Science of Computer Programming 242 (2025) 103261

Contents lists available at ScienceDirect

Science of Computer Programming

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

Integrating behavioral semantic analysis in usage-based equivalent tests generation for mobile applications

Shuqi Liu [a], Yu Zhou [a,*], Huiwen Yang [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 generation
Functional equivalence
Semantic information
Screen intimacy

ABSTRACT

Graphical user interface (GUI) testing is crucial to ensure the expected behaviors of mobile applications (apps). The burgeoning automated usage-based testing seeks to generate simulated human interactions tailored to functional features of apps. However, the difficulties in understanding UI semantics, along with the multiple implementation alternatives, significantly restrict the ability to exercise a specified usage. In this paper, we propose GUEST (**G**enerating **U**sage-based **E**quivalent Te**ST**s), which automates the generation of multiple equivalent tests for GUI usage to help developers more thoroughly test mobile apps' features. GUEST integrates textual information from state pages with the UI structure to express operational GUI widgets with semantic information. It leverages the semantic coverage of edge links within the state transition graph of state-machine encoding for the usage to match canonical screens for the current state page. To exploit behavioral semantics, GUEST treats the state transition graph as a social network and performs centrality analysis to identify key canonical screens in the state transition graph. By utilizing the intimacy between key screens and candidate widgets' reachable screens, GUEST grants higher priority to frequently used and more accessible actions. We evaluate GUEST on desired usages across 22 popular apps and the results reveal that GUEST can successfully exercise the desired usage in 88% of the tests and outperform the state-of-the-art baseline method in both screen and widget classification performance.

1. Introduction

In recent years, mobile applications (apps) have been pervasively used in daily life. Mainstream app stores such as the Apple AppStore [1] and Google Play [2] offer millions of apps across multiple categories. With the increasing significance of mobile apps, developers are increasingly emphasizing the maintenance of high quality and the fulfillment of user expectations in their apps. To alleviate the time and labor-intensive manual testing, automated Graphical User Interface (GUI) testing is employed to examine the behavior of mobile apps [3–8].

Some existing GUI testing techniques, such as random or search-based methods, aim to achieve high code coverage or discovering more defects [3,9–13]. However, these methods may fail to address developers' practical needs. Mobile apps typically revolve around

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

https://doi.org/10.1016/j.scico.2024.103261
Received 25 July 2024; Received in revised form 21 December 2024; Accepted 22 December 2024

Available online 30 December 2024

specific features (functionalities), and user interactions reveal their usage habits. Events generated by these methods often lack a tight correlation between interaction sequences, which may overlook the examination of critical issues in functional operations [14–17].

Developers prefer usage-based test cases [18] that closely relate to app use cases or features [19]. Such test cases consist of a sequence of events that simulate user operations, such as "adding an item to bookmarks". This approach effectively supports practical testing goals, such as regression testing or performance testing, which usually target common app use cases [19].

Test reuse has proven to be an effective way for leveraging existing usage-based test cases to test another app with similar functionality. In light of this, previous work [20–27] has focused on generating event sequences for target apps by reusing test cases from source apps. Nevertheless, test reuse poses several limitations in practical adoption, including reliance on manually crafted source tests and limited applicability across app categories [28]. Manually-written tests tend to focus on typical implementations of functionality, often neglecting alternatives user interaction paths. For instance, a developer might test adding a news article to bookmarks directly from the recommendation page but overlook adding it through the category menu. Furthermore, several methods [21,26,27] require complex program analysis, making them inapplicable when the source code of the app is unavailable.

The recent tool Avgust [28] overcomes the limitations of test reuse by leveraging a synthetic, app-agnostic state-machine encoding to generate usage-based tests for a new target app. It synthesizes a generic state machine intermediate-representation (IR) model using only screenshots and video frames from screen recordings, simplifying the creation of test scenarios. However, it still presents technical challenges that hinder test generation. First, Avgust leverages image classifiers to align screen and GUI widgets with the IR model, with performance heavily dependent on classifier training data. Unseen text and structural layout may interfere with classification results. Second, Avgust groups screen image into a standard representation based on visual and textual features, defining a typical classification known as the canonical screen [28,29]. Indeed, neglecting widget mapping can cause the classifier to fail to recommend the correct canonical screen, affecting widget selection. Third, Avgust focuses on the current screen when recommending triggerable widgets, without considering the broader contextual information or possible screen transitions. This lack of contextual understanding reduces its ability to fully capture app behavior, especially when state transitions depend on interactions across different screens.

To facilitate more effectively and thoroughly testing the functionality of mobile apps, we propose GUEST (**G**enerating **U**sage-based **E**quivalent te**ST**s), a novel lightweight approach that semantically understands app behavior to generate usage-based tests. GUEST mitigates the limitations of Avgust in three key aspects. First, rather than relying on neural models for image understanding, GUEST directly captures semantic information from the dynamically running app's GUI pages. The valuable information about the GUI widgets and page structure enables a more accurate and adequate characterization of the current screen and operable widgets. Second, GUEST determines the canonical screen by calculating the semantic overlap between the current screen and the edges link (in and out edges) of the canonical screen in the IR model, providing a more precise identification of screen states. Third, GUEST regards the state transition graph of state-machine encoding for the usage as a complex social network and applies network analysis to depict the graph semantics of app behavior. By conducting centrality analysis on each state transition graph, GUEST identifies key screen nodes within the network. It then identifies candidate event sets by measuring the intimacy (i.e., frequency of communication) between reachable screens and these key screens. This strategy allows GUEST to prioritize frequently used and more accessible widgets with higher intimacy-aware scores. Additionally, GUEST can generate multiple equivalent tests for the same feature, thus providing greater diversity and relevance in examining the behavior of the app.

We conduct comprehensive experiments to evaluate the effectiveness of our approach. We apply GUEST to generate tests for 18 usage scenarios across 22 popular apps available on Google Play.[1] To assess the performance of the test generation, developer interactions with GUEST's proposals are simulated by executing each usage scenario on three relevant apps. The experimental results demonstrate that GUEST is capable of generating at least two high-quality tests for each test scenario. Compared to the representative baseline approach, GUEST achieves superior performance in its classifier with higher coverage of screens and transitions and provides more accurate recommendations.

Our main contributions are summarized as follows.

- We propose GUEST,[2] a novel approach capable of generating usage-based equivalent tests to examine the behavior of mobile apps thoroughly without relying on pre-existing tests written by domain experts.
- GUEST utilizes network analysis to represent the graph semantics of dynamic behavior contained in the state-machine encoding of the usage, and applies intimacy analysis between key screens and widgets' reachable screens facilitates recommending the frequently used and more accessible interactions.
- We conduct comprehensive experiments to demonstrate the performance of GUEST on common usage scenarios across 22 popular apps. The evaluation results confirm the effectiveness of GUEST in generating usage-based equivalent tests.

The remainder of this paper is organized as follows. Section 2 and Section 3 give the necessary background and the motivation of our work. Section 4 introduces the overview of our approach and then details its components. Section 5 reports and discusses the evaluation results. Section 6 discusses potential limitations and threats to validity. Section 7 describes the related work, and Section 8 concludes the paper.

[1] https://play.google.com/.

[2] https://github.com/liushuqi-2022/GUEST-repo.

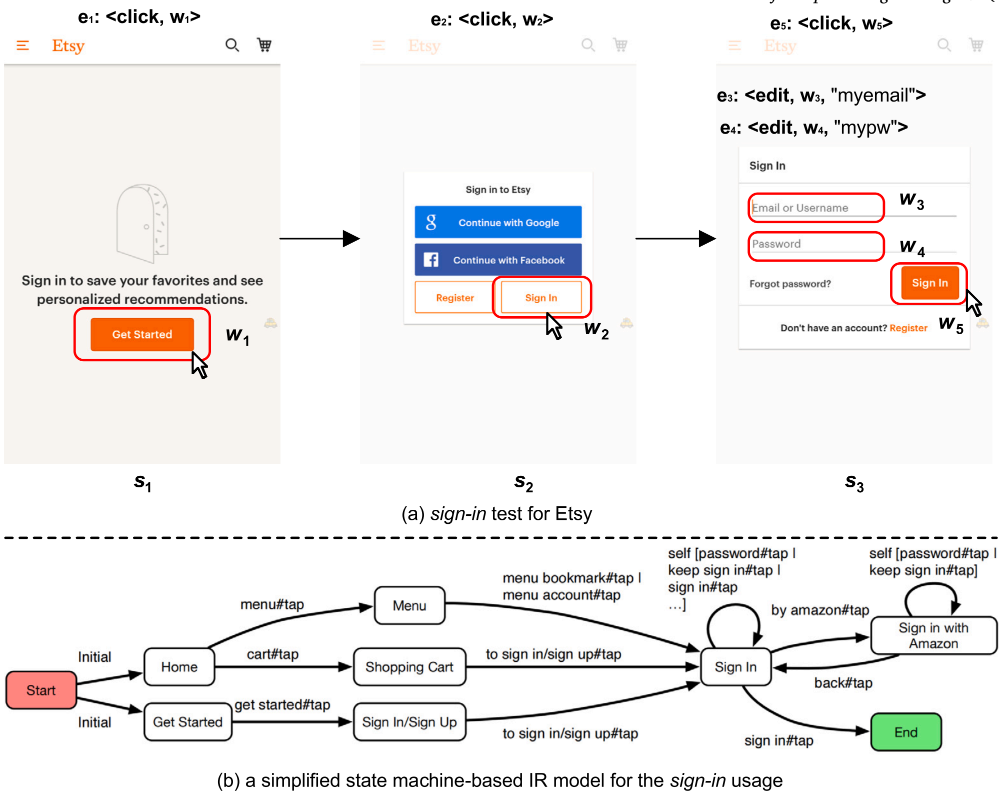


Fig. 1. A simplified state machine-based IR model (b) for the *sign-in* usage merged from the *6pm* and *Etsy* apps. (a) corresponds to a example of *sign-in* test from the *Etsy* app. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

2. Background

2.1. Usage-based test

Typically, users interact with mobile apps through GUIs, which contain various widgets, the fundamental elements of the interface. These widgets can be triggered by user actions, such as clicking buttons or entering text in text boxes, which are referred to as GUI events. A GUI event is represented as $e = \langle t, r, o \rangle$, where $e.t$ denotes the event type (e.g., click, edit), $e.r$ is a function that retrieves the target widget of the operation, and $e.o$ includes optional data related to the event operation, such as the string for `edit`. A usage-based test consists of a sequence of GUI events designed to execute a specific functionality in the app, such as “sign-in”.

Fig. 1(a) illustrates the “sign-in” test in the *Etsy*[3] app, represented by the event sequence $\{e_1, e_2, e_3, e_4, e_5\}$. Each screenshot shows the event operation on the corresponding GUI page, where the red box marks the widget being operated. For instance, e_1 represents a click on the widget w_1 with the rendered text “Get Started” on the GUI page s_1.

2.2. Model-based GUI testing

For GUI testing, various model-based methods have been proposed to ensure app quality. Stoat [12] employs a stochastic model learned from the app to optimize test suite generation. Ape [10] dynamically evolves its abstraction criterion using runtime information through a decision tree and generate UI events via a combination of random and greedy depth-first exploration strategies. In contrast, ComboDroid [13] acquires test cases through human input or from a GUI model built via GUI exploration. The data flows among these obtained tests are analyzed and combined to obtain the final tests. These model-based methods depict diverse behavioral patterns of apps, thereby improving test coverage.

Multiple events work together to accomplish the specific usage of functionality in an app. The GUI model is primarily focused on connecting discrete events to represent the behavior of a mobile app, which captures essential characteristics and interactions. A GUI model is essentially a Finite State Machine (FSM), represented by a tuple (S, s_0, F, E, T). It consists of a set of states S corresponding to specific screens or pages in the user interface (UI) of app, an initial state s_0 $(s_0 \in S)$, a set of desired final states F $(F \subset S)$, a set of events E, and a set of transitions between states T. Each transition $t \in T$ is defined as (s_s, e, s_t), where s_s and s_t are source and

[3] https://play.google.com/store/apps/details?id=com.etsy.android.

target states, and e represents the event that triggers the transition. The state transition graph of the GUI model is a labeled directed graph, where nodes correspond to the states of the app and edges represent event-driven transitions between these states.

