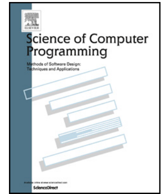
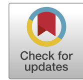
Science of Computer Programming 232 (2024) 103052

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico





Enhancing test reuse with GUI events deduplication and adaptive semantic matching

Shuqi Liu [a], Yu Zhou [a,*], Longbing Ji [a], Tingting Han [b], Taolue Chen [b,*]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] Department of Computer Science, Birkbeck, University of London, UK

ARTICLE INFO

Keywords:
Test reuse
GUI events deduplication
Semantic matching
Adaptive strategy
Test oracle

ABSTRACT

Developers typically employ Graphical User Interface (GUI) testing to ensure the expected behavior of applications, but they face the challenge of designing appropriate test cases with functional features. Recently, researchers have proposed several test reuse methods based on semantic matching to alleviate the burden. However, the limited text semantic information and semantic matching rules between events severely limit the existing test reuse methods. In this paper, we propose TREADROID (Test Reuse EnhAncer for anDROID applications), a framework that combines GUI events deduplication with the adaptive semantic matching strategy to enhance the usability of the reused tests. Considering the connection between widget attribute texts, we categorize attributes and measure widget similarity based on the same corresponding attributes as well as across attributes in the same group. In addition, we propose a deduplication strategy for GUI events to reduce the redundancy caused by reusing a test with unique functionality. To further bridge the semantic gap, we design a two-stage adaptive matching strategy to search for the target test with functionality closer to that of the source test. Experimental evaluation against the baseline methods on 25 applications demonstrates that: (i) the adaptive semantic matching strategy overall improves the performance of widget mapping; (ii) GUI events deduplication dramatically increases the precision of events on average, even reaching 100% for multiple tests; (iii) TREADROID can significantly reduce the manual effort of creating tests for similar applications.

1. Introduction

Graphical User Interface (GUI) testing plays a significant role in the development and maintenance of mobile applications [1–4]. It enables developers to rapidly identify potential functional exceptions in applications that affect user experience. To reduce the burden of manually designing GUI test cases (GUI tests in short), numerous automated test generation techniques [5–17] have been investigated to help verify the behavior of applications.

A well-known problem with automatic test generation is that it ignores much of the semantic information of applications. Typically, exploration methods employing different strategies, such as random-based [9,10,17] and search-based [5,13,15] techniques, are utilized for test generation. While these methods aim to maximize coverage and finding more bugs, they often generate tests that

* Corresponding authors.
E-mail addresses: liushuqi@nuaa.edu.cn (S. Liu), zhouyu@nuaa.edu.cn (Y. Zhou), jlb_nuaa@nuaa.edu.cn (L. Ji), t.han@bbk.ac.uk (T. Han), t.chen@bbk.ac.uk (T. Chen).

https://doi.org/10.1016/j.scico.2023.103052
Received 19 May 2023; Received in revised form 26 October 2023; Accepted 27 October 2023

Available online 31 October 2023
0167-6423/© 2023 Elsevier B.V. All rights reserved.

are less functional. In other words, generated tests should not only provide comprehensive coverage but also be as representative as possible to effectively verify the normal implementation of application functionalities. Therefore, generating meaningful tests with semantic information is of great necessity.

In recent years, existing methods have generated meaningful tests that verify the behavior of applications through test reuse [18–23]. These methods focus on reusing existing tests designed for one application to test other similar applications. Specifically, existing tests will automatically match sequences of events from similar applications that achieve the same functionality. As illustrated in Fig. 1, for instance, consider an existing test (a) created for application *Simple Tip Calculator*, which calculates tips. These methods attempt to generate event sequences (c) for testing application *Free Tip Calculator*. Researchers tend to design similarity calculation methods for exact matching, such as weighted average textual similarity of corresponding attributes [20,21] or further consider fuzzy strategy [22]. Matching two events semantically using limited widget information can be challenging. In specific scenarios, due to insufficient widget attribute information, the matched events may not perform the same functionality as the source events. For instance, for the "calculate total bill with tip" test scenario, the event "fill in the bill" from an existing test may match the "fill in the number of people" event of another similar application, without raising an exception. The semantic matching method limits the accuracy of event matching. Therefore, additional measures are necessary for increasing the chance of test reuse. However, existing works rarely support adopting additional strategies to achieve closer functional approximation of unsuccessful test reuse, which requires identifying incorrectly matched events. Another concern relates to differences in UI design, which may lead to variations in the operational steps required to implement specific functionality in two similar applications. Recent work [20] has considered supplementing events with additional steps in the target application compared to the source application. Nonetheless, reusing such existing tests that implement functionality with additional steps in the source application may result in redundant events and interfere with the implementation of the functionality in the target application.

To mitigate the problem above, in our prior work [24], we introduced an adaptive strategy aimed at optimizing the generated tests. However, this adaptive strategy had certain limitations in identifying incorrectly matched events, which could potentially lead to missed opportunities for improving the accuracy of event matching. Furthermore, the work mainly depended on the use of the same corresponding attributes to semantically match widgets across similar applications. This approach may ignore valuable textual information embedded in different attributes, resulting in incorrect event matching. In cases where the current research do not work well, it is necessary to employ more comprehensive measures.

Our analysis of state-of-the-art research in relevant fields has revealed that the utilization of semantic information and semantic matching strategy significantly affects the accuracy of generated events. In light of this observation, we propose an automated framework TREADROID (Test Reuse EnhAncer for anDROID applications) in this paper to enhance the usability of reused tests. This paper presents several noteworthy extensions to our preliminary work [24]: (1) We introduce new rules that make full use of the available semantic information defined in widgets. The attributes of widgets are grouped by leveraging the connections between the textual information they represent. We consider both the same attributes [20] and those across attributes within the same group together to characterize the textual similarity of widgets for semantic matching. (2) We introduce a restriction rule for deduplicating GUI events. The rule can help reduce the false removal of duplicate events to ensure the performance of subsequent adaptive semantic matching on the obtained tests. (3) We enhance the adaptive rules, which captures imprecise event matching resulting from semantic gaps, by implementing a two-stage process. The improved adaptive rules can effectively reduce the number of search iterations required during adaptive semantic matching. Furthermore, the adoption of a fitness function within the adaptive strategy enables a more thorough exploration to search tests with superior semantic matching compared to the obtained tests. Based on the above extensions, we present new experiments evaluation to identify enhancements that promote test reuse.

The main contributions of this paper are summarized as follows.

- We propose an automated test reuse framework TREADROID,[1] which integrates GUI events deduplication and adaptive semantic matching strategy to improve the usability of generated tests.
- During the process of exploring space with the adaptive matching strategy, TREADROID rewards the generated tests that are more semantically similar to the existing tests guided by the fitness function.
- We conducted extensive comparative experiments with the state-of-the-art baseline approaches on 25 popular open-source Android applications to evaluate the performance of TREADROID. The results demonstrate that TREADROID is superior to the baseline approaches in both fidelity (precision and recall) and utility (manual effort reduction), which enhances the performance of test reuse.

The rest of this paper is organized as follows. Section 2 presents the background knowledge and the motivating example of our work. Section 3 outlines the overall framework and details each component. Section 4 conducts experiments to evaluate our framework and analyze the experimental results. The potential threats to validity are discussed in Section 5. Finally, Section 6 and Section 7 include related work and conclusions, respectively.

[1] https://github.com/liushuqi-2022/TREADROID-repo.

2. Background and motivating example

Users interact with mobile application through the GUI. There are many widgets in a window, which are atomic GUI elements. Users can trigger an actionable widget to interact with the GUI by clicking a button or filling a text box with text, which forms a GUI event. Formally, an event can be expressed as $e_i = (w_i, a_i)$, where w_i denotes the widget, and a_i represents the action performed on w_i. Several ordered events form a test that verifies the application implements a specific functionality. These tests typically contain oracle events, which are designed to check whether the application successfully accomplishes the expected functionalities.

Recent studies suggest employing the test reuse technique, which reuses existing tests from similar applications that share common functionality, to reduce the cost of testing new applications. Fig. 1 shows that test (b) can be obtained by reusing an existing test (a) on the application *Free Tip Calculator* using the existing method [20]. Test (b) is a sequence of events searched in the target application (target app) *Free Tip Calculator*, based on the semantic similarity with each event from the source test (a).

Comparing the semantic similarity between the textual descriptions of two widgets to match GUI events and oracles across applications is a critical step in test reuse. The textual descriptions of widgets undeniably play a pivotal role in achieving semantic matching. The UI hierarchy provides these textual semantic descriptions of the widget through its defined attributes, such as 'resource id', 'text', and 'content-desc'. Earlier studies [20], [21] have addressed the challenge of mapping widgets with distinct UI designs but similar semantic characteristics. As an example, they have successfully established mappings between widgets such as w_4^s from *Simple Tip Calculator* and w_4^t from *Free Tip Calculator* as depicted in Fig. 1, despite variations in their distribution within the user interface.

By leveraging natural language processing, previous work has demonstrated the potential of resolving these important mappings between GUI elements for test reuse. However, these techniques still face challenges in semantic matching. As a terrible example, test (b) shows the results of an existing method [20] reusing test (a) through semantic matching, which reveals the challenges of test reuse. First, the textual semantics described between widget attributes may be similar. When dealing with various types of widgets, the text information in different attribute values affects the effective utilization of limited information to compute semantic similarity. For example, the text information of an image widget can be contained in the attributes 'resource-id' or 'content-desc'. Second, greedily selecting the widgets with the highest similarity for matching may lead to failure of test reuse. In test (b), event e_1^t matches event e_3^t with the highest similarity, but this is not the best result. Third, it is possible that the source app contains some unique functionalities, resulting in the matching of redundant events in the target tests. For instance, the event e_0^s in the source test (a) that represents the acceptance of the cache functionality matches the e_0^t of the target app. However, the matched event e_0^t in test (b) does not contribute to testing the functionality of splitting the tip.

We have designed TREADROID to address the aforementioned challenges. First, TREADROID groups attributes based on textual connections between widgets, allowing for a more accurate measure of similarity. It calculates similarity cross attributes within the same group as well as the corresponding attributes. Second, faced with the fact that the highest similarity may not be the best match, TREADROID employs an adaptive strategy to identify events in the generated test that may not be correctly matched. It guides the search direction using a flexible fitness function, which rewards tests with higher overall similarity. This way, it aims to discover tests that closely approximate the functionality of the source test. Furthermore, TREADROID overcomes redundancy arising from the unique functionality in the source test by employing a GUI event filtering rule. This rule can effectively capture and eliminate unrelated events from the generated test. Specifically, the next section will present TREADROID and describe how it overcomes these challenges.

3. Approach

Fig. 2 gives an overview of our framework TREADROID, which primarily consists of four components: *Data Processing*, *Test Generation*, *GUI Events Deduplication*, and *Adaptive Semantic Matching*. First, we process the inputs, i.e., a source test, a source app, and a target app, to prepare data for test reuse (Section 3.1, Data Processing). Second, the widgets in the source test are iteratively greedy matched with the highest similarity in the target app to generate an initial test (Section 3.2, Test Generation). Third, the initial test is further examined to remove duplicate GUI events that satisfy specially designed rules (Section 3.3, GUI Events Deduplication). Finally, we focus on identifying potentially incorrectly matched events in the obtained tests, and then extend the search to semantically match the target test with functionality that is closer to the source test. (Section 3.4, Adaptive Semantic Matching).

