed when studying the semantic matching in isolation, and (ii) uniformly sampled the configuration every X positions from the top. We choose X=5 that is the highest sampling step that guarantees all the semantic matching instances to be included at least once in the study. This process selected 68 configurations. Table 7.1 reports the statistical description of MRR and TOP1. The data in the table indicate that the sampled configuration set has almost the same statistical description as the complete set of the configuration in table 5.2.

Table 7.1. Statistical description of MRR and TOP1 in 68 SEMANTIC MATCHING CONFIGURATIONS

	Min	Q1	Q2	Q3	Max	Average
MRR	0.201	0.649	0.694	0.724	0.795	0.686
TOP1	0.065	0.464	0.511	0.540	0.671	0.510

We also considered random and perfect configurations. The random configuration assigns a random score between 0 and 1 to each pair of events, and serves as the baseline, to quantify the impact of semantic and syntactic based approaches. The perfect configuration assigns the score of 1 to the correct pairs of events, based on the ground truth, 0 otherwise.

ATM considers events with a similarity score greater than a threshold as matching events (line 8 in Algorithm 6). The original ATM paper ([10]) uses a threshold optimized for the configuration considered in the paper. Our experiments indicate that the same threshold may penalize the results of ATM for other configurations and applications. Thus we decided to use different thresholds across configurations and applications.

We derived unbiased thresholds for each pair of applications and each configuration from the similarity scores computed with respect to the considered configuration for the pairs of events that occur in the other applications. Intuitively, we compute the threshold for each pair of applications $\langle A^i, A^j \rangle$ as the similarity score that best separates correct from incorrect pairs of events, by considering all pairs of applications that do not include either A^i or A^j. In details, we consider the similarity scores computed for all pairs of events in the experiment *in-isolation*. We computed the threshold of each pair of applications $\langle A^i, A^j \rangle$ from the similarity score of all pairs of events that occur in any pair of applications, but the pairs that include either A^i or A^j. We compute the threshold as the similarity score that best separates the pairs of correct and incorrect matches, that is, the score that maximizes the F1-SCORE computed for pairs above and below the threshold.

We computed the threshold for all configurations that differ in the CORPUS OF DOCUMENTS, WORD EMBEDDING and SEMANTIC MATCHING ALGORITHM, components. We did not distinguish configurations that differ in EVENT DESCRIPTOR EXTRACTOR and EVENT SELECTOR, since the semantic score does not depend on EVENT SELECTOR, and only partially on EVENT DESCRIPTOR EXTRACTOR.

Figure 7.1 shows the process of finding a threshold for migration of test cases

from application A^1 to A^4 using SEMANTIC MATCHING CONFIGURATION C_2. In the example, we consider we have a small dataset of semantic matching results including two SEMANTIC MATCHING CONFIGURATIONS C_1 and C_2, and four applications with one or two queries that have been answered. Query answer qa_i contains a source event e^s and set of target event $E^t = \langle e_1^t, e_2^t, \dots e_n^t \rangle$ which are ordered by their semantic similarity to e^s. First we select the queries that are answered by C_2 excluding those related to either of A^1 or A^4. Queries q_{12} and q_{13} satisfy the criteria and we assume C_1 and C_2 differ in a component other than EVENT DESCRIPTOR EXTRACTOR, otherwise if all other components were the same we should have considered q_5 and q_6 as well. Then, we try different thresholds in the range of 0 to 1 with 0.01 steps. We calculate F1-SCORE of each query and consider the median of F1-SCORE values as the indicator of how well a threshold can separate correct from incorrect matches. In the example T=0.40 performed better than others and we consider it as the final threshold.

Figure 7.2 shows how we calculate F1-SCORE of a query based on a given threshold. We define the confusion matrix elements as below:

TP: It is 1 if the correct match is above the threshold, 0 otherwise.

FP: Number of incorrect events above the threshold.

TN: Number of incorrect events below the threshold.

FN: It is 1 if the correct match is below the threshold, 0 otherwise.

In the example, the *query answer qa* has four target candidates in which e_3^t is the correct match. We apply threshold T=0.4 and three events including the correct match reside above the threshold. The confusion matrix elements are: one true positive, two false positives, one true negatives, and no true negative. The F1-SCORE of the query for T=0.4 is equal to 0.5.

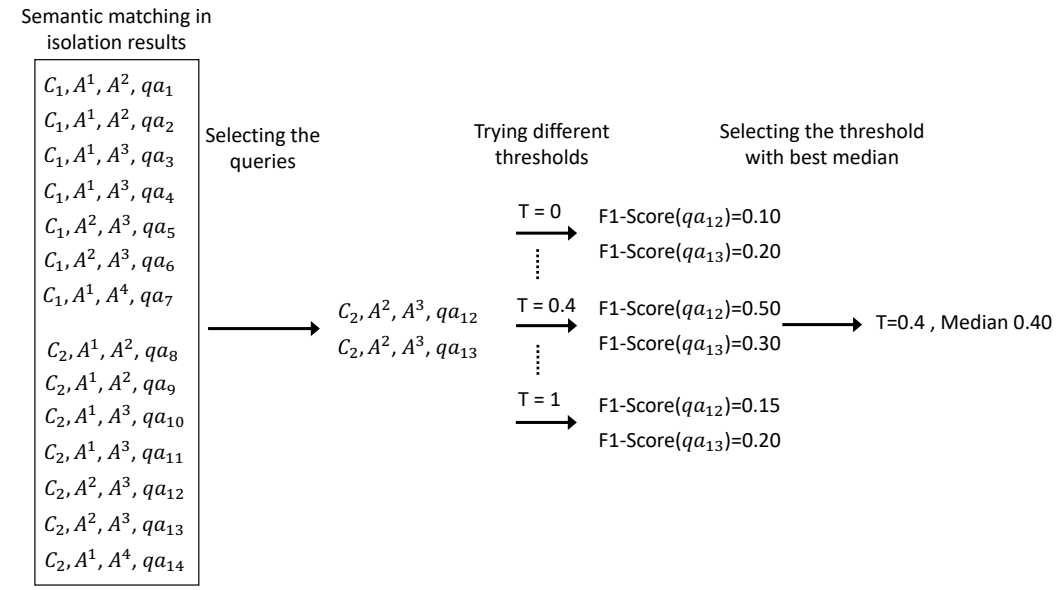


Figure 7.1. Finding the threshold for SEMANTIC MATCHING CONFIGURATION C_2 and migration from application A^1 to A^4

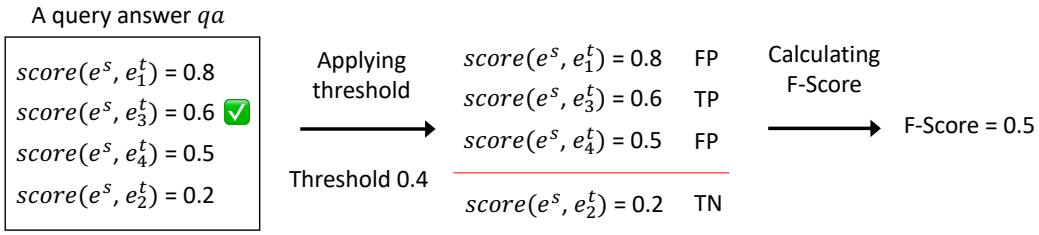
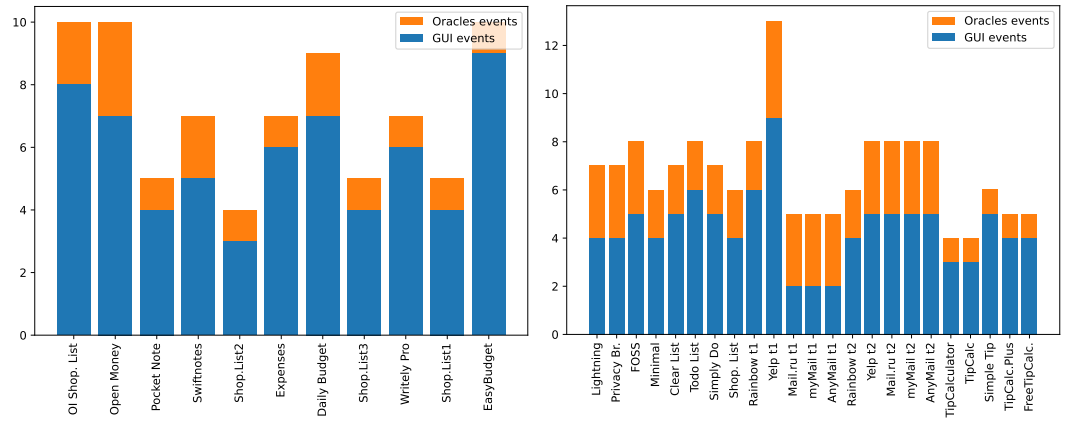


Figure 7.2. Calculating F1-SCORE of a query answer for threshold 0.4

We experimented with 34 unique source test cases, 11 shared by both ATM and CRAFTDROID and 23 test cases only with CRAFTDROID. The ground truth indicates the expected matching, as provided by the CRAFTDROID authors for CRAFTDROID test cases, and manually identified by us for the ATM scenarios.

Figures 7.3 (a) and (b) show the size of the ATM and ADAPTDROID test cases in terms of number of GUI events. Figure 7.4 shows the size of the ground truth of ATM scenarios, that is, the number of events that belong to the correct mappings across test cases of compatible applications. CRAFTDROID refers to the source test cases as the ground truth of compatible applications, thus the size of the ground truth is equal to the size of the source test cases. The figures explicitly distinguish events that correspond to assertions (oracle events). We consider both GUI and



(a) Source test cases of ATM scenarios, Mean = 7.1, STD = 2.27

(b) Source test cases of CRAFTDROID scenarios, Mean = 6.7, STD = 1.94

Figure 7.3. Size of the source test cases

oracle events, since TEST REUSE approaches migrate both GUI and oracle events with semantic matching. Oracle events commonly occur at the end of the test cases, thus the possibility of generating oracles depends on the ability to migrate all previous events in the test case. We evaluate the migrated test cases with oracles included and oracles excluded for an unbiased evaluation of both GUI and oracle events.

