Comprehensive Semantic Repair of Obsolete GUI Test Scripts for Mobile Applications

Shaoheng Cao
shaohengcao@smail.nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Minxue Pan*
mxp@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Yu Pei
csypei@comp.polyu.edu.hk
Department of Computing, The Hong Kong Polytechnic University
Hong Kong, China

Wenhua Yang
ywh@nuaa.edu.cn
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics
Nanjing, China

Tian Zhang
ztluck@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Linzhang Wang
lzwang@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

Xuandong Li
lxd@nju.edu.cn
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China

ABSTRACT

Graphical User Interface (GUI) testing is one of the primary approaches for testing mobile apps. Test scripts serve as the main carrier of GUI testing, yet they are prone to obsolescence when the GUIs change with the apps' evolution. Existing repair approaches based on GUI layouts or images prove effective when the GUI changes between the base and updated versions are minor, however, they may struggle with substantial changes. In this paper, a novel approach named COSER is introduced as a solution to repairing broken scripts, which is capable of addressing larger GUI changes compared to existing methods. COSER incorporates both external semantic information from the GUI elements and internal semantic information from the source code to provide a unique and comprehensive solution. The efficacy of COSER was demonstrated through experiments conducted on 20 Android apps, resulting in superior performance when compared to the state-of-the-art tools METER and GUIDER. In addition, a tool that implements the COSER approach is available for practical use and future research.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Human-centered computing** → *Graphical user interfaces.*

KEYWORDS

GUI test script repair, Android testing, regression testing

*Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639108

ACM Reference Format:
Shaoheng Cao, Minxue Pan, Yu Pei, Wenhua Yang, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Comprehensive Semantic Repair of Obsolete GUI Test Scripts for Mobile Applications. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639108

1 INTRODUCTION

With the rapid advances in internet technology, mobile applications have integrated deeply into all aspects of people's lives. Consequently, the demand for high-quality mobile applications has increased, making their quality assurance a top priority for app developers. Given the GUI-driven characteristics of mobile apps, GUI testing has become the primary testing method. This method leverages GUI test scripts to simulate user interactions with the application under test so that the testing execution can be automated with test harness/tools like Appium [1] and Robotium [30].

In these test scripts, GUI elements, or widgets, are selected based on their positions and/or properties, which makes the scripts highly vulnerable to changes in the apps' GUI during updates [36]. Meanwhile, the GUIs of mobile applications are updated frequently due to the rapid software development cycle and swift iteration process, which often causes the GUI test scripts designed for the base version apps to become obsolete and broken for the updated versions. Manual repairing of GUI test scripts is both labor-intensive and costly. In response to this challenge, a variety of techniques have been proposed to automatically repair broken test scripts.

Model-based approaches, such as Atom [19] and Chatem [3], assume the availability of high-quality behavioral models for apps under testing and utilize the models to guide the repairing of obsolete test scripts. Given the complexity of real-world mobile apps nowadays, it, however, is highly challenging (if not impossible) to construct and maintain such high-quality models for the apps, which greatly restricts the applicability of model-based approaches. Approaches like METER [27] and GUIDER [36] leverage the visual

and structural information about the GUI elements to guide the repair process. The information can be easily acquired at runtime with the help of testing frameworks like UI Automator, and we refer to such information as external semantic information since, while it reflects the apps' semantics, it is more closely related to the external appearance, rather than internal logic, of the apps. In particular, METER establishes the matching relation between elements on app GUIs based on their visual appearance, and it utilizes that relation to better locate the evolved GUI elements; GUIDER combines the structural and visual information to build the matching relation between the GUI elements to guide the process of repairing test scripts. Experimental results show that METER and GUIDER are more effective than model-based approaches in repairing obsolete GUI tests for mobile apps [27, 36].

However, the efficacy of existing approaches is limited when applications undergo substantial GUI changes during evolution. On the one hand, application vendors tend to make the changes between versions more visible to users for usability reasons [25], which often leads to substantial changes to the app GUIs and, in turn, the external semantic information. These substantial changes will cause the existing techniques that rely on external semantic information to produce fewer correct repairs. On the other hand, while external semantic information can often help match GUI elements at two specific screens, it falls short when handling complex situations involving, e.g., the adjustment of interaction logic and the removal of GUI elements.

To tackle these challenges, we propose a novel approach named COSER (**CO**mprehensive **SE**mantic **R**epair). We have observed that applications requiring regression testing with the same set of test scripts are usually developed within the vendors, implying the availability of source code. Compared with external semantic information, the information drawn from the source code better reflects the intended behaviors of the GUI elements but was neglected by the previous work. From the app source code, we can extract relevant code snippets that define the behaviors of GUI elements as internal semantics. COSER innovatively combines external and internal semantic information of GUI elements to identify, for each of the widgets in the base version app, which other widgets from the updated version are the probable matches. COSER analyzes the source code of applications to extract the GUI elements with internal semantics, and it also initializes a transition graph to facilitate the repairing process.

To evaluate the efficacy of the COSER approach, a tool by the same name, COSER, has been implemented and applied to repair GUI test scripts of 20 open-source Android apps. In the experiments, it helped preserve 90% test actions of all the test scripts to run correctly. This constitutes a significant improvement of 44% over METER and 23% over GUIDER, the current state-of-the-art tools, respectively.

The contributions of our work are as the following:

- We present COSER, a novel approach for automating the repair of GUI test scripts for Android apps. The approach harnesses both external and internal semantic information of GUI elements, offering targeted semantic matching methods for different semantic types, resulting in a unique and comprehensive way to produce high-quality repairs.
- We implement a supporting tool for the easy application of COSER, which is publicly available at https://github.com/Vallxy/COSER.
- We empirically evaluate COSER's effectiveness and efficiency by applying it to repair GUI test scripts for 20 open-source Android apps collected from other research works. The results clearly indicate that COSER outperforms existing approaches significantly.

2 COSER IN ACTION

In this section, we use an app named *Network Monitor* with a relevant test script to illustrate the repairing process of COSER. Figure 1 shows the screens of version 1.17.0 (Screen 1.1 and 1.2) and version 1.32.1 (Screen 2.1 and 2.2) for selecting all the fields to monitor.

Listing 1 presents a test script TS1 designed for the base version app. This script is written in Python for the Appium testing engine, and it tests the functionality of selecting all the fields to monitor. Test script TS1 first locates the button "More options" (W1) via its accessibility id and clicks the button to expand the menu (Screen 1.1). Then, the menu item "Select all" (W2) is clicked to select all the checkboxes (Screen 1.2). To locate the menu item, the script obtains all the GUI elements with id *android:id/title* and gets the first one from the returned list of elements.

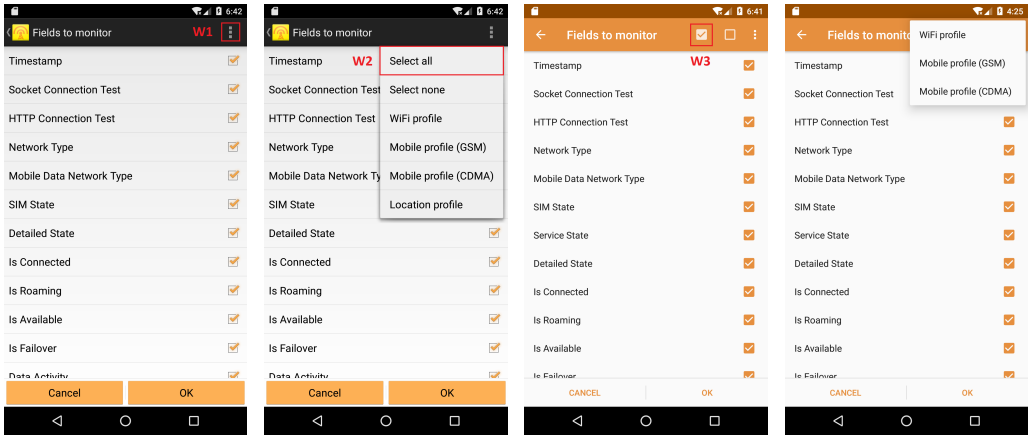
In the updated version app, the "Select all" menu item is removed from the dropdown menu, turned into a standalone checkbox, and assigned with a new id *android.networkmonitor:id/action_select_all*. With the change, the original test script becomes obsolete on the updated version app as it does not test the functionality of selecting all the fields to monitor now.