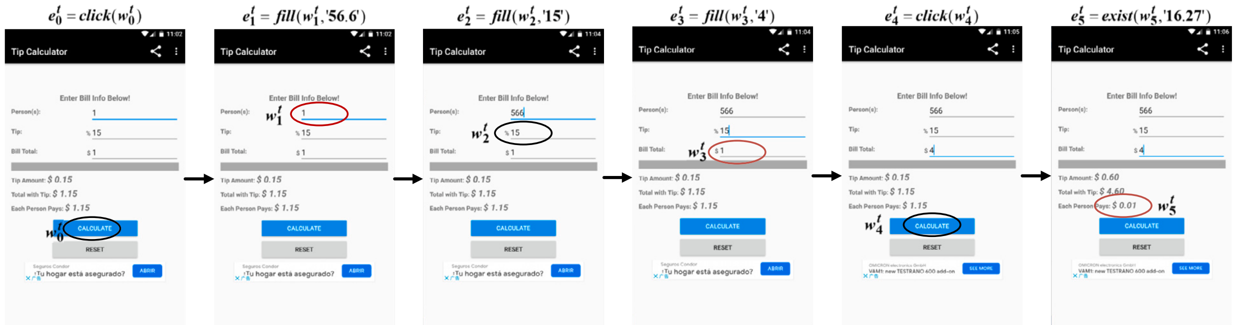
With a general understanding, we describe each part in detail below.

3.1. Data processing

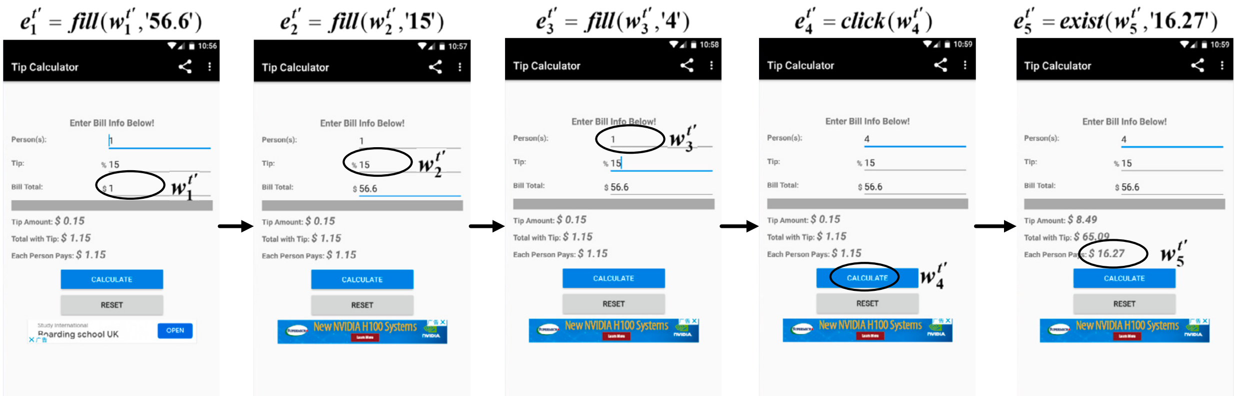
The Data Processing stage involves two key steps: test augmentation and model extraction. In this stage, TREADROID preprocesses both the source test and the target app to extract the relevant semantic information necessary for matching widgets. Since this component follows the step of the recent method CRAFTDROID, we only describe it briefly. For more specific details, please refer to the literature [20].

Test augmentation To facilitate test reuse across applications, we first perform test augmentation on the existing test to obtain an augmented test with semantic information. As mentioned in Section 2, we define a test t as $\{(w_1, a_1), (w_2, a_2), \cdots, (w_n, a_n)\}$, where (w_i, a_i) represents an event. These ordered events form a test designed to verify the behavior of the application.

(a) The existing test for *Simple Tip Calculator*



(b) The reused test for *Free Tip Calculator* using CRAFTDROID



(c) The expected reused test for *Free Tip Calculator*

Fig. 1. A simple example of test reuse. Test (b) is obtained by reusing the existing test (a) using CRAFTDROID, and test (c) is the expected test for *Free Tip Calculator* by reusing an existing test (a).

Test augmentation is to extract the semantic information of widgets in the source test. Through the *adb* tool,[2] we can extract the XML file corresponding to the UI hierarchy of the GUI screen where the widget is situated. Subsequently, the textual semantic information of the widgets can be parsed from this XML file.

An example of a GUI screen for application *Simple Tip Calculator* is shown in Fig. 3, where w_1^s is an editable bill widget that displays text "100" by default. We can extract textual semantic information represented by both attributes and their corresponding values. Generally speaking, we can extract the semantic information of the test (a) illustrated in Fig. 1 by acquiring following steps iteratively. After launching the application *Simple Tip Calculator*, (i) we extract the semantic information of the widget w_0^s from the

[2] https://developer.android.com/studio/command-line/adb.html.

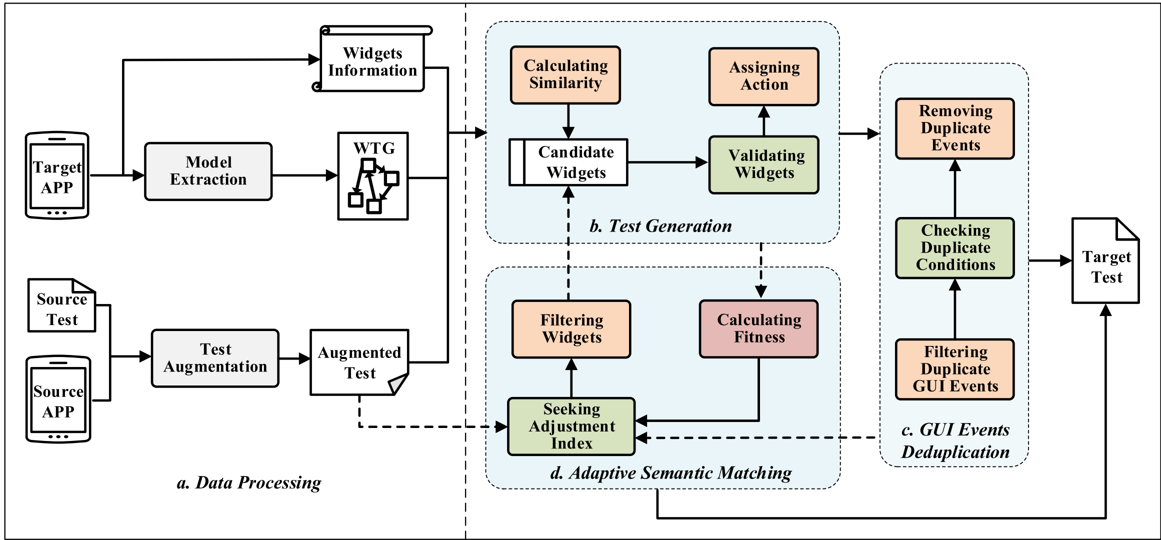
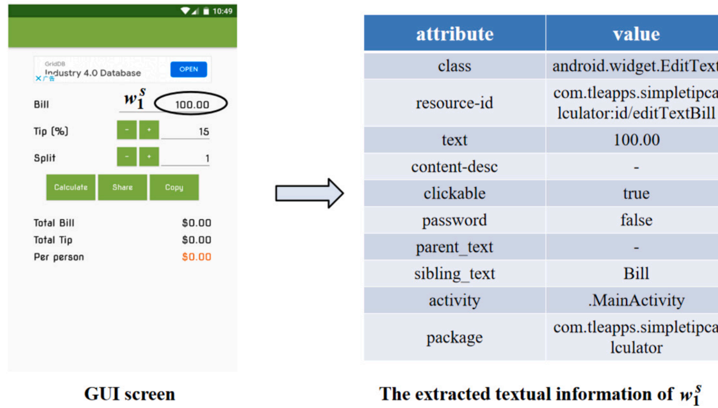


Fig. 2. The framework of TREADROID.



attribute	value
class	android.widget.EditText
resource-id	com.tleapps.simpletipcalculator:id/editTextBill
text	100.00
content-desc	-
clickable	true
password	false
parent_text	-
sibling_text	Bill
activity	.MainActivity
package	com.tleapps.simpletipcalculator

Fig. 3. The example of extracted textual information of widget w_1^s from GUI screen.

XML file corresponding to the current screen; (ii) the event is executed according to the action "click", and an updated GUI page for the application is obtained. Steps (i) and (ii) are repeated iteratively until the semantic information for all widgets is collected.

Model extraction Model extraction mainly analyzes the interaction between activities through the source code of the target app. These interactions are visually represented through a window transition graph (WTG). Formally, WTG consists of individual nodes and directed edges between nodes, where nodes and edges are composed of activities and events respectively. Leveraging the mapping established between widgets and activities in the WTG, the reachability of the current activity to the activity where candidate widgets are located can be validated.

WTG is built in two steps via the model extractor introduced by [25]. The model extractor first extracts activities and widgets from Manifest and XML-based meta-data files. It then analyzes the event handlers of the widgets to identify transitions between activities. For instance, in the case of testing the behavior of application *Minimal* for adding a task, the corresponding WTG triggered during its execution is depicted in Fig. 4. With triggering event e_1^m, i.e., clicking button w_1^m, *Minimal* switches to the activity page for adding a task. This transition corresponds to the activity change from 'Main' to 'AddToDo'. Similar activity transitions are triggered by the execution of subsequent events. Afterward, WTG is instrumental in obtaining and verifying candidate widgets for matching each event in the source test. As the application is executed, the WTG can be updated based on the feedback from the running information to make it more complete.

3.2. Test generation

The main purpose of test generation is to obtain an initial reused test. Each event in the augmented source test is iteratively matched with the target app to form an event sequence. **Algorithm 1** illustrates the whole process. The specific rule for matching

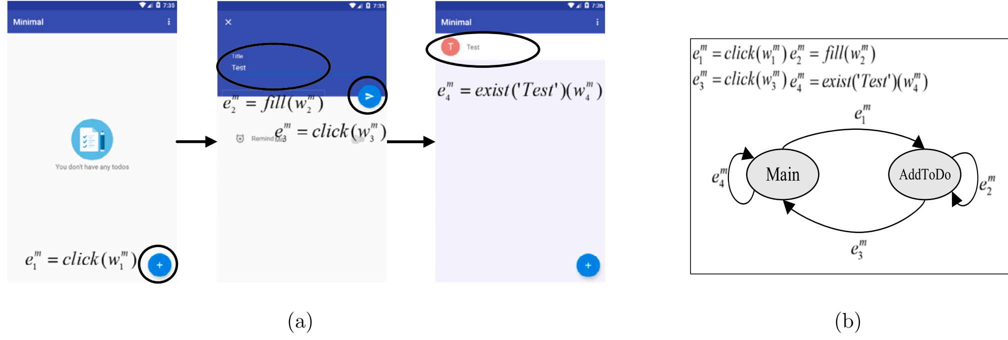


Fig. 4. The example of WTG corresponds to the test that validates adding a task. (a) The test for validating the function of adding a task. (b) The WTG is triggered by the execution test (a).

Algorithm 1 Test generation.

Input: target app a_t, WTG of target app a_t, widgets on the activities of target app w_{tb}, augmented source test $t'_s = \{(w_1^{s'}, a_1), (w_2^{s'}, a_2), \cdots, (w_n^{s'}, a_n)\}$

Output: Initial test $t_{init} = \{(w^t_{m_1}, a_{m_1}), (w^t_{m_2}, a_{m_2}), \cdots, (w^t_{m_n}, a_{m_n})\}$

1: Initialize: $t_{init} = \emptyset, fitness_{init} = -1$
2: **while** $\triangle fitness(t_{init}) > threshold_1$ or $fitness(t_{init}) \leq threshold_2$ **do**
3: **for** $(w_i^{s'}, a_i) \in t'_s$ **do**
4: $w_{candidates}$ = getCandidates($w_i^{s'}$, w_{tb}, WTG)
5: **for** $w^t_m \in w_{candidates}$ **do**
6: $leadingEvents$ = validateReach($w^t_m, WTG, w_{tb}, t_{init}$)
7: **if** $leadingEvents \neq$ null **then**
8: $a_m = generateAction(w^t_{m_i}, a_i, w^t_m)$
9: $t_{init} = t_{init} \cup \{leadingEvents, (w^t_m, a_m)\}$
10: **break**
11: **end if**
12: **end for**
13: **end for**
14: **end while**
15: **return** t_{init}

each event in the augmented source test is defined as follows. First, the semantic similarity between the widgets w_{tb} searched by the target app and the widget $w_i^{s'}$ in the augmented source test is calculated. The widgets w_{tb} are sorted by similarity to establish a set of candidate widgets $w_{candidates}$ (Line 4, details in Section 3.2.1). Then, starting from the widget with the highest similarity, we identify the reachable widget and return the leading events $leadingEvents$ to the widget $w_i^{s'}$ being validated (Line 6, details in Section 3.2.2). Finally, based on the action a_i performed on the source widget $w_i^{s'}$, we assign the appropriate action a_m to the reachable widget (w^t_m, a_m) as the target event, along with the leading events (Lines 7-9, details in Section 3.2.3). By following these steps until all source events have been traversed, an initial test t_{init} is generated. These steps are explained in more detail in the following parts.