We assessed semantic matching in the context of TEST REUSE, by comparing the quality of the test cases that ATM and CRAFTDROID migrate with the 68 sampled configurations. We executed some sample configurations for five hours, and we identified an upper bound for each run as the maximum execution time after which no tool migrates test cases. We set the maximum execution time to 2.5 hours.

7.2 Experimental Results

We migrated all 89 scenarios, that is, test cases with the ground truth, and target application, for the 68 selected configurations, for a total of over 6,000 migrations.

RQ4. Impact of Semantic Matching in the Context of Test Reuse.

We measure the impact of semantic matching on TEST REUSE as the correlation between semantic matching metrics (MRR and TOP1) and the TEST REUSE metric (F1-SCORE). The experimental results indicate a statistically significant

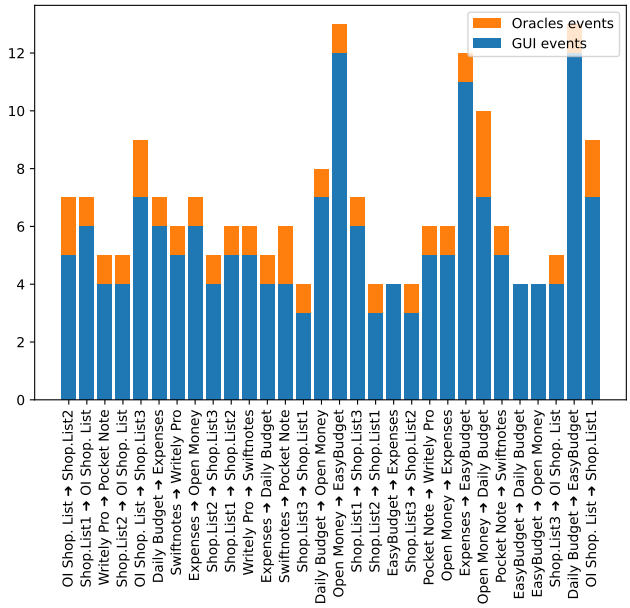


Figure 7.4. Size of the ground truth for ATM scenarios, Mean = 6.6, STD = 2.5

impact of semantic matching on TEST REUSE, and similar performance of most configurations on both ATM and CRAFTDROIDTEST GENERATOR. Tables 7.2 and 7.3 show the Pearson correlation ([82]) between the MRR and TOP1 metric values for semantic matching, on one side, and the F1-SCORE for TEST REUSE, on the other side. The tables report the correlations from experimenting with CRAFTDROID on both *all scenarios* and *shared scenarios*. They report the correlations from experimenting with ATM on the shared scenarios only, since we cannot adapt ATM for all scenarios. The tables also report the p-values that we compute to validate the statistical significance of the results. We indicate the correlation as weak ($\leq$0.3), moderate (0.3 - 0.5), or strong ($\geq$0.5), following the widely accepted classification of Cohen ([22]), and we indicate as statistically insignificant (-) results with p-vales smaller than 0.005. The results indicate either medium or strong correlation for all statistically significant cases. The results in the tables indicate that oracles do not impact significantly on the correlation.

Table 7.2. Correlation of semantic matching metrics and test reuse metric, with oracles

Test Generator	**CRAFTDROID**		**ATM**		**CRAFTDROID**	
Scenarios	**All**		**Shared**		**Shared**	
Semantic Matching Metric	**MRR**	**TOP1**	**MRR**	**TOP1**	**MRR**	**TOP1**
correlation with F1-SCORE	0.39063 [M]	0.51028 [S]	0.33981 [M]	0.42692 [M]	0.1505 [–]	0.22328 [–]
p-value	0.00099	0.00001	0.00458	0.00028	0.22057	0.06721

[S] strong correlation
[M] moderate correlation
[W] weak correlation
[–] statistically insignificant

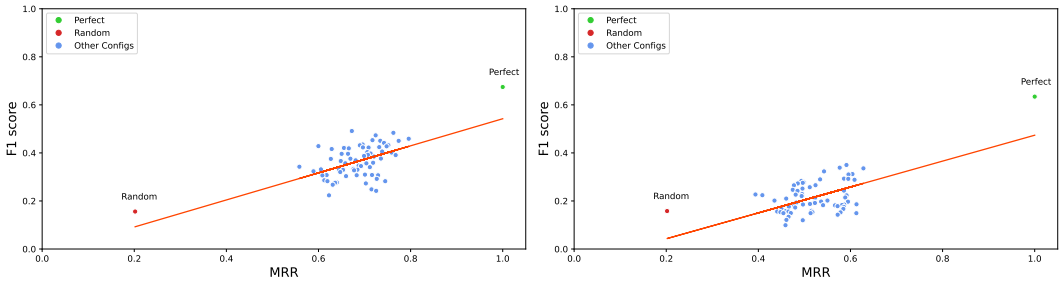
Table 7.3. Correlation of semantic matching metric and test reuse metric without oracles

Test Generator	**CRAFTDROID**		**ATM**		**CRAFTDROID**	
Scenarios	**All**		**Shared**		**Shared**	
Semantic Matching Metric	**MRR**	**TOP1**	**MRR**	**TOP1**	**MRR**	**TOP1**
correlation with F1-SCORE	0.39830 [M]	0.49938 [S]	0.34433[M]	0.45241 [M]	0.15759 [–]	0.24503 [–]
p-value	0.00077	0.00001	0.00404	0.00011	0.19934	0.04402

[S] strong correlation
[M] moderate correlation
[W] weak correlation
[–] statistically insignificant

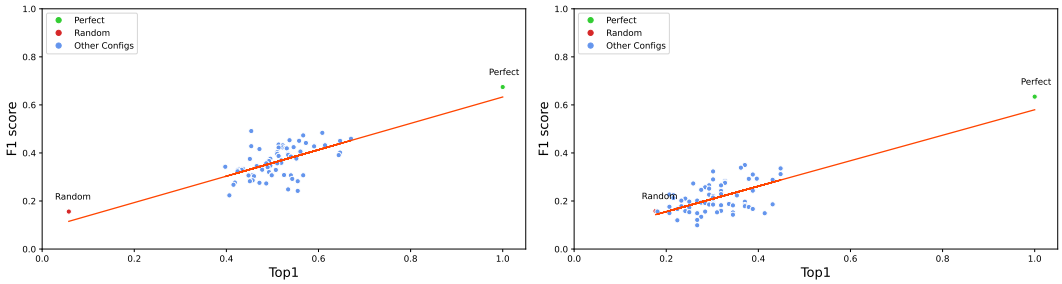
Figures 7.5 and 7.6 plot the MRR and TOP1 metrics with respect to the F1-SCORE for both CRAFTDROID and ATM executed with *all* and *shared* scenarios, respectively. The figures report also the correlations for the perfect (green dots) and random (red dots) baselines. The F1-SCORE of the perfect configuration is 0.6627 for CRAFTDROID and 0.557 for ATM. This shows SEMANTIC MATCHER is not the only factor for test migration; TEST GENERATOR plays an important role as well. The F1-SCORE of the random configuration is 0.1553 for CRAFTDROID and 0.1698 for ATM, less than most SEMANTIC MATCHING CONFIGURATIONS (blue) dots, thus confirming the effectiveness of semantic matching for TEST REUSE. The least squares polynomial fit with degree of one ([35]) (red lines) indicates the trends, which are positive in all cases, confirming the correlation between MRR and TOP1 metrics. The gap between SEMANTIC MATCHING CONFIGURATIONS

(blue) and perfect (green) dots suggests a space for improving semantic matching for TEST REUSE. The results for TOP1 are below the ones for MRR (TOP1 dots are skewed to the left comparing to MRR), since TOP1 scores 1 only if the correct candidate is exactly in the top position of the ranking while MRR scores positively also when the candidate occurs in a high position that may be different from the top position. Thus, TOP1 does not consider the case of many "good" but "not perfect" candiadtes



(a) CRAFTDROID on all scenarios (b) ATM on shared scenarios

Figure 7.5. Correlation between semantic matching (MRR) and TEST REUSE (F1-SCORE) with oracles



(a) CRAFTDROID on all scenarios (b) ATM on shared scenarios

Figure 7.6. Correlation between semantic matching (TOP1) and TEST REUSE (F1-SCORE) with oracles

We measure the effectiveness of semantic matching across ATM and CRAFTDROID TEST GENERATORS as the difference Δ (delta) between the F1-SCORE values of ATM and CRAFTDROID migrations for shared scenarios. We normalize Δ values for each approach, separately, and use a tailed t-test to determine if a mean of Δ is different from zero with p-value of 0.05. Non-zero mean values of

Δ indicate configurations with different performance for ATM and CRAFTDROID: ATM better than CRAFTDROID for positive Δ and vice versa for negative Δ.

Figure 7.7 plots all configurations sorted by normalized Δ values, and marks the negative (orange) and positive (blue) mean values. ATM and CRAFTDROID do not differ for most (average $\Delta = 0$) but six configurations: Three orange configurations on the left-hand side (CRAFTDROID better than ATM), and three blue on the right-hand side (ATM better than CRAFTDROID). Only the left most out of the six configurations that work better on either approaches corresponds to a p-value less than 0.001. Thus, the data indicate that no specific pattern between the configurations works better for one of two approaches.

