Semantic Matching of GUI Events for Test Reuse: Are We There Yet?

Leonardo Mariani
University of Milano - Bicocca
Milan, Italy
leonardo.mariani@unimib.it

Ali Mohebbi
USI Università della Svizzera italiana
Lugano, Switzerland
ali.mohebbi@usi.ch

Mauro Pezzè
USI Università della Svizzera italiana
Lugano, Switzerland
SIT Schaffhausen Institute of Technology
Schaffhausen, Switzerland
mauro.pezze@usi.ch

Valerio Terragni
The University of Auckland
Auckland, New Zealand
v.terragni@auckland.ac.nz

ABSTRACT

GUI testing is an important but expensive activity. Recently, research on test reuse approaches for Android applications produced interesting results. Test reuse approaches automatically migrate human-designed GUI tests from a source app to a target app that shares similar functionalities. They achieve this by exploiting semantic similarity among textual information of GUI widgets. Semantic matching of GUI events plays a crucial role in these approaches. In this paper, we present the first empirical study on semantic matching of GUI events. Our study involves 253 configurations of the semantic matching, 337 unique queries, and 8,099 distinct GUI events. We report several key findings that indicate how to improve semantic matching of test reuse approaches, propose SemFinder a novel semantic matching algorithm that outperforms existing solutions, and identify several interesting research directions.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Human-centered computing → Mobile phones**; • **Computing methodologies → Natural language processing**.

KEYWORDS

GUI testing, test reuse, mobile testing, Android applications, word embedding, NLP

ACM Reference Format:
Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic Matching of GUI Events for Test Reuse: Are We There Yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464827

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464827

1 INTRODUCTION

Automatically generating test cases for GUI applications (*GUI tests*) is an active research topic [3, 4, 26, 28, 31, 32, 49, 51, 64]. A GUI test consists of (i) a sequence of events that interact with the GUI, and (ii) one or more assertion oracles that predicate on the GUI state. Current GUI test generators suffer from two main limitations [21]. First, they often generate semantically meaningless GUI tests that miss many relevant behaviors of the application under test. As such, they likely miss the GUI event sequences that properly exercise functionalities and reveal faults. Second, current GUI test generators rely mostly on implicit oracles [56, 65, 94] that reveal system crashes and exceptions, while missing many failures related to the semantics of the app under test.

A recent research thread explores the **reuse of GUI tests** across similar applications as an alternative way to automatically generate GUI tests [13, 15, 16, 45, 72, 75, 76]. GUI test reuse approaches generate new tests for a *target app* by migrating tests designed for a *source app*, an application that shares similar functionalities with the *target app*. Figure 1 shows an example of test migration between two Android apps. When test migration succeeds, GUI test reuse approaches (i) generate semantically meaningful GUI tests that properly exercise the functionalities of the target app, and (ii) adapt semantically relevant oracle assertions to the target app [15, 45], thus addressing the main limitations of GUI test generators.

GUI test reuse approaches exploit the fact that many GUI applications share semantically similar functionalities [34, 54, 75]. Hu et al. report that 196 (63.4%) of the top 309 non-game mobile apps in the Google Play Store can be clustered into 15 groups each sharing many common functionalities [34]. GUI test reuse is grounded on the observation that *different apps expose common functionalities via semantically similar GUI events* [93]. As such, automatic approaches try to migrate GUI tests across apps by mapping semantically similar GUI events.

In this paper, we target test reuse for Android applications. The current test reuse approaches for Android apps are ATM [15] and CraftDroid [45]. These two approaches successfully migrate non-trivial test cases, showcasing the potential of test reuse.

ATM and CraftDroid combine *semantic matching of GUI events* with *test generation*. Semantic matching of GUI events identifies

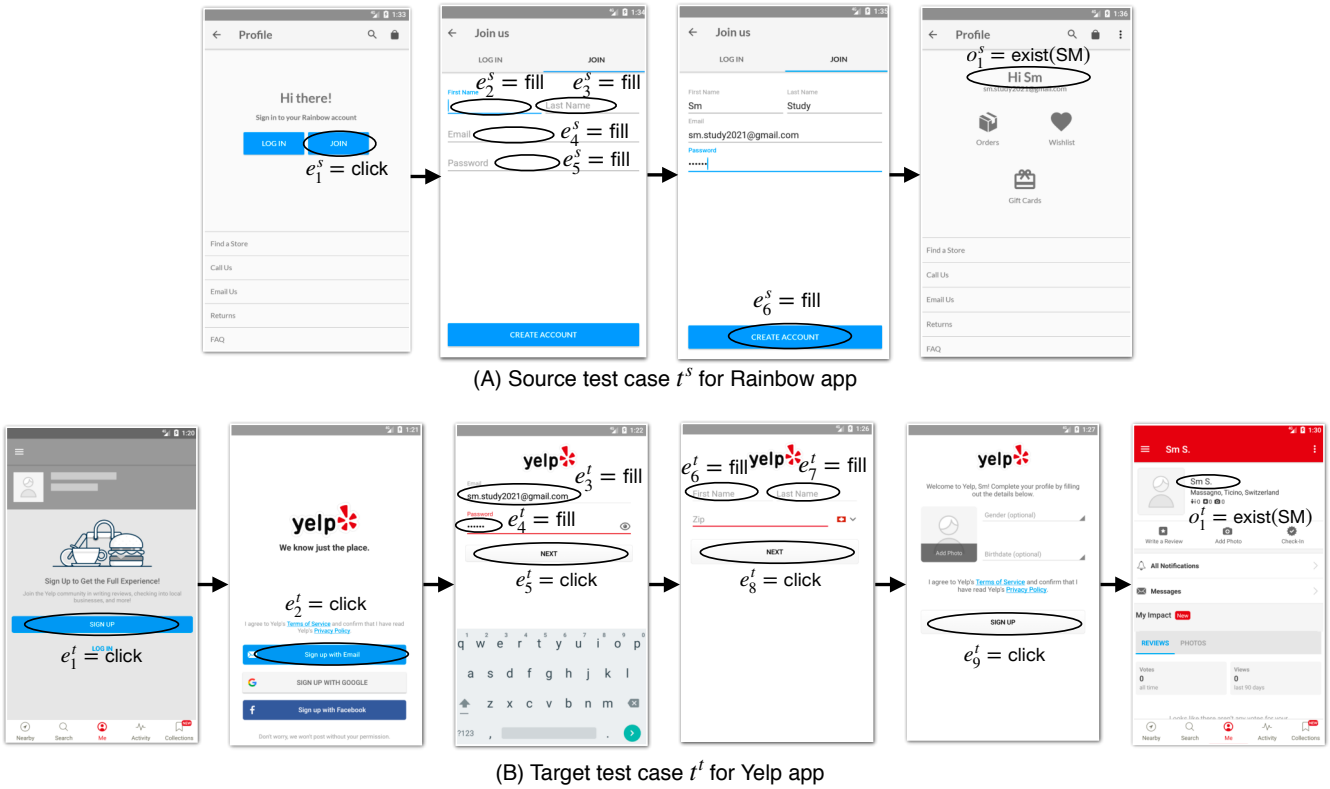

Figure 1: Test reuse example, the target test cases (B) is obtained by migrating the source test case (A)

semantically similar events across source and target apps, by applying word embedding techniques [61] to the textual descriptors of events in the GUI widgets. Test generation exploits the similarities identified with semantic matching to migrate GUI tests from the source to the target app.

The overall effectiveness of test reuse strongly depends on the effectiveness of semantic matching of GUI events. Indeed, the semantic matching is what drives the matching of the events between the source and the target test. Recently, Zhao et al. acknowledge the importance of reusing GUI tests, and propose the FRUITER framework [93] to comparatively evaluate test reuse techniques. FRUITER compares test reuse techniques as a whole, but does not support the evaluation of semantic matching in isolation.

In this paper, we present the first study on the semantic matching of GUI events for GUI test reuse/generation techniques. We identify four main components of the semantic matching, as illustrated in Figure 2: Corpus of Documents (Component 1), Word Embedding (Component 2), Event Descriptor Extractor (Component 3), and Semantic Matching Algorithm (Component 4). We then comparatively evaluate the impact of different choices for each component on the effectiveness of the semantic matching. Our study involves 253 configurations of these four components, 337 unique semantic matching queries, and 8,099 distinct GUI events, obtained from 30 ANDROID apps. Our configurations include the two configurations of ATM and CRAFTDROID and many other configurations that have not been investigated in the context of test reuse yet. We also propose a new semantic matching algorithm (SEMFINDER) and a new corpus of documents (GooglePlay) based on 900,805 app descriptions.