Taking the base and updated version app as well as test script TS1 as the input, COSER can automatically produce the test script TS2 shown in Listing 2 as the repaired version for TS1. The key to COSER's success in repairing the test script lies in the fact that it relies on both the properties and the code that are associated with the widgets, which we refer to as the external and internal semantic information of the widgets in this work, to find out how the widgets have evolved across app versions. In particular, COSER retains the first test action in TS1 since the updated version app also has a button with the same accessibility id "More options", and COSER correctly recognizes widget W3 in Screen 2.1 as the counterpart of menu item W2 in Screen 1.2 since the two widgets will trigger the execution of similar code when activated.

In contrast to that, neither of the two state-of-the-art GUI test repairing tools, namely METER and GUIDER, can correctly repair test script TS1. METER fails because the "Select all" menu item (W2) evolves from a textual element into a graphical element, and the visual dissimilarity between the two widgets prevents the tool from establishing the matching relation between them. GUIDER also fails because, since widgets W3 and W2 do not share any identity property, the tool cannot match the two widgets with each other based on the structural information either.

3 THE COSER APPROACH

Figure 2 illustrates an overview of the COSER approach. The input to COSER includes both the base and updated versions of an application, along with a test script for the base version, and the



(a) Screen 1.1 (b) Screen 1.2 (c) Screen 2.1 (d) Screen 2.2

Figure 1: Screens of the base and updated version of Network Monitor for selecting all the fields to monitor.

```
# TS1
driver.find_element_by_accessibility_id('More options').click()
driver.find_elements_by_id('android:id/title')[0].click()
...
```

Listing 1: Test script TS1 for the base version app.

```
# TS2
driver.find_element_by_accessibility_id('More options').click()
driver.back()
driver.find_element_by_id(
    'android.networkmonitor:id/action_select_all').click()
...
```

Listing 2: Repaired test script TS2 for the updated version app.

output is the repaired test script for the updated version. COSER has four main components: (1) A *Script Executor* that parses the input obsolete script into an internal representation and executes the script to provide enough information for subsequent stages; (2) A *Semantic Extractor* that extracts both the external and internal semantic information of each GUI element; (3) A *Transition Analyzer* that extracts a static transition graph from the source code and refines it using dynamic information; (4) A *Scripts Repairer* that incorporates the external semantic information, the internal semantic information, and the transition graph to repair the input script. The following subsections will introduce the individual components and how they work together in repairing obsolete test scripts.

3.1 Script Executor

Script Executor takes the input of an obsolete script for the base version application and parses it into a sequence of test actions. After parsing the test script, this component proceeds to execute the script with the testing engine and collect information for the *Semantic Extractor*. We separate the process of parsing and executing the script into distinct stages to prevent this component's dependence on a specific testing engine, thus ensuring its extensibility.

3.2 Semantic Extractor

Semantic Extractor plays the role of extracting the external and internal semantic information of GUI elements. The external semantic information of GUI elements is obtained from the properties related to the appearance or functionality of the elements. Meanwhile, the internal semantic information of GUI elements is code snippets that define the functionality of the elements. Notably, COSER combines both forms of semantic information in the script repair task, thereby providing a comprehensive approach to script repair.

3.2.1 External Semantics. External semantics of GUI elements is textual information related to appearance or functionality. Such information can be extracted from the properties of GUI elements during runtime. To accomplish this, COSER utilizes UI Automator, an integral part of the Android SDK, which provides APIs to access the properties of GUI elements on any given screen.

In this work, we lay emphasis on three specific properties including *text*, *content-desc*, and *resource-id*. These properties have been selected due to their strong correlation with our definition of external semantics.

For each GUI element, one of these three properties is chosen to represent its external semantics based on a predetermined priority, and the property is chosen if and only if there is no non-empty property whose priority is higher than it. The priorities are assigned in descending order: the *text* property takes precedence, followed by *content-desc*, and finally *resource-id*. We prioritize the properties like this for the following reasons. The property *text* is the most common one displayed to the users and frequently outlines the functionality of a GUI element; The property *content-desc* also has the same features as *text*, but it is less frequently used, so it has the second priority. The property *resource-id* is set to identify the GUI elements and is closer to the developer's view of functionality rather than the users' view so it has the lowest priority.

Although most GUI elements can get their external semantics with the strategy above, there are still some complex situations where the three properties of an element are all empty, thus having nothing to assign. We observed that such situations appear on specific types of GUI elements, e.g., *CheckBox*, *switchButton*, and *EditText*. These types of elements are standard widgets provided by the Android framework and often do not have meaningful text

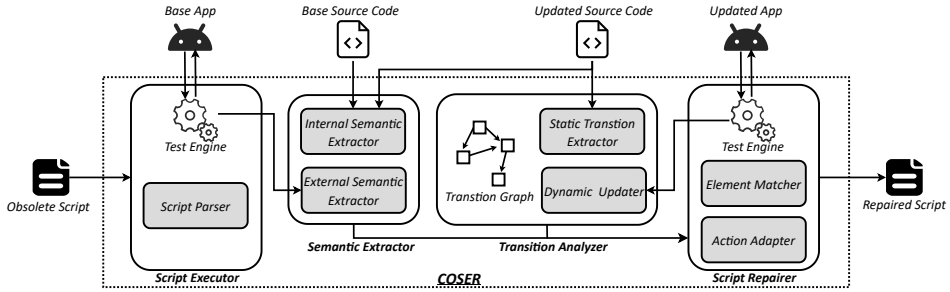


Figure 2: Overview of COSER

```
<Button android:id="@+id/main_menu_button_new"
        android:onClick="handleNewButton"
        android:text="@string/main_menu_new" />
```

Listing 3: Pattern 1: Direct method binding in the layout XML files.

```
createButton=(TextView)findViewById(R.id.create_list);
createButton.setOnClickListener(v->createNewList(...)));
```

Listing 4: Pattern 2: Callback methods bonded to the GUI elements.

```
case R.id.menu_voice_add:
    startVoiceRecognitionActivity(...);

if(id==R.id.menu_voice_add){
    startVoiceRecognitionActivity(...);}
```

Listing 5: Pattern 3: Context-sensitive code binding such as binding different menu items to different handlers.

properties. Instead, they need to be associated with the surrounding elements to express their semantics correctly.

To properly assign external semantics to these elements, a strategy is developed based on the rendering rules and typical layouts of these types of elements. This strategy involves identifying what we term a "perceptual group" for each target element. In this context, a perceptual group refers to GUI elements grouped to convey a specific functionality. The problem of grouping GUI elements in Android applications has drawn significant attention, resulting in some innovative works [35, 39]. In this work, we adopted the container recognition approach as detailed in [35]. This approach is based on the observation that a container in a GUI is visually identifiable as a (round) rectangular wireframe that encloses multiple children elements. This understanding facilitates the effective segregation of elements into distinct groups, thereby enabling the assignment of relevant external semantics to them.

A key characteristic of these groups in our context is the presence of a central element, surrounded by other different types of elements. Typically, only the central element has meaningful textual information with it. Once identifying the perceptual groups of target elements, the external semantics of the central element will be assigned to those elements that do not have external semantics yet within the group. During the GUI element matching stage, only a single interactable element performing the functionality of the group is considered (e.g., *CheckBox*), which effectively prevents multiple matches within a group.