To capture common usage patterns across apps, we leverage a state machine-based IR model, an FSM-based structure that summarizes key interaction flows. Fig. 1(b) presents a simplified state machine-based IR model for the "sign-in" usage case, extracted from the *6pm*[4] and *Etsy*[5] apps. This model summarizes various pathways to access the *Sign In* page, including through the menu, shopping cart, or get-started channels. Each state and transition is defined at an abstract level, allowing for generalization across different apps. This example comes from Avgust [28], which constructs a state machine-based IR model by merging different implementations across multiple apps for a specific usage.

2.3. Social network centrality and intimacy analysis

A social network is typically depicted as a graph, denoted by $G' = (V', E')$, where V' and E' represent the sets of nodes and edges, respectively. The nodes set V' indicates the entities within the social network, such as individuals or organizations, each node representing a participant. Meanwhile, the edges set E' indicates the connections or relationships between these nodes, depicting various social interactions among them. A mobile app usually consists of a set of screens (or pages). User actions on these screens trigger transitions between screens. If we consider screens as actors and their transition relationships as social interactions, the state transition graph of an app can be regarded as a social network.

The notion of centrality, originating from social network analysis, was developed to measure the importance of nodes within a network. Centrality measures are widely used in network analysis and have been applied in various domains, including biological networks [30], co-authorship networks [31], affiliation networks [32], etc. Researchers have introduced several centrality measures specific to social networks, for example:

1) Degree centrality [33] represents the number of direct connections a node has, which helps identify highly connected nodes in the network. Formally, the degree centrality $C_d(v_i)$ of a node v_i can be defined as:

$$C_d(v_i) = \frac{deg(v_i)}{n-1}$$

where $deg(v_i)$ is the number of direct connections of v_i, and n represents the number of all nodes.

2) Closeness centrality [33] quantifies the proximity of a node to others. It is represented by the average length of the shortest path between a node and the others in the network. Specifically, the closeness centrality $C_c(v_i)$ of node v_i can be calculated as:

$$C_c(v_i) = \frac{n-1}{\sum_{j=1, j\neq i}^{n} d(v_i, v_j)}$$

where $d(v_i, v_j)$ represents the shortest path distance between node v_i and node v_j.

3) Katz centrality [34] considers both direct and indirect connections to assess a node's influence. The Katz centrality $C_k(v_i)$ for node v_i is:

$$C_k(v_i) = \alpha \sum_{j=1}^{N} A_{i,j} C_k(v_j) + \beta, (\alpha < \frac{1}{\lambda_{max}})$$

where A represents the adjacency matrix of the network with eigenvalue λ, α and β are scaling factors. α controls the importance of direct connections, while β controls the influence of indirect connections. In general, the Katz centrality values capture the cumulative influence of a node and its neighbors on the entire network.

4) Harmonic centrality [35] is a variant of closeness centrality that addresses the challenges posed by disconnected graphs. It considers the sum of reciprocal distances rather than the average, allowing isolated nodes to be included in the calculation. Specifically, the harmonic centrality $C_h(v_i)$ of node v_i is defined as:

$$C_h(v_i) = \frac{\sum_{j=1, j\neq i}^{n} \frac{1}{d(v_i, v_j)}}{n-1}$$

5) Betweenness centrality [36] quantifies the importance of a node in the network. It is measured based on the frequency of the node's shortest path connecting other nodes. Nodes with high betweenness centrality usually act as bridges or intermediaries in the network. The betweenness centrality $C_b(v_i)$ of the node v_i can be expressed as:

$$C_b(v_i) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{g_{ij}(v_i)}{g_{ij}}$$

where g_{ij} indicates the total number of shortest paths between nodes i and j, and $g_{ij}(v_i)$ represents the number of those shortest paths from i to j passing through node v_i. In a nutshell, centrality analysis preserves the details of the graph and can potentially reflect the structural characteristics and behaviors.

[4] https://play.google.com/store/apps/details?id=com.zappos.android.sixpmFlavor.

[5] https://play.google.com/store/apps/details?id=com.etsy.android.

Table 1
The frequency of communication between *Get Started* state and key nodes.

Usage	*Sign In*	*Setting*
sign-in	8	1
terms	3	11

Intimacy indicates the communication frequency between two individuals in a social network. Recently, an Android malware detection method IntDroid [37] aims to reduce the overhead of static analysis and graph-matching by computing intimacy between important nodes and sensitive APIs in the function call graphs. This provides precise graph details for distinguishing malicious and benign apps.

In mobile apps, users typically access specific crucial screens, serving as expected entry points for executing desired usage. For instance, in Fig. 1, screens like *Menu* and *Shopping Cart* are vital for executing the "sign-in" usage. In our approach, we regard the state transition graph of a state machine-based IR model as a social network, and adopt centrality analysis to mine key screens within the usage scenario. Examining app behavior involves dynamic exploration, where the app's current state influences subsequent state transitions. Intimacy analysis between key and reachable screens prioritizes candidate events, allowing us to understand the dynamic behavior semantic implied in state-machine models and suggest actionable operations for generating usage-based equivalent tests.

3. Motivation

In this section, we present a motivating example involving two state machine-based IR models for the "sign-in" and "terms" usages. The usage-based IR model records the behaviors of multiple apps to accomplish a certain usage. The purpose of the GUEST is to generate equivalent usage-based tests for a previously unseen app by using the usage-based IR model for the desired usage.

We analyzed the state transition graph of state-machine encoding for the two usages. The state transition graph can be treated as a social network, where the nodes represent states and the edges represent transitions between them. For each state transition graph, a key node is selected based on the degree of the node, which is a common node in the both graphs. The key nodes of the state transition graphs for the "sign-in" and "terms" usages are identified as the *Sign In* and *Setting* states. Users generally reach these two app states to achieve specific functionalities.

Table 1 lists the communication frequencies between the state *Get Started* and the key nodes *Sign In* and *Setting*. In two different usage scenarios, the communication relationship between the state *Get Started* and the respective key nodes is different. It can be observed that, for the same state, the communication with the key nodes is frequent in different networks. These key nodes play an important role in the dynamic communication of the network.

As shown in the example, for the same app, the state of the app in different target usages has inconsistent communication with key nodes. This makes it difficult to generate usage-based tests for the app. In the following sections, we will describe in detail how the GUEST identifies key screens in the state navigation graph and generates test scenarios based on the communication relationship between app states and those key screens.

4. Approach

GUEST acts as a human-in-the-loop tool to assist developers in creating usage-based tests by suggesting input events. Since app functionality can be accessed in various ways, multiple test paths are possible for each usage scenario. By suggesting events for a particular screen, GUEST enables flexible test generation aligned with user behavior. Fig. 2 gives an overview of GUEST. It has three types of inputs, i.e., the desired usage scenario (defining the expected functionality of the app), a database of the state machine-based intermediate presentation (IR) models for multiple usage scenarios, and the target app for testing. To thoroughly test the desired usage scenarios, GUEST follows these steps to generate equivalent usage-based tests. (a) GUEST performs network analysis on the IR model corresponding to the desired usage retrieved from the IR models database to obtain key canonical screens (Section 4.1, Usage-Based Network Analysis). (b) GUEST extracts GUI information from the current state of the running target app (Section 4.2, GUI Information Extraction). (c) Finally, GUEST assists developers to generate usage-based equivalent tests by suggesting top-k actions on the current screens of the target app. It iteratively aligns actionable widgets with the canonical widgets of the IR model, and utilizes intimacy analysis to prioritize frequently used and more user-friendly widgets (Section 4.3, Test Scenario Generation).

4.1. Usage-based network analysis

GUEST leverages state machine-based IR models to assist developers in generating usage-based equivalent tests for target apps. Given a desired usage (e.g., "sign-in"), it retrieves the corresponding IR model from the database, and uses it to guide test generation. To facilitate the recommendation of operations consistent with user habits, GUEST extracts semantic information contained in the retrieved IR model by performing network analysis. Next, we describe the model retrieval and network analysis in detail.

Model Retrieval. To retrieve the relevant state machine-based IR model for a given usage, Model Retrieval utilizes the IR models database [38] built via Avgust [28], chosen for its strong generalization and reusability. This database covers state-machine-based IR

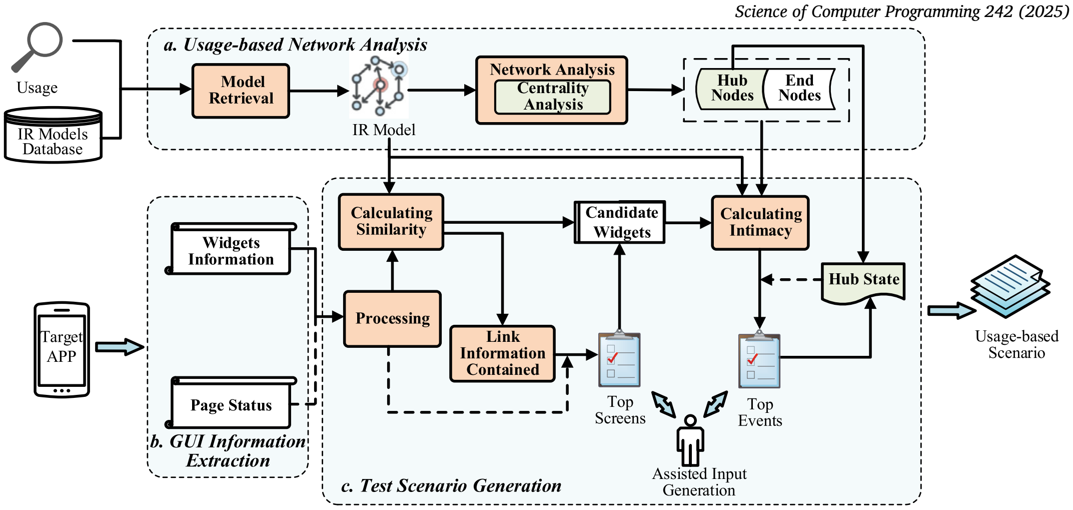


Fig. 2. The workflow of GUEST.

models for 18 types of usages, such as registration and adding items to a shopping cart, synthesized from diverse behaviors across 18 related apps, each learning from others' behavior. It captures potential usage pathways, effectively guiding test generation for new apps.

For example, given the desired usage "sign-in", the model retrieval provides the corresponding IR model, similar to what is shown in Fig. 1(b). This IR model captures the "sign-in" usage pattern through an FSM design with seven screen states, where transitions are triggered by user actions. Each state represents a canonical screen independent of specific apps, while each transition signifies a user interaction with a canonical widget. For example, the canonical screen "*Get Started*" is an abstract representation of commonly occurring screens. Similarly, canonical events, such as "sign in", correspond to user interactions represented by events e_2 and e_5, which are typical during the sign-in process.

By leveraging the IR model, GUEST assists developers in exploring different manners to accomplish the usage "sign-in" in the target app. This includes available through the *Menu*, *Shopping Cart*, or *Sign In/Sign Up* options within the *get started* area. For example, in an unseen app like *Aliexpress*,[6] a user might click buy button from the shopping cart, prompting a login page where he/she complete the login by entering their password and clicking the sign-in button. This helps understand user interactions during the "sign-in" process and supports building adaptive interfaces to improve user experiences. To generate more realistic tests that align with user behavior, GUEST analyzes the implied semantics within the transition graph of the IR model.

Network Analysis. GUEST regards the state transition graph of the IR model as a social network, and uses network analysis to identify the most important nodes, which help guide users to interact with different intentions.

To accomplish the desired usage of target app, it is crucial to first identify the entry point for the specific usage. GUEST primarily focus on two key types of screen nodes, i.e., hub screens and end screens. Hub screens play a crucial role in covering the critical functional implementation paths within the app, ensuring that test cases can contain pivotal interactions. On the other hand, taking end screens as key nodes guides generated tests towards successfully accomplishing the expected usage.