3.2.1. Similarity calculation

Considering the large number of GUI widgets, we design the similarity calculation to more quickly determine the best matching widget for each source widget. More specifically, the function `getCandidates` is responsible for constructing a set of candidate widgets (Line 4 in **Algorithm 1**). Measuring the semantic similarity between two widgets poses challenging as it plays a significant role in event matching. To better represent the similarity between two widgets, we take two main factors into consideration: (1) the textual information and (2) the position relative to current activity. Initially, the textual similarity between the source widget $w_i^{s'}$ and each widget in w_{tb} of the target app is calculated. Subsequently, utilizing the extracted WTG of the target app, the distance between the widget and the current activity window is further measured to obtain the final similarity score.

Textual similarity In Section 3.1, we collected the semantic information of widgets from the augmented source test and the target app. We calculate the textual similarity between two different widgets following the steps below.

The attributes of collected widgets may contain compound words as their values. To ensure the accuracy of the similarity calculation, text preprocessing is performed as a preliminary step. We employ techniques commonly used in natural language processing (NLP), such as tokenization and stopword removal, on the attribute values of each widget. Taking widget w_1^s in Fig. 3 as an example, the word list [‘*edit*’,‘*text*’,‘*bill*’] can be obtained by preprocessing the value ‘*editTextBill*’ from the attribute *resource-id*. To prevent the loss of crucial information, we refrain from applying stopword removal to the value of the attribute *text*, which represents the text displayed by the widget.

Once we have determined multiple attribute word lists of widgets, the next step is to calculate the textual similarity between two widgets. First, the word2vec model [26] is used to obtain the vector representations $\overrightarrow{V_w}$ and $\overrightarrow{V_{w'}}$ for the words w and w'. Next, we measure the textual similarity between words w and w' by calculating the cosine distance between the two vectors, as follows:

widget w_1

attribute	value
class	android.widget. ImageButton
resource-id	android:id/up
text	-
content-desc	-
clickable	true
password	false
parent_text	-
sibling_text	-
activity	.activity. MessageList
package	com.fsck.k9

The textual information of widget w_1

widget w_2

attribute	value
class	android.widget. ImageButton
resource-id	-
text	-
content-desc	Navigate up
clickable	true
password	false
parent_text	-
sibling_text	-
activity	ru.mail.ui.write mail. WriteActivity
package	com.my.mail

The textual information of widget w_2

Fig. 5. Semantic information of widget w_1 extracted from application *K-9* and widget w_2 extracted from application *myMail*.

$$sim(w, w') = \frac{\overrightarrow{V_w} \cdot \overrightarrow{V_{w'}}}{|\overrightarrow{V_w}||\overrightarrow{V_{w'}}|}$$

Then, the similarity between two attribute word lists a_s and a_t can be expressed integrally by:

$$sim(a_s, a_t) = \frac{\sum_{w \in a_s} \max_{w' \in a_t} sim(w, w')}{|a_s|}$$

where $\max sim(w, w')$ indicates the highest similarity between words w and w'.

Attributes and their corresponding values contribute to the textual semantic information of the widget. Assessing the similarity between two widgets based on limited textual information presents a challenge, as unique design styles may lead to differences in the semantic text attributes of the two widgets. For instance, consider two image buttons, w_1 and w_2, used for upward navigation in two similar applications, as depicted in Fig. 5. CRAFTDROID [20] only treats the same attributes to calculate the similarity, which potentially leads to the loss of essential semantic information. In their calculation, a similarity value of 0 indicates no semantic connection between the two widgets. To more effectively capture the overall characteristics of two widgets, we incorporate both cross-attributes and same corresponding attributes into the similarity calculation.

For any same attribute, a_s and a_t are considered the attribute word lists of widgets w_i^s and w_m^t, respectively. Based on the above, the textual similarity between widgets w_i^s and w_m^t is defined as:

$$sim_1(w_i^s, w_m^t) = \frac{\sum_{a_s \in w_i^s} sim(a_s, a_t) * wg(a)}{|w_i^s|}$$

where $wg(a)$ represents a weight, indicating the importance of attribute a among all attributes.

Textual similarity across attributes also makes a non-negligible contribution to semantic matching. We group attributes that describe similar content. For image type widgets, since they lack the attribute 'text' content, we group *resource-id*, *content-desc*, and *sibling-text*. For other types of widgets, we define *text*, *resource-id*, and *content-desc* as crossable attributes. Therefore, We calculate the textual similarity across attributes between widgets w_i^s and w_m^t as follows:

$$sim_2(w_i^s, w_m^t) = \max_{a'_{sp} \in w_i^s, a''_{tp} \in w_m^t} sim(a'_{sp}, a''_{tp})$$

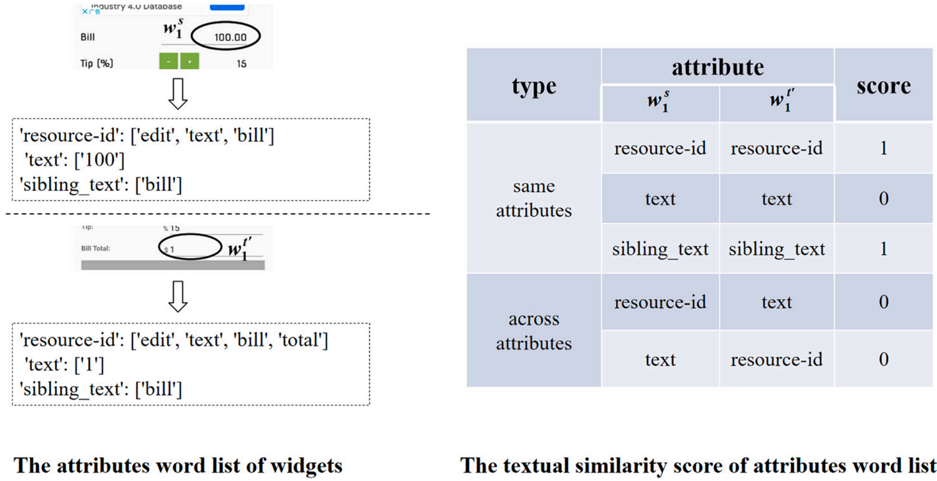
where a'_{sp} and a''_{tp} are the word lists of two different attributes a'_p and a''_p.

Finally, the textual information similarity between widgets w_i^s and w_m^t is:

$$sim(w_i^s, w_m^t) = sim_1(w_i^s, w_m^t) + sim_2(w_i^s, w_m^t)$$

We consider a simple example to calculate the textual similarity between two widgets w_1^s and $w_1^{t'}$ in Fig. 1. Assuming that we have obtained the meaningful attribute word lists for widgets w_1^s and $w_1^{t'}$ as shown in Fig. 6. First we calculate the similarity scores of word lists that include the same attributes, which are *resource-id*, *text*, and *sibling-text*, as 1, 0, and 1, respectively. If the weights of these three attributes are set as 1, 0.5 and 1.5 respectively, we can obtain a similarity score of $sim_1 = (1 * 1 + 1 * 1.5)/3 = 0.83$ for the same attributes of the two widgets. Similarly, we calculate the similarity score of word lists across attributes within the same group as shown in Fig. 6, and in this case, the largest score is $sim_2 = 0$. Then, we obtain the textual similarity between w_1^s and $w_1^{t'}$ as $sim = sim_1 + sim_2 = 0.83$.

Final similarity A widget may be similar to widgets in different GUI activities. To find the widgets that are closest to the current activity, we employ the WTG to obtain the shortest path d_s. In other words, we need to determine the distance between the current



type	attribute w_1^s	attribute $w_1^{t'}$	score
same attributes	resource-id	resource-id	1
	text	text	0
	sibling_text	sibling_text	1
across attributes	resource-id	text	0
	text	resource-id	0

Fig. 6. The textual similarity score of attributes word lists between widgets w_1^s and $w_1^{t'}$.

activity and the activity that contains the candidate widget w_m^t. Generally, the final similarity between widgets w_i^s and w_m^t can be expressed as:

$$sim'(w_i^s, w_m^t) = \frac{sim(w_i^s, w_m^t)}{1 + \log_2(d_s + 1)}$$

We prioritize the selection of candidate events as target events based on their closer semantic similarity to source events. An increase in the number of steps might deviate from the intended functionality. For example, through WTG, we can retrieve two candidate paths from the current activity S_{cur} to the activity S_4 where w_m^t is located as $path_1 = S_{cur} \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \xrightarrow{e_3} S_3 \xrightarrow{e_m^t} S_4$ and $path_2 = S_{cur} \xrightarrow{e_1} S_1 \xrightarrow{e_4} S_3 \xrightarrow{e_m^t} S_4$, respectively. Among them, S denotes a certain activity state, e denotes an event. We would prioritize the shorter path $path_2$, that is, the candidate GUI widget that is closer to the current activity, for extending the sequence of the events.

3.2.2. Widget reachability verification

Widget reachability verification aims to search for reachable candidate widgets through paths between activities. **Algorithm** 2 describes how the widget reachability verification works. For each candidate widget w_m^t, the matched events are executed first to obtain the current activity (Lines 1-2). The pre-collected widgets w_{tb} assist in locating the activity $destAct$ that w_m^t belongs to (Line 3). Next, all paths from the current activity $srcAct$ to $destAct$ in WTG are explored (Line 4).

Most importantly, we validate the reachability of widget w_m^t by executing each path (Lines 5-19). Function `getstepping` checks whether the path is reachable and provides the exploration result (Line 10). The verification results are divided into two cases: one is when the path is reachable, in which case the leading events are considered as stepping (Line 12); the other is when the path is unreachable, and the leading events are null (Line 16).

Several widgets are designed within the same activity, which can result in the repetitive execution of common paths during reachability validation. The baseline method [20] validates candidate widgets by exhaustively exploring all paths, which can be time-consuming. To avoid unnecessary time consumption, we store previously validated paths and their corresponding leading events to improve efficiency (Line 13). As a result, before validating a path, we first check whether it has been validated before (Line 6). If it has been validated, the associated leading events are directly utilized (Line 7).

3.2.3. Action assignment

We assign an appropriate action a_{m_i} to the reachable widget based on the action a_i of the source event. This allocation is generally performed regardless of whether it is a GUI or an oracle event. For example, we need to reuse event $e_5^s = exist(w_5^s, \text{'16.27'})$ of test (a) in Fig. 1, and finally the same action is assigned to widget $w_5^{t'}$. In other words, we check whether the text '16.27' exists in the current activity of target app. If not, an empty oracle is generated; otherwise, the same action is generated to form the target oracle event.

Due to the diverse UI designs of widgets, assigning the same action to a similar widget may not always be effective. For example, different list applications may require distinct actions for deleting a task, such as long-clicking or right-swiping. Simply allocating actions consistent with the source event is not appropriate. To alleviate this problem, we introduce a mutation operation for unique actions and utilize the fitness score to identify the most suitable action. This mutation operation is exemplified by transitions between actions like long-clicking and swiping. Specifically, we apply two operations to the widget with unique actions: (i) following the action of the source widget to generate t_1, and (ii) performing a mutation operation to obtain t_2. We then evaluate the two tests t_1 and t_2 by using the fitness function. Consequently, the test with the higher fitness score is considered more suitable for the assigned action. This mutation strategy for unique action allows us to dynamically adapt the action based on the performance of the overall test.