3.2.2 Internal Semantics. Internal semantics of a GUI element refers to the code snippets that define its functionalities. In Android applications, different screens are usually represented by *Activities*. Developers declare GUI elements in layout XML files and bind them to *Activities* through functions such as *setContentView()*. Subsequently, developers can retrieve the reference of a specific GUI element through the identifier with functions such as *findViewById*. They can write some code to define the behaviors of the elements. Some GUI elements are created dynamically during runtime. These elements are out of the scope of our current implementation.

Internal Semantic Extractor analyzes the source code of the applications to extract the internal semantics of GUI elements. To be specific, it first scans the source code to acquire the binding relationship between the *Activities*/*Fragments* and layout XML files. After that, the extractor recognizes all the GUI elements in the layout XML files. Following this, the matching relationship between *Activities*/*Fragments* and their inclusive GUI elements can be acquired. The extractor then continues to get the relevant code snippets of each GUI element as the internal semantics according to three pre-defined patterns shown in Listings 3 through 5.

We follow these patterns to extract the internal semantics because they are all standard or de-facto design guidelines and are widely used by developers. It is noteworthy that, in specific cases, a single GUI element might be linked with multiple code snippets, each defining a unique functionality within different contexts. As illustrated in Figure 3, when the system operates in fully offline mode, the "login" and "sync" buttons are associated with identical code snippets. In contrast, when the system operates outside of this mode, these buttons are tied to distinct functionalities. To acquire a comprehensive description of the GUI elements' functionalities and to correctly distinguish them, all of an element's code snippets will be aggregated as its internal semantics. By employing this strategy, we can obtain a comprehensive representation of the internal semantics for each GUI element.

3.3 Transition Analyzer

Transition Analyzer analyzes the source code to produce a transition graph of the applications. This transition graph provides information about how to navigate from one screen to another and how to reach some GUI elements within specific element groups. The transition graph is acquired statically and will be updated and used dynamically by the *Script Repairer*. To construct the transition graph, two types of analysis are applied in *Transition Analyzer*.

3.3.1 Inter-Activity Transition Analysis. Android apps can invoke activity transition by passing the *Intent* to the handler methods such as *startActivity()*. The *Intent* is an abstract description of an

```
if(PREFERENCE_FULL_OFFLINE){
  switch(menuItem.getItemId()){
    case R.id.sync:  showDrawerErrorToast("offline");
                     break;
    case R.id.login: showDrawerErrorToast("offline");
                     break;
    ...
  }
}
else{
  switch(menuItem.getItemId()){
    case R.id.sync:  // sync code here
                     break;
    case R.id.login: // login code here
                     break;
    ...
  }
}
```

Figure 3: An example of multiple code snippets bonded to a single GUI element

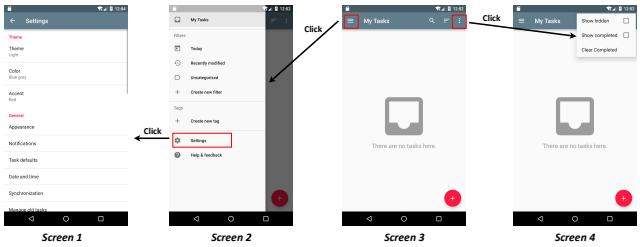


Figure 4: An example of Transition Graph for the app Tasks

operation to be performed. As described in the Android document, it can be categorized into two types:

- *explicit Intent*: explicitly declaring the transition relation from one *Activity* to another.
- *inexplicit Intent*: declaring the desired action for the *intent-filter* to catch, thus deciding the transition action.

As developers are suggested to use *Intent* to realize the transition between *Activities*, the analyzer then analyzes the relevant code snippets of *Intent* to acquire the transition information of *Activities*. As to *explicit Intent*, the analyzer can directly acquire the source and destination *Activities*. For *inexplicit Intent*, the analyzer first extracts the desired action and then gets the corresponding *Activity* that has the *intent-filter* to handle the action. With the analyzing method above, transitions between *Activities* and the GUI elements which can trigger the transitions can be acquired.

3.3.2 Intra-Activity Transition Analysis. Although the transition relation between *Activities* can provide an overview of the whole app, the information is still coarse-grained. It neglects the transitions within each *Activity*. To handle this, *Intra-Activity Transition Analysis* is introduced; this module focuses on analyzing the transition relations within an *Activity*.

One example is the menus in Android. There are different types of menus in Android. In this work, we consider the most common ones. *More Options Menu* is a widely used menu type that appears after Android 3.0, usually at the top of the application. Our analyzer detects initializing methods such as *inflateMenu()* to detect the presence of the *More Options Menu* within *Activities*, and subsequently constructs transition relations between the menu and its items. In addition to the *More Options Menu*, the analyzer supports analysis for other UI components such as the *Navigation Drawer (Side Bar)* and *Popup Windows*. The *Intra-Activity Transition Analysis* can be extended to support more types of UI components provided by the Android SDK or other third-party libraries, thus making it adaptable to a wide variety of Android applications.

Figure 4 shows an example of a transition graph that includes the outcome of both *Inter-Activity Transition Analysis* (Screen 2 to 1) and *Intra-Activity Transition Analysis* (Screen 3 to 2 and Screen 3 to 4). This graph will be continuously updated by the *Script Repairer* to incorporate more transition relations.

3.4 Script Repairer

The *Script Repairer* is the most critical component in the COSER approach. This component takes the information including the transition graph and the external and internal semantics of GUI elements as input to repair the obsolete scripts. In this section, we first introduce the mechanism *GUI element matching* which determines the matching relation between GUI elements. Subsequently, we elaborate on the design of the repair algorithm. Finally, we will discuss how our algorithm achieves the effect of *Action Adaptation*, which necessarily reduces or amplifies the test actions.

3.4.1 GUI Element Matching. In COSER, GUI element matching utilizes both external and internal semantics.

The external semantics, derived from properties set by developers and often expressed in natural language, have been previously shallowly leveraged via word-level semantic comparison or simple string comparison. However, many of these external semantics are short sentences that convey specific meanings, thus not suitable for word-by-word semantic comparison. Additionally, simple string comparison may not accurately capture the semantic similarity between two short sentences that have similar meanings but differ significantly in their wording. Consequently, we innovatively leverage large language models to calculate the similarity of these external semantics. In this work, we selected Sentence-BERT [28] to calculate the external semantic similarity. Sentence-BERT is a variant of the well-known large language model BERT [6]. It is specifically engineered to efficiently convert a sequence of words into a representative embedding vector. Within our study, in order to calculate the similarity between two external semantics, Sentence-BERT is utilized to generate embeddings of these semantics. The similarity score is subsequently calculated by applying cosine similarity to these vectors.

In terms of external semantics, the algorithm first calculates the similarity score of the external semantics between the old element and all elements present in the current screen of the updated version app. After that, one element with the highest score will be selected as the matching element. COSER then checks whether the similarity score is greater than a given threshold denoted as θ. If the score is indeed greater, the match is considered successful; otherwise, the matching element is discarded and the algorithm resorts to internal semantics for matching. The threshold θ is set to 0.7 empirically. The impact of different values of θ on the outcome will be examined in the evaluation section.

Internal semantics are extracted from the code snippets corresponding to the GUI elements. Notably, these code snippets are encapsulated into moderately grained functions, making the task of detecting similar internal semantics resembles the task of clone detection. Hence, we utilize a code pre-trained model and fine-tune

it on the clone detection task to measure the internal semantic similarity. The model we employ is UniXcoder [12], and we fine-tune it on the BigCloneBench [33] dataset.

UniXcoder is chosen because of its proven effectiveness in source code processing tasks. It is specifically designed to understand and model the underlying structure of the code, making it a fitting choice for our internal semantic comparison task. By fine-tuning UniXcoder on the clone detection task, we ensure the model is well-adjusted to identify semantically similar code snippets, thus enhancing the accuracy of our internal semantic similarity calculation. The fine-tuned model takes two code snippets as input and determines whether the pair are semantically similar so there is no threshold. We also extract the similarity score from the model, subsequently employing it to sort the matching elements if there are multiple matches.