Our study discloses some relevant findings that both help identify a better matching algorithm and offer important insights for future research on test reuse. The most important findings are: (i) the Semantic Matching Algorithm (Component 4 in Figure 2) is the component that impacts the most on the overall effectiveness of semantic matching, and our proposed algorithm (SEMFINDER) outperforms the algorithms of ATM and CRAFTDROID; (ii) sentence level word embedding techniques (such as, Word Movers distance [39]) perform much better than world level ones (such as, WORD2VEC [60], used by both ATM and CRAFTDROID); (iii) considering certain widget attribute types as textual descriptors of GUI events can negatively affect the results; (iv) training word embedding models with corpora of documents specific to the mobile app domain lead to better results.

In summary, this paper

- develops the first framework to automatically evaluate the semantic matching of GUI events;
- identifies and extracts the core components of the semantic matching exploited in current test reuse approaches;
- evaluates 253 configurations of the semantic matching, and reveals important insights;
- proposes a new semantic matching algorithm (SEMFINDER) and a corpus of documents that outperform existing ones;
- makes our framework implementation and all data publicly available, for future research in this area [53].

2 TEST REUSE ACROSS SIMILAR GUI APPS

This section gives the preliminaries of this paper and introduces the GUI test reuse problem with an example.

Preliminaries: This paper targets Graphical User Interface **(GUI)** applications for the ANDROID platform. A GUI is a forest of hierarchical windows where only a window is active at any time [59]. Windows host **widgets**, which are atomic GUI elements characterized by attributes (such as, *text* and *resource-id*). At any time, the active window has a **state S** that encompasses the attribute values of the displayed widgets. Some widgets expose user-actionable events to let users interact with the app [25]. An **event** is an atomic interaction on a widget. For instance, users can click on widgets of type *Button*, or can fill widgets of type *EditText*. Following previous test reuse approaches, we abstract the implemented widget type and group events into two types: *clickable* and *fillable*. 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 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 [10]. For example, by checking for the absence or presence of widgets with specific attributes values.

Test reuse approaches for GUI applications [93] automatically migrate GUI tests (including oracles) 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 t^t that tests A^t as t^s tests A^s. They create t^t by searching A^t for events that are semantically similar to events in t^s.

Figure 1 shows an example of a migration from a test designed for the source app *Rainbow* (A) to the target app *Yelp* (B). The two tests verify the same feature, namely the creation of a new user. The example is taken from the experiments of CRAFTDROID [45].

The migration process exploits a semantic similarity relation $\sim$ to determine corresponding events of different apps. In the example we have that $e_1^s \sim e_1^t, e_2^s \sim e_6^t, e_3^s \sim e_7^t, e_4^s \sim e_3^t, e_5^s \sim e_4^t$, and $e_6^s \sim e_9^t$. Current test reuse approaches define such a relation as a one-to-one mapping between a source and a target event. The notion of semantic similarity of GUI events largely influences the ability of test reuse techniques to recognize corresponding events, thus impacting on the whole migration process.

3 SEMANTIC MATCHING OF GUI EVENTS

Test reuse approaches need to match semantically similar GUI events across apps. Such a semantic matching should capture 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 tests for the target app that maximize the number of *semantically similar events*, possibly in the order prescribed by the source test.

Current approaches characterize the semantics of events by relying on the textual attributes found in the GUI. In particular, they associate each event with its *descriptor* that encompasses the textual attributes of the widget associated with the event. For instance, the attributes *text* of events e_1^s and e_1^t in Figure 1 are *"join"* and *"sign up"*,

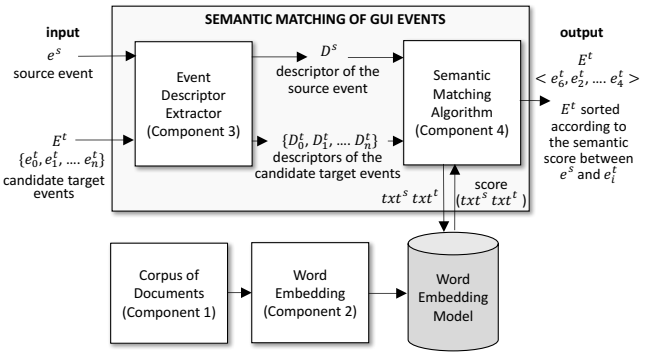

Figure 2: Logical workflow of the semantic matching

respectively. They then identify similar semantics by querying a word embedding model that recognizes words or sentences that express similar concepts. For instance, a word embedding model would recognize that *"join"* and *"sign up"* are semantically similar.

Figure 2 shows the logical workflow among the core components of the semantic matching of GUI events, which is shared by all test reuse approaches that rely on word embedding. Given a source event e^s and a set of candidate target events $E^t = \{e_0^t, e_1^t, \cdots e_n^t\}$, for each event in $e_i^t \in E^t$ the semantic matching computes a similarity score that expresses the degree of semantic similarity between e_i^t and e^s. The semantic matching computes the score by aggregating the scores returned by the word embedding model for each pair of attributes in their descriptors ($score(txt^s, txt4^t)$ in Figure 2). Different semantic matching algorithms use different aggregation functions [15, 45]. Then, test reuse approaches can consider the event(s) with the highest scores [45] and/or ignore all events that are below a predefined threshold [15]. The semantic matching of GUI events can thus be divided into four main components:

C1) Corpus of Documents that the approaches use to build a word embedding model.

C2) Word Embedding that relies on the corpus of documents to create a word embedding model that defines the semantic space of words/sentences in the corpus.

C3) Event Descriptor Extractor that extracts information from a source event e^s and a set candidate target events $E^t = \{e_0^t, e_1^t, \cdots e_n^t\}$. This component extracts the (textual) descriptors $D = \{\langle a_i, v_i \rangle\}$ of each event, both dynamically from the GUI states and statically from the GUI layout files. In a descriptor D, a_i is an attribute type (such as, *text*) and v_i is its value (such as, "PRESS OK"). Note that, the textual value of an attribute might be a full sentence, as in this example. For convenience of notation, we use $D[a_i]$ to refer to the value v_i of the attribute a_i of descriptor D.

C4) Semantic Matching Algorithm that returns a list of E^t elements sorted according to the similarity score computed from the descriptor of the source event (D^s) and the descriptors of the candidate target events ($\{D_0^t, D_1^t, \cdots D_n^t\}$). Internally, the semantic matching algorithm computes the similarity score between events by aggregating the similarity scores of corresponding attributes in the descriptors of the source and target events: $sim(e^s, e_i^t) = sim(D^s, D_i^t) = agg_j\{sim(D^s[a_j], D_i^t[a_j])\}$.

We now describe the component type implementations that we consider in our study. We refer to a specific implementation of a component as an *instance*. We include *all the component instances of ATM and CraftDroid* and *many other instances*, which were not investigated in the context of semantic matching of GUI events, yet.

3.1 Corpus of Documents

Our study considers three corpora of English documents:

Blog Authorship Corpus (Blogs) [79] 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 [1, 80].

User Manuals of Android apps (Manuals) [15] that consists of the user manuals of 500 Android applications. This corpus was built by the authors of ATM [15], 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.

The corpus of documents plays an important role in the semantic matching. Indeed, the quality of a word embedding model depends on the corpus of documents used to train the model.

There are two important characteristics that the corpus of documents should have to obtain an effective word embedding model.

First, the corpus should 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 [18]). Moreover, the words contained in the corpus should be words that are often found in the GUI of Android applications.

Second, the corpus should reflect the same word usage that mobile apps commonly adopt. In fact, a word can have a different meaning depending on the context of usage. Word embedding models trained with domain-specific corpora often outperform those trained with general corpora [42]. To study and quantify the importance of the context of usage, we considered both general (Blogs) and mobile apps specific corpora (Manuals and Google-play).

3.2 Word Embedding

Word embedding [60] is a class of unsupervised language modeling and feature learning techniques that map words or sentences from a corpus of documents to vectors of real numbers [85].

A word embedding assigns each unique word in the corpus to a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are close to one another. The resulting vector space is a word embedding model, which test reuse approaches use to identify semantically similar, although syntactically different words (the so-called *synonym problem*). In the context of test reuse, the synonym problem is a key issue, because we cannot expect that independent developers use the same words to express the same concepts.

Our study considers the following word embedding techniques:

Word2vec [60]: one of the most popular word embedding techniques developed in 2013 in Google. It implements a shallow (two-layer) neural network that is trained to reconstruct linguistic contexts of words. Both ATM and CraftDroid rely on models built with Word2vec [15, 45].

Global Vectors (Glove) [70]: 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) [39]: 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 sentences [39]. As such, WM considers distance between sentences (one or more words) [85] and not only among pairs of words like the distances based on Word2vec or Glove [85]. In the context of test reuse this could be useful, because event descriptors often contain multiple words [15, 45]. WM returns an integer greater than zero, that we normalize from 0 to 1, with a standard normalization $1/(1+\mathrm{WM}(\mathrm{txt}^s, \mathit{txt}^t))$.