Algorithm 2 Reachability verification.

```
Input: target app a_t, WTG of target app a_t, widgets on the activities of target app w_tb,
Initial test t_init, candidate widget w^t_m
Output: leading events leadingEvents
 1: execute(t_init)
 2: srcAct = getCurrentActivity()
 3: destAct = getActivity(w^t_m, w_tb)
 4: paths = getPaths(srcAct, destAct, WTG)
 5: for path ∈ paths do
 6:     if path ∈ steppings.keys() then
 7:         leadingEvents = steppings[path]
 8:         break
 9:     else
10:         stepping = getstepping(w^t_m, path, w_tb)
11:         if stepping then
12:             leadingEvents = stepping
13:             steppings[path] = stepping
14:             break
15:         else
16:             leadingEvents = null
17:         end if
18:     end if
19: end for
20: return leadingEvents
```

To determine which generated tests are most similar to the source tests, we conduct a quantitative evaluation using the fitness function. This function guides the exploration process to generate a target test that closely approximates the functionality of the source test. The fitness function is defined based on the similarity of the events, which provides an overall quantification of the similarity between the generated test and the source test. It can be represented as follows:

$$fitness(t) = w_1 \sum_{i=1}^{n} sim_{e_i} |e_i \in gui| + w_2 \sum_{j=1}^{n} sim_{e_j} |e_j \in oracle|$$

where $fitness(t)$ denotes the fitness score of test t, w_1 and w_2 denote the weights assigned to GUI and oracle events in fitness score respectively, $sim_{e_i}|e_i \in gui|$ and $sim_{e_j}|e_j \in oracle|$ denote the similarity of GUI event e_i and oracle event e_j respectively. The fitness score ranges from 0 to 1, with a higher fitness score indicating that test t closely approximates the functionality of the source test.

The Test Generation process terminates when the fitness score of a reused test reaches its maximum potential, cannot be further improved, or exceeds a specified expected similarity threshold.

3.3. GUI events deduplication

In contrast to the target app, the source app may involve additional steps after initiation to accomplish a task that the target app does not require, as previously explained in Section 2. In such scenarios, reusing these events from the source test could introduce redundant events into the reused test, potentially interfering with the execution of the intended functionality. To address this problem, we carry out GUI events deduplication to simplify the initial test t_{init}.

Algorithm 3 GUI events deduplication.

```
Input: Initial test t_init = {(w^t_m1, a_m1), (w^t_m2, a_m2), ⋯ , (w^t_mn, a_mn)}
Output: Simplified test t_simp

 1: Initialize: t_simp = ∅
 2: repeat_resid = getrepeat_resid(t_init)
 3: repeat_index = getrepeat_index(t_init, repeat_resid)
 4: for i ∈ repeat_index do
 5:     t_simp = deduplicate(t_init, i, repeat_index)
 6: end for
 7: exception = execute(t_simp)
 8: if exception then
 9:     t_simp = t_init
10: end if
11: return t_simp
```

Algorithm 3 provides pseudo-code for deduplicating GUI events in the initial test t_{init}. Considering that the attribute ‘resource-id’ of the widget is typically unique, it is efficient to identify the widget based on this attribute. First, we retrieve the ‘resource-id’ list $repeat_{resid}$ for the duplicated widgets in the initial test t_{init} through function `getrepeat_resid` (Line 2). Then, according to the $repeat_{resid}$ of the duplicated widgets, we obtain the list of indices for the duplicated events in the initial test t_{init} through function

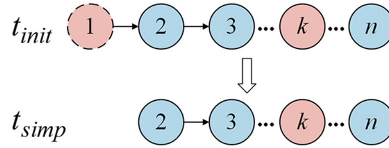


Fig. 7. The example of single event duplication appeared in the initial test t_{init}. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

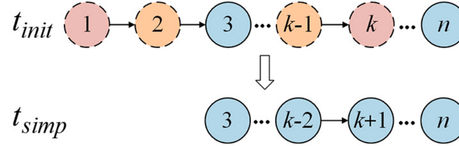


Fig. 8. The example of cyclic events duplication appeared in the target test t_{init}.

`getrepeat_index` (Line 3). To decide which duplicated event should be eliminated in cases of multiple occurrences, we design function `deduplicate`, which handles two potential situations.

One case involves only a single event duplication. For example, Fig. 7 illustrates an instance of the single event duplication that appeared in the initial test t_{init}, where each circle represents an event. Duplicated events are indicated by red circles in the figure. Using functions $getrepeat_resid$ and $getrepeat_index$, we determine that the repeated event indexes are 1 and k. Since the purpose of deduplication is to remove redundant events from the beginning of the initial test t_{init}, we eventually get a simplified events sequence t_{simp} by removing the first event.

A more complex situation involves deduplicating cyclic events. In such cases, when two or more redundant events are initially matched, the reverse event of these redundancies may be generated later to ensure proper functionality execution. This situation is exemplified in the initial test t_{init} depicted in Fig. 8. Following the steps to retrieve the index of repeated events, we can obtain the index lists of repeated events [1, k] and [2, k-1]. Since these duplicated events do not contribute to functional execution, removing them becomes necessary to save additional effort. After deduplication, a simplified test $t_{simp} = \{(w^t_{m_3}, a_{m_3}), \cdots, (w^t_{m_{k-2}}, a_{m_{k-2}}), (w^t_{m_{k+1}}, a_{m_{k+1}}), \cdots, (w^t_{m_n}, a_{m_n})\}$ is obtained.

To ensure that the functionality executes smoothly after removing duplicated events, we verify this by executing the simplified test t_{simp}. The deduplication operation will be abandoned if the application encounters an exception (Lines 8-10).

3.4. Adaptive semantic matching

The Adaptive Semantic Matching strategy aims to improve event matching, making it more aligned with the functionality of the source tests for unsuccessfully reused tests. As discussed in Section 1, the semantic problem of similarity calculation has a significant impact on the performance of test reuse. The test generation described in Section 3.2 relies on a greedy approach, selecting widgets with the highest similarity for matching, which can result in inappropriate matches. Therefore, we aim to make up for this deficiency by extending the search scope to optimize those tests that were not successfully reused. The specific workflow of adaptive semantic matching is described in **Algorithm 4**. Adaptive semantic matching consists of two stages: internal adaptation (Lines 2-12) and external adaptation (Lines 14-26). These stages differ in their strategy for searching the index of potentially incorrect matching events, while they both share the same steps for generating tests based on the retrieved indexes. In terms of index retrieval, internal adaptation focuses on retrieving event indexes from the generated tests that exhibit higher similarity to the source event than the current match. Conversely, external adaptation concentrates on retrieving indexes of the matched events with lower similarity. We will now provide detailed explanations of the two steps: adaptive index retrieval and adaptive test generation.

3.4.1. Adaptive index retrieval

Based on the observations of the event matching rules during the process of test reuse, we designed the adaptive matching in both internal and external adaptation. The main challenge lies in identifying events that might have been incorrectly matched. We address this limitation through the following two aspects:

Internal adaptation. Drawing inspiration from the implicit connection between events, we build the similarity matrix between simplified test t_{simp} and augmented source test t'_s (Line 2). The events within a test are sequential, which means the events that have already been matched can influence the selection of subsequent events to match. Considering this, we employ the parameter $adjrange$ to control the calculation range. That is to say, for each event in t_{simp}, we compare the similarity between it and the next $adjrange$ events of its corresponding matched source event. If other source events exhibit higher similarity to the current event, the index of the current event is added to adj_index1 (Line 3). For example, the widgets for simplified test t_{simp} and augmented source test t'_s are $w^t_{m_1}, w^t_{m_2}, w^t_{m_3}, w^t_{m_4}$ and $w^{s'}_1, w^{s'}_2, w^{s'}_3, w^{s'}_4$, respectively. The similarity matrix is calculated as follows:

Algorithm 4 Adaptive semantic matching.

Input: Simplified test t_{simp}, Augmented source test t'_s
Output: Target test t_{target}

1: Initialize: $t_{target} = \emptyset$, $adjrange = \text{int}\ (\frac{len(t'_s)+1}{2})$
2: $adj_matrix = \text{getsimilarity}(t_{simp}, t'_s, adjrange)$
3: $adj_index1 = \text{getindex1}(adj_matrix)$
4: **if** $adj_index1 \neq []$ **then**
5:     **for** $index \in adj_index1$ **do**
6:         $t_{proce} = \text{inter_testgen}(t_{simp}, index)$
7:         **if** $fitness(t_{proce}) > fitness(t_{simp})$ **then**
8:             $t_{simp} = t_{proce}$
9:             **break**
10:         **end if**
11:     **end for**
12: **end if**
13: **if** $cond_{fail}(t_{simp})$ **then**
14:     $adj_index2 = \text{getindex2}(t_{simp})$
15:     **for** $index \in adj_index2$ **do**
16:         $t_{proce} = \text{exter_testgen}(t_{simp}, index)$
17:         **if** $fitness(t_{proce}) > fitness(t_{simp})$ **then**
18:             **if** $cond_{fail}(t_{proce})$ **then**
19:                 $t_{simp} = t_{proce}$
20:                 $adj_index2 = \text{getindex2}(t_{simp})$
21:             **else**
22:                 $t_{target} = t_{proce}$
23:                 **break**
24:             **end if**
25:         **end if**
26:     **end for**
27: **else**
28:     $t_{target} = t_{simp}$
29: **end if**
30: **return** t_{target}

$$
\begin{array}{c} \\ w^t_{m_1} \\ w^t_{m_2} \\ w^t_{m_3} \\ w^t_{m_4} \end{array}
\begin{array}{c}
\begin{array}{cccc} w^{s'}_1 & w^{s'}_2 & w^{s'}_3 & w^{s'}_4 \end{array} \\
\begin{bmatrix} 0.25 & 0.19 & 0.15 & 0 \\ 0 & 0.22 & 0.14 & \mathbf{0.25} \\ 0 & 0 & 0.13 & 0.11 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\end{array}
$$

In this example, the similarity between $w^t_{m_2}$ and $w^{s'}_4$ is higher than that of the currently matched source widget. This suggests that $w^t_{m_2}$ might be a more suitable match for $w^{s'}_4$ than $w^{s'}_2$. Consequently, we include the index 2 of the simplified widget $w^t_{m_2}$ into adj_index1, which indicates that other widgets similar to the source widget $w^{s'}_2$ will be reconsidered for matching in a subsequent iteration.

External adaptation. The adopted similarity calculation method may not perform optimally for some widgets, and the events in the simplified test t_{simp} may match the source events with relatively low similarity. It is essential to determine whether a better event match exists for the simplified test, particularly in cases where test reuse fails. Specifically, the failure condition of test reuse $cond_{fail}$ is defined as the last oracle in the simplified test being empty or the displayed text being inconsistent with the source oracle (Line 13). For the simplified test t_{simp} that meets the condition $cond_{fail}$, we first obtain a list of the similarity scores of the matched events and retrieve the required indexes to form the list adj_index2 (Line 14). To select the indexes of events that may result in better matches, we design the rules for the following two cases.

First, we assess whether there are multiple low scores in the similarity score. For example, in the following given similarity score, the similarity falls below 0.05 from widget $w^t_{m_3}$. This situation raises suspicion about the possibility of incorrect matches between widgets $w^t_{m_3}$ and $w^t_{m_2}$, with a significant impact on the matching of subsequent widgets. In such cases, we add the index of the first widget with low similarity, along with its predecessor, to the index list adj_index2. Thus, $adj_index2 = [3, 2]$.

$$
\begin{array}{cccccc} w^t_{m_1} & w^t_{m_2} & w^t_{m_3} & w^t_{m_4} & w^t_{m_5} & w^t_{m_6} \\ [0.16 & \mathbf{0.11} & \mathbf{0.04} & 0.03 & 0.03 & 0.00] \end{array}
$$

Secondly, if the condition described above is not met, we consider that the adj_index2 corresponds directly to the indexes of the widgets with the lowest similarity scores. It is assumed that the similarity score is determined as follows:

$$
\begin{array}{cccccc} w^t_{m_1} & w^t_{m_2} & w^t_{m_3} & w^t_{m_4} & w^t_{m_5} & w^t_{m_6} \\ [0.23 & 0.27 & \mathbf{0.11} & 0.22 & \mathbf{0.08} & 0.00] \end{array}
$$

Assuming we adjust the adaptive event threshold to 2, following the above rules, we add the indexes of widgets $w^t_{m_3}$ and $w^t_{m_5}$ to adj_index2. In this case, $adj_index2 = [5, 3]$, based on the similarity scores sorted in ascending order.