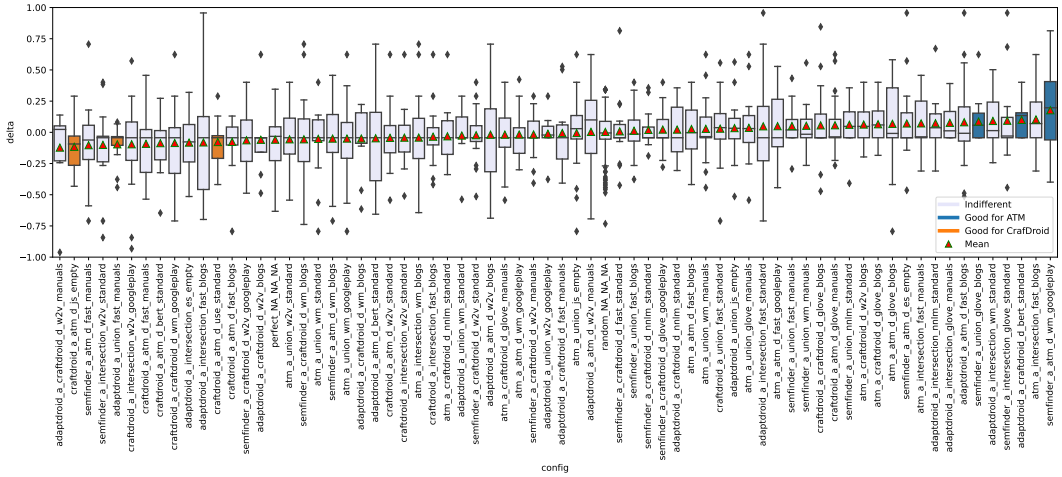


Figure 7.7. Configurations sorted by normalized Δ of F1-SCORE

The results show

(i) a medium to strong positive correlation between semantic matching and TEST REUSE, thus confirming that semantic matching impacts on test reuse,

(ii) a significant gap between the best SEMANTIC MATCHING CONFIGURATION and the perfect baseline that suggests a space for improving semantic matching for TEST REUSE.

(iii) a uniform behavior of the TEST GENERATOR in the different SEMANTIC MATCHING CONFIGURATIONS, thus confirming the generality of the TEST GENERATORS.

RQ5. Component Effectiveness in the Context of Test Reuse.

We measured the effectiveness of the components (SEMANTIC MATCHING ALGORITHM (C4), EVENT DESCRIPTOR EXTRACTOR (C3), WORD EMBEDDING (C2), CORPUS OF DOCUMENTS (C1)) on TEST REUSE, by grouping the SEMANTIC MATCHING CONFIGURATIONS by component instances, and ranking the groups according to the median F1-SCORE. Tables 7.4,7.5,7.6, 7.7 show how effective instances are with respect to a specific TEST GENERATOR and scenarios setup. The best values are in boldface, the worst in italics.

The experimental results indicate: (i) A more stable performance of SEMFINDER and ADAPTDROID_a across scenarios than ATM_a and CRAFTDROID_a SEMANTIC MATCHING ALGORITHMS; (ii) A clear ranking with ATM_d on top followed by Intersection, Union and CRAFTDROID_d as EVENT DESCRIPTOR EXTRACTOR, in this order; (iii) A better performance of WM and FASTTEXT over WORD2VEC and GLOVE WORD EMBEDDING; (iv) GOOGLE-PLAY as the best CORPUS OF DOCUMENTS for CRAFTDROID, with no relevant differences among corpora for ATM.

SEMANTIC MATCHING ALGORITHM

Table 7.4 reports the median F1-SCORE for the CRAFTDROID and ATM TEST GENERATORS grouped according to the SEMANTIC MATCHING ALGORITHM instances for both *all* and *shared* scenarios. The values for *all* scenarios benefit from the large size of the experiments. The homogeneity of values across configurations indicates an even behaviour of the different choices of the SEMANTIC MATCHING ALGORITHM component on TEST REUSE. The CRAFTDROID_a algorithm works best for the CRAFTDROID TEST GENERATOR and worst for the ATM TEST GENERATOR, while ATM_a works well for ATM TEST GENERATOR and worst for CRAFTDROID TEST GENERATOR. The SEMFINDER and ADAPTDROID algorithms work stably well with both approaches on all scenarios.

EVENT DESCRIPTOR EXTRACTOR

Table 7.5 reports the median F1-SCORE for the CRAFTDROID and ATM TEST GENERATOR grouped by instances of the EVENT DESCRIPTOR EXTRACTOR component. The median F1-SCORE values in the table clearly indicate a ranking: ATM_d, Intersection, Union, CRAFTDROID_d, with and without oracles, and for all sets of setups, for both ATM and CRAFTDROID but CRAFTDROID with oracles on shared scenarios.

WORD EMBEDDING

Table 7.4. Median F1-SCORE values of MRR grouped by SEMANTIC MATCHING ALGORITHM

Test Generator		**CRAFTDROID**			**CRAFTDROID**			**ATM**		
Scenarios		**All**			**Shared**			**Shared**		
Instance	**#conf.**[a]	oracles	no oracles	rank[b]	oracles	no oracles	rank	oracles	no oracles	rank
CRAFTDROID_a	13	**0.3974**	**0.4978**	1	**0.1936**	**0.2219**	1	*0.1782*	*0.2018*	4
SEMFINDER	18	0.3678	0.4503	2	0.1761	0.2050	3	0.1847	0.2160	3
ADAPTDROID_a	20	0.3496	0.4369	3	0.1841	0.2174	2	**0.2251**	**0.2498**	1
ATM_a	17	*0.3336*	*0.4192*	4	*0.1741*	*0.1963*	4	0.2127	0.2354	2

[a] Number of SEMANTIC MATCHING CONFIGURATIONS that use the instance
[b] Rank of the instance in the specific TEST GENERATOR and scenario setup

Table 7.5. Median F1-SCORE values of MRR grouped by EVENT DESCRIPTOR EXTRACTOR

Test Generator		**CRAFTDROID**			**CRAFTDROID**			**ATM**		
Scenarios		**All**			**Shared**			**Shared**		
Instance	**#conf.**[a]	oracles	no oracles	rank[b]	oracles	no oracles	rank	oracles	no oracles	rank
ATM_d	19	**0.4214**	**0.5169**	1	**0.2539**	**0.2888**	1	**0.2729**	**0.3036**	1
Intersection	14	0.3581	0.4434	2	0.1786	0.2072	2	0.2274	0.2680	2
Union	16	0.3316	0.4115	3	0.1656	*0.1644*	3,4	0.1888	0.2109	3
CRAFTDROID_d	19	*0.3070*	*0.3712*	4	0.1428	*0.2050*	4,3	*0.1818*	*0.2081*	4

[a] Number of SEMANTIC MATCHING CONFIGURATIONS that use the instance
[b] Rank of the instance in the specific TEST GENERATOR and scenario setup

Table 7.6. Median F1-SCORE values of MRR grouped by WORD EMBEDDING

Test Generator		**CRAFTDROID**			**CRAFTDROID**			**ATM**		
Scenarios		**All**			**Shared**			**Shared**		
Instance	**#conf.**[a]	oracles	no oracles	rank[b]	oracles	no oracles	rank	oracles	no oracles	rank
WM	13	**0.4190**	**0.5023**	1	**0.2019**	**0.2372**	1	0.2017	0.2234	2
FastText	17	0.3302	0.4192	3	0.1722	0.2153	3	**0.2099**	**0.2352**	1
W2V	14	0.3432	0.4236	2	0.1880	0.2197	2	*0.1513*	*0.1694*	4
GloVE	10	*0.3040*	*0.3700*	4	*0.1100*	*0.1224*	4	0.1870	0.2071	3

[a] Number of SEMANTIC MATCHING CONFIGURATIONS that use the instance
[b] Rank of the instance in the specific TEST GENERATOR and scenario setup

Table 7.6 reports the median F1-SCORE for the CRAFTDROID and ATM TEST GENERATORS grouped by the WORD EMBEDDING instances. The table reports the values for the instances that occur at least ten times in the sampled configurations: WM, GLOVE, FASTTEXT, and WORD2VEC. The values indicate that WM and FASTTEXT perform better than WORD2VEC and GLOVE.

Figures 7.8 and 7.9 compare syntactic and semantic configurations, by plotting the configurations sorted by mean F1-SCORE. We aggregate semantic configurations by means, since both MRR and TOP1 aggregate the results by means. The yellow box plots indicate the five configurations that implement syntactic techniques, the green box plot indicates the perfect configuration, and the red box plot the random configuration. The distribution of syntactic configurations in both figures indicates no relevant differences between syntactic and semantic configurations, however the limited amount of syntactic configurations does not allow us to generalize the results. While the perfect configuration subsumes all configurations, the random configuration subsumes some configurations for ATM. This explains the lower F1-SCORE values for *shared* scenarios than *all* scenarios in Table 7.6. We argue that the effectiveness of WORD EMBEDDING models depends on scenarios: The more the scenarios are extensive, the more they benefit from semantic approaches.