Centrality measures the importance of nodes in a network, which facilitates the identification of highly connected nodes (called hub nodes). GUEST performs centrality analysis to identify hub screen states in the specific IR model for the desired usage. Section 2.3 discusses the diverse range of centrality measures available for the network. Regarding centrality methods, five commonly used centrality metrics are selected: betweenness centrality, degree centrality, closeness centrality, katz centrality, and harmonic centrality. In the field of social network analysis, measuring the importance of a node often involves combining various centrality measures. With these centrality metrics, we address the above-mentioned challenge by constructing comprehensive centrality to obtain hub screens in two steps, as shown in Fig. 3.

Step 1: Candidate screens pre-filtering. Since hub screens control network communication and connect different screens, key screens should exist within these center locations to facilitate coordinated interactions among various features. Therefore, we pre-filter out all possible hub screens as candidate screens. Betweenness centrality is used to evaluate all screen nodes, as it quantifies how well a node acts as a bridge in state transitions. Screens with a betweenness score of zero are excluded from the candidate set.

Step 2: Centrality ranking. The number of identified candidate screen nodes is large. In fact, within the network, only a few nodes function as hub nodes. To determine the final hub screen nodes, we further measure the centrality of the candidate screens. Various centrality measures provide distinct perspectives on node importance. To achieve a more comprehensive measure for these candidate screens, we combine degree centrality, closeness centrality, katz centrality, and harmonic centrality metrics. Formally, the

[6] https://play.google.com/store/apps/details?id=ru.aliexpress.buyer.

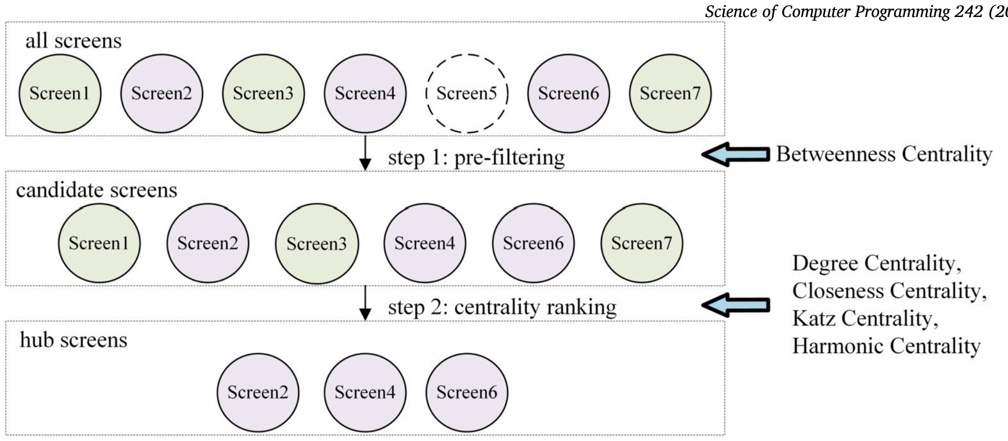


Fig. 3. Two steps in comprehensive centrality analysis.

comprehensive centrality $C_{com}(v_i)$ for the screen node v_i is calculated by averaging the values obtained from these four centrality metrics, denoted as:

$$C_{com}(v_i) = \frac{C_d(v_i) + C_c(v_i) + C_k(v_i) + C_h(v_i)}{4} \tag{1}$$

where $C_d(v_i)$, $C_c(v_i)$, $C_k(v_i)$, and $C_h(v_i)$ denote the degree centrality, closeness centrality, katz centrality, and harmonic centrality of v_i, respectively. Since high centrality represents greater importance within the network, we identify the candidate nodes with top-k comprehensive centrality scores as the hub screens. To investigate the impact of different numbers of hub screens on recommended events, we explore $k = 1$, 2, and 3. Taking Fig. 3 as an example, there are 7 different screen nodes in the state transition graph of the GUI model. First, we calculate the betweenness centrality for all screen nodes, filtering out *Screen*5 due to its value of 0. This lead to the candidate hub screens set {*Screen*1, *Screen*2, *Screen*3, *Screen*4, *Screen*6, *Screen*7}. Next, we combine Eq. (1) to measure the comprehensive centrality of these candidates and sort them in descending order. Assuming $k = 3$, the first three screen nodes with the highest centrality scores, *Screen*2, *Screen*4 and *Screen*6, will be designated as hub screens. These hub screens will work with the end nodes to prioritize possible candidate events for test generation.

4.2. GUI information extraction

GUEST suggests the next possible operations to accomplish the desired usage based on the current state of the target app. This necessitates GUEST to accurately and comprehensively extract GUI information that describes the current state. The diversity of app designs poses challenge in extracting representative details from various app states. Therefore, GUEST focuses on two aspects of GUI information extraction, i.e., widgets information and page status. On the one hand, we extract the inherent meaning of all operable widgets on the current screen, which helps identify actionable steps associated with these widgets. On the other hand, we characterize the GUI page to enhance the understanding of its functionalities. GUEST relies on Appium,[7] which utilizes Android's Accessibility API and UIAutomator [39] to build UI hierarchy file for deriving both types of information.

Widgets Information. Operable widgets, such as buttons and text fields, are crucial for user interaction. Widget information uniquely identifies elements on the page, enabling actionable steps for each widget's functionality. GUEST captures the UI layout hierarchy in XML format through the `driver.page_source`, which includes details about all widgets in the app's current state, such as `resource-id`, `text`, and `bounds` attributes. The UI layout is represented as a tree, with each node corresponding to a widget. The extraction of widget information involves three operations: (i) obtaining descriptive attributes for each widget, such as `class`, `content-desc`, `clickable`, `text`, and `resource-id`; (ii) extracting local structural information from the UI layout tree, including parent and child nodes, to determine the semantics of intermediate elements in the layout structure; and (iii) defining the cropping box using the boundary values representing the pixel coordinates of the widget on the screen, and extracting the specified region from the screenshot to create a vivid image of the widget.

For example, a screenshot of the login page of the *Wish* app and its UI layout three is depicted in Fig. 4. The extracted information for the "Sign In" button in the current app state in Fig. 4 is {"Text": "Sign In", "class": android.widget.TextView, "Clickable": true, "Bounds": [42,1006][1038,1111], "resource-id": "com.contextlogic.wish:id/sign_in_fragment_sign_in_button", "
numberOfChildren": 0, "numInParentLayout": 1, "screenshotPath": 1-SignIn
/wish/42_1006_1038_1111.png}.

Page Status. Page status provides a macro-level semantic overview of the current state of the app, providing an overall understanding of its functionality. It contains two key aspects: the activity name of the page and representative attribute textual information of all

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

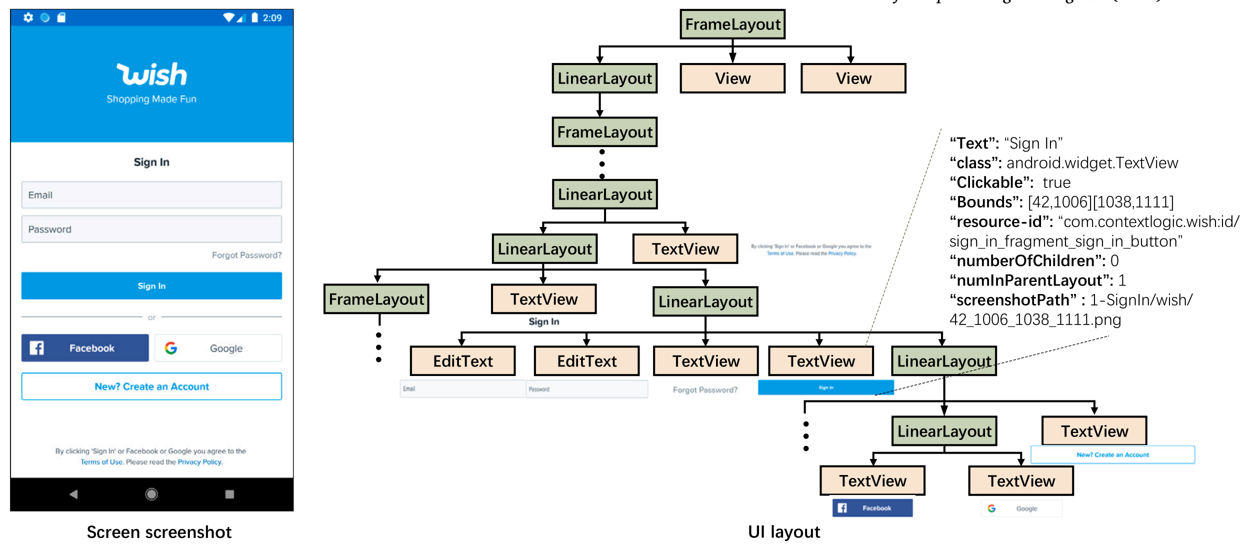


Fig. 4. UI state captured from Wish login page. The UI layout hierarchy elements highlighted on the right correspond to screenshot shown on the left.

operable widgets. The comprehensive API provided by Appium[8] facilitates the control of mobile devices and apps. Among its feature, the `Activities` class includes a `current_activity` property that enables the querying of the activities currently running on the device. GUEST utilizes this property to dynamically capture the current activity, which accurately identifies the state of the app.

4.3. *Test scenario generation*

The purpose of test scenario generation is to help developers create usage-based tests for the target app. Algorithm 1 outlines the iterative process for generating test scenarios (Lines 2-18) through GUEST. The process involves iteratively triggering widget based on the current state of the target app to accomplish usage features (Lines 5-15). First, extracted operable widgets information w_{tb} is adjusted based on the UI layout of the status page, and the page status s_{page} is updated accordingly (Line 6, details in Section 4.3.1). Second, GUEST measures the semantic similarity between candidate canonical screens in the IR model and the current state, recommending the top canonical screens for developers to select (Line 7, details in Section 4.3.2). Third, GUEST recommends the operations that are highly intimated with key nodes (i.e., hub nodes or end nodes) from the corresponding candidate widgets $widgets_{can}$ of the selected $screen$ (Lines 8-12, details in Section 4.3.3). Finally, GUEST executes the selected event, updates the app state, and adds the event to the explored test t_{tn} (Lines 13-14). This continues until the target usage scenario is achieved or the maximum number of events per generated test is reached (Line 5).

Fig. 5 demonstrates how GUEST assists developers by suggesting the top-k canonical screens that match the current state and recommending possible events. The green text indicates the developer's interactions with GUEST, guiding the generation of test scenarios. Each sequence of triggered events at different states forms a usage-based test. Subsequent sections detail how this process is implemented.

4.3.1. *Processing*

In this stage, we first preprocess the extracted GUI information to facilitate state matching and event generation. As mentioned in Section 4.2, the GUI information includes both widgets information and page status. The goal is to fully express the semantic information of operable widgets.

Due to the diversity of certain UI designs, such as the use of custom controls or complex layouts, extracting widget information solely from XML can result in the loss of crucial semantic information. For example, in Fig. 6, the layout tree of the news module shows a clickable element with type `ViewGroup`, which is responsible for arranging its child elements. However, among these child elements, only the element with rendered text of "Autos" of type `TextView` is clickable, while others are not, despite containing key news information. In such cases, custom controls or complex layouts may obscure the semantic relationship between interactive elements and their content. Therefore, directly using the semantic information of clickable events may not be suitable for handling these complex layouts. To address this, we adapt the semantic information of operable widgets, which are intermediate nodes defining the structural layout among elements, in combination with the layout tree of the page.