Fast Text (Fast) [18]: an extension of Word2vec developed in Facebook. While Word2vec treats words as the smallest unit to train on, Fast learns vectors for the n-grams that are found within each word. Fast 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". Fast is designed to alleviate the Out-of-Vocabulary issue [18]. In fact, even if the word "aquarius" is not present in the corpus, Fast would embed "aquarius" near to "aquarium" because they share seven n-grams.

Bidirectional Encoder Representations from Transformers (BERT) [24]: a context-sensitive word embedding technique that infers the meaning of a word from its surroundings, by learning how to predict 15% of masked words in a sentence.

Neural Network Language Model (NNLM) [8]: a family of neural network techniques that learn word embedding models jointly with the language model. In our study we consider the NNLM technique proposed by Google [35].

Universal Sentence Encoder (USE) [19]: a state-of-the-art context-sensitive word embedding technique proposed by Google.

3.3 Event Descriptor Extractor

This component collects the descriptors of the source event e^s and of the candidate target events $\langle e_0^t, e_1^t, \cdots e_n^t, \rangle$. An event descriptor D is a set of *textual attributes* $\{a_1, a_2 \cdots a_m\}$ extracted from the GUI states. Each attribute is defined as a $\langle type, value \rangle$ pair. Our study considers all the attribute types used in current test reuse approaches for Android (ATM and CraftDroid) [15, 45] as part of the descriptors. The attributes can be classified as primitive and derived. *Primitive* attributes are directly associated with the widget of the event under consideration. *Derived* attributes are obtained from primitive attributes of other widgets in the GUI state that contains the event under consideration.

The primitive attributes of a widget w are:

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

Table 1: Groups of event descriptors

attribute category	attribute type	ATM 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		✓		✓

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.

Sometimes the textual information that describes a widget is not found in the widget itself but in near widgets [11]. For instance, the widget associated with e_2^s in Figure 1 (A) does not have any visible textual attribute, but there is a neighbor widget with text attribute "First Name" that describes the semantic of the widget of e_2^s. ATM defines derived attributes from the spatial positions of the widgets [15]. CraftDroid defines derived attributes from the hierarchical structure of the Android GUI states [45], in which widgets have a parent-child-sibling relationship. The element that directly precedes another element in the hierarchy is the *parent* of the element below it, and the element below the parent is the *child*. Two elements at the same hierarchical level are *siblings*.

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 immediately before w in the hierarchical structure.

neighbor-text, the *text* attribute of the closest widget from w within a certain 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 did not consider each attribute individually, as the semantic matching algorithms require a set of attributes to be effective. Table 1 shows the four groups of attributes that we considered in our study, where "A" and "C" indicate the attributes used by ATM and CraftDroid, respectively. The "intersection" group (A ∩ C) are attributes used by both ATM and CraftDroid. We consider this group to evaluate the impact of the attributes used by only one approach. For example, we can evaluate the impact of the descriptors *neighbor-text* and *file-name*, by comparing the results of the groups "A" and "A ∩ C". The "union" group (A ∪ C) are attributes used by ATM, CraftDroid or both.

Algorithm 1: Semantic Similarity Calculator

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

```
1  function getSimScore
2      ⟨txt^s, txt^t⟩ ← preprocessing(txt^s, txt^t)
3      switch M do
4          case model at "word" level (Word2vec, Glove, FastText) do
5              score[][] ← ∅
6              for each word wd_1 ∈ txt^s do
7                  for each word wd_2 ∈ txt^t do
8                      score[wd_1][wd_2] ← cosineSim(M(wd_1), M(wd_2))
9              mappedScores ← getMatchedWords(score[][])
10             return aggr{mappedScores}
11         case model at "sentence" level (WMD, BERT, NNLM, USE) do
12             return sim(M( txt^s ), M( txt^t ))
```

3.4 Semantic Matching Algorithm

Test reuse approaches decide how to generate the target test case by analyzing the lists of target events sorted according to the similarity score computed for each event in the source test case. More specifically, the algorithm takes in input the descriptor D^s of the source event e^s and the set of descriptors $\{D_0^t, D_1^t, \cdots D_n^t\}$ of the candidate target events E^t, and returns a sorted list of E^t based on the similarity scores computed between D^s and each of the target descriptors D_i^t, where D_i^t denotes the descriptor of event e_i^t.

We now describe the three semantic matching algorithms of our study: the one used in ATM, the one used in CraftDroid, and SemFinder, a new algorithm that we propose in this paper.

All the three algorithms rely on a word embedding model ($\mathcal{M}$) to compute the semantic similarity scores among the attribute values of the source and target descriptors. Algorithm 1 illustrates the function that the three algorithms share (Function getSimScore). The function computes the similarity scores between two sentences txt^s and txt^t obtained from the values of the textual attributes of the source and target descriptors, respectively. More specifically, the function takes in input two sentences txt^s, txt^t, a model $\mathcal{M}$, and an aggregator function (average or sum), and returns a real number that expresses the similarity score between txt^s and txt^t.

A pre-processing phase removes stop words, performs lemmatization, and splits words in the case of camel case notation (line 2). If the model $\mathcal{M}$ is at word level, the algorithm computes the cosine similarity of `vector(`wd_1`)` and `vector(`wd_2`)` for all possible pairs of words of the two sentences $\langle wd_1 \in txt^s, wd_2 \in txt^t \rangle$ (lines 6–8). Then it identifies the best match among the pairs as (i) the pair with the highest cosine similarity, where (ii) every word is matched only once (line 9). It finally returns the similarity score using the aggregation function passed as an input (line 10). ATM uses *sum* as an aggregation function, while CraftDroid and SemFinder use *average*. If the model is at sentence level the algorithm does not consider each word individually, but queries the model $\mathcal{M}$ with the sentences as a whole. Notably, both ATM and CraftDroid use models at word level, we add lines 10 and 12 to make the algorithm compatible with the sentence level word embedding models that we considered in our study.

We now describe the three algorithms and their key differences.

Algorithm 2: Semantic Matching Algorithms

Input: source descriptor D^s, set of target descriptors $\{D^t_0, D^t_1, \cdots D^t_n\}$
Output: sorting of E^t based on the semantic similarity with e^s

```
13 function ATM
14   descScores[] ← ∅
15   label^s_1 ← getFirstDef(D^s[neighbor-text], D^s[resource-id] +
       D^s[file-name])
16   label^s_2 ← 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^t_i) then
19       label^t_1 ← getFirstDef(D^t_i[neighbor-text], D^t_i[resourse-id] +
           D^s[file name])
20       label^t_2 ← getFirstDef(D^t_i[text], D^t_i[content-desc] or D^t_i[hint])
21       scores ← ∅
22       for each label^s ∈ {label^s_1, label^s_2} do
23         for each label^t ∈ {label^t_1, label^t_2} do
24           add getSimScore(label^s, label^t, M, "sum") to scores
25       descScores[D^t_i] ← max{ scores }
26   return E^t sorted by descScore
27 function CraftDroid
28   descScores[] ← ∅
29   for each i from 1 to n do
30     if type(e^s) = type(e^t_i) 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^t_i[a_i], M, "avg") to scores
34       descScores[D^t_i] ← avg{ scores }
35   return E^t sorted by descScore
36 function SemFinder
37   descScores[] ← ∅
38   for each i from 1 to n do
39     if type(e^s) = type(e^t_i) then
40       ⟨txt^s, txt^t⟩ ← ⟨∅, ∅⟩
41       for each a_i ∈{ text, resource-id, content-desc, hint, file-name,
           neighbour-text} do
42         txt^s ← txt^s ∪ D^s[a_i]
43         txt^t ← txt^t ∪ D^t[a_i]
44       descScores[D^t_i] ← getSimScore(txt^s, txt^t, M, "avg")
45   return E^t sorted by descScore
```

Semantic Matching of ATM [15] Lines 13 to 26 of Algorithm 2 encode the semantic matching algorithm of ATM. The algorithm starts by collecting two textual representations of the source event: $label^s_1$ (line 15) and $label^s_2$ (line 16). $label^s_1$ is the first *defined* attribute among $\langle$*neighbor-text*, *resource-id* $\cup$ *file-name*$\rangle$ in D^s. If all of such attributes are undefined, $label^s_1$ is the empty string. Notably, ATM extracts the *neighbor-text* attribute only for filling events, for clicking events the attribute is always undefined. $label^s_2$ is the first *defined* attribute among $\langle$*text*, *content-desc*, *hint*$\rangle$ in D^s (line 16).