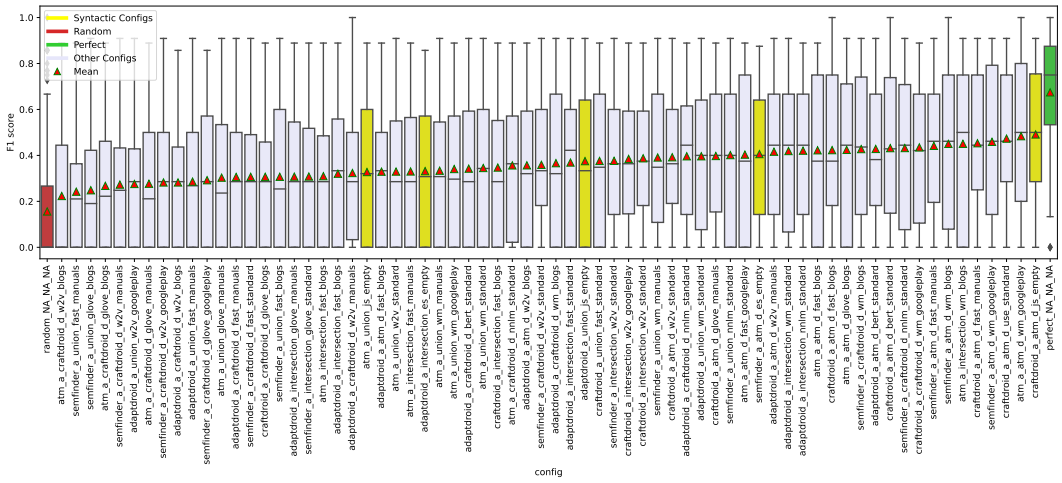


Figure 7.8. Range of F1-SCORE per SEMANTIC MATCHING CONFIGURATIONS in CRAFTDROID with all scenarios, Ordered by mean

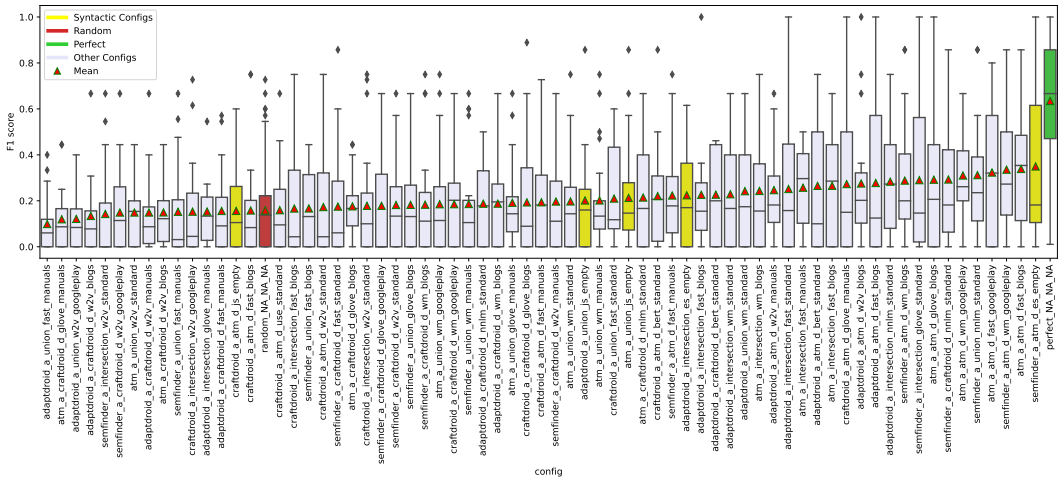


Figure 7.9. Range of F1-SCORE per SEMANTIC MATCHING CONFIGURATIONS in ATM with shared scenarios, Ordered by mean

CORPUS OF DOCUMENTS

Table 7.7 reports the median F1-SCORE for the CRAFTDROID and ATM TEST GENERATORS grouped by the CORPUS OF DOCUMENTS component. The values in the table indicate the independence of ATM from the choice of CORPUS OF DOCUMENTS, while CRAFTDROID TEST GENERATOR perform best for GOOGLE-PLAY and worst for MANUALS.

Table 7.7. Median F1-SCORE values of MRR grouped by CORPUS OF DOCUMENTS

Test Generator		**CRAFTDROID**			**CRAFTDROID**			**ATM**		
Scenarios		**All**			**Shared**			**Shared**		
Instance	**#conf.**[a]	oracles	no oracles	rank[b]	oracles	no oracles	rank	oracles	no oracles	rank
Google Play	9	**0.3848**	**0.4772**	1	**0.2019**	**0.2296**	1	*0.1851*	*0.2084*	3
Blogs	18	0.3382	0.4223	2	0.1804	0.2111	2	0.1906	**0.2233**	2,1
Manuals	15	*0.3234*	*0.3746*	3	*0.1548*	*0.1763*	3	**0.1916**	0.2190	1,2

[a] Number of SEMANTIC MATCHING CONFIGURATIONS that use the instance
[b] Rank of the instance in the specific TEST GENERATOR and scenario setup

The results indicate that

(i) semantic matching instances that are effective in isolation perform well also in TEST REUSE, however, semantic matching instances that are less effective in isolation may still perform well in TEST REUSE,

(ii) both SEMFINDER and ADAPTDROID algorithms work consistently well across TEST GENERATORS,

(iii) ATM_d and WM are the best instances for EVENT DESCRIPTOR EXTRACTOR and WORD EMBEDDING, respectively,

(iv) GOOGLE-PLAY performs best with CRAFTDROID TEST GENERATOR, while all CORPUS OF DOCUMENTS instances work similarly with ATM TEST GENERATOR.

RQ6. Component Impact Analysis in the Context of Test Reuse. Table 7.8 reports the median of standard deviation for both CRAFTDROID and ATM TEST GENERATOR grouped by components. The values in the table indicate that EVENT DESCRIPTOR EXTRACTOR is the most impactful component on TEST REUSE, with F1-SCORE values far higher than the other components for all setups. SEMANTIC MATCHING ALGORITHM, and WORD EMBEDDING follow EVENT DESCRIPTOR EXTRACTOR with different comparative performance depending on the TEST GENERATOR and scenarios. The F1-SCORE values ranks CORPUS OF DOCUMENTS as the least impactful component on TEST REUSE for all setups.

The results in the table depend on the scenarios: SEMANTIC MATCHING ALGORITHM ranks third for CRAFTDROID with *shared* scenarios, and fourth with *all* scenarios. The results with *all* scenarios indicate a similar impact of SEMANTIC MATCHING ALGORITHM and WORD EMBEDDING for CRAFTDROID. The plots in

Table 7.8. Median values of F1-SCORE grouped by Components

Test Generator	**CRAFTDROID**			**CRAFTDROID**			**ATM**		
Scenarios	**All**			**Shared**			**Shared**		
Component	oracles	no oracles	rank[b]	oracles	no oracles	rank	oracles	no oracles	rank
EVENT DESCRIPTOR EXTRACTOR	**0.0536**	**0.0835**	1	**0.0688**	**0.0779**	1	**0.0517**	**0.0704**	1
WORD EMBEDDING	0.0352	0.0436	2	0.0360	0.0481	3	0.0173	0.0226	3
SEMANTIC MATCHING ALGORITHM	0.0333	0.0293	3	0.0538	0.0574	2	0.0195	0.0289	2
CORPUS OF DOCUMENTS	*0.0139*	*0.0245*	4	*0.0299*	*0.0331*	4	*0.0161*	*0.0189*	4

[b] Rank of the instance in the specific TEST GENERATOR and scenario setup

Figure 7.10 confirm the similar performance of SEMANTIC MATCHING ALGORITHM and WORD EMBEDDING for CRAFTDROID and ATM TEST GENERATOR.

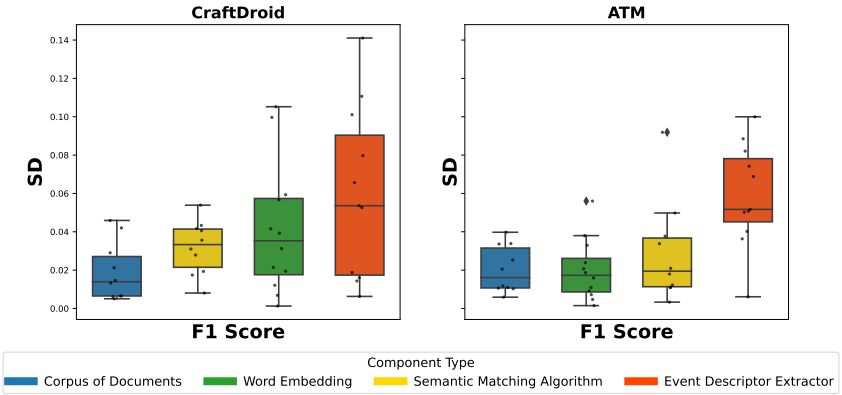


Figure 7.10. Impact analysis of the components of CRAFTDROID TEST GENERATOR (all scenarios) and ATM TEST GENERATOR (shared subjects only)

The results indicate that

(i) EVENT DESCRIPTOR EXTRACTOR is the most impactful component in TEST REUSE,

(ii) SEMANTIC MATCHING ALGORITHM and WORD EMBEDDING are the next impactful components and their impact depends on both the TEST GENERATOR and the subjects,

(iii) CORPUS OF DOCUMENTS is the least impactful component.

7.2.1 Discussion

The experimental results that we discuss in this section indicate the importance of semantic matching in TEST REUSE. They also show a substantial gap between the best SEMANTIC MATCHING CONFIGURATIONS and the perfect mapping (Figures 7.8 and 7.9). The gap indicates space for improvement.

The results clearly indicate the contribution of the different instances for the semantic matching components. Here we compare the results of the experiment of semantic matching in isolation (RQ1, RQ2 and RQ3) that we discuss in detail in Section 5.2, to the results in the context of TEST REUSE (RQ5, RQ6 and RQ7) that we discuss in detail in Section 7.2. Since different metrics captured the effectiveness of semantic matching in isolation (MRR and TOP1) and in the context of TEST REUSE (F1-SCORE), we compare the results qualitatively.

- SEMANTIC MATCHING ALGORITHM: The evaluation in isolation indicates that SEMFINDER performs best and ADAPTDROID worst among the evaluated SEMANTIC MATCHING ALGORITHM. The evaluation in the context of TEST REUSE does not reveal substantial differences among the evaluated SEMANTIC MATCHING ALGORITHM, all of which perform well. SEMFINDER and ADAPTDROID semantic matching algorithms are not paired with a specific TEST REUSE approach, and perform evenly well when paired with any TEST GENERATOR. ATM_a and CRAFTDROID_a SEMANTIC MATCHING ALGORITHMS perform best when paired with the corresponding TEST GENERATOR, worst otherwise. These results suggest that SEMFINDER *could be a safe choice since it performed well in all contexts*, regardless of the test generation approach used.