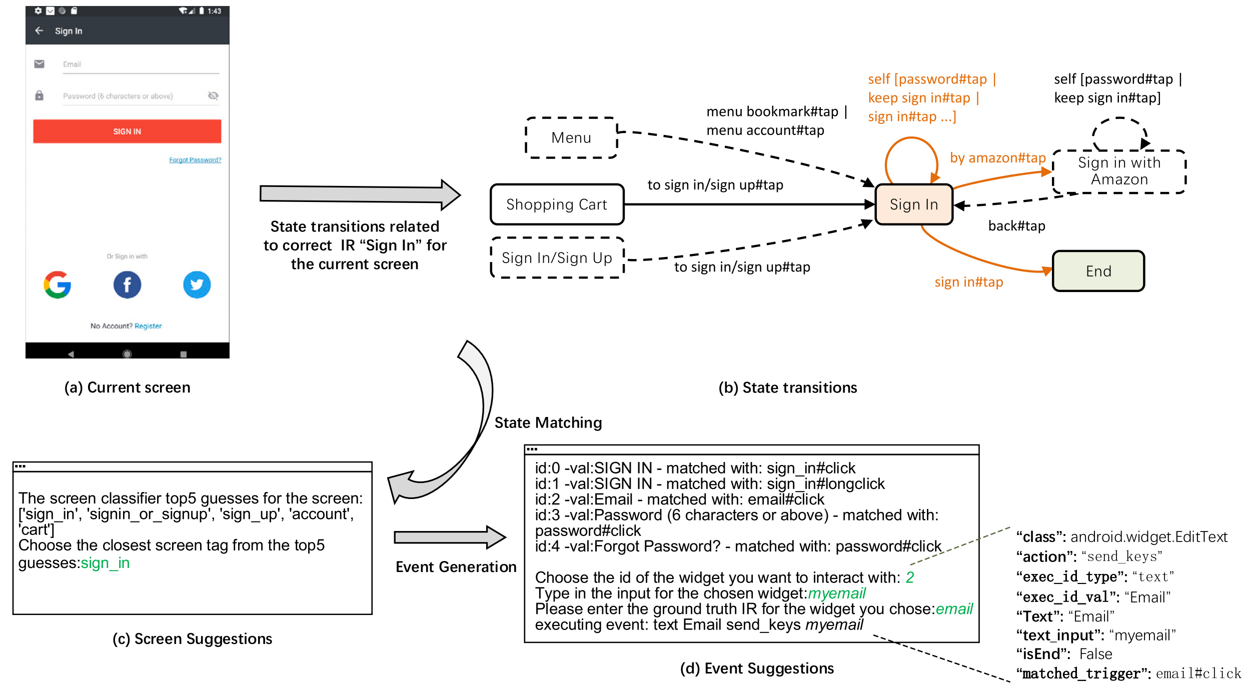
Operable widgets, particularly those of types such as `ViewGroup`, `Linear Layout`, `FrameLayout`, `LinearLayoutCompat`, `RelativeLayout` and `Tab`, are categorized as intermediate elements. These intermediate elements typically have at least two child nodes, including at least one of the type `TextView`. Moreover, in the layout structure, there is usually an element with the longest

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

Algorithm 1 Test Generation.

Input: target app a_t, the IR model IR for the desired usage, hub nodes of IR model n_{hubs}, end nodes of IR model n_{ends}, desired number of generated tests n_{Max}, widgets on the current state of target app w_{tb}, page status of the state page s_{page}
Output: usage-based tests $t_u = \{t_1, t_2, \ldots\ldots, t_{tn}\}$

```
 1: Initialize: $t_u = \emptyset$, $v_{hs} = \emptyset$, $tn = 1$, $t_{tn} = \emptyset$
 2: while $tn < n_{Max}$ do
 3:     $IsHub = False$
 4:     $IsEnd = False$
 5:     while $len(t_{tn}) < Max_{Action}$ or not $IsEnd$ do
 6:         $w'_{tb}, s'_{page}$ = getProcess($w_{tb}$, $s_{page}$)
 7:         $screen, widgets_{can}$ = getScreen($w'_{tb}$, $IR$, $s'_{page}$ )
 8:         if not $IsHub$ then
 9:             $event$ = getEvent($n_{hubs}, widgets_{can}, IR, v_{hs}, screen$ )
10:         else
11:             $event$ = getEvent($n_{ends}, widgets_{can}, IR, v_{hs}, screen$ )
12:         end if
13:         $IsHub, IsEnd, v_{hs}$ = updateState($n_{hubs}, v_{hs}, screen, event$)
14:         $t_{tn} = t_{tn} \cup \{event\}$
15:     end while
16:     $t_u = t_u \cup t_{tn}$
17:     $tn += 1$
18: end while
19: return $t_u$
```



```
The screen classifier top5 guesses for the screen:
['sign_in', 'signin_or_signup', 'sign_up', 'account',
'cart']
Choose the closest screen tag from the top5
guesses:sign_in
```

```
id:0 -val:SIGN IN - matched with: sign_in#click
id:1 -val:SIGN IN - matched with: sign_in#longclick
id:2 -val:Email - matched with: email#click
id:3 -val:Password (6 characters or above) - matched with:
password#click
id:4 -val:Forgot Password? - matched with: password#click

Choose the id of the widget you want to interact with: 2
Type in the input for the chosen widget:myemail
Please enter the ground truth IR for the widget you chose:email
executing event: text Email send_keys myemail
```

```
"class": android.widget.EditText
"action": "send_keys"
"exec_id_type": "text"
"exec_id_val": "Email"
"Text": "Email"
"text_input": "myemail"
"isEnd": False
"matched_trigger": email#click
```

Fig. 5. An example is used to illustrate how the GUEST assists developers in triggering a widget on the current GUI page.

text, representing the main content of the module. Notably, due to the specific functionalities of the widgets, the maximum text length associated with different intermediate elements is unique within the same GUI. Recall from Section 4.2 that GUEST supports the extraction of local structural information about widgets, including parent-child relationships. To enhance the semantic information of intermediate nodes, GUEST incorporates the most representative information from their child elements. It captures the representative text from these intermediate elements by applying the following rules:

(1) If a child widget's text length exceeds that of the current representative text, the widget and its text are updated as the new representative information.

(2) If the child widget doesn't satisfy the above condition but belongs to an intermediate element type, GUEST continues traversing through the layout structure until it extracts the most representative text from all child nodes.

Taking Fig. 6 as an example, there are two clickable widgets ① and ② presented on the GUI screen. Leveraging its UI structure layout, we can identify that the representative text information of clickable widget ① is "*New EV plans for major oil company*", which is the longest text within the entire module. This method overcomes the limitation that mainstream OCR engine may identify interfering information like 'Autos' and 'Aug 1,12:06pm' as textual semantic information.

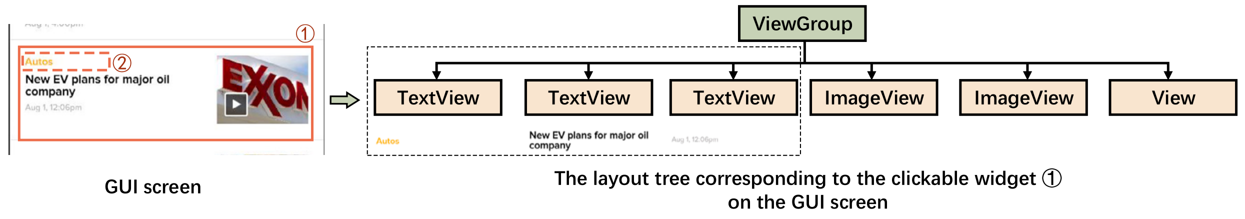

①
②
Autos
New EV plans for major oil company
Aug 1, 12:06pm
GUI screen
ViewGroup
TextView
TextView
TextView
ImageView
ImageView
View
The layout tree corresponding to the clickable widget ① on the GUI screen

Fig. 6. A simple example of layout tree of operable widget ① from GUI screen.

This iterative process gathers key semantic information from intermediate elements, effectively reducing the interference of irrelevant textual information on the similarity calculation. By following this strategy, GUEST enriches the semantic information associated with operable widgets and enhances the representation of the page state.

4.3.2. State matching

By aligning the GUI information depicting the current state of the target app with the IR model for the desired usage, state matching suggests candidate canonical screens. Once the developer selects the most appropriate screen from these candidates, it determines the scope for the subsequent actions. As the widgets information contains semantic text while the canonical widgets and screens within the IR model are iconic, semantic similarity measure is adopted to facilitate the state matching.

To more accurately approximate the semantics contained in the corresponding canonical screens of the IR model, three primary factors of the current state are considered:

Out-Degree Edges Semantic Coverage. The operable widgets on the screen reflect the functionality of the target app's current state. Fig. 5 depicts a simple example of triggers associated with the login page of apps recorded in the IR model. The *Sign In* canonical screen enables state transitions via actions such as tapping on *password*, *keep sign in*, *sign in* or *by amazon*. This indicates that widgets corresponding to these canonical widgets are available for user interaction on pages with login feature.

Preprocessing. The values of widget attributes like `text`, `resource-id`, and `content-desc` are processed for accurate similarity computation. We employ techniques such as tokenization and stopword removal from natural language processing (NLP) to preprocess attribute values. Importantly, we preserve stopwords in the `text` attribute (i.e., the displayed text) to retain essential information. In addition, there may be multiple widgets sharing the same value for the attribute `resource-id`. To minimize semantic interference, we do not consider this attribute for multiple widgets where the same `resource-id` value occurs.

Similarity Calculation. The key challenge lies in utilizing limited textual information to establish semantic approximation between the widgets on the screen and the canonical widgets of each canonical screen in the IR model. To better match widgets on the screen, we extend the description of canonical widgets introduced in Avgust [28]. This extension contains a more comprehensive textual description of the behavioral attributes associated with the canonical widgets. We employ a pre-trained BERT language model [40] to conduct semantic similarity measurement.

We compute the similarity score between each attribute value of a widget W_i on the target app's screen and the description of a canonical widget WIR_i from a canonical screen. To achieve this, we convert the words list of attribute values into a sentences list A, and s' corresponds to a sentence description of any canonical widget. For each sentence $s \in A$, $sim_1(A, s')$ denotes the maximum value of $sim(s, s')$. Formally,

$$sim_1(A, s') = \max_{s \in A} sim(s, s')$$

where $sim(s, s')$ is the semantic similarity between the two sentences s and s', obtained from the cosine distance of the vector representations $\overrightarrow{V_s}$ and $\overrightarrow{V'_s}$ of s and s', viz.,

$$sim(s, s') = \frac{\overrightarrow{V_s} \cdot \overrightarrow{V'_s}}{|\overrightarrow{V_s}||\overrightarrow{V'_s}|}$$

Then, we compute the similarity between all attribute values of a widget W_i and the description of a canonical widget WIR_i by

$$sim(s_O, s') = \frac{\overrightarrow{V_{s_O}} \cdot \overrightarrow{V'_s}}{|\overrightarrow{V_{s_O}}||\overrightarrow{V'_s}|}$$

The over similarity between a widget W_i and a canonical widget W_IR_i is computed as the weighted average:

$$sim(W_i, WIR_i) = \frac{sim_1(A, s') + sim(s_O, s')}{2}$$

Similarity matrix analysis. After calculating similarity, a similarity matrix is generated to compare the widgets on the current screen with canonical widgets. The most similar pairs are identified by the following two parts:

(1) The heuristic rules, associating the textual data of a widget with similar terms, potential position, and types linked to each canonical widget.

(2) The highest cosine similarity, representing the likelihood of a target widget matching the expected canonical widget. Candidate pairs are selected using a predefined similarity threshold of 0.65, which is adjusted to 0.45 to expand the selection range if no suitable matches are found.

The summed average scores of the candidate association pairs is regarded as the out-degree edges semantic coverage of the canonical screen corresponding to the current screen. Formally,

$$cov_{out} = \frac{\sum_{c=1}^{m} sim(W_c, WIR_c)}{m}$$

where $sim(W_c, WIR_c)$ is a candidate association pair and m denotes the total count of these candidate pairs.

In-Degree Edges Semantic Coverage. More importantly, the current state is inherently influenced by the functionality of the previous operation. An intuitive observation from Fig. 5 indicates that the current state page is closely resembles to the *Sign In* and *Sign in with Amazon* canonical screens. Assume that the current page is reached by triggering the widget corresponding to *sign in* from the state *Shopping Cart*. At this point we believe that the possibility of canonical screen *Sign In* being the closest match is highest, which is supported by the established connection between the *Sign In* and *Shopping Cart* canonical screens based on past user interactions for the usage. In light of this, we record the canonical widget WIR_{pre} corresponding to the previously triggered widget during dynamic testing process. Subsequently, we determine the in-degree edges semantic coverage as follows:

$$cov_{in} = \begin{cases} 1, & \text{if } WIR_{pre} \in inedges, \\ 0, & \text{if } WIR_{pre} \notin inedges, \end{cases}$$

where $inedges$ indicates all triggers that may transition to canonical screen.