For each event $e^t_i \in E^t$ that has the same type of e_s (either both filling or both clicking events), the algorithm collects $label^t_1$ and $label^t_2$ in the same way it collects $label^s_1$ and $label^s_2$, respectively. Then, the algorithm invokes Function getSimScore (Algorithm 1) for each combination of $\langle\ label^s \in \{label^s_1, label^s_2\}, label^t \in \{label^t_1, label^t_2\}\rangle$, using "sum" as aggregation function. The algorithm assigns the highest returned value to the score of the current target event ($score[D^t_i]$ line 25). After the algorithm analyses each target event, it sorts E^t based on the final scores (line 26).

Semantic Matching of CraftDroid [45] Lines 27 to 35 of Algorithm 2 encode the semantic matching algorithm of CraftDroid. For each target event e^t_i 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, *resource-id* of the source descriptor is compared to *resource-id* of the target descriptor. CraftDroid assigns the average of List *scores* to the final score of the current target descriptor (line 34).

SemFinder Lines 36 to 45 of Algorithm 2 encode the semantic matching algorithm SemFinder that we propose in this paper. For each event $e^t_i \in E^t$ that has the same type of e_s (either both filling or both clicking events), SemFinder builds two sentences txt^s and txt^t. It builds txt^s by concatenating all the values of the attributes of D^s (separated with a space), and txt^t with the values in D^t_i. It then removes words that are repeated in the same sentence. It finally aggregates the similarity score between txt^s and txt^t using *average*, and assigns the result to the final score of the current target descriptor (line 44).

Key differences While sharing the same general idea, the three algorithms differ in three important aspects[1]:

I. *The attributes of source and target descriptors that they compare.* Both ATM and CraftDroid compute the semantic similarity only between certain types of source and target attributes. CraftDroid computes the semantic similarity only between attributes of the same type. However, there is no guarantee that across different apps the relevant semantic information is always contained in the same attribute type. Indeed, a typical test reuse scenario involves source and target apps implemented by different developers, who might follow different software development styles and standards. ATM allows some flexibility on the attribute types, for instance, it considers the pair $D^s[text]$ and $D^t[resource\text{-}id]$, but it is still restricted to some combinations. For example, given a source event e^s with $D^s[text]$ = "address", and target event e^t_i with $D^t_i[neighbor\text{-}text]$ = "find" and $D^t_i[resource\text{-}id]$ = "location", both ATM and CraftDroid would not consider the pair of attributes $\langle D^s[text], D^t_i[resource\text{-}id]\rangle$, thus missing the semantic similarity of "address" and "location". ATM misses this pair of attributes because the first not empty attribute at Line 20 of Algorithm 2 is $D^s[text]$, and thus will only consider the pair $\langle D^s[text], D^t_i[neighbor\text{-}text]\rangle$.

SemFinder computes the semantic similarity scores among all attributes, regardless of their type. Our intuition is that the relevant semantic information can be in any of the considered attribute types. As such, SemFinder merges all source attribute values into a single sentence, all target attribute values into another sentence, and computes the semantic similarity of the two sentences.

II. *The way they aggregate the similarity scores of multiple pairs of attribute types.* ATM uses the *max*imum (line 25 of Algorithm 2), while CraftDroid uses the *average* (line 34 of Algorithm 2) to aggregate the similarity score of multiple pairs of attribute types. Both aggregation functions have their pros and cons [15, 45]. SemFinder does not consider attributes separately, but it groups them into sentences, thus does not need to combine the similarity scores of multiple pairs of attribute types. By comparing sentences, SemFinder to

[1] we exclude the difference of attribute types, which is considered individually by C3.

Table 2: Subjects of our experiment

subject from	category	app id	# of DL	subject from	category	app id	# of DL
ATM	Expense Tracker	EasyBudget [17]	100k	CraftDroid	To-Do List	Minimal [78]	-
		Expenses [48]	1K			Clear List [27]	-
		Daily Budget [40]	50K			Todo List [82]	-
		Open Money [89]	1K			Simply Do [38]	-
	Note Taking	Swiftnotes [2]	-			Shop. List [37]	-
		Writely Pro [71]	-		Shopping	Rainbow [73]	0.5M
		Pocket Note [77]	-			Yelp [90]	50M
	Shopping List	Shop.List1 [5]	-		Mail Client	Mail.ru[50]	50M
		Shop.List2 [86]	100K			myMail [67]	10M
		Shop.List3 [81]	5K			AnyMail [22]	10M
		OI Shop. List [68]	1M		Tip Calculator	TipCalculator	
CraftDroid	Browser	Lightning [6]	10K			TipCalc [7]	500
		Privacy [83]	1K			Simple Tip [84]	1K
		FOSS [30]	-			TipCalc.Plus [91]	500
		FirefoxFocus [66]	5M			FreeTipCalc. [36]	1K

leverage the full capacity of sentence level embedding techniques. Sentence level techniques can handle semantic relation of words when they appear together.

III. *The way they aggregate the similarity scores in case of word-level word embedding models.* ATM aggregates the similarity scores of different words in the same sentence (Lines 4 to 10 of Algorithm 1) with the *sum* (line 24 of Algorithm 2), while CraftDroid with the *average* (line 33 of Algorithm 2). Even if SemFinder combines all attribute types in single sentences, it also needs to aggregate scores of words for word-level word embedding models (Lines 4 to 10 of Algorithm 1). SemFinder aggregates the similarity scores with the *average* (Line 44 of Algorithm 2), like CraftDroid, since the *average* often works better than *sum* (used in ATM). This is because the *sum* privileges (assign high score to) sentences with many words, as there is always a positive score between two words, if both words are represented in the model. Thus, two attributes with many unrelated words usually get a higher score than two attributes with fewer highly related (semantically similar) words.

4 EXPERIMENT

In this paper, we study the effectiveness and limitations of the semantic matching of GUI events for test reuse approaches. We conducted a set of experiments involving 253 different configurations of the semantic matching (Figure 3), aiming to answer three research questions:

- **RQ1 Baseline Comparison:** *Do semantic approaches based on word embedding outperform syntactic and random approaches?*
- **RQ2 Component Effectiveness:** *What are the most effective instances of each component?*
- **RQ3 Impact Analysis:** *Which component type(s) have the greatest impact on the semantic matching of GUI events?*

RQ1 checks whether the use of semantic approaches is justified, by comparing the effectiveness of semantic approaches to both syntactic (edit-distance and Jaccard similarity) and random approaches. RQ2 identifies which instances achieve the best performance. RQ3 studies which component type has the largest impact on the effectiveness of semantic matching, thus suggesting where the research community should focus its effort.

4.1 Implementation

We implemented a fully automated tool in Python that runs the different configurations of the four component types on a set of source and target events. The tool represents a framework to evaluate the semantic matching of GUI events, which can be easily extended to add new component instances.

The source code of ATM and CraftDroid is publicly available, ATM is written in Java [12], while CraftDroid in Python [44]. We re-implemented the semantic matching algorithm of ATM in Python referring to the original Java implementation [12]. For the algorithm of CraftDroid, we reused the original Python code as much as possible [44]. It is important to mention that the semantic matching algorithms of ATM and CraftDroid are internal algorithms of test reuse tools and can be hardly executed in isolation. As such, one of the contributions of this work is a framework for comparing different component instances of the semantic matching, similar to what Fruiter achieved in the context of test reuse [93].

We implemented the Event Descriptor Extractor instances with a tool that executes the source and target tests and extracts from the GUI states the values of the nine widget attributes considered in our study (Table 1). We used the framework Appium (1.1.13) to read the GUI states at runtime. We implemented our own extractor, rather than rely on the implementations of ATM or CraftDroid, to have a common tool to collect all the descriptors. Our event extractor considers all types of click and fill events used by state-of-the-art test reuse approaches (ATM and CraftDroid). In particular, click events include simple click, swipe and long click, and are applicable to a wide range of Android widget types such as Button, ListView, Dialog, and ImageButton. Fill events insert a text into an EditText widget.

4.2 Subjects

We considered all the publicly available test migration scenarios (pairs of source and target test cases $\langle t^s, t^t \rangle$) used in the experiments of ATM and CraftDroid. Such scenarios involve test cases from 41 Android apps. We considered all the test scenarios[2] that belong to the 30 Android apps that we could run. We could not

[2] ATM considers 10 test scenarios for each pair of source and target apps. However, such scenarios cannot be considered in isolation, because the scenarios are dependent on one another. For this reason, we consider only the first scenario.