3.4.2. Adaptive test generation

Once the adaptive index for the widgets is obtained, we describe how internal and external adaptation search for tests with functionality closer to the source test. Note that when the adaptive range is determined, we only need to start exploring from the obtained index. Firstly, we follow a process similar to Section 3.2 for iterating over each index and rematching. To improve efficiency, we also filter out the unmatched widgets that were previously validated in Section 3.2 from the candidate widgets associated with the current index. Based the above operations, we validate the reachability of candidate widgets and assign corresponding actions to the reachable widget following the steps (Lines 5-12) of **Algorithm 1**. We iterate over matching events until all the events in the source test have been traversed. After the above steps, we get the processed test t_{proce} obtained from the new exploration (Lines 5-6, 15-16).

Next, it is critical to determine which generated tests can be considered. As previously discussed, we utilize the fitness function to guide the search towards a direction more similar to the source test. We first perform internal adaptation (Lines 2-12). For this stage, we updated the simplified test t_{simp} to the one with a high fitness score (Lines 7-10). Our primary focus then shifts to external adaptation (Lines 14-26). If the simplified test t_{simp} does not satisfy the condition $cond_{fail}$, the obtained test is the final target test t_{target}, which indicates a successful test reuse (Line 28). However, if it does meet this condition, we perform external adaptation to search for more semantically similar event matches. Leveraging the fitness function to guide exploration, a test with the highest similarity is identified as the target test (Lines 17-25). To ensure the correctness of the in-depth exploration direction, we update the index list for the newly explored test (Lines 18-20).

4. Evaluation

To demonstrate the performance of our proposed framework TREADROID, we evaluated TREADROID by considering the following four research questions:

- **RQ1:** How effective is TREADROID in terms of the precision and recall for widget mapping?
- **RQ2:** Is TREADROID more effective than baseline methods for widget mapping?
- **RQ3:** How do the semantic matching method and the adaptive strategy contribute to effective widget mapping?
- **RQ4:** Is TREADROID more efficient than baseline methods?
- **RQ5:** How much effort can be saved by TREADROID to generate tests?

In this paper, we propose a novel framework TREADROID to improve the usability of the reused tests. RQ1 assesses the performance of TREADROID for reusing tests across applications in various categories. In RQ2, we perform comparisons in terms of widget mapping effectiveness to demonstrate that our TREADROID outperforms the state-of-the-art baseline methods. In RQ3, we investigate the contributions of the improved semantic matching method (Section 3.2) and adaptive strategy (Section 3.4) in enhancing test reuse. Moreover, RQ4 indicates the efficiency of our TREADROID in comparison to baseline methods. Furthermore, RQ5 focuses on quantifying the manual effort saved when using TREADROID to reuse tests for each test scenario.

4.1. Experimental setup

Implementation Details. TREADROID requires a source app, an existing test associated with the source app, and a similar application of the same category as the source app, that is, the target app, as input. Its purpose is to reuse the existing test designed for the source app to test another similar target app. Our experiment was conducted on an Ubuntu desktop, with 3.4 GHz Intel Core i7 CPU and 32 GB RAM. We used a Nexus 5X emulator running Android 6.0 (API 23) to install the applications. The GUI states of the applications were recorded using Appium.[3]

As discussed in the context of test generation in section 3.2, TREADROID has adjustable parameters in three aspects: the weights w_1 and w_2 associated with GUI and oracle events in fitness score, the thresholds $threshold_1$ and $threshold_2$ for determining search termination, and the weight of each attribute among all available attributes. According to [20], the GUI events responsible for triggering actions and the oracles evaluating the achieved GUI states have equal importance in the generated test. Consequently, w_1 and w_2 are both configured to 0.5 to maintain balance. Aligned with [20], TREADROID leverages two different criteria to stop its exploration during the search process. The first condition for search termination is when the fitness scores of two tests generated by TREADROID are identical, defined by setting $threshold_1$ as 0.001. The second condition, denoted by $threshold_2$, was arrived at through multiple experimental runs on randomly selected pairs of similar applications. Ultimately, it was determined to be 0.12, the optimal balance between ensuring the presence of correctly matched and mismatched event pairs in the test.

Finally, concerning the relative importance weights of each attribute among all available attributes, we empirically determined the best performing values. Specifically, for the attributes 'resource-id', 'text', 'content-desc', 'parent_text', and 'sibling_text', the weights were set to 0.5, 1, 1, 1, and 0.5, respectively. In particular, the weights of these attributes were adjusted to 1, 0.5, 1, 1, and 1.5 in the case of tip applications, where the text of editable widgets plays little role in the similarity calculation. Our results reported in the next section were obtained based on these parameter settings.

[3] https://github.com/appium/appium.

Table 1
The specific details about subject applications.

Category	Application	Version	Source
Browser	a11-Lightning	V4.5.1	F-Droid
	a12-Browser for Android	V6.0	Google Play
	a13-Privacy Browser	V2.10	F-Droid
	a14-FOSS Browser	V5.8	F-Droid
	a15-Firefox Focus	V6.0	Google Play
To Do List	a21-Minimal	V1.2	F-Droid
	a22-Clear List	V1.5.6	
	a23-To-Do List	V2.1	
	a24-Simply Do	V0.9.1	
	a25-Shopping List	V0.10.1	
Shopping	a31-Geek	V2.3.7	F-Droid
	a32-Wish	V4.22.6	
	a33-Rainbow Shops	V1.2.9	
	a34-Etsy	V5.6.0	
	a35-Yelp	V10.21.1	
Mail Client	a41-K-9	V5.403	Google Play
	a42-Email mail box fast mail	V1.12.20	
	a43-Mail.Ru	V7.5.0	
	a44-myMail	V7.5.0	
	a45-Email App for Any Mail	V6.6.0	
Tip Calculator	a51-Tip Calculator	V1.1	Google Play
	a52-Tip Calculator	V1.11	
	a53-Simple Tip Calculator	V1.2	
	a54-Tip Calculator Plus	V2.0	
	a55-Free Tip Calculator	V1.0.0.9	

Subject Applications. We conduct test reuse on five categories of applications to perform the experimental evaluation. These selected subject categories of the applications are also commonly used by users in their daily life and professional activities. The categories consist of browsers, to-do list, shopping, mail client, and tip calculator. Each of these categories contains five applications, all of which share similar functionalities. The applications were sourced from Google Play [27] and F-droid [28], which are popular platforms frequently used by researchers [5,18,19,29–33] for studying GUI testing or the functionality of applications. Specific details about these subject applications are listed in Table 1.

Existing tests. Our proposed framework TREADROID is inspired by the idea of state-of-the-art method CRAFTDROID [20] to enhance test reuse. To ensure a fair comparison, we choose method CRAFTDROID as the baseline and reuse its test suites for comparative evaluation. These test suites consist of tests designed to verify two main functionalities for each application. That is, we have collected two tests that verify similar functionalities for the five applications within each category. The specifics of the collected tests for each application, including the average number of GUI and oracle events and the total test steps, are presented in Table 2.

On such a basis, it is convenient to achieve test reuse from one application to another by employing the TREADROID. In the experimental evaluation, we perform a total of $4(tests) * 5(target_apps) * 2(functionalities) = 40$ test reuses for each category of applications. Thus, the number of 200 is attempted for test reuse across similar applications.

Baselines. To demonstrate the competitiveness of TREADROID, we select three state-of-the-art baselines for comparison. CRAFTDROID [20] and ATM [21] are widely recognized for test reuse, while TRASM [24] represents the conference version before the expansion of TREADROID. In common, they have the capability to migrate GUI tests, including oracles, for mobile applications with similar functionality. All of these baselines utilize the Word2Vec [26] word embedding model for semantic matching and analyze the Window Transition Graph to identify unmatched widgets across different application pages. In particular, ATM triggers only the first actionable element within the shortest path that exists; otherwise, it semi-randomly selects one of the actionable elements for matching that has not been triggered previously. CRAFTDROID includes the validation of reachable paths and supports supplementary intermediate events for event matching. TRASM follows the test generation step similar to that of CRAFTDROID, but additionally integrates an adaptive strategy to enhance the performance of semantic matching.

Performance Metrics. To ensure a fair evaluation of our framework, we adopt four widely used metrics in test reuse.

- **Precision** is the number of correct target events generated. Formally,

$$precision = \frac{tp}{tp + fp}$$

  where tp (true positive) indicates that the event of the target test generated is correct, fp (false positive) means an incorrect event is generated.
- **Recall** represents the number of source events that are correctly reused. Formally,

$$recall = \frac{tp}{tp + fn}$$

Table 2
The collected tests for each application.

Category	Functionality	Avg GUIs	Avg Oracles	Test Steps
Browser	b11-Access website by URL	3.4	1	1. Locate the search bar 2. Input URL and press Enter 3. Specific content should appear
	b12-Back button	7.4	3	1. Locate the search bar 2. Input URL1 and press Enter 3. Specific content about URL1 should appear 4. Input URL2 and press Enter 5. Specific content about URL2 should appear 6. Click the back button 7. Specific content about URL1 should appear
To Do List	b21-Add task	4	1	1. Click the add task button 2. Fill the task title 3. Click the add button 4. The task title should appear in the task list
	b22-Remove task	6.8	2	1. Add a new task 2. Action the remove button 3. Click the confirm button if exists 4. The task should not appear in the task list
Shopping	b31-Registration	14.2	5	1. Click the register button 2. Fill out necessary data 3. Click the submit button 4. Personal data should appear in the profile page
	b32-Login with valid credentials	9	4	1. Click the login button 2. Fill out valid credentials 3. Click the submit button 4. Personal data should appear in the profile page
Mail Client	b41-Search email by keywords	5	3	1. Start the inbox activity 2. Click the search button 3. Input keywords and press Enter 4. Specific email should appear
	b42-Send email with valid data	8	3	1. Start the inbox activity 2. Click compose button 3. Input unique subject ID and valid email address 4. Click send button 5. The unique ID should appear in the spent
Tip Calculator	b51-Calculate total bill with tip	3.8	1	1. Start the tip calculation activity 2. Input bill amount and tip percentage 3. Total amount of bill should appear
	b52-Split bill	4.8	1	1. Start the tip calculation activity 2. Input bill amount and tip percentage 3. Input number of people 4. Total amount of bill per person should appear

where fn (false negative) indicates no suitable matching event is found.

- **Reduction** measures how much effort developers can save by adopting TREADROID to generate tests instead of writing them from scratch. Formally,

$$Reduction(t_n) = 1 - \frac{ED(t_n, t_g)}{|t_g|}$$

where $ED(t_n, t_g)$ represents the Levenshtein distance [34] of the generated test t_n and its ground truth t_g, measuring how close the generated test is to the ground truth test. In our case, the Levenshtein distance calculates the minimum number of edits required to transform the transitions of the generated test to those of the ground truth. Each edit operation corresponds to an action such as inserting, deleting, or replacing transitions.
- **Avg. Reuse Time** indicates the time it takes for TREADROID to reuse a complete source test on average. It is measured in seconds. The shorter the average reuse time, the higher the efficiency.

Furthermore, the correctness of the events in the target test is evaluated by comparing with the events in the collected existing tests of the corresponding application that serve the same functionality. To facilitate the evaluation process, we employ manually constructed tests from the baseline method CRAFTDROID [20], which are considered as the ground truth for the reused tests. Moreover, we conduct manual inspections to minimize errors during the proofreading process. Our experimental results are presented under the guarantee of the above procedure.

Table 3
Effectiveness evaluation of TREADROID.

Functionality	GUI Event		Oracle Event	
	Precision	Recall	Precision	Recall
b11	100%	100%	100%	100%
b12	100%	100%	100%	100%
b21	87%	100%	85%	100%
b22	80%	98.6%	88.88%	91.4%
b31	40.3%	93.05%	24.67%	65.52%
b32	54.12%	82.14%	54.09%	70.21%
b41	100%	100%	100%	100%
b42	94.8%	97.74%	93.1%	100%
b51	100%	100%	100%	100%
b52	90.5%	98%	100%	80%
Total	84.6%	96.9%	84.5%	90.7%

4.2. Experimental results

RQ1: How effective is TREADROID in terms of the precision and recall for widget mapping?