Global Similarity. The screen pages of the app reflect the overall functional modules and the services provided. For a more comprehensive measurement, we analyze the global similarity between the current screen and canonical screens, using the activity name ac of the page and the textual information aw of all operable widgets with the description s_{nd} of the canonical screen nd. The final global similarity sim_{global} is computed as,

$$sim_{global} = \max(sim(s_{ac}, s_{nd}), sim(s_{aw}, s_{nd}))$$

The final score for any screen is computed by weighting the above factors:

$$sim_{screen_i} = \alpha * cov_{out} + \beta * cov_{in} + \gamma * sim_{global}$$

where α, β, γ represent weights assigned to each factor, indicating their respective importance among all factors, and $\alpha + \beta + \gamma = 1$.

Finally, candidate canonical screens are ranked by their final score, and GUEST recommends the top choices to developers. As illustrated in Fig. 5, GUEST suggests the top-5 canonical screens for the current screen, which include *sign_in*, *signin_or_signup*, *sign_up*, *account*, and *cart*. The canonical screen *sign_in* is selected as the most appropriate option for state matching.

4.3.3. Event generation

Once the canonical screen that best matches the current screen is identified, GUEST determines candidate widgets aligned with the desired usage. Instead of directly presenting candidate widgets as events for developers, we introduce the concept of intimacy between screens to prioritize frequently used and easily accessible screen transitions.

Intimacy Analysis. The interactions among multiple screens reflect app behavior. When there are multiple reachable paths between two screens, their transitions are considered frequent. If the distance between the two screens is short, it indicates easier access. Two screens with both frequent transitions and shorter paths are seen as intimate pairs, representing a high degree of interaction and ease of access. Specifically, we define the intimacy of screens p and q as

$$intimacy(p, q) = \frac{n_{pq}}{al(p, q) + 1} \tag{2}$$

where n_{pq} indicates the total number of reachable paths between screens p and q, and $al(p, q)$ denotes the average length of these paths.

Considering that the behavior of the app is a continuously triggered process, we divide event generation into two stages (Lines 8-12 of Algorithm 1). In the first phase, starting from the initial state of the app, GUEST relies on the guidance of the hub screen nodes identified from the IR model to reach the hub state. In the second phase, once the hub state is reached, GUEST towards the desired usage, guided by end screen nodes.

To determine which widget to trigger, we prioritize candidate widgets based on the intimacy between reachable screens and the key screens (i.e., hub screens or end screens). The intimacy-aware score of a widget w_i is obtained as:

$$score_{inti}(w_i) = \frac{1}{|P||Q|} \sum_{p \in P} \sum_{q \in Q} \frac{n_{pq}}{al(p, q) + 1}$$

where P represents the set of reachable screens of w_i, and Q is the set of key screens. For example, we take the candidate widget 'account' which has a single reachable screen labeled as $account$. From $account$ to the key screens, like $sign\_in$ and $signin\_or\_signup$

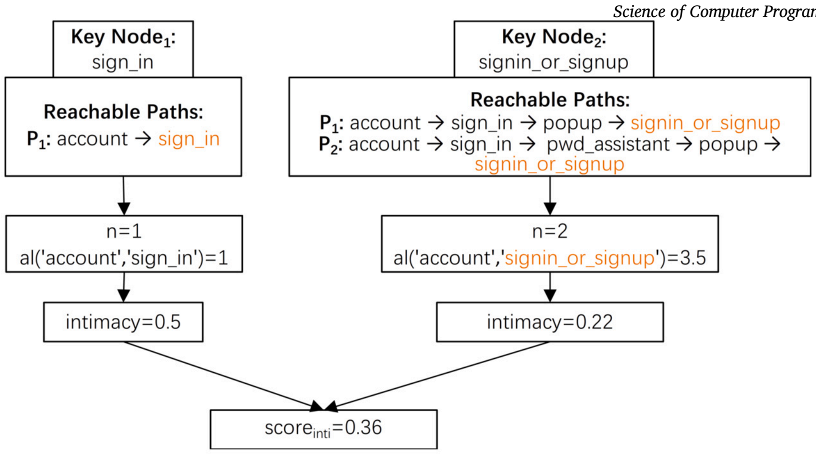

Key Node$_1$: sign_in
Reachable Paths:
P$_1$: account → sign_in
n=1
al('account','sign_in')=1
intimacy=0.5
Key Node$_2$: signin_or_signup
Reachable Paths:
P$_1$: account → sign_in → popup → signin_or_signup
P$_2$: account → sign_in → pwd_assistant → popup → signin_or_signup
n=2
al('account','signin_or_signup')=3.5
intimacy=0.22
score$_{inti}$=0.36

Fig. 7. The process of calculating the intimacy-aware score of candidate widget 'account'.

identified via network analysis, there are three distinct paths, as depicted in Fig. 7. Notably, there is only one path between $account$ and $sign\_in$, represented as $account \rightarrow sign\_in$. Through Eq. (2), we determine the intimacy between $account$ and $sign\_in$ as 0.5. Similarly, the intimacy between $account$ and $signin\_or\_signup$ is calculated as 0.22. Hence, the intimacy-aware score of the candidate widget 'account' is obtained as 0.36.

Candidate widgets are sorted based on their intimacy-aware scores, ensuring that frequently used and easily accessible widgets are prioritized. For each widget, GUEST generates an executable event, considering whether user input is required based on the widget type. The options selected by developers from GUEST's suggested candidate events are treated as target events. For instance, to execute a click event, GUEST first locates the UI element using methods like `find_element_by_id()` or `find_element_by_xpath()`, and then simulates a tap by calling the `click()` method.

Once the termination condition is met, a usage-based test scenario is generated through the event generator's sequence of triggered events based on the interactions. Since test generation is guided by hub screens, GUEST has the capability to generate multiple equivalence tests. During the test generation process, GUEST record the status of visited hub screens v_{hs}, prioritizing unexplored paths. For instance, in Fig. 1, the first test triggers the hub state *Menu* by selecting the *menu* on *Home* screen. For the second test, GUEST will prioritize a different widget *cart* on the *Home* screen to activate another hub state *Cart*.

5. Evaluation

In this section, we evaluate the proposed GUEST approach. We mainly explore the following research questions:

- **RQ1:** How effective is the GUEST at generating tests that achieve the desired usage?
- **RQ2:** How much effort can be saved by using GUEST to generate tests?
- **RQ3:** How accurate are GUEST's screen and widget classifiers?
- **RQ4:** How does the number of hub nodes and different centrality metrics affect the performance of GUEST for widget recommendations?
- **RQ5:** How does the size of the generated tests affect their desired usage implementation effectiveness?

5.1. Experimental setup

Implementation Details. GUEST is implemented in Python, and utilizes the Appium[9] framework for test generation. GUEST requires input including a desired usage, an IR model database of state machine-based models with various usages, and a target app. Its purpose is to employ a state machine-based IR model to guide the generation of equivalence tests for accomplishing the desired usage of the target app. GUEST employs the `networkx`[10] library to obtain the state transition graph of the IR model for the desired usage and conducts network centrality analysis. Additionally, the `sentence_transformers`[11] library is used to generate embeddings and `sklearn`[12] library calculates the similarity between embeddings for sentence-level semantic similarity calculation based on BERT. The experiment was conducted on a Windows desktop equipped with a 2.1 GHz Intel Core i7 CPU and 32 GB RAM. For app installation and GUI status recording, we utilized a Nexus 5X emulator running Android 6.0 (API 24).

[9] https://github.com/appium/appium.
[10] https://github.com/networkx.
[11] https://github.com/UKPLab/sentence-transformers.
[12] https://github.com/topics/sklearn.

Table 2
The specific details about subject apps.

App ID	App Name	Version	App ID	App Name	Version
A1	AliExpress	V7.2.1	A12	USA Today	V5.23.2
A2	Ebay	V6.112.0.2	A13	Zappos	V10.4.0
A3	Etsy	V5.6.0	A14	BuzzFeed	V2020.2
A4	Dailyhunt	V17.1.5	A15	Fox News	V3.29.2
A5	Geek	V2.3.7	A16	BBC News	V5.10.0
A6	Groupon	V19.16.204451	A17	Reuters	V3.4.2
A7	Home	V2.4.0	A18	News Break	V13.0.2
A8	6PM	V2.1.1	A19	Chess	V3.86
A9	Wish	V4.22.6	A20	Andttt	V0.6.6
A10	The Guardian	V6.15.1903	A21	Reversi	V1.73
A11	ABC News	V5.4.6	A22	Sudoku	V2.10.7

As outlined in the context of test generation in Section 4.3, GUEST provides two types of parameters, i.e., the maximum number of events Max_{Action} for each generated test and the weight of each contributing factor to the final score of any screen. Similar to the settings of the baseline [28], each generated test is limited to 20 events, so we also set Max_{Action} to 20. The final score of the screen is determined by three factors: out-degree edges semantic coverage, in-degree edges semantic coverage, and global similarity. We assigned equal weights to these factors. In certain cases where in-degree edges semantic coverage is zero, global similarity becomes more important than out-degree edges semantic coverage, as it considers the semantics of all widgets and the overall activity. To determine the appropriate parameter values, we conducted experiments using different settings. We randomly selected five apps from each category, initially setting α to 0.5 and then adjusting it incrementally to 0.45, 0.55, and so on, with an interval of 0.05. The results indicated that a weight of $\alpha = 0.45$ yielded the best performance for screen classification. Consequently, we adjusted the weights to $\alpha = 0.45$, $\beta = 0$, and $\gamma = 0.55$ respectively to reflect their importance. These adjusted weights provided the best performance in our experiments, and the reported results are based on these values.

Subject Apps. We conduct usage-based test generation on 22 apps for experimental evaluation. 18 subject apps were selected from the open-source evaluation dataset of Avgust [28]. This selection was based on two key considerations: (1) the shopping and news categories have been identified as sharing common functionalities across all categories on Google Play; and (2) among the closest existing approaches [18,28,29], these apps were evaluated by defining the most functional tests, considering the different implementations of the same functionality. In addition, considering that the game is a typical category that contains varied interactions, we collect four apps to validate the generalization of GUEST to other app types. Table 2 provides fundamental information about these apps, including their names and version details. Notably, apps A1 to A9, A10 to A18, and A19 to A22 belong to the shopping, news, and game categories, respectively. These apps are available on widely used platforms such as Google Play [2] and F-Droid [41], and have been intensively studied in the literature for similar purposes [3,18,28,29,42–44].

Usages and IR models database. As aforementioned, the idea of GUEST is similar to Avgust [28]. Avgust is the first technique capable of generating usage-based tests by leveraging synthetic IR Models learned from app videos. Therefore, we selected Avgust as the baseline and leveraged its usages and IR models database for comprehensive comparative evaluation. From prior work [18,28,29], the selected 18 usages were identified manually within shopping and news apps. Table 3 lists the specific test case names and functionalities for these identified usages. Specifically, we conducted test generation based on 15 usages within the shopping category (U1-U15), 14 usages within the news category (U1-U11, U16-U18), and 3 usages within the game category (U3, U5, U8). For each of these usages, the IR models database comprises state machine models commonly learned from behaviors across 18 apps (A1-A18) detailed in Table 2, which can be utilized to guide test generation for another new app. Furthermore, human tests [38] are incorporated to establish a ground truth for assessing how closely the generated tests matched analogous tests sourced from Avgust. Their empirical study collected 374 video recordings covering 18 different usage scenarios with three relevant apps for each scenario. They conducted an extensive labeled dataset comprising 2,478 ground-truth labels for screens and 2,434 labels for widgets. Their findings revealed the usage-based tests generated by Avgust are close to the tests crafted by humans in their user study. For the newly collected game apps, we determined the human tests that implemented usage after careful discussion.

Performance Metrics. For a fair comparison, we utilize the same metrics (i.e., Precision, Recall, and Top-k Accuracy) used in the baseline work [28]. Additionally, to measure the utility of the tests generated by GUEST, we employ a widely used metric in literature (e.g., [18,25,45]).

- **Precision** refers to the proportion of states and transitions in the generated tests that occur in the most similar human-obtained tests. Formally,

$$precision = \frac{|t_{gen} \bigcap t_{hum}|}{|t_{hum}|}$$

where t_{gen} indicates the set of states and transitions in the test generated by our GUEST, t_{hum} is the set of states and transitions in the most similar human-obtained tests. Here, $|t_{gen} \bigcap t_{hum}|$ represents the number of elements in the intersection between sets t_{gen} and t_{hum}.