- EVENT DESCRIPTOR EXTRACTOR: Both the evaluation in isolation and in the context of TEST REUSE indicate that EVENT DESCRIPTOR EXTRACTORS perform with almost the same order of effectiveness, with ATM_*d outperforming the others*.

- WORD EMBEDDING: Both the evaluation in isolation and in the context of TEST REUSE indicate that WM performs better than the other WORD EMBEDDING techniques. FASTTEXT performs well, although often worse than WM, only in the context of TEST REUSE. We speculate that this result might derive from the capability of FASTTEXT to handle out-of-vocabulary issues

more effectively than the other word embedding that we consider in the study, being out-of-vocabulary a phenomenon that occurs often when processing a large variety of diverse words. Overall, *the results clearly indicate WM as the best option for the task of semantic matching, among the considered techniques*.

- CORPUS OF DOCUMENTS**:** Both the evaluation in isolation and in the context of TEST REUSE suggests that *the best corpora to use may depend on how the corpora is used*, and this on the TEST REUSE technique. In fact, Google Play is the best corpora for CRAFTDROID, while it performs worst for ATM.

- **Impact of the components on TEST REUSE:** The four components (CORPUS OF DOCUMENTS, WORD EMBEDDING, SEMANTIC MATCHING ALGORITHM, EVENT DESCRIPTOR EXTRACTOR) have a different impact on TEST REUSE, depending on the TEST GENERATOR (ATM, CRAFTDROID), and the impact varies from evaluation in isolation and in the context of TEST REUSE (Figures 5.4, and 7.10). The evaluation in isolation ranks SEMANTIC MATCHING ALGORITHM first, then WORD EMBEDDING, EVENT DESCRIPTOR EXTRACTOR and CORPUS OF DOCUMENTS. The evaluation in the context of TEST REUSE ranks EVENT DESCRIPTOR EXTRACTOR first, then SEMANTIC MATCHING ALGORITHM and WORD EMBEDDING between the second and third place depending on the TEST REUSE TEST GENERATOR, and finally CORPUS OF DOCUMENTS. From these results, we can deduce that *the set of extracted descriptors and the way these descriptors are processed are the most important and impactful part of the semantic matching process*. While, *the* WORD EMBEDDING*, and even more the choice of the* CORPUS OF DOCUMENTS*, play a minor role* in the semantic matching process. Interestingly, results also highlight how there is a *gap to fill* with respect to the performance of the ideal semantic matching process.

7.2.2 Implications of the results

We conclude this section by discussing the implications of the results from three different viewpoints that can drive future research in improving semantic matching for test reuse: choice of instances, impact of components and test reuse.

Choice of instances: We observe that best results when relying on the instances that are robust in both isolation and test reuse context. The results suggest that

the most robust instances of the different components are SEMFINDER for the WORD EMBEDDING, WM and in general sentence level embedding for the WORD EMBEDDING, ATM_d for the EVENT DESCRIPTOR EXTRACTOR, Google-Play for the CORPUS OF DOCUMENTS.

Impact of components: We observe that the components impact very differently in the context of semantic matching and test reuse. The EVENT DESCRIPTOR EXTRACTOR is the most impactful component in the context of test reuse, even if it follows both the SEMANTIC MATCHING ALGORITHM and the WORD EMBEDDING when considering semantic matching only. We speculate that that choice of the next events highly depends on the current ones, and the EVENT DESCRIPTOR EXTRACTOR plays the most important role because it it the gateway of necessary information for the SEMANTIC MATCHING EVALUATOR. Based on these results we encourage exploring new attributes to add more available information. We observe that the CORPUS OF DOCUMENTS is the least impactful, and ATM is not very sensitive to the choice of corpus.

Test Reuse: The gap between the perfect semantic matching configuration and the perfect Test Reuse indicates a relevant space for improving the TEST GENERATOR. Our investigation indicates that the incompleteness of the Target Application Model is an important obstacle of finding the correct match. The instances of the TEST GENERATOR that we evaluated find it difficult to find the correct matching when the events belong to different windows.

Chapter 8

Conclusions

Automatically reusing test cases across similar applications is a recent and promising way to efficiently test interactive applications. Current TEST REUSE approaches migrate test cases from a source to a target application, and rely on semantic matching to find corresponding events between source and target applications. Semantic matching plays a crucial role not only in TEST REUSE, but also in some of the PATTERN-BASED approaches that use semantic matching to match elements of the patterns to events of target applications [66, 42].

Current TEST REUSE approaches combine different techniques and present different results. Comparing the different approaches offers the opportunity to define new approaches with better performance than current ones. FrUITeR, the only publicly available framework we are aware of, allows to automatically compare TEST REUSE approaches as a whole, but does not support the comparison of the contribution of the different components of the approaches, and in particular of the core semantic matching component.

In this thesis, we propose a general ARCHITECTURE of TEST REUSE approaches. We define a framework to comparatively evaluate the core components of TEST REUSE approaches and the different combinations of the instances of the different components. We discuss combination of new instances of the components to produce new efficient TEST REUSE approaches that supersede current approaches, and we offer the framework in an open replication package to evaluate new combinations of instances and obtain new TEST REUSE approaches.

The ARCHITECTURE that we proposed in this thesis encompasses a SEMANTIC MATCHER and a TEST GENERATOR. The SEMANTIC MATCHER is composed of four subcomponents: i) CORPUS OF DOCUMENTS ii) WORD EMBEDDING iii) EVENT DESCRIPTOR EXTRACTOR iv) SEMANTIC MATCHING ALGORITHM. The TEST GENERATOR includes a EVENT SELECTOR. The SEMANTIC MATCHER identifies events

of the source and the target application that are semantically similar by relying on word embedding techniques, which score similarity of the textual descriptor of events. The TEST GENERATOR leverages the similarities between events to generate the test cases for the target application based on the specific strategies. The ARCHITECTURE provides the ability to study and evaluate TEST REUSE approaches at the fine granularity of the components. In this thesis we identified 25 relevant instances for the SEMANTIC MATCHER components and two instances of EVENT SELECTOR. Each combination of SEMANTIC MATCHER instances creates a unique semantic matching strategy (SEMANTIC MATCHING CONFIGURATION).

We evaluated semantic matching in isolation with the SEMANTIC MATCHING EVALUATOR, a framework that automatically evaluates the impact of the components and the effectiveness of the instances. We conducted our empirical evaluation with 147 unique test migration scenarios from 30 applications. The results of our experiments show that i) sentence level word embedding techniques perform better than world level ones, ii) all semantic matching components are impactful, iii) the quality of semantic information is more important than the quantity, iv) word embedding models trained on corpora of documents specific to mobile app domain perform generally better than general corpora.

Following our insights on the effectiveness of domain specific CORPUS OF DOCUMENTS, we hypothesized that highly specialized corpora of documents can improve the effectiveness of semantic matching. To evaluate our hypothesis, we used topic modeling to partition GOOGLE-PLAY corpus into 27 domain specific clusters of documents. We created word embedding models with FASTTEXT, GLOVE, WM, and WORD2VEC techniques, and evaluated 240 SEMANTIC MATCHING CONFIGURATIONS that use the specialized WORD EMBEDDING models. The results of our experiments suggests there exists an optimal point for the specialized corpuses. We also observed the complementarity between the models trained with GOOGLE-PLAY and the ones trained with clusters.

To evaluate semantic matching in TEST REUSE context, we proposed TEST MIGRATION EVALUATOR, a framework that automatically evaluates both the impact of the components and the effectiveness of the instances, based on the quality of the generated test cases. The results of an empirical evaluation with 89 unique test migration scenarios show that i) the semantic matching of GUI events is highly correlated with the performance of TEST REUSE ii) SEMANTIC MATCHING ALGORITHM and EVENT DESCRIPTOR EXTRACTOR are the most impactful components in context of Test Reuse, iii) our new SEMANTIC MATCHING ALGORITHM performs consistently well both in-isolation and in the context of TEST REUSE. iv) there is a relevant space for further improving performance of TEST REUSE.

8.1 Contributions

This thesis offers the following contributions.

A general ARCHITECTURE of TEST REUSE approaches: We define a general conceptual ARCHITECTURE for TEST REUSE approaches that abstracts internal component of the approaches and their interactions. The ARCHITECTURE embodies both the workflow of the current approaches and the main processes that we modularize as components. The components are implemented with interchangeable instances; From the object-oriented prospective, the components play the role of classes, and the instances of objects.

New instances for the SEMANTIC MATCHER components: We evaluated all instances proposed in the state-of-the-art TEST REUSE approaches as well as both new instances for the identified components and popular artifacts of NLP community: SEMFINDER a new SEMANTIC MATCHING ALGORITHM, GOOGLE-PLAY a CORPUS OF DOCUMENTS, and TOPICS a highly specialized corpora of documents. The current SEMANTIC MATCHING ALGORITHMS are either too restrictive and thus miss contextual information or too general and thus suffer from unavoidable noise. SEMFINDER offers a good trade-off, and outperforms the current SEMANTIC MATCHING ALGORITHM in semantic matching both in-isolation and in the context of TEST REUSE.
We built the GOOGLE-PLAY corpus by crawling Google Play application descriptors, and we used the GOOGLE-PLAY corpus to train WORD EMBEDDING models specialized for the mobile app domain. We defined SEMANTIC MATCHING CONFIGURATIONS that uses GOOGLE-PLAY to obtain high MRR and TOP1 scores for semantic matching in-isolation.
We defined TOPICS, corpora of documents with apps descriptions specific to the domain of mobile applications. We used topic modeling to cluster documents such that each cluster corresponds to the corpus of documents containing description of apps with similar functionalities. We trained WORD EMBEDDING models with TOPICS corpora to obtain models for specific domains of apps.