For RQ1, we aim to evaluate the effectiveness of our proposed novel framework TREADROID for reusing tests across Android applications. As described in Section 3, for the 25 commonly used applications belonging to five categories, we reuse tests from one application on four other similar applications within the same category. To ensure the correctness of the experimental results, we recorded the experimental results in Table 3 following manual inspection.

The effectiveness of test reuse for the fundamental functionalities outlined in Table 2 is quantified through the precision and recall metrics of matched GUI and oracle events. Overall, among the results of 200 tests reused, the precision and recall of GUI events are 84.6% and 96.9% respectively, and those of oracle events are 84.5% and 90.7% respectively. These findings prove that the effectiveness of our TREADROID for test reuse on these five categories of applications can be recognized.

The results presented in Table 3 reveal a noteworthy observation that there are significant variations in the performance achieved by test reuse for different categories of applications. Notably, TREADROID exhibits its best performance in both Browser and Mail categories. This is due to the fact that the achievement process of these functionalities, including b11, b12, b41, and b42, incorporates distinct semantic features that facilitate widget matching. In these categories, the reuse of all existing tests related to the functionalities b11, b12, b41, and b51 demonstrates near-perfect success, with both GUI events and oracles achieving close to 100% precision and recall. In contrast, the precision and recall of event matching are as low as 50% on the functionality b31, which belongs to the shopping category. Nevertheless, considering that TREADROID still achieves over 84% and 90% in precision and recall for both GUI and oracle events mapping, respectively, the feasibility of test reuse within the functionalities is encouraging.

In addition, we investigate the factors contributing to the noticeable differences in experimental results and attribute them to the following three key aspects:

- **UI design differences between source and target app.** For example, for the functionality ‘b31-Registration’, application *Etsy* necessitates a password confirmation step after entering the password, a feature not present in the source test. Since the “Confirm Password” test step cannot be matched, reusing the existing test for subsequent registration steps becomes challenging for the TREADROID.
- **Non-optimal event matching.** The word-level similarity calculation method and limited semantic information lead to the incorrect matching of widgets. For example, the text attributes for the two widgets that enter the different functionalities are “sign in” and “sign up”, respectively. However, our method weights the highest similarity between words on average, which results in incorrect matching. A promising solution lies in considering more comprehensive and coarse-grained similarity computation.
- **Length of existing source test.** The existing test with complex steps increases the difficulty of test reuse. Compared the functionality ‘b51-Calculate total bill with tip’ with the functionality ‘b52-Split bill’, although the latter has only one additional test step ‘Input number of people’, the recall of the oracle drops to 80%.

RQ2: Is TREADROID more effective than baseline methods for widget mapping?

After confirming the effectiveness of TREADROID in reusing tests across Android applications, we conduct comparative experiments to assess its performance relative to baseline methods. We employed the original dataset from CRAFTDROID for our experimental evaluation. To avoid any errors during repeated experiments, we directly compare our experimental results with those presented in the original literature [20].

Table 4 provides a comparison of widget mapping results between TREADROID and the baseline methods in terms of each functionality. The results indicate that TREADROID have overall outperformed the baselines, which indicates that our TREADROID has higher flexibility for test reuse than the baseline methods. In summary, the higher flexibility of the TREADROID can be supported from the following aspects. Compared with the baseline, TREADROID achieves the highest improvements of 31.25%, 18.75% in precision for GUI and oracle events, respectively, along with recall improvements of 17.74% and 26.03%. On average, the precision

Table 4
Comparison of test reuse between CRAFTDROID, ATM, TRASM and TREADROID.

Functionality	Approach	GUI Event		Oracle Event	
		Precision	Recall	Precision	Recall
b11	CRAFTDROID	79%	100%	100%	100%
	ATM	68.75%	100%	90%	100%
	TRASM	100%	100%	100%	100%
	TREADROID	**100**%	100%	**100**%	100%
b12	CRAFTDROID	85%	100%	100%	100%
	ATM	76.47%	100%	93.33%	100%
	TRASM	100%	100%	100%	100%
	TREADROID	**100**%	100%	**100**%	100%
b21	CRAFTDROID	78%	100%	85%	100%
	ATM	76.78%	100%	75%	100%
	TRASM	87%	100%	85%	100%
	TREADROID	**87**%	100%	**85**%	100%
b22	CRAFTDROID	69%	**100**%	85%	80%
	ATM	70.24%	98.33%	77.77%	70%
	TRASM	77.01%	97.52%	**89.65**%	76.47%
	TREADROID	**80**%	98.6%	88.88%	**91.4**%
b31	CRAFTDROID	**44**%	90%	**34**%	**67**%
	ATM	39.08%	**97.14**%	21.25%	62.96%
	TRASM	40.11%	94.3%	25.33%	61.29%
	TREADROID	40.3%	93.05%	24.67%	65.52%
b32	CRAFTDROID	53%	82%	**56**%	61%
	ATM	48.27%	80.76%	37.25%	44.18%
	TRASM	52.43%	76.78%	52.54%	65.95%
	TREADROID	**54.12**%	**82.14**%	54.09%	**70.21**%
b41	CRAFTDROID	100%	100%	100%	100%
	ATM	100%	100%	100%	100%
	TRASM	100%	100%	100%	100%
	TREADROID	100%	100%	100%	100%
b42	CRAFTDROID	85%	80%	89%	89%
	ATM	81.34%	94.78%	81.48%	89.79%
	TRASM	90.29%	95.27%	85.71%	97.95%
	TREADROID	**94.8**%	**97.74**%	**93.1**%	**100**%
b51	CRAFTDROID	82%	100%	100%	80%
	ATM	90%	100%	83.33%	88.23%
	TRASM	93%	100%	100%	90%
	TREADROID	**100**%	100%	**100**%	**100**%
b52	CRAFTDROID	80%	**100%**	100%	65%
	ATM	85%	100%	81.25%	76.47%
	TRASM	86.2%	96.15%	100%	75%
	TREADROID	**90.5%**	98%	**100%**	**80%**

and recall of matched GUI events show an improvement of 7.44% and 0.85%, while those of matched oracle events increased by 3.64% and 6.03%. Therefore, TREADROID demonstrates superior overall performance in terms of precision and recall for widget matching compared to the baseline methods.

However, as observed in Table 4, our TREADROID does not achieve the highest widget matching performance in all functionalities compared to the baselines. For functionalities b22 and b52, the recall of GUI events slightly decreases on average by nearly 0.01% and 0.7% respectively compared with the baselines. This can be attributed to the adaptive semantic matching strategy employed in our method tends to result in the rematching of a few events incorrectly, increasing the number of false negative events. In addition, our method exhibits lower oracle precision compared to its conference version TRASM for functionality b22. This indicates that TREADROID introduce some interference as it broadens the exploration scope with the adaptive matching strategy. Ultimately, the 14.93% increase in recall reflects TREADROID's ability to generate tests effectively validating the functionality 'b22-Remove task'.

Moreover, for the functionality 'b31-Registration', our results are generally lower than those of the baselines. And CRAFTDROID achieves nearly the best performance for widget matching. After a thorough analysis, this phenomenon can be attributed to the presence of several incomplete tests generated in the CRAFTDROID, with a total number of events lower than the actual number [35]. This results in high precision and recall. Moreover, this is a key factor that reduces the precision of the oracle in our method. It is worth noting that ATM demonstrates a notable GUI recall, achieved by incorporating path and semi-random exploration to enhance the possibility of correct widget matching. Nevertheless, the considerable improvement in the recall of oracle events for our TREADROID indicates its superior performance in test reuse when compared to the baselines.

RQ3: How do the semantic matching method and the adaptive strategy contribute to effective widget mapping?

Table 5
The results of the TREADROID under different settings.

ISM	GED	AS	GUI Event		Oracle Event	
			Precision	Recall	Precision	Recall
-	-	-	75.5%	95.2%	84.9%	84.2%
✓	-	-	79.31%	97.12%	83.71%	85.86%
✓	✓	-	81.88%	97.12%	83.71%	85.86%
✓	-	✓	81.94%	96.95%	84.57%	90.71%
✓	✓	✓	84.67%	96.95%	84.57%	90.71%

RQ2 illustrates that our TREADROID increases the flexibility of the generated tests. As mentioned in Section 1, TREADROID first improves the semantic matching method based on the baseline CRAFTDROID to generate an initial test. It then combines an adaptive strategy to optimize the obtained test after deduplication. We aim to enhance widget matching performance for test reuse through an improved semantic matching method and adaptive strategy. To clarify the contributions of these two components to test reuse, we present the results of the proposed TREADROID under the following settings:

Only with ISM. Based on the baseline CRAFTDROID, we solely employ the test generation phase to generate the test. We integrate cross-attributes and same-attributes calculations to assess the similarity for matching and adopt a mutation strategy for unique action widgets to allocate appropriate action. This strategy allows us to investigate the impact of improved test generation, namely the improved semantic matching (ISM), on the matching performance of widgets.

Without AS. This strategy means we generate tests by carrying out test generation and GUI events deduplication (GED), without further considering test optimization.

Without GED. This strategy means we generate tests by performing the test generation and the adaptive strategy (AS), without simplifying the reused test. This variant helps us understand the influence of the combined improvements in semantic matching and adaptive strategy on the performance of TREADROID.

With ISM, GED and AS. This combination represents our comprehensive framework TREADROID. We apply the adaptive strategy (AS) to further optimize the test obtained from the previous strategy. Through this, we mainly discuss the roles of GUI events deduplication and adaptive strategy in test reuse, respectively.

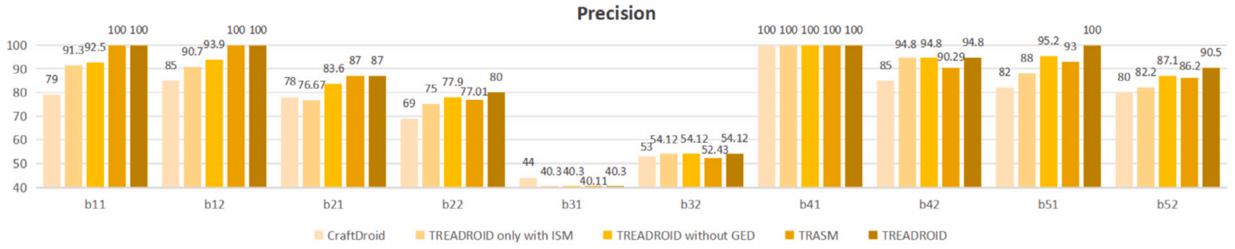
Table 5 compares overall the effectiveness of the TREADROID with four key settings and the baseline method CRAFTDROID. At the overall level, in terms of precision and recall of GUI and oracle events, TREADROID has improved by 9.17%, 1.75%, -0.33%, and 6.51%, respectively. These improvements were discussed in RQ2. The improvement of the second row of results over the first row, and the improvement of fourth row over the second row represent the contributions of the improved semantic matching method, and the adaptive strategy, respectively. Specifically, the consideration of the improved semantic matching method results in increases of 3.81%, 1.92%, -1.19%, and 1.66%, while the adaptive strategy contributes an increase of 2.63%, -0.17%, 0.86%, and 4.85%, respectively. This indicates that both of these two semantic-based enhancing strategies indeed improve the performance of event matching. Moreover, the findings between the fourth and fifth rows suggest that GUI events deduplication primarily improves the precision of GUI events. This is because the redundancy of GUI events in the test increases the probability of false positive, leading to a decrease in the precision of GUI events. According to the results of the last column, GUI events deduplication does not contribute to a closer approximation of the functional implementation.