Table 3
The 18 desired usages used in GUEST's evaluation.

Usage ID	Test Case Name	Tested Functionalities
U1	Sign In	Provide username and password to sign in
U2	Sign Up	Provide required information to sign up
U3	Category	Find first category and open browsing page for it
U4	Search	Use search bar to search a product/news
U5	Terms	Find and open legal information of the app
U6	Account	Find and open account management page
U7	Detail	Find and open details of the first search result item
U8	Menu	Find and open primary app menu
U9	About	Find and open about information of the app
U10	Contact	Find and open contact page of the app
U11	Help	Find and open help page of the app
U12	Add Cart	Add the first search result item to the cart
U13	Remove Cart	Open cart and remove the first item from the cart
U14	Address	Add a new address to the account
U15	Filter	Filter/sort search results
U16	Add Bookmark	Add the first search result item to bookmarks
U17	Remove Bookmark	Open bookmarks and remove the first item from it
U18	Textsize	Change text size

- **Recall** quantifies the proportion of states and transitions from the most similar human-obtained tests that appear in the generated tests. Formally,

$$recall = \frac{|t_{gen} \bigcap t_{hum}|}{|t_{gen}|}$$

- **Top-k Accuracy** is the percentage of recommended states and transitions with at least one correlation among the top k results. Formally,

$$acc_{top\_k} = \frac{cor(k)}{|t_{gen}|}$$

  where $cor(k)$ represents the number of ground-truth labels for states and transitions that appear in the top-k recommended results.
- **Effort Reduction** measures how much effort developers can save by adopting GUEST to generate tests instead of writing them from scratch. Formally,

$$Reduction(t_{gen}) = 1 - \frac{ED(t_{gen}, t_{hum})}{|t_{hum}|}$$

  where $ED(t_{gen}, t_{hum})$ represents the Levenshtein distance [46] of t_{gen} and t_{hum}, measuring how close the generated test is to its ground-truth test. In our case, the Levenshtein distance calculates the minimum number of edits required to change the transitions of the generated test to the closest human test. Among them, a single edit is defined as an operation of inserting, deleting, or replacing for transitions.

5.2. Experimental results

RQ1: How effective is the GUEST at generating tests that achieve the desired usage?
In this RQ, we aim to evaluate the effectiveness of GUEST for generating usage-based equivalent tests for Android apps. GUEST employs IR models learned from the other apps to guide test generation, demonstrating its capability to generate tests for previously unseen apps. For this evaluation, we randomly selected three news and shopping apps to generate tests for 16 usage cases, while data extraction restrictions limited two additional cases to two apps each, resulting in 52 test scenarios (16 cases $*$ 3 apps + 2 cases $*$ 2 apps). We utilized the first two generated tests from each of the 18 apps, leading to 102 unique tests after merging duplicates. GUEST identified two cases (specifically, "U5-Terms" for the *Etsy* app and "U6-Account" for the *6pm* app) where functionalities had a single implementation path. For game category, we randomly selected two usages from four different game apps, generating 15 unique tests. To ensure fair evaluation, tests generated by GUEST were compared with similar human-generated tests, following the baseline approach [28]. Three mobile app developers, each with 1-3 years of Android development experience, are hired to mimic developers and interact with the GUEST. At the start of the test, participants was required to watch a brief tutorial video and get acquainted with the apps.

We use the precision and recall metrics for both states and transitions as well as the number of successful executions to evaluate the effectiveness of GUEST. Table 4 lists the results for 117 tests across 18 usages. Manual inspection confirms that these tests fulfill the desired functionality, exhibiting distinct and identifiable characteristics. For example, a test is successful for "search" usage if the app displays relevant search content. Overall, GUEST achieved an 88% successful execution, with 103 tests effectively demonstrating the intended functionality. As shown in Table 4, the precision of states and transitions are 91% and 85% respectively, indicating that

Table 4
Effectiveness comparison of test generation between GUEST and Avgust.

Usage	Approach	Precision States	Precision Transitions	Recall States	Recall Transitions	Successful Executed
U1-SignIn	Avgust	0.89	0.86	0.7	0.73	5/6
	GUEST	0.89	**0.89**	**0.72**	**0.77**	**6/6**
U2-SignUp	Avgust	0.76	0.82	0.89	0.78	6/6
	GUEST	**0.88**	**0.83**	**0.92**	**0.79**	6/6
U3-Category	Avgust	0.87	0.93	0.84	0.81	12/12
	GUEST	**0.97**	**0.94**	**0.90**	**0.82**	12/12
U4-Search	Avgust	1	0.97	0.71	0.6	4/5
	GUEST	1	**1**	**0.89**	**0.86**	**4/6**
U5-Terms	Avgust	0.66	0.77	0.77	0.85	7/10
	GUEST	**0.81**	**0.92**	**0.79**	**0.92**	**9/9**
U6-Account	Avgust	1	1	1	1	5/5
	GUEST	1	1	1	1	5/5
U7-Detail	Avgust	0.69	0.71	0.52	0.67	6/6
	GUEST	**0.75**	**0.75**	**0.57**	0.67	6/6
U8-Menu	Avgust	1	0.83	0.76	0.69	11/11
	GUEST	1	0.83	**0.86**	**0.82**	11/11
U9-About	Avgust	1	0.69	**1**	0.78	4/6
	GUEST	1	**0.83**	0.96	**0.89**	4/6
U10-Contact	Avgust	0.89	0.69	0.78	0.56	1/4
	GUEST	0.89	**0.76**	**0.85**	**0.77**	**3/6**
U11-Help	Avgust	1	0.89	0.86	0.66	6/6
	GUEST	1	**0.92**	**0.94**	**0.83**	6/6
U12-AddCart	Avgust	0.86	0.51	0.79	0.62	3/6
	GUEST	**0.89**	**0.66**	**0.91**	**0.82**	**6/6**
U13-RemoveCart	Avgust	0.9	0.84	**0.96**	0.88	4/4
	GUEST	**0.93**	**0.96**	0.86	**0.90**	4/4
U14-Address	Avgust	0.63	0.44	0.70	0.64	0/4
	GUEST	**0.88**	**0.69**	**0.86**	**0.87**	0/4
U15-Filter	Avgust	0.83	0.63	0.79	0.74	2/5
	GUEST	**1**	**0.9**	**0.83**	**0.83**	**3/6**
U16-AddBookmark	Avgust	0.67	0.42	**0.89**	0.47	3/3
	GUEST	**0.9**	**0.76**	0.81	**0.78**	**6/6**
U17-RemoveBookmark	Avgust	0.64	0.44	0.75	0.48	4/4
	GUEST	**0.87**	**0.77**	**0.96**	**0.78**	**6/6**
U18-Textsize	Avgust	0.69	0.81	0.7	0.71	6/6
	GUEST	**0.83**	**0.89**	**1**	**0.87**	6/6
Avg.	Avgust	0.83	0.74	0.80	0.70	89/109(0.81)
	GUEST	**0.91**	**0.85**	**0.86**	**0.83**	**103/117(0.88)**

GUEST rarely accesses incorrect states but often triggers GUI widgets not interacted with by users. On average, the generated tests contain 86% of the states and 83% of the transitions present in the closest manual tests. This shows that GUEST effectively explores most screens identified in manual tests, but exhibits some discrepancy in executing the expected widgets that drive appropriate transitions. GUEST demonstrates strong adaptability in highly dynamic GUI scenarios by utilizing semantic information from key widgets and context after each action to check whether the current state aligns with a predefined state in the IR model. We observed that typical components associated with usage are usually static and obvious, which motivates GUEST to generate tests on game apps that accomplish the desired usage. The presence of self-rendering UI elements in game GUIs has little impact on test generation when semantic matching is applied. Moreover, during searching and filtering, filtering irrelevant text based on the UI structure improves the accuracy of state matching. This enables GUEST to track frequent UI changes and adapt to various states, maintaining accuracy even in dynamic content updates.

We analyzed the tests that did not fully accomplish the desired usage to examine the specific edge cases where GUEST underperforms. We identified three primary factors contributing to these failures. (1) Incomplete execution of functional steps. Some IR models do not capture all necessary steps for certain functionalities. For instance, for the usage "U4-Search", the test navigates successfully to the search page. However, due to limitations in the pathes recorded within the IR model, GUEST mistakenly interprets this as the completion of the desired usage and suggests prematurely ending the process. As a result, the search is left incomplete due to missing input for the search query. (2) Insufficient semantic matching for targeted widgets. In certain cases, GUEST encountered challenges in identifying appropriate widgets due to variations in UI design. For example, in the *abc* app, after reaching the settings page, GUEST was unable to locate the "about" widget because it lacked a clear and explicit label. (3) Difficulty in handling custom UI elements. Some UI pages contained custom controls essential for executing the desired functionality, such as drop-down menus for address input, category selection options, or toggle switches for filtering items. GUEST struggled to handle these custom elements, which prevented it from completing the expected operations.

After verifying the effectiveness of GUEST in generating usage-based tests, we conduct comparative experiments to determine its superiority over the baseline method, Avgust, which is the closest state-of-the-art work. To ensure a fair evaluation, we directly apply GUEST for the mobile apps in the original dataset [38] to generate usage-based tests. Moreover, to avoid inadvertent errors during replicating, we reuse results reported in the original literature [28]. Only on the newly collected four game apps, we run Avgust to generate the tests. Table 4 presents a comparison of the results between GUEST and Avgust for each desired usage. The results indicate that GUEST generates over 7.3% more tests and achieves an 7% increase in successful executions compared to Avgust. This reflects GUEST's capacity to explore and validate paths toward expected functional usages that Avgust failed to identify. Furthermore, GUEST demonstrates a 8% and 6% improvement in precision and recall for states of tests, and a remarkable 11% and 13% improvement in precision and recall for transitions. When the IR model records complex interactions, GUEST performs effectively, particularly for the usages such as adding or removing items from the cart or bookmarks, which require navigating to an item page first. By applying centrality analysis on the state transition graph, GUEST guides the app to a hub state first and then completes the target tasks in stages, simplifying the steps required for complex state transitions. These findings demonstrates the effectiveness of our GUEST in generating tests that more fully exercise the desired usage compared to those generated by Avgust.

However, from Table 4, we also notice that for the usages “U9-About”, “U13-RemoveCart”, and “U16-AddBookmark”, the states recall of tests generated by our GUEST is 0.04, 0.1, and 0.08 lower than Avgust, respectively. We investigated the cause of this phenomenon, which can be explained from the following two aspects. On the one hand, GUEST explores new paths compared to Avgust, while the IR model for the usage does not cover the canonical categories corresponding to the newly explored screens. This situation leads to a reduction in the number of the closest human test for the state categories explored by the generated tests, despite accomplishing the expected usage. On the other hand, the closest human test might be too long, such as continuing operations after removing a shopping cart. In contrast, GUEST tends to generate shorter tests by prioritizing frequently used and more accessible operations, which helps determine the most effective paths to achieve the desired usage. The above results indicate that, overall, GUEST is significantly more effective than Avgust in generating tests that achieve the desired usage.

Summary for RQ1

GUEST can significantly outperform Avgust in terms of precision, recall, and successful execution, which is effective at generating tests that achieve the desired usage.

RQ2: How much effort can be saved by using GUEST to generate tests?

The generated test is considered useful in practice if it can execute specific transitions as expected in the ground-truth test. To answer this research question, we assessed the power of converting the usage tests generated by GUEST into the most similar human tests and compared them with those produced by Avgust.

Fig. 8 illustrates the edit distance, required to convert the generated tests into the most similar human tests. Correspondingly, the potential reduction in effort achievable by employing tests generated by our GUEST also be showed. In general, the results depicted in Fig. 8 demonstrate that GUEST reduces the edit distance by 0.9, corresponding to a significant 18% improvement over Avgust. On average, GUEST can save approximately 61% of manual effort compared to writing usage-based tests from scratch. This validates the superior performance of our GUEST in test generation compared to Avgust.