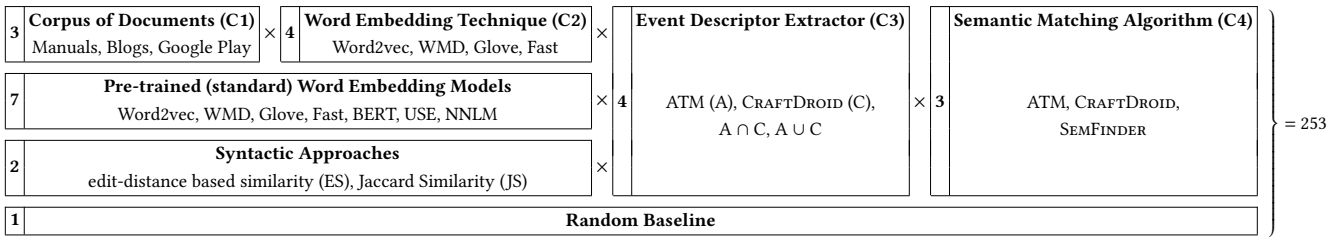

Figure 3: The 253 configurations of components' instances considered in our study

run five Android apps of ATM because we encountered various errors when compiling their source code. Despite a lot of effort, we could not provide the correct environment for these subjects. We could not run six Android apps of CraftDroid because they require communication with a server, but the API or Security protocol changed. Table 2 shows the 30 Android apps that we considered in our study.

We consider all 139 pairs of source and target test cases $\langle t^s, t^t \rangle$ among the test migration scenarios provided by ATM and CraftDroid that involve these 30 apps. The 139 scenarios include a ground-truth annotation from the ATM and CraftDroid authors that specifies which events in the source test case match which events in the target test case. Given a source event $e^s \in t^s$, we use $e^t_{gt} \in t^t$ to denote the event that semantically matches e^s as annotated by the ground truth. Notably, not all events in source test cases have an equivalent counterpart in the target app. Some events (called ancillary events [93]) are specific to the source app only, but are needed in the source test to reach certain app states or windows. Since our goal is to evaluate the semantic matching only, we removed them.

Because some source tests share the same app, the same event could be repeated across multiple source tests. We remove redundant events by considering two events e_a and e_b to be equivalent iff all the nine event descriptors considered in our study are identical across e_a and e_b. After removing all redundant events, we obtained 337 unique source events, and thus 337 unique queries.

There are multiple ways to define the set of candidate target events $E^t = \{e^t_0, e^t_1, \cdots e^t_n\}$ for each $e^s \in t^s$. We define E^t as the set of events that are actionable in all the GUI states traversed by the target test t^t. 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^t. Our definition of E^t leads to semantic matching queries that are coherent with test reuse, which match events across applications considering target events that span multiple windows [15, 45]. Simply defining E^t as the set of events actionable in the window of e^t_{gt} would create an artificial and unrealistic scenario. This is because a test reuse technique cannot know in advance which window of the target application should contain events semantically similar to a given source event.

According to our definition, if multiple events in t^t belong to the same window, we can have many redundant events within the same E^t. We remove all of them by applying the equivalent relation described above. The cardinality of resulting E^t ranges from 5 to 80, with an average of 24.03 and median of 19 events.

4.3 Experimental Setup

Figure 3 shows the 253 configurations of the semantic matching that we used in our experiment.

We considered the 12 pairwise combinations of the three corpora of documents and four word embedding techniques: Word2vec, WMD, Glove and Fast, building 12 word embedding models. Before running the word embedding techniques we used the same pre-processing steps used at Line 2 of Algorithm 1.

For all seven word embedding techniques we considered the pre-trained (standard) models provided by the authors of such techniques. Notably, these pre-trained models are obtained using different corpora of documents (such as, different versions of Google News and Twitter datasets), which are not publicly available. As such, we were not able to consider such corpora as individual components, like we did for Manuals, Blogs, and Google-play.

We decided not to build models with BERT, USE and NNLM using the three corpora (Manuals, Blogs, and Google-play), and thus relying only on the pre-computed models. This is because these word embedding techniques require a non-trivial parameter tuning that goes beyond the scope of this paper.

RQ1 considers two canonical syntactic approaches that compute the syntactic similarity of words/sentences: edit-distance based similarity (ES), and the Jaccard Similarity index (JS). Because both ES and JS do not use the corpus of documents, we ignore the combinations of ES and JS with the three corpora.

ES computes the (normalized) similarity of two words relying on the "Levenshtein distance" [41] that quantifies the dissimilarity of two words as the minimum number of operations (deletion, insertion and substitution) required to transform a word into the other. Given two words wd_1 and wd_2,

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

where LD(wd_1 wd_2) is the "Levenshtein distance" of wd_1 and wd_2. ES returns 1 if the words are identical. ES operates at word level, and thus replaces the query of the word embedding model at line 8 of Algorithm 1.

JS computes the similarity of two sets by dividing the number of elements that are shared between both sets by the total number of (unique) elements (both shared and not shared). In our context, sets are sentences and the elements of the sets are words. Given two sentences txt_1 and txt_2,

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

JS returns 1 when txt_1 and txt_2 have all identical words, regardless of their position in the sentences. JS operates at sentence level, and

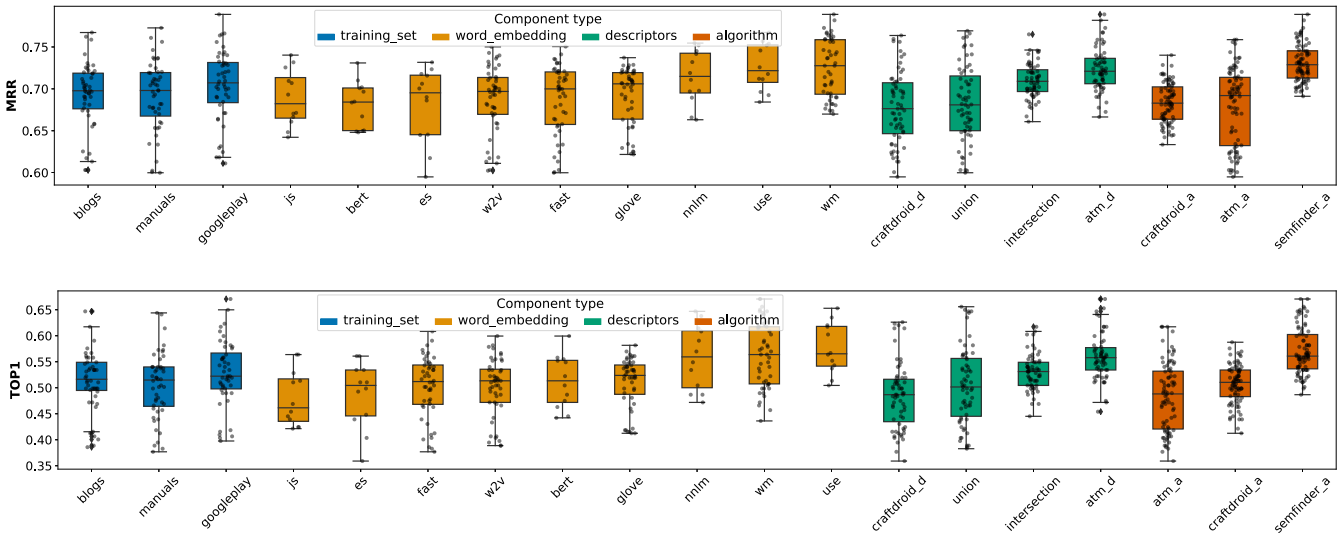

Figure 4: Distribution of MRR (top) and TOP1 (bottom) for each component

thus replaces the interrogation of the word embedding model at line 12 of Algorithm 1.

We experiment with the Event Descriptor Extractor instances (Component 3) by combining the four groups of descriptors summarized in Table 1 with all three Semantic Matching Algorithms instances (Component 4). To distinguish the descriptors and algorithms when they share the same name, we added the suffix *"_d"* and *"_a"*. For instance, ATM_d denotes the descriptor group and ATM_a the algorithm of ATM.

An important design choice is how to modify the semantic matching algorithms to accept a group of descriptors that differs from the groups used by the original algorithms. We modify the semantic matching algorithms as follows: If the group of descriptors does not contain an attribute a that is considered in the original algorithm, we remove a from the algorithm. For instance, when combining the "intersection" group to CraftDroid_a, we remove the *activity-name*, *parent-text* and *sibling-text* from the set of attribute types at Line 32 of Algorithm 2. If the group of descriptors contains an attribute a that is not considered in the original algorithm, we add a to the algorithm by appending it at the end of the *text* attribute. For instance, when combining the CraftDroid_d group with ATM_a, we append the attribute types *activity-name*, *parent-text* and *sibling-text* to the attribute *text* at Lines 16 and 20 of Algorithm 2. The rationale of using this approach is twofold: (i) ATM does not ignore the new attributes, since ATM gives highest priority to the *text*, and (ii) the choice corresponds to the way the original algorithm of CraftDroid handles the *hint* attribute (line 32 of Algorithm 2).

Our last configuration is a random baseline that assigns a random score between 0 and 1 to each pair of events. To cope with the stochastic nature of the random baseline, we repeated this process 100 times and we report the median result.