Specifically, Fig. 9 depicts the precision and recall of GUI and oracle events for each functionality. To show the effects of adaptive strategy and semantic matching method more significantly, we compare the results of CRAFTDROID, TREADROID only with ISM, TREADROID without GED, TRASM, and TREADROID. Oracle events used to check the correctness of intended functionality are the primary focus. For most functionalities, there is a varying degrees improvement in both the precision and recall of oracle events. However, it should be noted that for functionalities b11, b12, and b21, although the precision or recall of the oracle shows a noticeable downward trend only when the semantic matching improves, the precision of the GUI events is increased or reduced slightly. This observation suggests that combining cross-attributes to comprehensively represent widget textual similarity may introduce some interference in certain cases. When focusing solely on the improved semantic mapping, the recall of oracle achieved by TREADROID is comparable to TRASM for 7 out of 10 functionalities. This phenomenon confirms that the semantic matching adopted by TREADROID improves the performance of widget mapping between similar applications. In addition, the data change from the second column to the third column in Fig. 9 characterizes the performance the impact of adaptive semantic matching during optimization. With the addition of the adaptive strategy, the matching of events is significantly improved, while the recall of oracle increases. This results in a closer functional approximation to the source test. Moreover, in general, the recall of oracle achieved by TRASM is lower than that of TREADROID without GED, which incorporates semantic matching and adaptive strategy. This explains that compared to TRASM, the semantic matching rule and improved adaptive strategy adopted by TREADROID have substantially enhanced the usability of the generated tests.

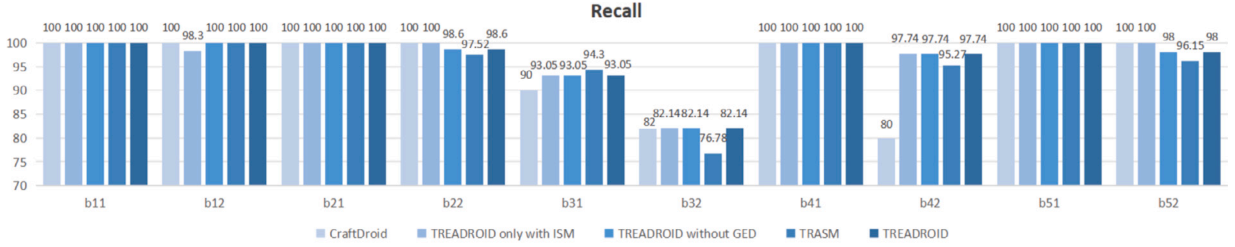
RQ4: Is TREADROID more efficient than baseline methods?

Efficiency is also a crucial measure of test reuse. In RQ4, our purpose is to investigate the time cost of our TREADROID compared with the baseline methods. Fig. 10 presents the average time required for test reuse by CRAFTDROID, ATM, TRASM, and TREADROID for each functionality.

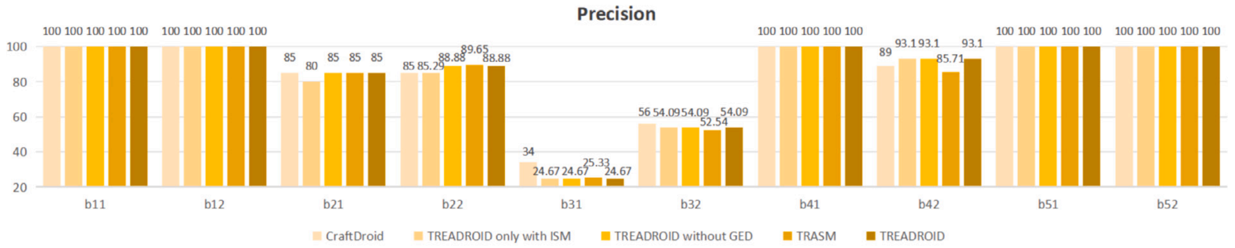
Remarkably, TREADROID takes significantly less time for test reuse on average than the baseline methods. TREADROID achieved a minimum time cost of 43%, 37.1%, 53.9%, 36.4%, 48.9%, 46.9%, 38.9%, 62.7%, 25.9%, and 33.1% of the baseline methods for



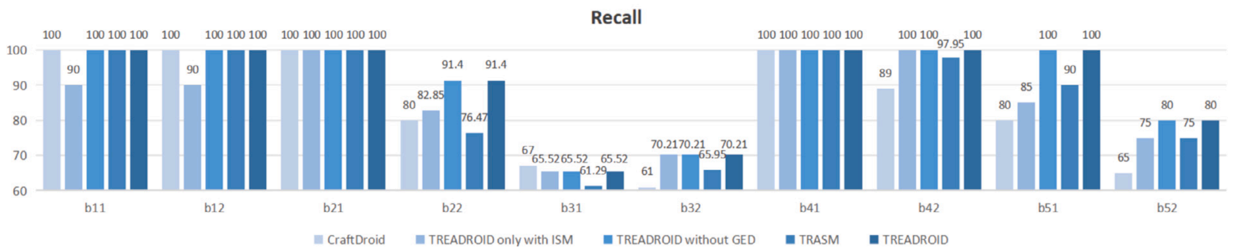
(a) Comparison of average precision for GUI events



(b) Comparison of average recall for GUI events



(c) Comparison of average precision for Oracle events



(d) Comparison of average recall for Oracle events

Fig. 9. Comparison of results for each functionality.

each functionality. These results highlight the efficiency gains developers can gain when using TREADROID, enabling them to potentially uncover more vulnerabilities in less time.

Several factors contribute to these efficiency improvements in TREADROID. We conclude that the following two aspects can explain these results. Firstly, we store the explored paths and stepping widgets when checking the reachability of widgets in Section 3.2.2, which significantly reduces the time required for repeated path exploration. Although it is mentioned in [20] that parallel execution on multiple devices or simulators can greatly reduce the time, TREADROID can achieve higher efficiency under the same settings. Additionally, the strategy employed by the TREADROID enhances the accuracy of widget matching, which avoids exploring more space. Moreover, compared with TRASM, TREADROID takes less time on average to reuse a test. This phenomenon explains

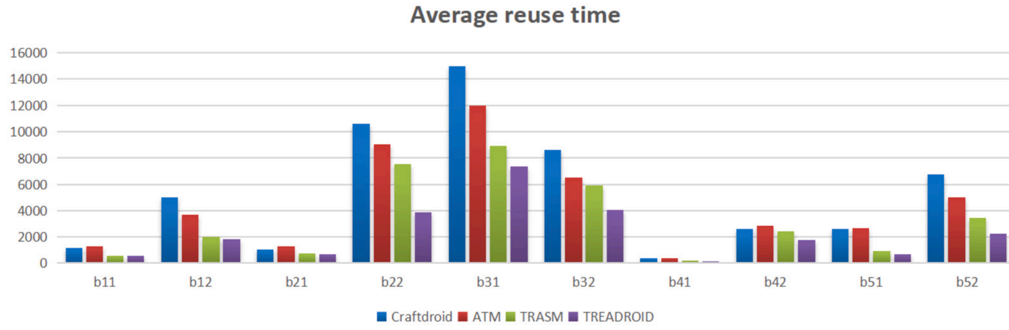
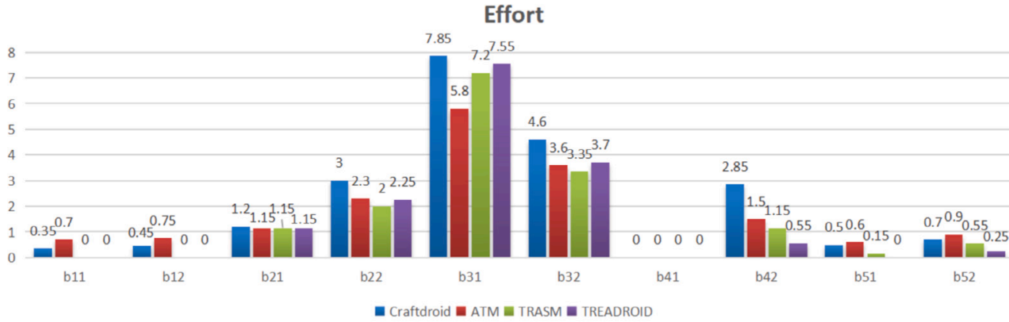
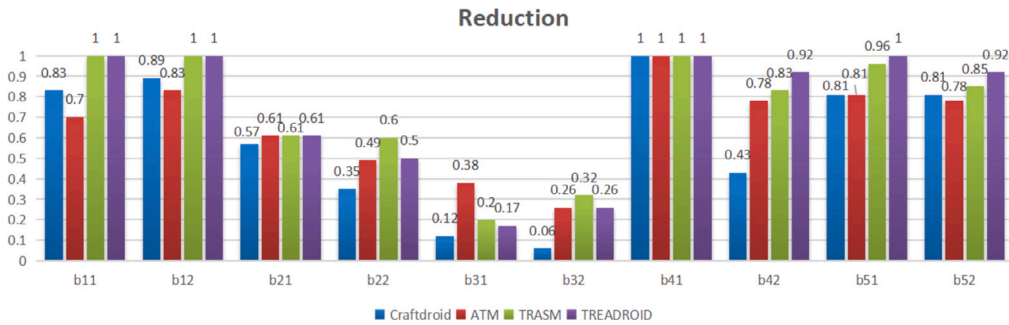


Fig. 10. The average reuse time of CRAFTDROID, ATM, TRASM, and TREADROID.



(a) Comparison of average effort



(b) Comparison of average reduction

Fig. 11. Comparison of average effort and reduction for each functionality.

that the improved adaptive semantic matching in TREADROID can effectively save the time consumption of exploring the additional space.

RQ5: How much effort can be saved by TREADROID to generate tests?

The reused test is considered successful in reducing manual effort for developers if it is able to trigger the intended execution of a specific functionality. To answer this research question, we conducted the assessment by converting all reused tests generated by TREADROID into ground truth, and then compared them against the tests generated by the baseline methods.

Fig. 11 indicates the average edit distance (denoted as effort) required to convert the reused test for each functionality into its ground truth test. It also presents the reduction in manual effort achieved by employing the tests generated through these methods. The results show that CRAFTDROID, ATM, TRASM, and TREADROID can save an average of 58.7%, 66.4%, 73.7%, and 73.8% in manual effort overall compared to writing tests from scratch. This suggests that our approach is comparable to TRASM while offering a 25% and 11% reduction in effort compared to CRAFTDROID and ATM. Taking "b42-Send email with valid data" as an example, TREADROID requires an average of 0.55 manual edits to convert the 20 generated tests into their ground truth tests. This significantly reduces the manual effort required to create tests from scratch by 92%.

Furthermore, we conducted a comparative analysis between our TREADROID and the baseline methods, focusing on the effort reduction in generating tests for each functionality. It is evident that our method consistently outperforms CRAFTDROID in terms of

reducing effort for all functionalities. Furthermore, our TREADROID also shows significant effort reduction compared to ATM, except for functionality b31. In comparison to TRASM, our TREADROID requires slightly more effort for the three functionalities b22, b31, and b32. Upon manual inspection, we attribute this to our introduced adaptive strategy, which led to a few events deviating from the source path as it explored a broader scope. Nonetheless, these generated tests do not adversely affect the successful achievement of the desired functionalities. On the whole, our approach remains a viable option as it offers the advantages of enhanced flexibility without increasing additional manual effort.

5. Threats to validity

Internal threats The main internal threats concern the quality of the source tests and potential errors in the implementation of our framework TREADROID. To ensure a fair comparison, we utilized data consistent with the baseline [20] in our evaluation. In addition, we conducted manual inspections to verify the correctness and quality of the source tests. To minimize the impact of randomness and specificities in the results, for each functionality in each category, we reused the existing test from one application to the remaining four applications respectively. The recorded results are the average of 20 reuse outcomes. To maintain quality, the experimental results, i.e., the generated target tests, are manually verified and further checked against the existing source tests to avoid any potential errors.

External threats The main external threat focuses on the generalization of TREADROID for mobile applications and their test cases. Our framework targets Android applications, which are widely used by the majority of users. The selected popular applications are various categories and sourced from Google Play and F-Droid. Predecessors [5,18,19,29–33] have also commonly utilized these applications for Android testing research. Moreover, the existing tests represent the fundamental functionalities of the application categories they belong to. Since our TREADROID is designed to reduce the high cost of manually designing test cases, it may not be less effective for specific functional test reuse compared to common and basic functionalities. Nevertheless, our evaluation demonstrates the effectiveness of proposed framework in supporting test reuse across Android mobile applications in multiple categories.