Evaluation Frameworks: We propose two frameworks to automatically evaluate semantic matching both in isolation and in the context of TEST REUSE. The frameworks integrate different instances of components to evaluate any combination of instances. The SEMANTIC MATCHING EVALUATOR framework assesses SEMANTIC MATCHING CONFIGURATIONS in isolation from TEST REUSE and relies on a data set of source and target candidate events. The TEST MIGRATION EVALUATOR framework integrates a modified version of the SEMANTIC MATCHING EVAL-

UATOR for assessing semantic matching in isolation, and includes the TEST GENERATOR component to assess semantic matching in the context of TEST REUSE. The frameworks can systematically explore possible combination of instances to assess impact of components and effectiveness of instances.

In-depth insight and empirical evaluation: We discuss the results of our experiments with 337 SEMANTIC MATCHING CONFIGURATIONS in isolation with a data set of 8,099 GUI events, and of our experiments with 68 configurations on 6,000 test migrations. Our comprehensive empirical study provides important insights about the relative impact of the different components, and the dependency of TEST REUSE approaches on semantic matching. The results of our experiment show that even the best SEMANTIC MATCHING CONFIGURATIONS score a max of 0.79 and 0.67 with MRR and TOP1 metrics, respectively, quite far from a perfect semantic matching. Our results also indicates that the perfect semantic matching achieves F1-SCORE of 0.65 in TEST REUSE context, thus indicating that semantic matching is not the only impactful component on TEST REUSE.

We presented the results of the study of semantic matching in isolation and the SEMANTIC MATCHING EVALUATOR framework at the 2021 International Symposium on Software Testing and Analysis [70]. We presented the results of domain specific word embedding study at 2022 International Conference on Program Comprehension [46]. We present the results of the study of semantic matching in the context of TEST REUSE and the TEST MIGRATION EVALUATOR framework in a paper that we submitted to the Empirical Software Engineering, and that is under review at the time of writing.

8.2 Open Research Directions

The results presented in this thesis opens new research directions.

Toward perfect semantic matching: Our empirical study indicates a relevant gap between the performance of the semantic matching approaches that we evaluated in the context of TEST REUSE and the ideal SEMANTIC MATCHING CONFIGURATION, perfect semantic matching baseline. Thus, the results indicate a big space for improvement. A possible research direction moves towards leveraging other sources of semantic information that are available in the GUI to reduce the gap between current semantic matching and the perfect baseline. Another relevant research direction moves towards the use of non-textual information. Current semantic approaches rely on textual information only, and ignore the

relevant visual information often available in the GUI. For example, the image of a pencil is often attached to a button to indicate an *edit* functionality. Some random approaches [105] use computer vision to find actionable events, while some PATTERN-BASED approaches [42] use computer vision to match widgets to patterns elements. Leveraging computer vision to take advantage of visual information available in the GUI can enhance semantic matching of GUI events.

From one-to-one to many-to-many mapping: The current semantic matching techniques consider only one-to-one mappings. A relevant challenge of TEST REUSE is the presence of many-to-many mapping between source and target events. The current TEST REUSE approaches identify some one-to-many mapping, since they consider ancillary events required to reach a state that contains one-to-one mapping. However, in many scenarios finding ancillary events is difficult, and the current TEST REUSE approaches miss many ancillary events. Considering macro events, that is, compositions of multiple events, can enhance semantic matching.

Improving TEST REUSE beyond semantic matching: The results of our empirical study show that even the perfect matching baseline does not result into a perfect test migration. Studying other aspects of TEST REUSE beyond semantic matching may improve TEST REUSE. Our intuition is that the dependency of TEST REUSE approaches on TARGET APPLICATION MODEL in states where semantic matching cannot find the correct match plays an important role in TEST REUSE. Reinforcement learning can be a suitable solution to produce effective models of the GUI to query for the next events in the generated test.

Bibliography

[1] Ahmed Abbasi, Hsinchun Chen, and Arab Salem. “Sentiment analysis in multiple languages: Feature selection for opinion classification in web forums.” In: *ACM Transactions on Information Systems (TOIS)* 26.3 (2008), pp. 1–34.

[2] Domenico Amalfitano et al. “Testing android mobile applications: Challenges, strategies, and approaches.” In: *Advances in Computers*. Vol. 89. Elsevier, 2013, pp. 1–52.

[3] Domenico Amalfitano et al. “Using GUI Ripping for Automated Testing of Android Applications.” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE ’12. ACM, 2012, pp. 258–261.

[4] Saswat Anand et al. “Automated Concolic Testing of Smartphone Apps.” In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’12. ACM, 2012, pp. 1–11.

[5] Ebru Arisoy et al. “Deep neural network language models.” In: *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*. 2012, pp. 20–28.

[6] Stephan Arlt, Andreas Podelski, and Martin Wehrle. “Reducing GUI test suites via program slicing.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 270–281.

[7] Earl T. Barr et al. “The Oracle Problem in Software Testing: A Survey.” In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525.

[8] Giovanni Becce et al. “Extracting Widget Descriptions from GUIs.” In: *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. FASE ’12. Springer, 2012, pp. 347–361.

[9] Farnaz Behrang and Alessandro Orso. “Poster: Automated Test Migration for Mobile Apps.” In: *Proceedings of the International Conference on Software Engineering*. ICSE Poster ’18. ACM, 2018, pp. 384–385.

[10] Farnaz Behrang and Alessandro Orso. “Test migration between mobile apps with similar functionality.” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE’19. IEEE Computer Society. 2019, pp. 54–65.

[11] Farnaz Behrang and Alessandro Orso. “Test Migration for Efficient Large-scale Assessment of Mobile App Coding Assignments.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’18. ACM, 2018, pp. 164–175.

[12] Iz Beltagy, Kyle Lo, and Arman Cohan. “SciBERT: A Pretrained Language Model for Scientific Text.” In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3615–3620.

[13] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. “Latent Dirichlet Allocation.” In: *Journal of machine Learning research* 3 (Jan. 2003), pp. 993–1022.

[14] Marcel Böhme and Paul Soumya. “On the Efficiency of Automated Testing.” In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’14. ACM, 2014, pp. 71–80.

[15] Piotr Bojanowski et al. “Enriching word vectors with subword information.” In: *Transactions of the association for computational linguistics* 5 (2017), pp. 135–146.

[16] Daniel Cer et al. *Universal Sentence Encoder*. 2018. arXiv: 1803.11175 [cs.CL].

[17] Jonathan Chang et al. “Reading Tea Leaves: How Humans Interpret Topic Models.” In: *Advances in Neural Information Processing Systems*. Ed. by Y. Bengio et al. Vol. 22. Curran Associates, Inc., 2009.

[18] Yong Chen et al. “Experimental explorations on short text topic mining between LDA and NMF based Schemes.” In: *Knowledge-Based Systems* 163 (2019), pp. 1–13.

[19] Lin Cheng et al. “GUICat: GUI testing as a service.” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE ’16. ACM, 2016, pp. 858–863.

[20] Wontae Choi, George Necula, and Koushik Sen. “Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning.” In: *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’13. ACM, 2013, pp. 623–640.

[21] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. “Automated Test Input Generation for Android: Are We There Yet?” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE ’16. IEEE Computer Society, 2015, pp. 429–440.

[22] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[23] MJ Crick and MD Hill. “The role of sensitivity analysis in assessing uncertainty.” In: *Uncertainty analysis for performance assessments of radioactive waste disposal systems*. 1987.

[24] Matthew J Denny and Arthur Spirling. “Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it.” In: *Political Analysis* 26.2 (2018), pp. 168–189.

[25] Matthew J Denny and Arthur Spirling. “Text preprocessing for unsupervised learning: Why it matters, when it misleads, and what to do about it.” In: *Political Analysis* 26.2 (2018), pp. 168–189.

[26] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding.” In: *arXiv* (2018).

[27] Sergio Di Martino et al. “Comparing the effectiveness of capture and replay against automatic input generation for android graphical user interface testing.” In: *Software Testing, Verification and Reliability* 31.3 (2021), e1754.

[28] Alan Dix. “Human-computer interaction.” In: *Encyclopedia of database systems*. Springer, 2009, pp. 1327–1331.

[29] Zhen Dong et al. “Time-travel testing of android apps.” In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE. 2020, pp. 481–492.

[30] Susan T Dumais et al. “Using Latent Semantic Analysis to Improve Access to Textual Information.” In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’88. Association for Computing Machinery, 1988, pp. 281–285.

[31] Fahimeh Ebrahimi, Miroslav Tushev, and Anas Mahmoud. “Classifying mobile applications using word embeddings.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2021), pp. 1–30.

[32] Markus Ermuth and Michael Pradel. “Monkey see, monkey do: Effective generation of GUI tests with inferred macro events.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’16. ACM. 2016, pp. 82–93.

[33] George Forman. “A Pitfall and Solution in Multi-Class Feature Selection for Text Classification.” In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML ’04. Association for Computing Machinery, 2004, p. 38.

[34] Svetoslav Ganov et al. “Event listener analysis and symbolic execution for testing GUI applications.” In: *Formal Methods and Software Engineering*. Springer, 2009, pp. 69–87.

[35] Joseph Diaz Gergonne. “The application of the method of least squares to the interpolation of sequences.” In: *Historia Mathematica* 1.4 (1974), pp. 439–447.

[36] Florian Gross, Gordon Fraser, and Andreas Zeller. “Search-based system testing: high coverage, no false alarms.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’12. ACM, 2012, pp. 67–77.

[37] Tianxiao Gu et al. “Practical GUI Testing of Android Applications Via Model Abstraction and Refinement.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. ICSE ’2019. May 2019, pp. 269–280.