4.4 Evaluation Metrics

In our study, a query q is a pair of a source event and a set of candidate target events $\langle e^s, E^t \rangle$ that returns the list of events in E^t sorted by their final score. In our context, we have only one correct answer (e^t_{gt}), and thus the *rank* of a query q_i, denoted by $rank_i$, is the position of e^t_{gt} in the list returned by the query q_i. Following standard practice, if multiple events have identical final scores, their rank is the average of their positions. For instance, if the top three events have identical final scores, their rank is equal to two ($1+2+3/3 = 2$).

We evaluate the semantic matching effectiveness of each of the 253 combinations using two metrics based on the ranks: (i) the Mean Reciprocal Rank (MRR) [47], and (ii) the ratio of queries in which the rank of the correct answer is one (*TOP1*).

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 [47].

The metric TOP1 is the ratio of queries in which the ground truth (e^t_{gt}) is at the first position of the returned list of events. TOP1 is less informative than MRR, because it does not make any difference whether a query returns the ground truth event at the second or last position in the list. However, TOP1 remains an important metric to evaluate the semantic matching of GUI events, as often test reuse approaches choose the first event in the list.

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

Table 3: Distributions of the 253 combinations sorted by MRR and TOP1 based on the percentiles 1%, 5%, 10%. [1:x] denotes the configurations from position 1 to x of the list of 253 configurations ordered by MRR or TOP1.

type	instance	MRR [1:3]	MRR [1:13]	MRR [1:26]	TOP1 [1:3]	TOP1 [1:13]	TOP1 [1:26]
C1	blogs	0%	23%	12%	0%	**15%**	12%
	manuals	**33%**	15%	15%	0%	**15%**	12%
	googleplay	**33%**	**31%**	**23%**	**33%**	**15%**	**19%**
C2	w2v	0%	0%	4%	0%	0%	0%
	glove	0%	0%	0%	0%	0%	0%
	wm	**100%**	**92%**	**62%**	**100%**	**69%**	**62%**
	fast	0%	0%	4%	0%	0%	4%
	bert	0%	0%	0%	0%	0%	0%
	nnlm	0%	0%	12%	0%	15%	15%
	use	0%	8%	19%	0%	15%	19%
	js	0%	0%	0%	0%	0%	0%
	es	0%	0%	0%	0%	0%	0%
C3	atm_d	**100%**	**46%**	**42%**	**67%**	**46%**	**42%**
	craftdroid_d	0%	15%	19%	0%	8%	19%
	intersection	0%	8%	12%	0%	0%	15%
	union	0%	31%	27%	33%	**46%**	23%
C4	atm_a	0%	8%	15%	0%	0%	15%
	craftdroid_a	0%	0%	0%	0%	0%	0%
	semfinder_a	**100%**	**92%**	**85%**	**100%**	**100%**	**85%**

4.5 Results

We run our 337 queries for each of the 253 configurations, we thus execute 85,261 semantic matching queries in total. MRR ranges from 0.201 to 0.789 across all configurations with an average of 0.696. The quartiles of MRR are: Q1: 0.674, Q2: 0.702, Q3: 0.724. In the set of the 253 configurations sorted according to the MRR values, the original configuration of ATM [manuals (Component 1), w2v (Component 2), ATM_d (Component 3), ATM_a (Component 4)] is in position 184 (MRR = 0.677), while the original configuration of CraftDroid [standard (Component 1), w2v (Component 2), CraftDroid_d (Component 3), CraftDroid_a (Component 4)] is in position 196 (MRR = 0.670).

TOP1 ranges from 0.065 to 0.671 across all configurations (Q1: 0.484, Q2: 0.522, Q3: 0.558), with an average of 0.518. In the set of the 253 configurations sorted according to the TOP1 values, the original configuration of ATM is in position 200 (TOP1 = 0.472), while the original configuration of CraftDroid is in position 191 (TOP1 = 0.484).

For both metrics the best configuration is [googleplay (Component 1), WMD (Component 2), ATM_d (Component 3), SemFinder_a (Component 4)] and the worst is random.

Figure 4 shows the distributions of MRR and TOP1 by instance. For example, the box plot of SemFinder_a on the top right of Figure 4 shows the distribution of the MRR values of all the 84 configurations with SemFinder_a as the semantic matching algorithm. The box plots of the same Component type are sorted by median.

Note that some instances among the same component type belong to less configurations than others. For instance, WM is present in 48 configurations, while USE only in 12. This is because for WM we considered the pre-computed standard model and three models built from the three corpora of documents, while for BERT we only considered the pre-computed model.

Table 3 shows the distributions of the various component instances for three percentiles 1% (top 3 entries), 5% (top 13 entries), and 10% (top 26 entries). The values in the cells indicate the percentage of entries in the selected percentile (column) that uses a given component (row). For instance, every entry in the first percentile (1%) uses WM in both the lists sorted by the MRR and TOP1 metric.

We tested for *statistical significance* using a parametric two-sided t-test [74]: if *p-value* <= 0.05 we reject the null hypothesis that the two distributions are the same. We used a parametric test as the normality D'Agostino's K^2 test [69] confirmed that most distributions are normally distributed.

4.6 RQ1: Baseline Comparison

Random has the worst performance, much worse than the other configurations, and this confirms our expectation. When the 253 configurations are ordered by MRR values, the second last configuration has value 0.595, while random 0.201. When they are sorted by TOP1 values, the second last configuration has value 0.359, while random 0.065.

The syntactic based similarity metrics (ES and JS) generally perform significantly worse than word embedding models (see Figure 4). Indeed, none of the 24 configurations with either JS or ES appear in the top 10% configurations sorted by either MRR or TOP1 values (see Table 3).

This result confirms the hypothesis that often different developers use different words to express the same logical GUI action [15, 45]. This result motivates the use of word embedding models that help identify semantically similar albeit syntactically different words/sentences in widgets attributes.

4.7 RQ2: Component Effectiveness

Corpus of Documents (Component 1) Table 3 shows that the googleplay corpus dominates the other two corpora for all percentiles (although the comparisons of the Component 1 distributions in Figure 4 are without statistical significance). Notably in Table 3, the sum of the three percentages of blogs, manuals and googleplay never reaches 100%. This is because the remaining configurations involve multiple pre-computed models obtained with different corpora of documents. In general the pre-computed models performed better, but we cannot draw generally valid conclusions, because these models are obtained with different corpora of documents.

Interestingly, the googleplay corpus suffers less from the Out Of Vocabulary (OOV) issue than the other two corpora. OOV issues occur when we ask the model to compute the similarity between two words, out of which at least one does not belong to the corpus. We considered all the 36 configurations that use Word2vec as word embedding technique, and use blogs, manuals, or googleplay as corpora of documents. We divided these 36 configurations into three groups, according to the corpus of documents used. Then, we counted the cumulative number of OOV issues for each group. OOV issues are easy to identify, because Word2vec returns 0.0 at Line 8 of Algorithm 1. The group of configurations that use googleplay triggered 92,032 OOV issues, while the manuals and blogs corpora 279,370 and 163,036 issues, respectively.

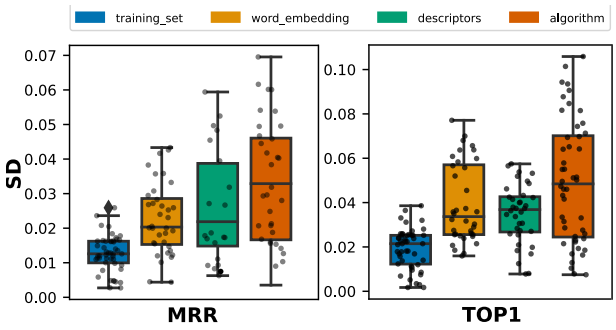

Figure 5: impact analysis

Word Embedding (Component 2) WM and USE are the best word embedding techniques according to both MRR and TOP1 (see Figure 4). The difference between WM and USE with Fast, Word2vec, Glove, BERT, JS, ES is always statistically significant (for both MRR and TOP1). Interestingly, WM dominates USE and all other techniques according to the percentiles reported in Table 3. We observe that sentence-level word embedding techniques perform statistically significant better than word-level ones. This result is supported by the observation that many GUI textual attributes have multiple words.

Event Descriptor Extractor (Component 3) ATM_d and *intersection* perform much better than *union* and CraftDroid_d, among the four groups of event descriptors (the difference is statistically significant). We root the poor performance of *union* and CraftDroid_d in the *activity-name* attribute (which is defined for each event). In fact, in our experiments many source and target events shared the same default *activity-name* (`main.activity`), and this affects the final scores. This is because if an unrelated event happens to have the same activity name of the source event, this event might yield a similarity score higher than the correct match (e^t_{gt}).