In terms of internal semantics, the algorithm detects all potential matching elements extracted by static analysis of the entire updated version app. If there are multiple matching elements, the one with the highest similarity score will be chosen.

The external semantics are employed to handle the cases where the GUI changes are minor. These changes typically involve slight adjustments to the GUI widgets, including e.g., changing their texts, appearances, and/or locations. Given these minor modifications, actions are relatively easy to repair using solely external semantics. The internal semantics of a GUI element represents its functionality. Throughout an app's evolution, while the GUI may experience minor or major changes, the core functionalities provided by a specific app typically remain consistent. Furthermore, the aim of our work is to repair the test script written for the base version, enabling it to test the same functionalities in the updated version. GUI element matching with internal semantics facilitates the identification of similar functionalities across the base and updated versions, thus enhancing the script repairing process. Internal semantic matching relies on static analysis. There are some GUI elements generated automatically by the Android framework, which is out of the scope of our static analysis method. These auto-generated elements' external semantics are often more complete and easier to acquire so external semantic matching works. COSER utilizes both external and internal semantic information to more accurately match the GUI elements across different versions of mobile apps.

3.4.2 Repairing Algorithm. In this work, we use a pair $\langle loc, evt \rangle$ to denote a test action a where loc is an element locator for pinpointing a particular GUI element on a given screen, and evt is an event to be triggered on that element when a is executed. Following the practice in previous work [18, 27, 36], we define a test script as a sequence $\mathcal{K}$= $a_1, a_2, ..., a_n$ where each a_i is a test action.

Executing a test action $a = \langle loc, evt \rangle$ on a screen S involves first applying the locator loc to identify on S a target GUI element to interact with and then triggering the event evt on the element. If the execution terminates successfully, it should transit the app to a (possibly different) destination screen.

Algorithm 1 explains how COSER repairs a test script to preserve its original intentions on the updated version. The algorithm takes the base version app $\mathcal{P}$, the updated version app $\mathcal{P}'$, a test script $\mathcal{K}$ for $\mathcal{P}$, and a transition graph extracted from the source code of $\mathcal{P}'$ denoted as $\mathcal{T}$ as input, producing a repaired test script $\mathcal{R}$ as output.

Algorithm 1: Test Script Repair

```
   Input: Base Version 𝒫, Updated Version 𝒫', A test script 𝒦 written for 𝒫,
          Transition graph extracted from the source code of 𝒫' denoted as 𝒯
   Output: Repaired test script ℛ
1  ℛ ← [];
2  for a in 𝒦 do
3      src ← The screen before executing a on 𝒫;
4      dest ← The screen after executing a on 𝒫;
5      src' ← The current screen of 𝒫';
6      oldEL ← findElement(a.loc, src);
7      newEL, exScore ← getExSemanticMatch(oldEL, src');
8      if exScore < θ then
9          newEL ← getInSemanticMatch(oldEL, 𝒫');
10         if isReachable(𝒯, src', newEL) then
11             path ← findPath(𝒯, src', newEL);
12             src' ← executePath(path, src');
13         else
14             newEL = NULL;
15         end
16     if newEL.exist() then
17         loc'← constructLOC(newEL, src');
18         ℛ.append(⟨loc', a.evt⟩) ;
19         dest' ← executeEvent(newEL, a.evt);
20         𝒯.update();
21     else
22         continue;
23     end
24 end
25 return ℛ;
```

COSER repairs the script action by action (line 2). Since $\mathcal{K}$ has been executed on $\mathcal{P}$ by *Script Executor* before the algorithm starts, the screens of $\mathcal{P}$ before and after executing the action a also have been recorded. Information about the GUI layout and the properties of all GUI elements is also included in each screen. We denote the screens before and after executing the action a as *src* and *dest*, respectively (line 3-4). Also, current screen of $\mathcal{P}'$ before the repairing process is recorded as *src'* (line 5). After acquiring the information above, the repair process is just launched. First, the element for a is acquired from the screen *src* using the locator of a. We denote this element as *oldEL* (line 6). Next, the algorithm uses the external semantics of *oldEL* and tries to find a matching element on *src'*. The matching element's external semantics has the highest similarity score with that of *oldEL* in the scope of *src'*. This matching element will be denoted as *newEL* and the similarity score will be denoted as *exScore* (line 7). If the similarity score is less than a given threshold θ, then the matching is considered not convincing and we resort to using internal semantics for matching (line 8-9). At the stage of element matching with internal semantics, all elements of the updated version $\mathcal{P}'$ will be considered. The final matching element will be the one with the highest similarity score as to the internal semantics. After acquiring the matching element through internal semantics, the algorithm then determines whether the *newEL* is reachable in the scope of the transition graph (line 10). Note that the matching element obtained by internal semantic matching is not necessarily on *src'* and might be on any screen of the updated version $\mathcal{P}'$. As a result, a path should be constructed to navigate from *src'* to where the matching element resides if the element is reachable (line 11). After getting the path, we then execute the path on the *src'* and update the *src'* (line 12). Once all steps have been undertaken, the algorithm then checks whether a matching element *newEL* has been found. If a match is found, a repaired test action

will be constructed and executed (line 16-19). Simultaneously, the transition graph is also updated (line 20). However, if there is no matching element found, the algorithm skips the test action and proceeds to the next one (line 21-22). This strategy is employed because as GUIs evolve, the interaction steps associated with a specific functionality may be reduced, or even the functionality itself can be removed from the app. If no matching element can be found through all of the methods previously mentioned, we determine that the interaction step is no longer needed in the updated version, so we skip the action and proceed to repair the next test action.

3.4.3 Action Adaptation. During the evolution of app GUIs, not only do the GUI elements themselves change but there are also adjustments to the interaction logic and the removal or addition of elements. Previous works such as METER and GUIDER only have a simple mechanism to handle such situations. METER relies on a global matching mechanism; it still leverages solely visual information, and the mechanism contributes little to the final results. GUIDER randomly selects a GUI element to interact with when it fails to find a match, which is also ineffective as we observed in the experiments. Consequently, when it comes to major GUI changes, most scripts cannot be repaired correctly by METER and GUIDER, thus making subsequent test actions no longer executable. To solve this problem, COSER's repair algorithm introduces the unique mechanism of *Action Adaptation*.

Action Adaptation includes the reduction and amplification of test actions. On the one hand, elements or functionalities might be removed. The test actions interacting with these elements should also be removed to make the subsequent actions run correctly. However, it can be challenging to determine whether an element has been completely removed or if it has simply been relocated or evolved into a different element. COSER leverages the internal semantics of GUI elements to handle the removal of GUI elements. If no internal semantic matching element exists for a GUI element of the base version, then there should be no corresponding functionality in the updated version either; thus removing the original test action is reasonable. Through this strategy, COSER achieves the reduction of test actions. On the other hand, when the interaction sequences become more complex, repairing the test actions also becomes a challenge. To handle this situation, COSER applies internal semantic matching to find a matching element that is reachable via the transition graph. Through this strategy, COSER achieves the amplification of test actions.

3.5 Implementation

We have implemented the approach described above into a tool, also named COSER, to automate the repair of GUI test scripts for Android apps and analyze their source code. COSER currently exploits Appium to drive mobile apps and run test scripts written in Python. In our implementation, COSER supports analyzing the apps' source code written in Java using the JavaParser [16].

One of the key features of COSER is its loose coupling with all employed frameworks, libraries, and models. This architectural decision facilitates extensibility, enabling COSER to incorporate any other testing and analyzing techniques in the future. Also, COSER does not heavily rely on any specific model. The models used in our implementation can be easily replaced by others if needed. Furthermore, the extensibility of COSER opens the possibility to support other types of source code as long as corresponding language analysis modules are developed.

To improve COSER's efficiency, we built a cache for the matching relations of GUI elements because many of the matching relations are reused a lot of times. By implementing this strategy, COSER could search the cached data and obtain the semantic matching relations, substantially reducing repeated computations.