Taking the example of the “U10-Contact” usage, the tests generated by GUEST for the *abc* app do not fully satisfy the specific usage. They still require an average of 2.5 manual edits to transform them into the most similar human tests. We compare these results with those from Avgust to demonstrate the performance of our GUEST in reducing effort. Among the tests that cannot satisfy the desired usage, GUEST covers 14 tests across five usages, while Avgust covers 20 cases across eight usages. Excluding “U9-About” and “U14-Address”, GUEST can achieve varying degrees of improvement in reduced effort. Although GUEST achieves a reduction of 1, it does not accomplish the intended usage “U4-Search”. We attributed this to the closest human test and the tests GUEST generated did match the corresponding search widget, but failed to thoroughly verify that the functionality worked. The above findings suggest the potential utility of GUEST, even in scenarios where it may not achieve successful test generation satisfying specific usage requirements.

Summary for RQ2

GUEST can significantly reduce the manual effort required for test generation compared to creating tests from scratch, demonstrating its utility even in scenarios where it may not fully achieve successful test generation for specific usage requirements.

RQ3: How accurate are GUEST's screen and widget classifiers?

As described in Section 4.3, GUEST classifies the app's current state screen and suggests candidate widgets to assist developers in selecting options for test generation. To evaluate the classification performance in the context of test generation, we compared GUEST with the state-of-the-art method Avgust, chosen due to its demonstrated classification performance in literature [28]. To ensure fairness in the evaluation and avoid errors introduced by subjective interpretations, we utilized the paths of tests generated

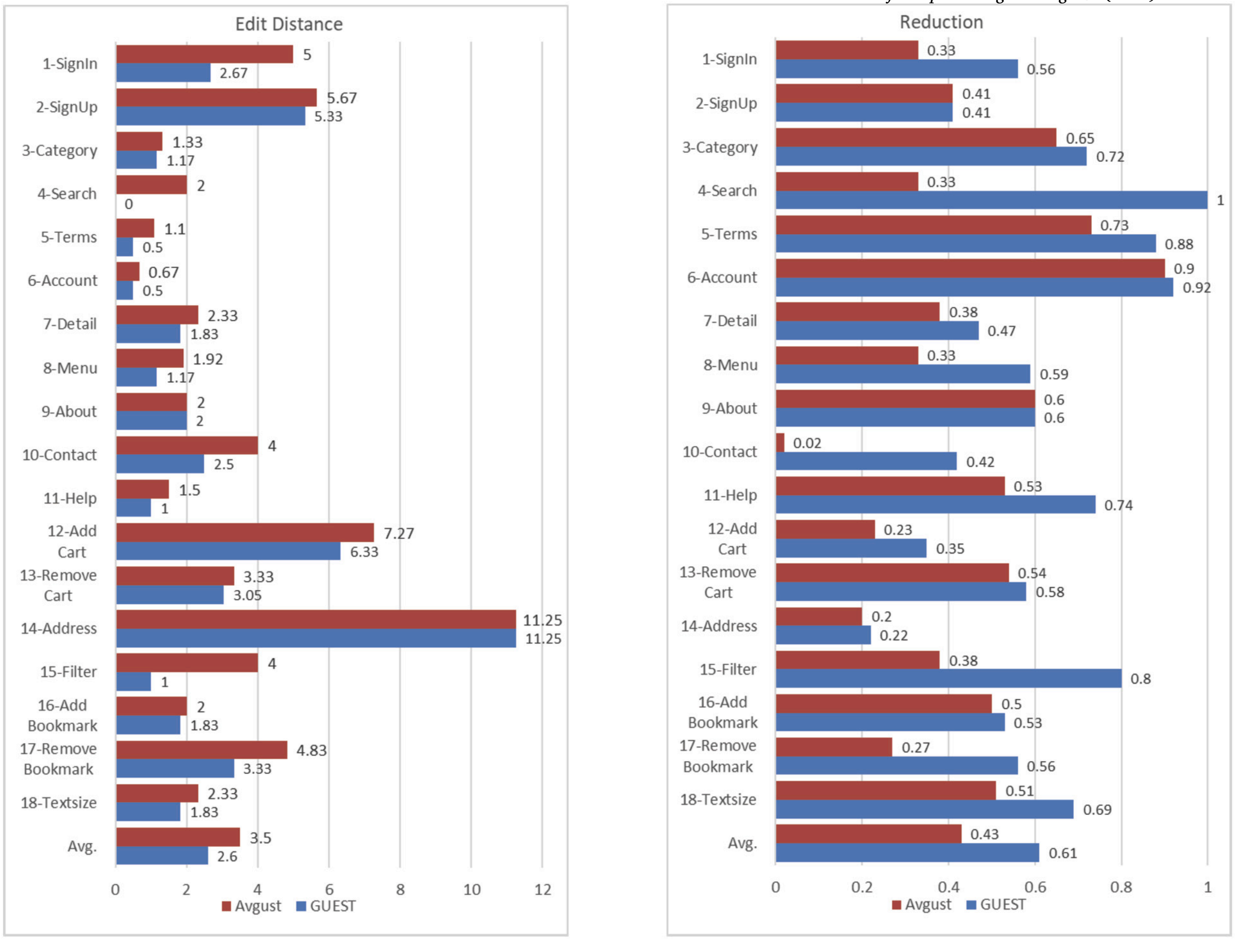


Fig. 8. The edit distance and reduction of Avgust and GUEST.

by Avgust from publicly available data to guide our test generation. We then evaluated the recommended states and widgets for each test step based on both top-1 and top-5 accuracy.

Screen Classification Evaluation. Fig. 9 illustrates the accuracy of the screen classifiers of Avgust and GUEST during the test generation phase when generating the same usage-based tests for the target app. Indeed, due to the dynamic runtime nature, state pages expose more new textual information. Consequently, the classifier within Avgust exhibits greater accuracy when relying solely on visual compared to utilizing runtime-captured dynamic information. Notably, on average, GUEST demonstrates significantly improved top-1 accuracy for screen classification. More importantly, in terms of top-5 accuracy, GUEST outperforms two variants of Avgust, achieving superior classification performance. This indicates that by combining structural and semantic information from pages, it is possible to classify states effectively and make GUEST readily available for new and unseen states as well.

As the testing phase of GUEST incorporates runtime information, the action it triggers affect the range of states the app will be transitioned to. Besides comparing two variants of Avgust, we further introduced a modified GUEST variant that does not take into account the information from the previous trigger. The aim was to explore whether incorporating contextual information from the previous action could enhance the classifier's accuracy. From Fig. 9, we can observe that the classification performance of the GUEST that does not consider the previous triggered action is worse than the standard one. This explains the state transition behavior semantics implicit in the state-machine encoding, which facilitates the prediction of the app current state. Furthermore, we can find that only on the top-1 minimum, the accuracy of GUEST is lower than Avgust which relies solely on vision. Overall, GUEST demonstrates competitive screen classification performance compared to the two variants of Avgust.

Widget Classification Evaluation. Once the developer selects the closest one from the recommended candidate screens, the GUEST recommends possible widgets for suggested interactions. We record the candidate triggers suggested at each step in the test generation process. Following that, we performed a meticulous manual examination of all recommended widgets with their cropped images to ensure their alignment with the specified categories. This thorough evaluation allowed us to assess GUEST's widget classification performance. Overall, among all the generated tests, GUEST can accurately recommend the corresponding widgets in 201 out of 226 steps. Compared to Avgust, GUEST achieved an improvement of 26 steps for successful widget recommendations, resulting in an

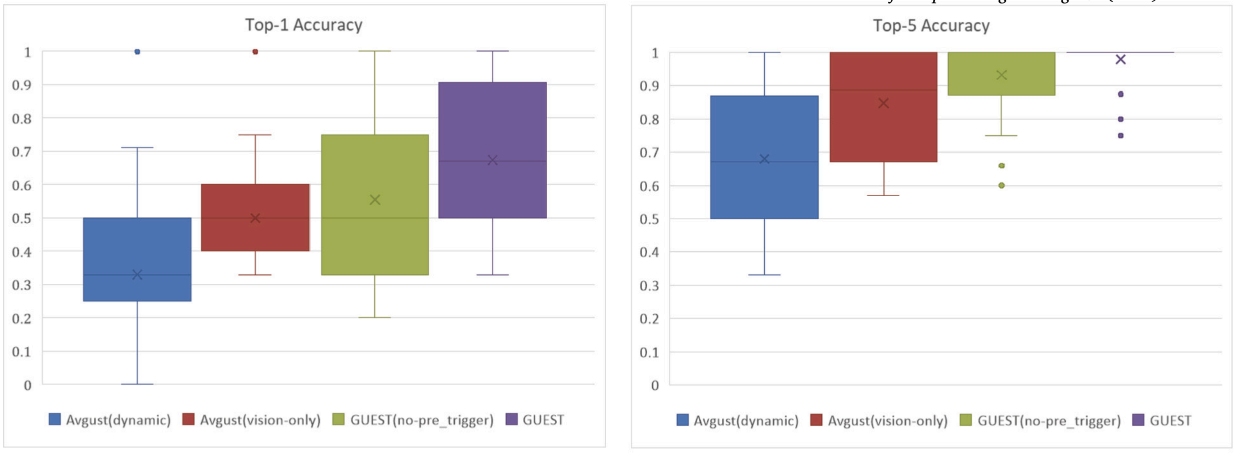


Fig. 9. GUEST screen classification outperforms the Avgust classifier variants overall in both top-1 and top-5 accuracy.

Table 5
The proportion of incorrect widgets recommended by GUEST that first appear at indexes 1 to 7 when choosing the top-k hub nodes.

	1	2	3	4	5	6	7
$k = 1$	**0.17**	**0.23**	**0.26**	0.20	0.11	0.01	0.02
$k = 2$	0.13	0.21	0.25	**0.25**	0.14	0.02	0
$k = 3$	0.14	0.21	0.24	0.21	**0.17**	**0.03**	0

Table 6
The proportion of incorrect widgets recommended by GUEST that first appear at indexes 1 to 7 when choosing different centrality measures.

	1	2	3	4	5	6	7
betweenness	0.13	**0.22**	0.25	0.22	0.15	0.02	**0.01**
degree	0.13	0.21	0.25	0.24	**0.15**	0.02	**0.01**
closeness	0.13	0.21	**0.26**	0.24	0.14	0.02	0
katz	0.13	**0.22**	0.25	0.24	0.14	0.02	0
harmonic	**0.14**	0.21	0.25	0.23	**0.15**	0.02	0
average	**0.14**	0.21	**0.26**	0.23	0.14	0.02	0
comprehensive	0.13	0.21	0.25	**0.25**	0.14	0.02	0

impressive 11.5% increase in widget classification accuracy. Therefore, we conclude that GUEST achieves better widget classification performance than Avgust.

Summary for RQ3

GUEST can significantly outperform Avgust in both screen and widget classification accuracy.

RQ4: How does the number of hub nodes and different centrality metrics affect the performance of GUEST for widget recommendations?
Throughout the test generation process, the interactive widgets suggested by GUEST might occasionally include incorrect options, potentially influencing the selection. To explore the factors affecting the performance of GUEST for widget recommendations, we conduct a detailed analysis of the incorrectly recommended widgets by our approach under different numbers of hub nodes and different centrality metric settings.

As described in Section 4.1, the candidate nodes with top-k comprehensive centrality are identified as the hub screens. Due to the considerable size of the state of an app, we limit our examination to three different values of k, excluding large numbers. Table 5 presents the proportion of indexes where the GUEST recommends incorrect widgets for varying k values. In Table 5, we can observe that, as the value of k increases, the proportion of incorrect widgets recommended by the GUEST appearing in the top-3 widgets decreases. That is to say, the index of incorrect widgets recommended in the test step is moved back. In general, the accuracy of the GUEST's prioritized widget recommendations improves as k increases. When k is 3, the accuracy of the first widget recommended by our GUEST is not further improved. According to several recommended widgets, user tends to prefer the foremost recommended widget for triggering. Consequently, improvements in the accuracy of widget recommendations begins to provide diminishing benefits in user choices. This explains GUEST can maintain good effectiveness when selecting hub nodes in the state transition graph with k set to 2.

Table 7
Growth of the generated test size, percentage of state coverage, and percentage of transitions coverage.