Semantic Matching Algorithm (Component 4) SemFinder_a outperforms both ATM_a and CraftDroid_a (always with statistical significance). Indeed, the MRR and TOP1 medians of SemFinder are higher than the median of both ATM_a and CraftDroid_a (Figure 4). Moreover, in the 10% percentiles of both MRR and TOP1, 85% of the entries use SemFinder_a as the semantic matching algorithm (Table 3). Each of the 84 configurations with SemFinder_a completed all 337 queries in 255 seconds on average, 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 reduces runtime while improving the results of semantic matching.

4.8 RQ3: Impact Analysis

We identified the component type with the highest impact on the semantic matching of GUI events with a so-called "local" sensitivity analysis [23], which varies the instance of one component type at a time while holding the others fixed [33]. For each of the four component types (Component 1, Component 2, Component 3 and Component 4), we clustered the 253 configurations, so that only the component under consideration varies, while the instances of the other three components are fixed. For example, if we consider Component 2 and exclude the random baseline, we have nine possible instances. Every time we fix the values for components Component 1, Component 3, Component 4, we define a new cluster with nine configurations (in which only Component 2 varies). Then, we compute the *standard deviation* (SD) of the MRR values of these nine configurations. This SD value represents the impact of Component 2 in the cluster (if the choice of Component 2 has high impact, the SD value is high, otherwise it is low) [33]. We repeated this process 28 times for every possible combination of the values of Component 1, Component 3, and Component 4, obtaining 28 SDs that globally capture the impact of Component 2 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 shows the distributions of the SDs values for category type. Semantic Matching Algorithm (Component 4) is the configuration with the highest impact for both MRR and TOP1 values, followed by Event Descriptor Extractor (Component 3), Word Embedding Technique (Component 2), and Corpus of Documents (Component 1). Researchers should consider this to prioritize their research effort on the most relevant components.

4.9 Threats to Validity

External validity A possible threat to the external validity is that our results may not generalize to other Android apps and test cases. We mitigated this threat by considering a large number of unique queries (337). The number of test migration scenarios in our study (139) is comparable with the scenarios used in the evaluation of test reuse approaches [15, 45]. Moreover, we collected the subjects from two benchmark datasets built by two independent teams, spanning several app categories and functionalities (see Table 2).

Internal validity A possible threat to the internal validity is that there might be errors in our framework that led to wrong results. We mitigated this threat by manually validating the correctness of the descriptors and metrics on a few queries. We manually scanned 337 queries and selected 30 queries with following characteristics: having empty descriptors, abnormally high or low MRR and TOP1 values. For such queries, we manually inspect the GUI of the app to check that the descriptors are correctly extracted. We also checked if the embedding models return the computed similarity scores. Moreover, we released our data and scripts and we welcome external validation [53].

Construct validity A possible threat to the construct validity is that we might not have faithfully re-implemented the semantic matching algorithms of ATM and CraftDroid. We mitigated this threat by referring to their original source code of the implementations provided by the authors of ATM and CraftDroid.

5 RELATED WORK

To the best of our knowledge, this paper is the first study on the semantic matching of GUI events for test reuse approaches. Recently, Zhao et al. propose the Fruiter framework [93] to comparatively evaluate test reuse techniques. Fruiter compares test reuse techniques as a whole, but does not support the evaluation and study of semantic matching in isolation. Fruiter alone cannot tell whether a test reuse technique works better than another because of a more

effective test generation or semantic matching of GUI events. In principle, our framework could be combined with Fruiter, to evaluate and investigate different combinations of test generation and semantic matching.

Techniques for reusing GUI tests are gaining popularity, as a valid solution to generate semantically meaningful test cases [15, 45, 75, 76]. In this study, we considered the test reuse approaches for Android applications: ATM [13, 15, 16] and CraftDroid [45]. We did not consider GUITestMigrator [14], because ATM is an extension of GUITestMigrator, which focuses on migrating GUI test cases of apps with the same specification. We also excluded test reuse approaches for Web apps [75, 76], and for adapting GUI tests across the Android and iOS versions of the same app [72]. We also excluded the GUI test reuse AdaptDroid [55] that was published after we conducted this study.

Some studies in the NLP community compared various word embedding techniques [9, 42, 87]. Li et. al report that word embedding techniques trained on domain specific corpora perform better on the related specialized tasks [42]. Their conclusion is inline with the results of this paper. Our study is the first one comparing word embedding techniques in the context of GUI events matching.

6 CONCLUSIONS AND FUTURE WORK

This paper presents the first study on semantic matching of GUI events for GUI test reuse/generation techniques. Our study involves 253 configurations of the semantic matching, 337 unique queries, and 8,099 distinct GUI events. We now highlight some of our key findings:

I. Sentence level word embedding techniques (WM, USE) perform generally much better than world level ones (Word2vec, Glove, and Fast). This is because many widget attributes are composed of multiple words (sentences). In fact, in our experiments, the widget attributes that we extracted are described with on average 2.39 words.

II. All component types impact on the effectiveness of the semantic matching. However, the semantic matching algorithm is the component type that impacts the most. Researchers should focus their effort in designing new and better algorithms. Moreover, SemFinder, the new algorithm proposed in this paper outperforms both the one of ATM [15] and the one of CraftDroid [45]. SemFinder consolidates both ATM and CraftDroid algorithms addressing some of their limitations. Differently from both ATM and CraftDroid, SemFinder is specifically designed for sentence-level word embedding models, which performs much better than word-level word embedding models for the semantic matching of GUI events.

III. When considering which widget attribute types should be used in the semantic matching of GUI events, the more is not always the better. Our experiments show that some attributes (such as, activity-name) can negatively affect the results. Also, the configurations that consider the largest number of attributes (union) are not the ones providing the best results. More research is needed to understand which widget attributes better describe the semantics of widgets. This would be especially important for derived attributes, as they could introduce meaningless and conflicting information. In our study we did not investigate which derived attributes would better describe a target widget, we followed the way both ATM and CraftDroid select the derived attributes. Automatically recognizing and removing meaningless and conflict information is an important future work.

IV. It might be preferable to train word embedding models with a corpora of documents specific to the mobile app domain. In fact, our proposed corpus of documents collected from the app descriptions in Google Play is more effective than general purpose corpora (although without statistical significance).

An important open research issue is related to the impact of the semantic matching of GUI events on the overall effectiveness of test reuse. This can be studied by combining our framework, which compares the semantic matching step, with Fruiter [93], which compares the whole test reuse activity (semantic matching + test generation). It would also be interesting to study the semantic matching of GUI events in other testing contexts that benefit from semantic matching, like GUI pattern-based test generation [34, 46, 52, 54], and GUI test repair [29, 43, 57, 58, 62, 63, 92].

Current test reuse approaches define the semantic similarity relations of GUI events as a one-to-one mapping between a source and a target event. However, there could be cases of one-to-many or many-to-one mappings, in which a source (or a target) event matches multiple target (or source) events. Although, test reuse approaches may spontaneously create one-to-many or many-to-one mapping during test generation because reaching a particular window or state requires the execution of *auxiliary events* [93]. Studying one-to-many or many-to-one mappings of GUI events would be an important future work.

Yet another promising research direction is the study of images and graphical representations of widgets as semantic descriptors. Indeed, images carry important semantic information about GUI widgets [20, 34, 88]. One could rely on ML techniques to classify images and graphical representations of widgets and convert them into textual representations of widgets (to be used as additional semantic descriptors). Also, one could leverage image analysis techniques to identify those widgets across apps that have similar graphical representations.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project *ASTERIx: Automatic System TEsting of InteRactive software applications* (SNF 200021_178742).

REFERENCES

[1] Ahmed Abbasi, Hsinchun Chen, and Arab Salem. 2008. Sentiment analysis in multiple languages: Feature selection for opinion classification in web forums. *ACM Transactions on Information Systems (TOIS)* 26, 3 (2008), 1–34.
[2] Adrian Chifor. 2021. Swiftnotes. https://play.google.com/store/apps/details?id=com.moonpi.swiftnotes. Last access: Jan 2021.
[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE '12)*. ACM, 258–261.
[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '12)*. ACM, 59:1–59:11.
[5] Andrzej Grzyb. 2021. Shopping List. https://play.google.com/store/apps/details?id=pl.com.andrzejgrzyb.shoppinglist. Last access: Jan 2021.
[6] Anthony Restaino. 2021. Lightning Browser. https://play.google.com/store/apps/details?id=acr.browser.lightning. Last access: Jan 2021.

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

[8] Ebru Arisoy, Tara N Sainath, Brian Kingsbury, and Bhuvana Ramabhadran. 2012. 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*. 20–28.

[9] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. 2014. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 238–247.

[10] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[11] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. 2012. Extracting Widget Descriptions from GUIs. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE '12)*. Springer, 347–361.

[12] Farnaz Behrang and Alessandro Orso. [n.d.]. ATM implementation. https://sites.google.com/view/apptestmigrator.