4 EVALUATION

To evaluate the effectiveness and efficiency of COSER, we conducted experiments that apply COSER to repair GUI test scripts for real-world apps. Based on the experimental results, we address the following research questions:

- **RQ1**: How effective is COSER in repairing GUI test scripts?
- **RQ2**: How efficient is COSER in repairing GUI test scripts?
- **RQ3**: How does COSER compare with the state-of-the-art approaches in terms of effectiveness and efficiency?
- **RQ4**: How do different components and configurations affect COSER's effectiveness?

4.1 Experimental Subjects

To answer RQ1 to RQ4, we collected in total 20 open-source Android apps as subjects from previous studies on testing for mobile apps.

Considering that previous studies on mobile applications systematically collected their subject apps to reflect the diversity of mobile applications and to reduce the influence of our bias in subject selection, we collected the Android apps from 13 papers [7, 8, 13, 15, 17, 21, 26, 27, 29, 32, 34, 36, 37] on mobile applications published in the past 3 years at major software engineering conferences. There were 171 apps initially. Among the 171 apps, 62 were excluded because they were no longer available, 17 were excluded because they only had one version accessible, 12 were excluded because the source code was not accessible, 13 were excluded because they were not written in Java, 6 games were excluded because the randomness and time-sensitiveness involved in their behaviors made them unsuitable to be tested using scripts, and 41 were excluded because their source code failed in compiling. After all these exclusions, there were 20 applications left.

We then chose the base and updated versions for apps following the strategies used by METER. We chose the latest stable versions as the updated versions and then manually constructed test scripts for them. In particular, we recruited five postgraduate students in Computer Science, each with at least two years of experience in mobile application development and testing to construct the scripts. Each student was randomly assigned four apps, and they were required to compose test scripts for each app. The objective was to ensure that these scripts adequately covered the functionalities to a level that met their satisfaction. For each application, we executed the associated test scripts on its preceding versions in a reverse chronological manner until we identified a version that acquired test results differing from the updated version (i.e., succeeding or failing). This version was then selected as the base version. After that, the same students manually revised the test scripts originally for the updated versions so that they provided a comparable level

Table 1: Experimental Subjects

ID	APP	BASE		UPDATED		$\#\mathcal{K}$	$\#\mathcal{A}$	$\mathcal{SC}$
		V	LOC	V	LOC			
S1	A2DP	2.12.4	8529	2.13.0.4	9357	8	163	50%
S2	AnkiDroid	2.6	66146	2.13.0	103211	31	808	53%
S3	AntennaPod	2.1.3	69039	2.5.0	68483	29	506	37%
S4	Better Battery Stats	2.3	18591	2.6	36109	5	285	25%
S5	ConnectBot	1.8.6	23203	1.9.8	26505	6	170	23%
S6	DNS66	0.4.1	3671	0.6.8	6752	6	69	69%
S7	Easy xkcd	5.3	10619	7.3.9	14158	12	166	37%
S8	GnuCash	2.0.0	30367	2.4.0	43724	15	200	40%
S9	KeePassDroid	2.0.6.4	14927	2.2.0.9	16710	7	200	47%
S10	Mgit	1.5.1	8502	1.6.1	8916	27	157	50%
S11	Network-monitor	1.17.0	12103	1.32.1	13019	3	44	61%
S12	OI File Manager	2.0.5	8976	2.2.2	9304	28	215	47%
S13	OpenBikeSharing	1.0	1405	1.10.0	3182	2	19	49%
S14	Open Camera	1.40	29508	1.49.2	78692	8	55	51%
S15	SoundBoard	0.91	581	0.92	665	3	13	68%
S16	SuperGenPass	2.2.2	1878	3.0.0	1936	5	31	59%
S17	Tasks	4.9.14	24784	6.0	56946	21	204	48%
S18	Vlille Checker	3.1.1	5466	4.5.0	6027	3	23	58%
S19	WhoHasMyStuff	1.0.24	1764	1.0.38	2176	9	55	65%
S20	WiFiAnalyzer	1.9.2	12393	2.1.2	29442	10	96	39%
Total			352452		535314	262	3485	44%

of statement coverage on the base version apps. These revised test scripts were then used in our experiments.

Table 1 lists the basic information of the 20 apps. For each app, the table gives the app ID (ID), the name (APP), the number of test scripts for its base version ($\#\mathcal{K}$), the number of test actions in all those test scripts ($\#\mathcal{A}$) and the percentage of statements covered by the test scripts on the base version ($\mathcal{SC}$). For both the base (BASE) and updated (UPDATED) versions of each app, the table also lists the version number (V) and the size in number of lines of code (LOC). In total, 262 test scripts of 3485 test actions were prepared for the 20 base version apps, covering 44% of the statements.

4.2 Experimental Protocol

The experiments were conducted on a Windows PC equipped with an AMD Ryzen 9 5950X CPU, 64GB RAM, and an RTX 3050 GPU. The subject Android applications were executed on an Android emulator with API level 25 provided by Android Studio.

To answer RQ1 and RQ2, we applied COSER to repair the GUI test scripts for all updated versions of the apps. The input of COSER includes the base and updated versions of the apps, denoted as $\mathcal{P}$ and $\mathcal{P}'$ respectively, the test script set written for $\mathcal{P}$ denoted as $\mathcal{K}$, the transition graph of $\mathcal{P}'$ denoted as $\mathcal{T}$ and the source code of both versions. Particularly, we first ran $\mathcal{K}$ on $\mathcal{P}$ and recorded the GUI elements's semantics and the transition of the screens before and after the execution of each test action from $\mathcal{K}$, then applied COSER to get the repaired test scripts set $\mathcal{R}$ as a derivation of $\mathcal{K}$. Finally, the previously mentioned five postgraduates manually reviewed and checked the repair results to determine the number of test actions that were correctly repaired. A repair is considered to be correct only when all the postgraduates engaged have a consensus on that.

To answer RQ3, we compared COSER with two state-of-the-art GUI test repair tools for mobile applications: METER and GUIDER. Atom [19] and Chatem [3], the other approaches that also focus on mobile GUI test repair, were excluded from our comparison due to their requirement for complete or extensive manual effort in generating application models. GUI test repair approaches targeting other application domains, such as WATER [4], which is designed for web apps, have demonstrated less efficacy in the mobile domain, attributable to the inherent differences between web and mobile applications [36]. Furthermore, we observed that test migration techniques often employ external information, a strategy that may be applicable to test repair. As such, we included AppTestMigrator[2] in our study, since it represents one of the state-of-the-art test migration techniques and is posited as being applicable to test script repair tasks [2]. It leverages the textual information of GUI elements to match the GUI elements between different applications with similar functionalities. While the original implementation of this tool was in Java, we developed a Python equivalent based on the existing source code and associated paper. Additionally, we implemented a null test repair tool that returns the same test action for each input test action. The repair results produced by this null tool can thus reflect the execution of the test scripts on updated versions without any modification, serving as a baseline for comparison.

To answer RQ4, we did the following settings: (1) We conducted an ablation study to understand how the external and internal semantics contributed to the repairing result. (2) We changed the value of θ and compared the effectiveness.

During the experiments, we recorded the same measures utilized in the other script repair studies:

- $\#\mathcal{K}_c$: The number of test scripts executed correctly after being repaired, confirmed by all the postgraduates.
- $\#\mathcal{A}_c$: The number of test actions executed correctly after being repaired, confirmed by all the postgraduates.
- T: The overall wall-clock repairing time in minutes.

4.3 Experimental Results

4.3.1 RQ1: Effectiveness. Table 2 shows the repairing results. Among the 262 test scripts consisting of 3485 test actions, only 81 (31%) test scripts and 1379 (40%) test actions could still be executed correctly on the updated versions. COSER was capable of correctly repairing a considerable number of test actions and making 215 (82%) test scripts and 3121 (90%) test actions execute correctly. This result represents an increase of 51% in the number of test scripts and 50% in the number of test actions compared with the NULL.