[38] Yu Gu et al. “Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing.” In: *ACM Trans. Comput. Healthcare* 3.1 (Oct. 2021). ISSN: 2691-1957.

[39] Giovani Guizzo et al. “A multi-objective and evolutionary hyper-heuristic applied to the Integration and Test Order Problem.” In: *Applied Soft Computing* 56 (2017), pp. 331–344.

[40] DM Hamby. “A comparison of sensitivity analysis techniques.” In: *Health physics* 68.2 (1995), pp. 195–204.

[41] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. “Search-based software engineering: Trends, techniques and applications.” In: *ACM Computing Surveys* 45.1 (2012), p. 11.

[42] Gang Hu, Linjie Zhu, and Junfeng Yang. “AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests.” In: *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE ’18. ACM, 2018, pp. 269–282.

[43] Yuening Hu et al. “Interactive topic modeling.” In: *Machine learning* 95.3 (2014), pp. 423–469.

[44] Qilu Jiao and Shunyao Zhang. “A Brief Survey of Word Embedding and Its Recent Development.” In: *2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. Vol. 5. 2021, pp. 1697–1701. DOI: 10.1109/IAEAC50856.2021.9390956.

[45] Elias Khalil, Mustafa Assaf, and Abdel Salam Sayyad. “Human Resource Optimization for Bug Fixing: Balancing Short-Term and Long-Term Objectives.” In: *Search Based Software Engineering*. Ed. by Tim Menzies and Justyna Petke. Springer International Publishin, 2017, pp. 124–129.

[46] Farideh Khalili et al. “The Ineffectiveness of Domain-Specific Word Embedding Models for GUI Test Reuse.” In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 2022, pp. 560–564.

[47] Matt J. Kusner et al. “From Word Embeddings to Document Distances.” In: *Proceedings of the International Conference on International Conference on Machine Learning*. ICML ’15. 2015, pp. 957–966.

[48] Andrei Kutuzov et al. “Word vectors, reuse, and replicability: Towards a community repository of large-text resources.” In: *Proceedings of the 58th Conference on Simulation and Modelling*. Linköping University Electronic Press. 2017, pp. 271–276.

[49] Vladimir I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals*. Tech. rep. 8. Soviet Physics Doklady, 1966, pp. 707–710.

[50] Guodong Li, Esben Andreasen, and Indradeep Ghosh. “SymJS: Automatic Symbolic Testing of JavaScript Web Applications.” In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Association for Computing Machinery, 2014, pp. 449–459.

[51] Hongmin Li et al. “Comparison of word embeddings and sentence encodings as generalized representations for crisis tweet classification tasks.” In: *Proceedings of ISCRAM Asia Pacific* (2018).

[52] Yuan-Fang Li, Paramjit K. Das, and David L. Dowe. “Two decades of Web application testingA survey of recent advances.” In: *Information Systems* 43 (2014), pp. 20–54.

[53] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. “Test Transfer Across Mobile Apps Through Semantic Mapping.” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE’19. IEEE Computer Society, 2019, pp. 42–53.

[54] Jun-Wei Lin and Sam Malek. “GUI Test Transfer from Web to Android.” In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2022, pp. 1–11.

[55] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. “Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing.” In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 399–410.

[56] Mario Linares-Vásquez et al. “Mining android app usages for generating actionable gui-based execution scenarios.” In: *Proceedings of the Working Conference on Mining Software Repositories*. MSR ’15. IEEE Computer Society, 2015, pp. 111–122.

[57] Chi-Yu Liu et al. “Topic Modeling for Noisy Short Texts with Multiple Relations.” In: *SEKE*. 2018, pp. 610–609.

[58] Tie-Yan Liu. “Learning to Rank for Information Retrieval.” In: *Foundations and Trendső in Information Retrieval* 3.3 (2009), pp. 225–331.

[59] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. “Dynodroid: An input generation system for android apps.” In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’13. ACM, 2013, pp. 224–234.

[60] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. “Evodroid: Segmented evolutionary testing of android apps.” In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE ’14. ACM, 2014, pp. 599–609.

[61] Daniel Maier et al. “Applying LDA topic modeling in communication research: Toward a valid and reliable methodology.” In: *Communication Methods and Measures* 12.2-3 (2018), pp. 93–118.

[62] Daniel Maier et al. “How document sampling and vocabulary pruning affect the results of topic models.” In: *Computational Communication Research* 2.2 (2020), pp. 139–152.

[63] Masoud Makrehchi and Mohamed S Kamel. “Extracting domain-specific stopwords for text classifiers.” In: *Intelligent Data Analysis* 21.1 (2017), pp. 39–62.

[64] Henry B Mann and Donald R Whitney. “On a test of whether one of two random variables is stochastically larger than the other.” In: *The annals of mathematical statistics* (1947), pp. 50–60.

[65] Ke Mao, Mark Harman, and Yue Jia. “Crowd intelligence enhances automated mobile testing.” In: *Proceedings of the International Conference on Automated Software Engineering*. ASE ’17. IEEE Computer Society, 2017, pp. 16–26.

[66] Qun Mao et al. “User behavior pattern mining and reuse across similar Android apps.” In: *Journal of Systems and Software* (2021), p. 111085.

[67] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. “Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles.” In: *Proceedings of the International Conference on Software Engineering*. ICSE ’18. 2018, pp. 280–290.

[68] Leonardo Mariani et al. “An Evolutionary Approach to Adapt Tests Across Mobile Apps.” In: *International Conference on Automation of Software Test*. AST ’21. 2021, pp. 70–79.

[69] Leonardo Mariani et al. “Automatic testing of GUI-based applications.” In: *Software Testing, Verification and Reliability* 24.5 (2014), pp. 341–366.

[70] Leonardo Mariani et al. “Semantic Matching of GUI Events for Test Reuse: Are We There Yet?” In: *Proceedings of the 30th International Symposium on Software Testing and Analysis*. ISSTA 21. ACM, 2021.

[71] Thainá Mariani and Silvia Regina Vergilio. “A systematic review on search-based refactoring.” In: *Information and Software Technology* 83 (2017), pp. 14–34.

[72] Phil McMinn. “Search-based software test data generation: a survey.” In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156.

[73] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. “GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing.” In: *Proceedings of The Working Conference on Reverse Engineering*. WCRE ’03. IEEE Computer Society, 2003, pp. 260–269.

[74] Ali Mesbah, Engin Bozdag, and Arie van Deursen. “Crawling AJAX by Inferring User Interface State Changes.” In: *Proceedings of the International Conference on Web Engineering*. ICWE ’08. ACM, 2008, pp. 122–134.

[75] Tomas Mikolov et al. “Efficient estimation of word representations in vector space.” In: *arXiv* (2013).

[76] Rodrigo MLM Moreira, Ana CR Paiva, and Atif Memon. “A pattern-based approach for GUI modeling and testing.” In: *Proceedings of the International Symposium on Software Reliability Engineering*. ISSRE ’13. IEEE Computer Society, 2013, pp. 288–297.

[77] Rodrigo MLM Moreira et al. “Pattern-based GUI testing: Bridging the gap between design and quality assurance.” In: *Software Testing, Verification and Reliability* 27.3 (2017), e1629.

[78] Inês Coimbra Morgado and Ana C. R. Paiva. “The IMPAcT Tool for Android Testing.” In: *Proc. ACM Hum.-Comput. Interact.* 3.EICS (June 2019).

[79] David Newman et al. “Automatic Evaluation of Topic Coherence.” In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. HLT ’10. Los Angeles, California: Association for Computational Linguistics, 2010, pp. 100–108.

[80] Bao N Nguyen et al. “GUITAR: an innovative tool for automated testing of GUI-driven software.” In: *Automated Software Engineering* 21.1 (2014), pp. 65–105.

[81] Minxue Pan et al. “Reinforcement learning based curiosity-driven testing of android applications.” In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 153–164.

[82] Egon S Pearson. “The test of significance for the correlation coefficient.” In: *Journal of the American Statistical Association* 26.174 (1931), pp. 128–134.

[83] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation.” In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543.

[84] Xue Qin, Hao Zhong, and Xiaoyin Wang. “TestMig: Migrating GUI Test Cases from iOS to Android.” In: *Proceedings of the International Symposium on Software Testing and Analysis*. ISSTA ’19. ACM, 2019, pp. 284–295.

[85] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. “A survey of many-objective optimisation in search-based software engineering.” In: *Journal of Systems and Software* 149 (2019), pp. 382–395.

[86] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. “Efficient GUI test generation by learning from tests of other apps.” In: *Proceedings of the International Conference on Software Engineering*. ICSE Poster ’18. ACM, 2018, pp. 370–371.

[87] Michael Röder, Andreas Both, and Alexander Hinneburg. “Exploring the Space of Topic Coherence Measures.” In: *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. WSDM ’15. Association for Computing Machinery, 2015, pp. 399–408.

[88] EB Roessler et al. “Expanded statistical tables for estimating significance in pairedpreference, paired–difference, duo–trio and triangle tests.” In: *Journal of food Science* 43.3 (1978), pp. 940–943.

[89] Andrea Romdhana et al. “Deep Reinforcement Learning for Black-Box Testing of Android Apps.” In: *ACM Transactions on Software Engineering and Methodology* 31.4 (July 2022).

[90] Jonathan Schler et al. “Effects of age and gender on blogging.” In: *AAAI spring symposium: Computational approaches to analyzing weblogs*. Vol. 6. 2006, pp. 199–205.

[91] H Andrew Schwartz et al. “Personality, Gender, and Age in the Language of Social Media: The Open-Vocabulary Approach.” In: *PLOS ONE* 8.9 (Sept. 2013), pp. 1–16.