[13] Farnaz Behrang and Alessandro Orso. 2018. Poster: Automated Test Migration for Mobile Apps. In *Proceedings of the International Conference on Software Engineering (ICSE Poster '18)*. ACM, 384–385.

[14] Farnaz Behrang and Alessandro Orso. 2018. 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, 164–175.

[15] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *Proceedings of the International Conference on Automated Software Engineering (ASE'19)*. IEEE Computer Society, 54–65.

[16] Farnaz Behrang and Alessandro Orso. 2020. AppTestMigrator: a tool for automated test migration for Android apps. In *Proceedings of the International Conference on Software Engineering (ICSE DEMO '20)*. ACM, 17–20.

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

[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching Word Vectors with Subword Information. *arXiv preprint arXiv:1607.04606* (2016).

[19] Daniel Cer, Yinfei Yang, Sheng yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, and Ray Kurzweil. 2018. Universal Sentence Encoder. arXiv:1803.11175 [cs.CL]

[20] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhut, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: Predicting natural-language labels for mobile GUI components by deep learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 322–334.

[21] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. 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, 429–440.

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

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

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[25] Alan Dix. 2009. Human-computer interaction. In *Encyclopedia of database systems*. Springer, 1327–1331.

[26] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020 (ICSE'20)*. ACM, 481–492.

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

[28] Markus Ermuth and Michael Pradel. 2016. 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, 82–93.

[29] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42, 2 (2016), 170–186.

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

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

[32] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications via Model Abstraction and Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE '19)*. IEEE Computer Society, 269–280.

[33] DM Hamby. 1995. A comparison of sensitivity analysis techniques. *Health physics* 68, 2 (1995), 195–204.

[34] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. 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, 269–282.

[35] Tensor Flow Hub. [n.d.]. Token based text embedding trained on English Google News 200B corpus. https://tfhub.dev/google/nnlm-en-dim128/2. Last access: 2020-09-30.

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

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

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

[39] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In *Proceedings of the International Conference on International Conference on Machine Learning (ICML '15)*. 957–966.

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

[41] Vladimir I. Levenshtein. 1966. *Binary codes capable of correcting deletions, insertions, and reversals*. Technical Report 8. Soviet Physics Doklady. 707–710 pages.

[42] Hongmin Li, Xukun Li, Doina Caragea, and Cornelia Caragea. 2018. Comparison of word embeddings and sentence encodings as generalized representations for crisis tweet classification tasks. *Proceedings of ISCRAM Asia Pacific* (2018).

[43] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '17)*. IEEE Computer Society, 161–171.

[44] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. [n.d.]. Craftdroid implementation. https://github.com/seal-hub/CraftDroid.

[45] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *Proceedings of the International Conference on Automated Software Engineering (ASE'34)*. IEEE Computer Society, 42–53.

[46] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. 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, 111–122.

[47] Tie-Yan Liu. 2011. Learning to rank for information retrieval. (2011).

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

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

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

[51] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 94–105.

[52] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE '17)*. IEEE Computer Society, 16–26.

[53] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. *Semantic Matching of GUI Events for Test Reuse: Are We There Yet?* https://doi.org/10.5281/zenodo.4725222

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

[55] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2021. An Evolutionary Approach to Adapt Tests Across Mobile Apps. In *International Conference on Automation of Software Test (AST '21)*. 70–79. https://doi.org/10.1109/AST52587.2021.00016

[56] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. What test oracle should I use for effective GUI testing?. In *Proceedings of the International Conference on Automated Software Engineering (ASE '03)*. IEEE Computer Society, 164–173.

[57] Atif Memon, Adithya Nagarajan, and Qing Xie. 2005. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 1 (2005), 27–64.

[58] Atif M Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology* 18, 2 (2008), 4.

[59] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of The Working Conference on Reverse Engineering (WCRE '03)*. IEEE Computer

Society, 260–269.
[60] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[61] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the International Conference on Neural Information Processing Systems (NIPS '13)*. 3111–3119.
[62] M. Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2010. Automatically Repairing Test Cases for Evolving Method Declarations. In *ICSM'10: Proceedings of 26th IEEE International Conference on Software Maintenance*.
[63] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting Test Suite Evolution through Test Case Adaptation. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, 231–240.
[64] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. SIG-Droid: Automated System Input Feneration for Android Applications. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '15)*. IEEE Computer Society, 461–471.
[65] Kevin Moran, Mario Linares Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '16)*. IEEE Computer Society, 33–44.
[66] Mozilla. 2021. Firefox Focus. https://play.google.com/store/apps/details?id=org.mozilla.focus. Last access: Jan 2021.
[67] My.com B.V. 2021. myMail. https://play.google.com/store/apps/details?id=ru.mail.mailapp. Last access: Jan 2021.
[68] OpenIntents. 2021. OI Shopping list. https://play.google.com/store/apps/details?id=org.openintents.shopping. Last access: Jan 2021.
[69] Egon S Pearson, Ralph B D "'AGOSTINO, and Kimiko O Bowman. 1977. Tests for departure from normality: Comparison of powers. *Biometrika* 64, 2 (1977), 231–246.
[70] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543.
[71] plafu. 2021. Writeily Pro. https://f-droid.org/en/packages/me.writeily. Last access: Jan 2021.
[72] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from iOS to Android. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '19)*. ACM, 284–295.
[73] rainbowshops. 2021. Rainbow. https://play.google.com/store/apps/details?id=com.rainbowshops. Last access: Jan 2021.
[74] Dieter Rasch and Volker Guiard. 2004. The robustness of parametric statistical methods. *Psychology Science* 46 (2004), 175–208.
[75] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Efficient GUI test generation by learning from tests of other apps. In *Proceedings of the International Conference on Software Engineering (ICSE Poster '18)*. ACM, 370–371.
[76] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Transferring tests across web applications. In *International Conference on Web Engineering*. Springer, 50–64.
[77] roxrook. 2021. Pocket Note. https://github.com/roxrook/pocket-note-android. Last access: Jan 2021.
[78] Ruben Roy. 2021. Minimal. https://f-droid.org/en/packages/com.rubenroy.minimaltodo/. Last access: Jan 2021.
[79] Jonathan Schler, Moshe Koppel, Shlomo Argamon, and James W Pennebaker. 2006. Effects of age and gender on blogging.. In *AAAI spring symposium: Computational approaches to analyzing weblogs*, Vol. 6. 199–205.
[80] H Andrew Schwartz, Johannes C Eichstaedt, Margaret L Kern, Lukasz Dziurzynski, Stephanie M Ramones, Megha Agrawal, Achal Shah, Michal Kosinski, David Stillwell, Martin EP Seligman, et al. 2013. Personality, gender, and age in the language of social media: The open-vocabulary approach. *PloS one* 8, 9 (2013), e73791.
[81] SECUSO Research Group. 2021. Shopping List (Privacy Friendly). https://play.google.com/store/apps/details?id=privacyfriendlyshoppinglist.secuso.org.privacyfriendlyshoppinglist. Last access: Jan 2021.
[82] SECUSO Research Group. 2021. Todo List. https://f-droid.org/en/packages/douzifly.list/. Last access: Jan 2021.
[83] Stoutner. 2021. Privacy Browser. https://play.google.com/store/apps/details?id=com.stoutner.privacybrowser.standard. Last access: Jan 2021.
[84] TLe Apps. 2021. Simple Tip Calculator. https://play.google.com/store/apps/details?id=com.tleapps.simpletipcalculator. Last access: Jan 2021.
[85] Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. 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, 384–394.
[86] Vansuita. 2021. Shopping List. https://play.google.com/store/apps/details?id=br.com.activity. Last access: Jan 2021.
[87] Yanshan Wang, Sijia Liu, Naveed Afzal, Majid Rastegar-Mojarad, Liwei Wang, Feichen Shen, Paul Kingsbury, and Hongfang Liu. 2018. A comparison of word embeddings for the biomedical natural language processing. *Journal of biomedical informatics* 87 (2018), 12–20.
[88] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 257–268.
[89] xorum. 2021. Open Money Tracker. https://play.google.com/store/apps/details?id=com.blogspot.e_kanivets.moneytracker. Last access: Jan 2021.
[90] Yelp, Inc. 2021. Yelp. https://play.google.com/store/apps/details?id=com.yelp.android. Last access: Jan 2021.
[91] ZaidiSoft. 2021. Tip Calculator Plus. https://play.google.com/store/apps/details?id=com.zaidisoft.teninone. Last access: Jan 2021.
[92] Sai Zhang, Hao Lü, and Michael D Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '13)*. ACM, 45–55.
[93] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE 20)*. 1190–1201.
[94] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the International Conference on Software Engineering (ICSE '19)*. IEEE Computer Society, 128–139.