Although COSER successfully enabled 90% of test actions to run correctly, there were still 364 (10%) test actions that could not be repaired. We carefully examined these actions and found there were four categories of situations that could cause ineffectiveness.

The most common cause for repairing failure is functionality modification, which led to 314 (86.26%) of 364 test actions failing to be repaired. We found that those modifications involved the addition of functionalities. This addition could change the sequence of interactions or even trigger a refactor of the GUI. In the original script, there were no such actions for testing these newly added elements, but these elements existed exactly in the updated version. Interacting with them might become a prerequisite for interacting with other elements, thus leading to repair failures. For example, some GUI elements are activated only when specific settings are enabled in the updated version. However, in the base version, these particular settings do not exist. Consequently, interaction with these elements in the updated version needs additional test actions to activate the corresponding settings. Although the *Action Adaptation*

Table 2: Results on 20 Android apps

	NULL			METER			AppTestMigrator			GUIDER			COSER		
ID	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$	T	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$	T	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$	T	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$	T	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$	T
S1	1	40	18.0	2	28	18.3	3	107	21.0	2	78	23.5	6	160	22.2
S2	4	325	98.2	11	442	147.4	6	486	114.5	17	684	150.5	19	754	121.5
S3	28	297	65.1	11	274	108.3	15	376	68.3	21	381	87.9	27	489	69.5
S4	2	34	25.7	2	36	52.1	2	126	34.6	2	53	33.6	3	160	38.8
S5	0	8	16.3	2	17	18.8	1	55	20.9	2	25	17.1	3	87	29.1
S6	2	13	8.2	2	18	11.7	1	16	9.8	1	18	13.5	4	52	11.0
S7	5	47	26.6	6	104	37.1	2	46	25.9	9	123	36.0	12	166	28.8
S8	6	111	26.3	7	134	37.1	10	123	26.2	8	125	32.4	11	183	28.7
S9	11	86	20.3	2	11	32.3	22	151	23.6	21	122	32.9	26	202	23.8
S10	0	8	29.7	1	34	38.0	25	144	33.8	22	141	43.7	26	147	34.5
S11	0	19	5.2	0	15	8.7	0	24	6.0	0	19	8.2	1	41	7.4
S12	0	117	45.4	17	163	55.8	14	183	46.6	13	146	59.9	24	205	49.7
S13	0	2	4.6	0	20	7.4	1	24	5.4	2	21	7.1	5	29	5.8
S14	1	6	12.2	3	21	17.5	7	54	13.4	8	55	16.5	8	55	13.3
S15	1	8	2.1	1	11	4.1	1	10	2.2	1	10	3.0	2	11	2.3
S16	3	25	4.3	0	14	5.5	1	20	4.8	1	24	6.3	3	25	5.5
S17	12	146	28.9	9	159	44.7	5	63	28.8	14	184	41.6	16	189	31.6
S18	0	3	3.3	2	15	5.3	1	11	3.7	2	16	5.3	2	20	4.0
S19	8	52	13.3	0	22	11.1	8	52	8.6	8	52	11.5	8	52	10.0
S20	7	32	18.9	7	60	18.4	5	75	15.2	2	49	23.5	9	94	17.4
Total	81	1379	462.3	85	1598	679.6	131	2146	560.2	161	2326	648.9	215	3121	555.5

in COSER partially solves the problem, there is still a small part of the mentioned situations difficult to solve. We will explore solutions to cover these scenarios in the future work.

Besides, the incorrect repairs for 19 (5.22%) and 8 (2.2%) out of 364 test actions were caused by incorrect external and internal semantics matches, respectively, due to the imprecision of the models. In our current implementation, we use sentence-BERT for external semantic matching and fine-tuned UniXcoder for internal semantic matching. With the rapid development of large pre-trained models, there will be more precise ones in the future. COSER's architecture supports swapping to other models without much modification.

In addition, the incorrect repairs of 23 (6.32%) test actions were due to the logical bugs within the updated versions. These bugs typically result in app crashes. Although COSER is not designed to detect these bugs, its inability to repair these actions can reveal the bugs' existence.

4.3.2 RQ2: Efficiency. As shown in Table 2, it cost COSER 555.48 minutes to repair all the scripts for the 20 apps, with the longest repairing time being 121.46 minutes and the shortest time being 2.25 minutes. On average, it took COSER about 9.56 seconds to repair one test action. We examined the performance of different components of COSER and found that the most time-consuming part is the external and internal semantic matching using deep learning models, especially when it comes to internal semantics. As previously noted, we incorporated a cache to mitigate this issue. Given the full automation of the repair process, we argue that the efficiency is acceptable and we will show the comparison with other state-of-the-art approaches in the next RQ.

4.3.3 RQ3: Comparison. **Comparison for effectiveness**.

In total, METER was able to make 85 (32%) test scripts and 1598 (46%) test actions execute correctly. AppTestMigrator was able to make 131 (50%) test scripts and 2146 (62%) test actions executed correctly. GUIDER was able to make 161 (61%) test scripts and 2326 (67%) test actions executed correctly. In comparison, COSER outperformed the other approaches, enabling 215(82%) test scripts and 3121(90%) test actions to execute correctly, representing 50%, 32%, and 21% more correct test scripts than METER, AppTestMigrator, and GUIDER, respectively. Similarly, COSER correctly repaired 44%, 28%, and 23% more test actions than the other three approaches.

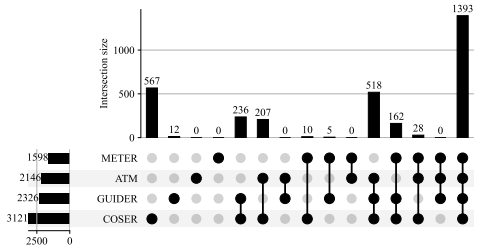


Figure 5: Partition of the test actions based on whether they can be correctly repaired by METER, AppTestMigrator, GUIDER, and COSER. Each column within the plot indicates the number of test actions that are correctly repaired by the techniques denoted by black dots but failed to be repaired by the techniques denoted by grey dots.

We examined all the repair results produced by the four approaches to reveal the reasons for their differences in effectiveness. As is shown in Figure 5, only 1393 (39.97%) out of 3485 test actions could be repaired by all approaches. There were 567 (16.27%) that could be repaired only by COSER. We examined these test actions and found that these actions were those with relatively major GUI changes including the adjustment of interaction logic and removal of elements. The combination of external and internal semantics and the action adaptation mechanism helped repair these actions. Overall, COSER repaired all test actions repaired by AppTestMigrator and most actions repaired by METER and GUIDER.

However, there were 17 test actions that could not be repaired by COSER but were repaired by METER or GUIDER. We found that 5 of these 17 test actions were from app S18, where some button widgets were not properly assigned any properties, thus making it difficult to acquire their external semantics and extract relevant internal semantics. This omission deviates from the standard and recommended Android programming practices. METER and GUIDER leveraged visual information to repair these. The remaining 12 test actions were from app S2, caused by mismatches between the external semantics and actual functionalities. With these mismatches, the external semantic matching was misled to produce incorrect repairs before even considering internal semantics. We consider these mismatches to be bugs in the apps themselves rather than a reflection of COSER's effectiveness. GUIDER leveraged visual information to repair these test actions. In conclusion, despite these instances, COSER managed to repair more test actions overall than either of the other four approaches, suggesting its superior effectiveness.

Comparison for efficiency. In total, it cost METER 679.6 minutes to repair all the scripts, and it cost AppTestMigrator and GUIDER 560.2 and 648.9 minutes to repair all the scripts, respectively. As to COSER, it took COSER 555.5 minutes to repair all the test scripts, which is 124.1, 4.7, and 93.4 minutes faster than METER, AppTestMigrator and GUIDER, respectively. Also, the average repairing time for a single test action with COSER was 9.56 seconds, which was 2.14, 0.08, and 1.61 seconds faster than METER (11.70 seconds), AppTestMigrator (9.64 seconds), and GUIDER (11.17 seconds), respectively.