[92] Ting Su, Jue Wang, and Zhendong Su. “Benchmarking Automated GUI Testing for Android against Real-World Bugs.” In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Association for Computing Machinery, 2021, pp. 119–130.

[93] Afnan A Al-Subaihin et al. “Clustering Mobile Apps Based on Mined Textual Features.” In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’16. Association for Computing Machinery, 2016.

[94] Didi Surian et al. “App Miscategorization Detection: A Case Study on Google Play.” In: *IEEE Transactions on Knowledge and Data Engineering* 29.8 (2017), pp. 1591–1604.

[95] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[96] Shaheen Syed and Marco Spruit. "Full-text or abstract? Examining topic coherence scores using latent dirichlet allocation." In: *2017 IEEE International conference on data science and advanced analytics (DSAA)*. IEEE. 2017, pp. 165–174.

[97] Saghar Talebipour et al. "UI Test Migration Across Mobile Platforms." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 756–767.

[98] Yee Whye Teh et al. "Hierarchical Dirichlet Processes." In: *Journal of the American Statistical Association* 101.476 (2006), pp. 1566–1581.

[99] Joseph Turian, Lev Ratinov, and Yoshua Bengio. "Word representations: a simple and general method for semi-supervised learning." In: *Proceedings of the 48th annual meeting of the association for computational linguistics*. Association for Computational Linguistics. 2010, pp. 384–394.

[100] Tanja EJ Vos et al. "Testar: Tool support for test automation at the user interface level." In: *International Journal of Information System Modeling and Design* 6.3 (2015), pp. 46–83.

[101] Thi Anh Tuyet Vuong and Shingo Takada. "A Reinforcement Learning Based Approach to Automated Testing of Android Applications." In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST '18. Association for Computing Machinery, 2018, pp. 31–37.

[102] Thomas A Walsh, Gregory M Kapfhammer, and Phil McMinn. "Automatically identifying potential regressions in the layout of responsive web pages." In: *Software Testing, Verification and Reliability* 30.6 (2020), e1748.

[103] Xukun Wang et al. "Where does LDA sit for GitHub?" In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE. 2019, pp. 94–97.

[104] Wei Yang, Mukul R. Prasad, and Tao Xie. "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications." In: *Fundamental Approaches to Software Engineering*. Ed. by Vittorio Cortellessa and Dániel Varró. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–265.

[105] Faraz YazdaniBanafsheDaragh and Sam Malek. “Deep GUI: Black-box GUI Input Generation with Deep Learning.” In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021, pp. 905–916.

[106] Yuanchun Li et al. “DroidBot: a lightweight UI-Guided test input generator for android.” In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 23–26.

[107] Xia Zeng et al. “Automated test input generation for android: Are we really there yet in an industrial case?” In: *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE’16. 2016, pp. 987–992.

[108] Yixue Zhao et al. “Avgust: Automating Usage-Based Test Generation from Videos of App Executions.” In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Association for Computing Machinery, 2022, pp. 421–433.

[109] Yixue Zhao et al. “FrUITeR: a framework for evaluating UI test reuse.” In: *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 20. 2020, pp. 1190–1201.

[110] Aimin Zhou et al. “Multiobjective evolutionary algorithms: A survey of the state of the art.” In: *Swarm and Evolutionary Computation* 1.1 (2011), pp. 32–49.

[111] Hengshu Zhu et al. “Exploiting Enriched Contextual Information for Mobile App Classification.” In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*. CIKM ’12. Association for Computing Machinery, 2012, pp. 1617–1621.

[112] Hengshu Zhu et al. “Mobile app classification with enriched contextual information.” In: *IEEE Transactions on mobile computing* 13.7 (2013), pp. 1550–1563.

Sitography

[113] Adrian Chifor. *Swiftnotes*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.moonpi.swiftnotes.

[114] Andrzej Grzyb. *Shopping List*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=pl.com.andrzejgrzyb.shoppinglist.

[115] Anthony Restaino. *Lightning Browser*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=acr.browser.lightning.

[116] Apps By Vir. *Tip Calc*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.appsbyvir.tipcalculator.

[117] Benoit Letondor. *EasyBudget*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.benoitletondor.easybudgetapp.

[118] Craigpark Limited. *Email App for Any Mail*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=park.outlook.sign.in.client.

[119] douzifly. *Clear List*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/douzifly.list/.

[120] Gaukler Faun. *FOSS Browser*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/de.baumann.browser/.

[121] Google. *Monkey Runner*. Accessed: 2023-03-07.

[122] Tensor Flow Hub. *Token based text embedding trained on English Google News 200B corpus*. 2020. URL: https://tfhub.dev/google/nnlm-en-dim128/2.

[123] JPStudiosonline. *Free Tip Calculator*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.jpstudiosonline.tipcalculator.

[124] keith kildare. *Shopping List*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/com.woefe.shoppinglist/.

[125] keith kildare. *Simply Do*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/kdk.android.simplydo/.

[126] Kvannli. *Daily Budget*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.kvannli.simonkvannli.dailybudget.

[127] Luan Kevin Ferreira. *Expenses*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=luankevinferreira.expenses.

[128] Mail.Ru Group. *Mail.ru*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=ru.mail.mailapp.

[129] Mozilla. *Firefox Focus*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=org.mozilla.focus.

[130] My.com B.V. *myMail*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=ru.mail.mailapp.

[131] OpenIntents. *OI Shopping list*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=org.openintents.shopping.

[132] plafu. *Writeily Pro*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/me.writeily.

[133] rainbowshops. *Rainbow*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.rainbowshops.

[134] roxrook. *Pocket Note*. Last access: Jan 2021. 2021. URL: https://github.com/roxrook/pocket-note-android.

[135] Ruben Roy. *Minimal*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/com.rubenroy.minimaltodo/.

[136] SECUSO Research Group. *Shopping List (Privacy Friendly)*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=privacyfriendlyshoppinglist.secuso.org.privacyfriendlyshoppinglist.

[137] SECUSO Research Group. *Todo List*. Last access: Jan 2021. 2021. URL: https://f-droid.org/en/packages/douzifly.list/.

[138] Stoutner. *Privacy Browser*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.stoutner.privacybrowser.standard.

[139] TLe Apps. *Simple Tip Calculator*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.tleapps.simpletipcalculator.

[140] Vansuita. *Shopping List*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=br.com.activity.

[141] xorum. *Open Money Tracker*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.blogspot.e_kanivets.moneytracker.

[142] Yelp, Inc. *Yelp*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.yelp.android.

[143] ZaidiSoft. *Tip Calculator Plus*. Last access: Jan 2021. 2021. URL: https://play.google.com/store/apps/details?id=com.zaidisoft.tenin one.

Appendices

.1 Mann-Whitney U test

MMR

Mann-Whitney U tests of pairs of instances for different distribution (*p-value* $\leq$ 0.05),according to MMR metrics. The entries with *p-value* $\leq$ 0.05 are indicated in bold.

Component C1 Corpus of Documents

	googleplay	**manuals**
blogs	0.0954	0.3228
googleplay		**0.0479**

Component C2 Word Embedding

	es	**fast**	**glove**	**js**	**nnlm**	**use**	**w2v**	**wm**
bert	0.3188	0.1022	0.0770	0.2670	**0.0125**	**0.0006**	0.0899	**0.0007**
es		0.1519	0.1547	0.2796	**0.0285**	**0.0075**	0.2016	**0.0032**
fast			0.4012	0.3524	**0.0361**	**0.0015**	0.4086	**0.0001**
glove				0.4308	**0.0316**	**0.0027**	0.4291	**0.0001**
js					0.0506	**0.0028**	0.3887	**0.0031**
nnlm						0.1103	**0.0361**	0.1982
use							**0.0011**	0.2718
w2v								**0.0001**

Component C3 Event Descriptor Extractor

	CraftDroid_d	**intersection**	**union**
ATM_d	**0.0000**	**0.0209**	**0.0000**
CraftDroid_d		**0.0000**	0.3689
intersection			**0.0001**

Component C4 Semantic Matching Algorithm

	ATM_a	**CraftDroid_a**	**SemFinder**
adaptdroid_a	**0.0000**	**0.0000**	**0.0000**
ATM_a		0.3931	**0.0000**
CraftDroid_a			**0.0000**

Top1

Mann-Whitney U tests of pairs of instances for different distribution (*p-value* $\leq$ 0.05),according to Top1 metrics. The entries with *p-value* $\leq$ 0.05 are indicated in bold.

Component C1 CORPUS OF DOCUMENTS

	googleplay	**manuals**
blogs	0.1219	0.1293
googleplay		**0.0123**

Component C2 WORD EMBEDDING

	es	**fast**	**glove**	**js**	**nnlm**	**use**	**w2v**	**wm**
bert	0.2427	0.1982	0.1802	0.1778	**0.0046**	**0.0001**	0.2638	**0.0003**
es		0.0735	**0.0478**	0.3886	**0.0028**	**0.0000**	0.0989	**0.0001**
fast			0.4962	**0.0246**	**0.0033**	**0.0000**	0.4030	**0.0000**
glove				**0.0213**	**0.0031**	**0.0000**	0.3675	**0.0000**
js					**0.0004**	**0.0000**	**0.0337**	**0.0000**
nnlm						0.0787	**0.0019**	0.3749
use							**0.0000**	0.0787
w2v								**0.0000**

Component C3 EVENT DESCRIPTOR EXTRACTOR

	CRAFTDROID_d	**intersection**	**union**
ATM_d	**0.0000**	**0.0004**	**0.0000**
CRAFTDROID_d		**0.0000**	0.0964
intersection			**0.0005**

Component C4 SEMANTIC MATCHING ALGORITHM

	ATM_a	**CRAFTDROID_a**	**SEMFINDER**
adaptdroid_a	0.4464	**0.0000**	**0.0000**
ATM_a		**0.0048**	**0.0000**
CRAFTDROID_a			**0.0000**