6. Related work

Test Generation. Test generation is a key topic in the current field of software testing. It holds significant importance in improving development efficiency and reducing testing costs. GUI testing is widely used to verify the behavior and functionality of GUI applications. We focus discussion on GUI test generation.

Current automated GUI test generators tend to maximize coverage and identify defects by employing either random or structured information to generate tests. One of the widely used fuzzing tools, monkey [10], selects the widgets within the GUI that exhibit erratic behavior to obtain events sequence. Aravind et al. [17] employed a novel random strategy that penalizes frequently chosen widgets when selecting a widget for test generation. These random approaches often result in the generation of unrealistic tests. Alternatively, researchers [36–38] relied on structural information derived from static analysis of source code to extract widgets. While these two methods can effectively identify exceptions, they often lack automatic oracles.

Recently, researchers [39–41] have explored the utilization of usage information from applications to improve the quality of generated tests. Mao et al. [39] aimed to generate replicable test scripts through crowd-based testing, which involved extracting events from various applications. By recording and mining test execution traces, Linares et al. [40] generated execution scenarios combined with the usage state of the application. Our work aims to improve the quality of tests generated by reusing existing tests from one application and adapting them for use on another similar application.

Test Reuse. In recent years, test reuse, an alternative approach to test generation, has attracted the attention of researchers. Hu et al. [18] proposed a machine learning method that can synthesize full tests from the modular ones in a library to test the behavior of the new application within the same category. However, their approach still requires manual intervention for widget recognition. Rau et al. [19] utilized the semantic similarity between UI elements to transfer tests across web applications. Afterwards, Lin [20] and Behrang [21] proposed methods that matched the widgets in existing tests with those in similar applications through the highest semantic similarity, forming event sequences in a greedy manner. These approaches exploit semantic similarity to generate meaningful tests and have the capability to automatically transfer oracles. More prominently, the CraftDroid [20] supports both dynamic and static analysis, which provides a solid foundation for in-depth research on enhancing test reuse.

The most common feature of these methods is that they achieve widget matching across applications to obtain event sequences based on semantic similarity. What brings great convenience but also presents challenges is that the accuracy of test reuse depends on how semantic similarity is calculated. In our prior work [24], we initially employed an adaptive measure to identify incorrectly semantically matched event indexes in the generated tests. However, this strategy has limited scope of application and still does not solve the problem of incorrect event matching. Our work extends to CRAFTDROID [20] and TRASM [24], which seeks feasible solutions to address the existing semantic challenges. The FRUITER framework introduced by Zhao et al. [42] incorporates fidelity and utility metrics to assess the effectiveness of UI test reuse methods on the Android platform. Derived from the idea of FRUITER, our approach recognizes the need to consider the utility of transferring a test, which may require more effort than writing it from scratch.

Existing work mainly focuses on accurately computing semantic similarity to match widgets. For instance, Mao et al. [22] introduced a more accurate event fuzzy matching strategy based on semantics, which overcomes the problem that the attributes of two events may not have a one-to-one correspondence. While their work improves the quality of the reused tests, there is still room for

further improvement. Mariani et al. [23] conducted the first empirical study on the semantic matching of GUI events and prospected in several research directions. They report attributes that better describe the semantics of widget can help reduce meaningless and conflicting information. Recently, Li et al. [43] investigated the challenges of cross semantic understanding in GUI event-based record and replay. They indicated that using attributes to represent the semantic information of widgets and applications can offer compatibility cross different devices and versions. Following this idea but different from existing work, TREADROID works on the association between widget attributes to further improve the similarity calculation method.

A recent work ADAPTDROID [44] utilizes an evolutionary algorithm to improve an initial set of greedily matched tests. Drawing inspiration from ADAPTDROID's test reduction, filtering out irrelevant events in GUI events deduplication increases the precision of reused tests. Undeniably, the iteration of population evolution in ADAPTDROID can be time-consuming, and the inherent stochastic evolution process may result in different results across multiple runs. In contrast, TREADROID employs a semantically driven two-stage adaptive strategy to optimize tests, ensuring exploration aligns with the intended functionality of the application. Distinct from TRASM, TREADROID not only improves the adaptive strategy but also integrates new semantic matching rules to better match widgets in similar applications. The adaptive semantic matching strategy further reduces the gap where current methods struggle to accurately represent the similarity of widgets. Indeed, ADAPTDROID completed 100 generations in 24 hours on average. The findings from RQ4 suggest that the worst-case time for TREADROID to reuse a test is approximately 3 hours. While maintaining the effectiveness, TREADROID demonstrates remarkable efficiency as a lightweight approach.

7. Conclusion

In this paper, we propose TREADROID, a novel framework to enhance the usability of the reused tests. TREADROID leverages the connection between different attributes to group attributes, which enables a more accurate calculation of semantic similarity across attributes. It performs GUI events deduplication to simplify the initial test generated through an improved semantic matching approach. Through the adaptive strategy, TREADROID further searches for the target test with functionality closely aligned to the source test. Experimental results demonstrate that TREADROID can achieve superior flexibility in generating tests compared to the state-of-the-art methods. In addition, we also identified the contributions of the improved semantic matching and the adaptive strategy in enhancing test reuse. Moreover, TREADROID also exhibits the ability to reduce the manual effort of creating tests for similar applications.

In future work, we plan to comprehensively consider the granularity of similarity calculation at the sentence-level to improve the accuracy of event matching. Generating tests without relying on existing test scripts [45,46] is also the next step to make test reuse more applicable.

CRediT authorship contribution statement

Shuqi Liu: Data curation, Software, Writing – original draft. **Yu Zhou:** Conceptualization, Methodology, Supervision, Writing – review & editing. **Longbing Ji:** Software, Validation. **Tingting Han:** Writing – review & editing. **Taolue Chen:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No. 61972197, No. 62372232), the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), Birkbeck BEI School Project (EFFECT), and National Natural Science Foundation of China (No. 62272397).

References

[1] M.E. Joorabchi, A. Mesbah, P. Kruchten, Real challenges in mobile app development, in: 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, IEEE, 2013, pp. 15–24.

[2] P.S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, D. Lo, Understanding the test automation culture of app developers, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), IEEE, 2015, pp. 1–10.

[3] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, How do developers test Android applications?, in: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 613–622.

[4] Y. Zhou, Y. Su, T. Chen, Z. Huang, H.C. Gall, S. Panichella, User review-based change file localization for mobile applications, IEEE Trans. Softw. Eng. (2020).

[5] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for Android applications, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 94–105.

[6] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for Android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 224–234.

[7] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, Z. Su, Practical gui testing of Android applications via model abstraction and refinement, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 2019, pp. 269–280.

[8] N. Mirzaei, H. Bagheri, R. Mahmood, S. Malek, Sig-droid: automated system input generation for Android applications, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2015, pp. 461–471.
[9] M. Ermuth, M. Pradel, Monkey see, monkey do: effective generation of gui tests with inferred macro events, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, 2016, pp. 82–93.
[10] Ui/application exerciser monkey, http://developer.android.com/tools/help/monkey.html.
[11] Z. Dong, M. Böhme, L. Cojocaru, A. Roychoudhury, Time-travel testing of Android apps, in: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, 2020, pp. 481–492.
[12] A. Memon, I. Banerjee, A. Nagarajan, Gui ripping: reverse engineering of graphical user interfaces for testing, in: 10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings, IEEE, 2003, pp. 260–269.
[13] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based gui testing of Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 245–256.
[14] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, A.M. Memon, Using gui ripping for automated testing of Android applications, in: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2012, pp. 258–261.
[15] J. Wang, Y. Jiang, C. Xu, C. Cao, X. Ma, J. Lu, Combodroid: generating high-quality test inputs for Android apps via use case combinations, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 469–480.
[16] F. Behrang, A. Orso, Automated test migration for mobile apps, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, 2018, pp. 384–385.
[17] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for Android apps, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 224–234.
[18] G. Hu, L. Zhu, J. Yang, Appflow: using machine learning to synthesize robust, reusable ui tests, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 269–282.
[19] A. Rau, J. Hotzkow, A. Zeller, Transferring tests across web applications, in: International Conference on Web Engineering, Springer, 2018, pp. 50–64.
[20] J.-W. Lin, R. Jabbarvand, S. Malek, Test transfer across mobile apps through semantic mapping, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 42–53.
[21] F. Behrang, A. Orso, Test migration between mobile apps with similar functionality, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2019, pp. 54–65.
[22] Q. Mao, W. Wang, F. You, R. Zhao, Z. Li, User behavior pattern mining and reuse across similar Android apps, J. Syst. Softw. 183 (2022) 111085.
[23] L. Mariani, A. Mohebbi, M. Pezzè, V. Terragni, 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, 2021, pp. 177–190.
[24] S. Liu, Y. Zhou, T. Han, T. Chen, Test reuse based on adaptive semantic matching across Android mobile applications, in: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2022, pp. 703–709.
[25] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, S. Malek, Reducing combinatorics in gui testing of Android applications, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 559–570.
[26] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Adv. Neural Inf. Process. Syst. 26 (2013).
[27] Googleplay, https://play.google.com/store/.
[28] F-droid, https://f-droid.org/.
[29] R. Mahmood, N. Mirzaei, S. Malek, Evodroid: segmented evolutionary testing of Android apps, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 599–609.
[30] L. Mariani, M. Pezzè, D. Zuddas, Augusto: exploiting popular functionalities for the generation of semantic gui tests with oracles, in: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 280–290.
[31] A. Rosenfeld, O. Kardashov, O. Zang, Automation of Android applications functional testing using machine learning activities classification, in: Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 122–132.
[32] S.R. Choudhary, A. Gorla, A. Orso, Automated test input generation for Android: are we there yet?(e), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 429–440.
[33] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, S. Malek, Reducing combinatorics in gui testing of Android applications, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 559–570.
[34] V.I. Levenshtein, et al., Binary codes capable of correcting deletions, insertions, and reversals, Sov. Phys. Dokl. 10 (1966) 707–710.
[35] Craftdroid, https://github.com/seal-hub/CraftDroid/tree/master/.
[36] A. Memon, I. Banerjee, A. Nagarajan, Gui ripping: reverse engineering of graphical user interfaces for testing, in: 10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings, IEEE, 2003, pp. 260–269.
[37] N. Mirzaei, H. Bagheri, R. Mahmood, S. Malek, Sig-droid: automated system input generation for Android applications, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), IEEE, 2015, pp. 461–471.
[38] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based gui testing of Android apps, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 245–256.
[39] K. Mao, M. Harman, Y. Jia, Crowd intelligence enhances automated mobile testing, in: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2017, pp. 16–26.
[40] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, Mining Android app usages for generating actionable gui-based execution scenarios, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 111–122.
[41] X. Lu, X. Liu, H. Li, T. Xie, Q. Mei, D. Hao, G. Huang, F. Feng, Mining usage data from large-scale Android users: challenges and opportunities, in: 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2016, pp. 301–302.
[42] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, N. Medvidovic, Fruiter: a framework for evaluating ui test reuse, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1190–1201.
[43] C. Li, Y. Jiang, C. Xu, Gui event-based record and replay technologies for Android apps: a survey, J. Softw. 33 (5) (2022) 1612–1634, https://doi.org/10.13328/j.cnki.jos.006551.
[44] L. Mariani, M. Pezzè, V. Terragni, D. Zuddas, An evolutionary approach to adapt tests across mobile apps, in: 2021 IEEE/ACM International Conference on Automation of Software Test (AST), IEEE, 2021, pp. 70–79.
[45] J. Liang, S. Wang, X. Deng, Y. Liu, Rida: cross-app record and replay for Android, https://cse.sustech.edu.cn/faculty/~liuyp/files/ICST2023-Rida.pdf.
[46] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S.A. Khan, Y. Brun, N. Medvidovic, K. Moran, Avgust: automating usage-based test generation from videos of app executions, in: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Singapore, 2022, pp. 421–433.