In conclusion, the overall efficiency of AppTestMigrator and COSER is alike. Moreover, COSER outperforms METER and GUIDER in terms of efficiency. This is because the CV methods METER and

Table 3: Ablation study for external and internal semantics

External Semantic	Internal Semantic	$\#\mathcal{K}_c$	$\#\mathcal{A}_c$
✓		169(65%)	2266(65%)
	✓	114(44%)	1804(52%)
✓	✓	215(82%)	3121(90%)

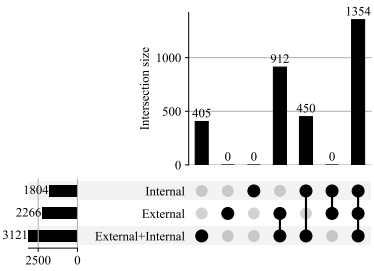


Figure 6: Partition of the test actions based on whether they can be correctly repaired by using different semantic types. Each column within the plot indicates the number of test actions that are correctly repaired by using the semantic type denoted by black dots but failed to be repaired by using the semantic type denoted by grey dots.

Table 4: COSER's effectiveness with different values of θ

θ	0.5	0.6	0.7*	0.8	0.9
$\#\mathcal{K}_c$	201(77%)	211(81%)	215(82%)	213(82%)	200(76%)
$\#\mathcal{A}_c$	2968(85%)	3049(88%)	3121(90%)	3036(87%)	2960(85%)

GUIDER use for visual matching are time-consuming and fragile. In contrast, COSER leverages pre-trained models to conduct external and internal semantic matching, leading to a more efficient matching process. Additionally, optimizations like caching semantic matching relations also contribute to COSER's efficiency.

4.3.4 RQ4: Factors affecting COSER's effectiveness. To better understand how the combination of external and internal semantics contributes to the repairing result, an ablation study was conducted. We devised two variants of COSER: one without the usage of external semantics (by omitting lines 7-8 of the repairing algorithm) and the other without the usage of internal semantics (by omitting lines 9-13 of the repairing algorithm). This experimental setup allows us to examine the individual impact of both semantic types on the overall script repair effectiveness.

As can be seen in Table 3, relying solely on external semantics made 169 (65%) test scripts and 2266 (65%) test actions be executed correctly while relying solely on internal semantics made 114 (44%) test scripts and 1804 (52%) test actions to be executed correctly on the updated versions. It can be observed that solely relying on only external or internal semantic information still achieved comparable results to the other approaches. The reason for this is that the version using external semantics is capable of repairing the test actions with their corresponding elements changed slightly, for example, those elements that stay on the same screen and have only minor modifications to their appearance. On the other hand, the other version only using internal semantics is capable of repairing complex situations with large GUI changes. The two versions are generally good at handling different scenarios.

Observations also indicate that solely using external semantics achieved slightly better results than solely using internal semantics. This is because external semantic matching is capable of addressing the match of those auto-generated elements, while internal semantic matching, as mentioned above, fails to match these elements.

Our detailed analysis reveals the different roles of both semantic types in the script repairing process. As illustrated in Figure 6, relying solely on external semantics repairs 912 unique test actions that can not be repaired by the other version. As mentioned above, this outcome primarily arises due to our current implementation's limitation in acquiring the internal semantics of auto-generated GUI elements. On the other hand, relying solely on internal semantics produces 450 unique test actions that can not be repaired by the other version. We found that these test actions are mostly related to relative major GUI changes.

Moreover, the combination of external and internal semantics not only covered all test actions solely using either of them but also enabled the repair of an extra 405 test actions. The combination of both semantic types allows COSER to cover all the scenarios that can be repaired solely using either semantic type. As a result, the probability of repairing more test actions for each test case is increased. Since test actions in a test case are sequential, the incorrect repair of earlier actions can have an impact on the repair of subsequent actions. Therefore, when more actions are correctly repaired, the probability of correctly repairing subsequent actions also increases. This finding highlights the effectiveness of combining both semantic types regardless of the extent of GUI changes, whether minor or major.

This analysis supports our argument that leveraging both external and internal semantics is far more effective than depending solely on one of them. In COSER, the external and internal semantics are well combined to repair obsolete test scripts. The combination of the two helps cover both simple and complex situations. As can be seen in Table 3, with the combination of external and internal semantics, the repairing result improves significantly.

Besides two types of semantics, we also investigated how different values of θ could affect the effectiveness. Table 4 shows the effects of various choices of the parameter θ. θ is the external semantic threshold; the default value for θ is marked with an asterisk (*). As can be seen from the table, the trend of the values of $\#\mathcal{K}_c$ and $\#\mathcal{A}_c$ resembles: they initially increase with the increment of θ and then decline. We can infer that the effectiveness of the repair process is related to the different values of θ. The farther θ from the default value, the poorer repair results COSER gets. This is because if θ becomes too small, more GUI elements are treated as matches through external semantics even if the matches are incorrect; if θ becomes too large, fewer auto-generated GUI elements are treated as matches through external semantics, COSER then resorts to internal semantics. As previously mentioned, internal semantics is not effective enough to handle these elements in our current implementation, thus leading to poorer results. We chose 0.7 as the appropriate default value as suggested in the table.

4.4 Threats to Validity

4.4.1 Construct Validity. The principal threats to construct validity in our study stem from the evaluation measures we adopted. In our experiments, we evaluate the effectiveness of the approaches by the number of test scripts and test actions that can be executed correctly after the scripts get repaired. The correct execution means

that the original test intention is maintained and correctly executed. Whether the test intention of a test action is preserved may vary according to different criteria. To reduce the risk, five postgraduates were recruited to manually review and check the repair results. All the students had more than two years of experience in mobile testing and did not participate in the design of COSER. A repair is considered correctly repaired only when all the postgraduates engaged have a consensus on that.

4.4.2 Internal Validity. The factor that may affect internal validity in the experiments is the possible defects in our implementation of COSER. To mitigate this threat, we reviewed and tested our implementation to reduce the risk of incorrectness.

4.4.3 External Validity. One major threat that may affect the external validity of the experiments is the applications we use may not reflect the broad spectrum of Android applications. To collect a diversified set of Android apps, we resorted to 13 papers on mobile testing from the top software conferences and selected 20 in total. Also, we plan to conduct larger-scale experiments to evaluate COSER more thoroughly in the future.

Another threat to external validity lies in the test scripts that COSER repaired in the experiments. To ensure that the preparation of test scripts is not biased in favor of COSER, we recruited postgraduates experienced in mobile development and testing to create these scripts. Test scripts written by other engineers or those generated by automatic tools might possess different characteristics, which could influence the behavior and effectiveness of COSER.

5 RELATED WORK

In this section, we review works closely related to COSER in GUI test repair and GUI test migration.

5.1 GUI Test Repair

The domain of GUI test script repair has drawn significant attention due to the user-friendly nature of GUI applications and the fragility of GUI test scripts. Memon and Soffa [23] first propose the idea of GUI test script repair and develop a model-based approach named GUI Ripper based on four user-defined transformations. Later, Memon et al. [22] develop a mechanism to obtain the application model through reverse engineering. Huang et al. [14] propose to use a genetic algorithm to generate new, feasible test cases as repairs to existing GUI test suites.

Besides model-based approaches, several white box approaches have also been developed for this task. Daniel et al. [5] propose to record GUI code refactorings as they are conducted in an IDE to repair test scripts. Grechanik et al. [11] propose a tool that extracts GUI changes by analyzing the source code and test scripts, generating repair candidates for GUI test scripts to be selected by testers. Fu et al. [9] devised a type-inference technique for locating GUI test script errors. Zhang et al. [38] combine dynamic and static analysis to generate recommendations for replacement actions when the GUI workflow changes. Gao et al. [10] propose a semi-automated approach named SITAR that relies on human input to improve the quality of the extracted models used in script repair.

In comparison with these works, COSER explores a different approach of using source code to extract intended behaviors for each GUI element as its internal semantics, facilitating the matching process. This approach enables COSER to effectively handle major GUI changes and achieve higher effectiveness.

Compared with desktop apps, research on GUI test repair for web and mobile apps has gained better results, mainly because the DOM tree of a web application's web page and the layout hierarchy of a mobile application provides valuable information to guide the repair of tests. Choudhary et al. [4] propose the WATER technique to repair GUI test scripts for a web application. When a test action fails, WATER repairs it by finding a replacement GUI element that shares at least one key property with the original element. Stocco et al. [31] propose the Vista technique to repair broken locators in GUI tests for web apps based on visual information. METER [27] establishes the matching relation between elements on app GUIs based on their visual appearance and uses it to guide the repair. GUIDER [36] exploits the structural and visual information of Android apps to build the matching relation between the widgets to repair scripts.

These works rely solely on the external semantics of GUI elements, disregarding the significance of internal semantics, which can limit their effectiveness particularly when dealing with major GUI changes. In contrast, COSER innovatively incorporates internal semantics and combines the two semantic types, resulting in a more efficient and effective production of correct repairs.

5.2 GUI Test Migration

GUI test migration is gaining more research interest. These techniques aim to migrate test scripts for specific functionalities to other applications which share similar functionalities. AppTestMigrator leverages a Word2Vec model [24] with a Window Transition Graph to migrate scripts based on textual similarity between GUI elements. It categorizes applications into several types and extracts some typical test cases for each category. Similarly, CraftDroid [20] utilizes textual information of GUI elements in conjunction with a UI Transition Graph to facilitate migration. These works also use semantic matching and show promising results in test migration.

However, these works rely solely on the external semantics of GUI elements and do not incorporate internal semantics. In our experiment, we chose AppTestMigrator as one of our comparison subjects. The results show that COSER has superior effectiveness over AppTestMigrator. In future work, we aim to explore the potential of adjusting our approach to accommodate the task of test migration, which will expand the applicability of our approach.

6 CONCLUSION

In this paper, we propose COSER—a novel approach that combines external and internal semantics of GUI elements to automatically repair GUI test scripts for Android apps. Experimental evaluation on 20 open-source Android apps suggests COSER's superior effectiveness and efficiency over state-of-the-art approaches METER, AppTestMigrator, and GUIDER.

ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China under Grant Nos. 62372227, 62232014, and 62032010.

REFERENCES

[1] Appium. 2023. Appium Documentation. http://appium.io/ [online, accessed 10-Jul-2021].

[2] Farnaz Behrang and Alessandro Orso. 2020. Test Migration between Mobile Apps with Similar Functionality. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 54–65. https://doi.org/10.1109/ASE.2019.00016

[3] Nana Chang, Linzhang Wang, Yu Pei, Subrota K. Mondal, and Xuandong Li. 2018. Change-Based Test Script Maintenance for Android Apps. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 215–225. https://doi.org/10.1109/QRS.2018.00035

[4] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (Toronto, Ontario, Canada) *(ETSE '11)*. Association for Computing Machinery, New York, NY, USA, 24–29. https://doi.org/10.1145/2002931.2002935

[5] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. 2011. Automated GUI Refactoring and Test Script Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering* (Toronto, Ontario, Canada) *(ETSE '11)*. Association for Computing Machinery, New York, NY, USA, 38–41. https://doi.org/10.1145/2002931.2002937

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-Travel Testing of Android Apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 481–492. https://doi.org/10.1145/3377811.3380402

[8] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky Test Detection in Android via Event Order Exploration. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/3468264.3468584

[9] Chen Fu, Mark Grechanik, and Qing Xie. 2009. Inferring Types of References to GUI Objects in Test Scripts. In *2009 International Conference on Software Testing Verification and Validation*. 1–10. https://doi.org/10.1109/ICST.2009.12

[10] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42, 2 (2016), 170–186. https://doi.org/10.1109/TSE.2015.2454510

[11] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*. 408–418. https://doi.org/10.1109/ICSE.2009.5070540

[12] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[13] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. *Understanding and Detecting Callback Compatibility Issues for Android Applications.* Association for Computing Machinery, New York, NY, USA, 532–542. https://doi.org/10.1145/3238147.3238181

[14] Si Huang, Myra B. Cohen, and Atif M. Memon. 2010. Repairing GUI Test Suites Using a Genetic Algorithm. In *2010 Third International Conference on Software Testing, Verification and Validation*. 245–254. https://doi.org/10.1109/ICST.2010.39

[15] Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek. 2020. *Automated Construction of Energy Test Oracles for Android.* Association for Computing Machinery, New York, NY, USA, 927–938. https://doi.org/10.1145/3368089.3409677

[16] JavaParser. 2023. Tools for your Java code. https://javaparser.org/ [online, accessed 10-Jul-2021].

[17] Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android Testing: They Matter. In *Proceedings of the 1st International Workshop on Advances in Mobile App Analysis* (Montpellier, France) *(A-Mobile 2018)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3243218.3243219

[18] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. 2013. Comparing the Maintainability of Selenium WebDriver Test Suites Employing Different Locators: A Case Study. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation* (Lugano, Switzerland) *(JAMAICA 2013)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/2489280.2489284

[19] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 161–171. https://doi.org/10.1109/ICST.2017.22

[20] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2020. Test Transfer across Mobile Apps through Semantic Mapping. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 42–53. https://doi.org/10.1109/ASE.2019.00015

[21] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-Wise Testing for Android Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 268–278. https://doi.org/10.1145/3338906.3338980

[22] Atif M. Memon. 2008. Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. *ACM Trans. Softw. Eng. Methodol.* 18, 2, Article 4 (nov 2008), 36 pages. https://doi.org/10.1145/1416563.1416564

[23] Atif M. Memon and Mary Lou Soffa. 2003. Regression Testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Helsinki, Finland) *(ESEC/FSE-11)*. Association for Computing Machinery, New York, NY, USA, 118–127. https://doi.org/10.1145/940071.940088

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems*, C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.), Vol. 26. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf

[25] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. 2016. Release Practices for Mobile Apps – What do Users and Developers Think?. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 552–562. https://doi.org/10.1109/SANER.2016.116

[26] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. 2020. *Static Asynchronous Component Misuse Detection for Android Applications.* Association for Computing Machinery, New York, NY, USA, 952–963. https://doi.org/10.1145/3368089.3409699

[27] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2022. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering* 48, 3 (2022), 910–929. https://doi.org/10.1109/TSE.2020.3007664

[28] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 3982–3992. https://doi.org/10.18653/v1/D19-1410

[29] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data Loss Detector: Automatically Revealing Data Loss Bugs in Android Apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/3395363.3397379

[30] Robotium. 2023. Android UI Testing. https://github.com/RobotiumTech/robotium [online, accessed 10-Jul-2021].

[31] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 503–514. https://doi.org/10.1145/3236024.3236063

[32] Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. 2019. Event Trace Reduction for Effective Bug Replay of Android Apps via Differential GUI State Analysis. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1095–1099. https://doi.org/10.1145/3338906.3341183

[33] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[34] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/3377811.3380382

[35] Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-Inspired, Unsupervised Inference of Perceptual Groups of GUI Widgets from GUI Images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/

3540250.3549138

[36] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. 2021. GUIDER: GUI Structure and Vision Co-Guided Test Script Repair for Android Apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/3460319.3464830

[37] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-Entry Testing of Android Applications by Constructing Activity Launching Contexts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 457–468. https://doi.org/10.1145/3377811.3380347

[38] Sai Zhang, Hao Lü, and Michael D. Ernst. 2013. Automatically Repairing Broken Workflows for Evolving GUI Applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 45–55. https://doi.org/10.1145/2483760.2483775

[39] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 275, 15 pages. https://doi.org/10.1145/3411764.3445186