	$N_{max} = 1$	$N_{max} = 2$	$N_{max} = 3$	$N_{max} = 4$
Test Suit Size	60	117	164	193
Transitions Coverage	0.29	0.41	0.46	0.46
States Coverage	0.40	0.57	0.62	0.63

To study the impact of different centrality metrics on the performance of widget recommendations, experiments were conducted on seven centrality methods (betweenness centrality, degree centrality, closeness centrality, katz centrality, harmonic centrality, average centrality, and comprehensive centrality). Among them, average centrality is constructed by taking the average value of the five centrality metric values. In each method, the top-2 nodes are selected as hub nodes, corresponding to different intimacy relationships. The experimental results are shown in Table 6, which describes the index of incorrect widgets that appear in each test step. From Table 6, it can be observed that selecting hub nodes based on degree centrality and comprehensive centrality results in better widget recommendation performance. Overall, using hub nodes identified through comprehensive centrality to calculate intimacy yields the best result. This is primarily because comprehensive hub nodes are selected through betweenness centrality filtering and by combining four individual centrality metrics. These findings indicate that selecting comprehensive centrality metrics outperforms other six metrics in widget classification.

Summary for RQ4

The performance of GUEST for widget recommendations improves as the number of hub nodes increases, and using comprehensive centrality metrics for hub node selection yields the best results for accurate widget recommendations.

RQ5: How does the size of the generated tests affect their desired usage implementation effectiveness?

As mentioned in Section 4.3, the desired number of tests generated by GUEST can be adjusted by configuring the corresponding variable n_{Max}. RQ1 measures the proportion of states and transitions in a generated test that occur in its most similar human test with precision. More importantly, how closely the generated number n_{Max} of tests approximates the total human-created tests also reflects the capability of GUEST in generating usage-based equivalent tests. Specifically, we utilize states and transitions coverage to measure the proportion at which the states and transitions in the generated tests occur within all human tests. While increasing the number of test suites can enhance states and transitions coverage, it can lead to additional expenses in terms of both test execution and maintenance. For this research question, we analyze the effect on the effectiveness of GUEST generating different numbers of usage-based tests, by setting $n_{Max} = 1$, $n_{Max} = 2$, $n_{Max} = 3$ and $n_{Max} = 4$.

In our experiments, we investigated the influence of test suite size on state and transition coverage, and the findings are depicted in Table 7. Notably, as n_{Max} grows from 1 to 2, we can observe that the total number of test suites increases by 57 for the desired usages on the target apps. Correspondingly, state and transition coverage increases by 0.17 and 0.12, respectively. However, with an additional usage-based test generated for each target app, the total test sizes increases by a reduced amount of 47. This suggests that certain target apps have at most two execution paths for specific usages, indicating limited exploration of new tests. Under such scenarios, increasing the test size resulted in only a slight improvement in states and transitions coverage, with incremental gains of just 0.05 for each. After checking the generated tests, we observed that for 11 of the 18 usages, there is only a slight fluctuation in state and transition coverage, with some cases even remaining unchanged. Overall, the growth in coverage from n_{Max} increasing from 2 to 3 begins to exhibit a declining trend compared to the increase from 1 to 2. This phenomenon also explains that when the number of generated tests is set as 2, GUEST has already explored most of the execution paths for the desired usage. Finally, increasing n_{Max} from 3 to 4 did not result in an improved transition coverage. Therefore, we can conclude that increasing the size of generated tests to values of n_{Max} greater than four may not yield significant additional benefits. Considering their desired balance between the size of generated test and test coverage, developers can opt to set n_{Max} within the range of 1 to 4.

Summary for RQ5

Increasing the test size improves coverage, but beyond a certain point, there are diminishing returns, suggesting that a moderate test size is optimal.

6. Discussion

6.1. Limitations

Requiring Human Intervention. While GUEST achieves an 88% success rate in generated tests, it is not fully automated. As discussed in Section 4.3, human input is necessary for selecting screens and widgets, which can introduce subjectivity and potentially lead to repetitive tests if options are limited. To enhance automation, we will explore integrating large language models (LLMs) into the process, leveraging recent advancements in natural language understanding and question-answering.

Inaccurate Semantic Information. The extraction of semantic information from widgets by GUEST may be less effective in GUIs with unique features. For instance, highly visual apps that utilize significant animations or dynamically loaded content can produce inconsistent semantic information over time. Additionally, custom controls such as sliders, toggle switches and drop-down menus may lack essential attributes like `resource-id` or `content-desc` in their XML definitions, which complicates the identification of their functionality. The UI elements in the scroll bar change over time, which may cause the semantic information of the GUI to be inconsistent with that when it is triggered. We will enhance GUEST to improve the accuracy of semantic information extraction.

6.2. Threats to validity

Internal Threats. The first internal threat relates to the quality of the state-machine encoding for the desired usage. To mitigate this threat, we ensure that the state-machine encoding for the desired usages and target apps used for evaluation are consistent with the baseline [28]. The state-machine encoding of each app is derived by learning the interactions from all other apps, covering a variety of functional usage scenarios to ensure comprehensiveness and representativeness. The second internal threat concerns human-in-loop guided test generation. To make tests closer to real-world scenarios, screens and widgets that meet expectations are manually selected. To avoid result specificity, for each desired usage, we conducted test generation randomly on three different apps and used the average results for comparison. All generated tests are manually verified and compared with the most closest human tests to ensure reliability. The third internal threat involves the extraction of semantic information of GUI widgets. To improve the accuracy in extracting semantic information from complex UI layouts, we integrate the structure of layout tree elements to extract representative semantic information from intermediate nodes.

External Threats. A major external threat to validity is the generalizability of our findings across different app types. The experimental evaluation primarily focuses on shopping and news Android apps [29], which are widely used and share common functionalities on Google Play. We include four popular game apps representing categories with varied GUIs to further assess the applicability of GUEST. Another external threat arises from the reliance on Appium for automation. Appium is a widely adopted framework in both industry and academia, consistent with other Android testing works [21,27,28]. GUEST focuses on core functionalities that are widely supported by Appium. Additionally, certain older devices, such as those running Android 4.1, or apps relying on restricted APIs with limited configurations, may not be fully supported, potentially impacting the applicability of GUEST. The IR model database for desired usage also poses a threat. We refer to existing studies [18,28,29] to comprehensively cover 18 usage scenarios in news and shopping apps. The IR model is essentially an FSM that records the relevant canonical widgets and state transitions, which facilitates mapping to the app. Given that our GUEST is focused on automating test generation to alleviate the labor-intensive task of manual test case design, it may not cover all the paths produced by human testing. Our evaluation demonstrates that the proposed approach has the ability to generate equivalent tests for multiple usages.

7. Related work

Mobile app testing has been extensively investigated recently. In this section, we primarily discuss the related work on two key areas, i.e., test generation and test reuse.

Test Generation. Existing automated GUI test generators emphasize maximizing code coverage and detecting potential defects. They typically achieve test generation by employing either random or based on structured information. For instance, Monkey [47], the widely used fuzzing testing tool, creates event sequences by selecting GUI widgets displaying erratic behavior. In contrast, Aravind et al. [4] adopted a novel random strategy that penalizes frequently chosen widgets during the selection process. From a different perspective, some researchers [6,12,48] leverage structural information from source code through static analysis to extract and identify widgets. Despite their ability to detect exceptions, neither of the above methods is particularly effective for examining specific functionalities of the app.

A recent automated testing technique Genie [49], proposed by Su et al., focuses on identifying non-crashing functional bugs in Android apps. Despite this significant breakthrough, Genie relies on a random-based fuzz testing approach, which limits its capacity to generate usage-based tests. Our work aims to generate tests to examine specific functionality of a mobile app, such as the process of logging in to an account.

Test Reuse. Rau et al. [42] enabled test transfer across web apps by exploiting textual semantic similarity between UI elements. Similarly, Lin et al. [21] and Behrang et al. [22] proposed a greedy matching approach based on the highest semantic similarity to pair widgets from existing tests with those in similar apps. In our previous work [26,27], we enhanced the usability of the reused tests by employing an adaptive strategy to alleviate semantic challenges in event matching. These approaches utilize semantic similarity to generate meaningful tests, providing valuable insights for future research related to this area. They can only explore a test to examine similar functionality in the target app based on existing tests, which makes it insufficient to examine the app's functionality thoroughly. Route [50] generates complementary high-quality tests by employing test augmentation techniques to explore different ways of examining the same functionality. However, the above methods depend on existing tests from the source app, typically designed to target common user interactions, limiting their scope to explore diverse functional paths.

The recent work Avgust [28] classifies screen and widget images from videos to synthesize a generalized state machine-based intermediate representation (IR) model for the usage. Then they employ the IR model to guide the generation of tests for the target app. Although Avgust's generalized state machine-based IR model addresses the significant challenges of relying solely on existing tests, neural models used for image understanding may struggle with novel and unseen screens encountered in real scenarios. In contrast to Avgust, GUEST focuses on leveraging semantic information from app screens to improve the classification accuracy of screens. Specifically, GUEST aims to semantically approximate screen classification through integrating the UI hierarchy structure and widgets information of the state pages. Moreover, Avgust's widgets recommendation relies solely on screen visual information, which overlooks the semantic information present in the IR model. GUEST works on preferentially recommending actions that emulate user behavior by leveraging guidance from key screen nodes obtained through network analysis for the state transition graph of state machine-based IR model.

8. Conclusion and future work

In this paper, we propose GUEST, a novel approach to generate usage-based equivalent tests for mobile apps. GUEST regards the state transition graph of state-machine encoding for the usage as a social network and employs network analysis to obtain key nodes, which fully exploits the graph semantics of dynamic behaviors. It combines the textual semantics of state pages with UI structure to semantically match operable GUI widgets. Moreover, GUEST prioritizes frequently used and more accessible actions through intimacy analysis between key screens and reachable screens of candidate widgets within the state machine model. Experimental results indicate that GUEST can test the desired usage of the app more effectively and achieve superior classification performance compared to the state-of-the-art baseline.

For future work, we plan to incorporate the visual information from the screen to represent the semantics of the UI screen more accurately. We also aim to enhance the generalizability of our approach by expanding the range of app categories. Additionally, generating oracles automatically to examine the behavior of apps is also the next step in our agenda.

CRediT authorship contribution statement

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

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. 62372232), the Fundamental Research Funds for the Central Universities (No. NG2023005), and 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).

References

[1] App store, https://www.apple.com.cn/app-store/.
[2] Google play, https://play.google.com/store/apps/.
[3] 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.
[4] 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.
[5] W. Yang, M.R. Prasad, T. Xie, A grey-box approach for automated gui-model generation of mobile applications, in: International Conference on Fundamental Approaches to Software Engineering, 2013, pp. 250–265.
[6] 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.
[7] 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.
[8] Y. Zhou, Y. Su, T. Chen, Z. Huang, H. Gall, S. Panichella, User review-based change file localization for mobile applications, IEEE Trans. Softw. Eng. 47 (12) (2020) 2755–2770.
[9] 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.
[10] 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.
[11] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, D. Poshyvanyk, Automatically discovering, reporting and reproducing Android application crashes, in: IEEE International Conference on Software Testing, IEEE, 2016, pp. 33–44.
[12] 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.
[13] 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.

[14] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, R. Kumar, Rico: a mobile app dataset for building data-driven design applications, in: Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, UIST '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 845–854.
[15] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, Z. Su, Large-scale analysis of framework-specific exceptions in Android apps, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 408–419.
[16] F. Pecorelli, G. Catolino, F. Ferrucci, A. De Lucia, F. Palomba, Software testing and Android applications: a large-scale empirical study, Empir. Softw. Eng. 27 (2) (mar 2022), https://doi.org/10.1007/s10664-021-10059-5.
[17] K. Rubinov, L. Baresi, What are we missing when testing our Android apps?, Computer 51 (4) (2018) 60–68, https://doi.org/10.1109/MC.2018.2141024.
[18] 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.
[19] 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.
[20] F. Behrang, A. Orso, Automated test migration for mobile apps, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 384–385.
[21] 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.
[22] 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.
[23] 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.
[24] 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), 2021, pp. 70–79.
[25] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, N. Medvidović, Ui test migration across mobile platforms, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2021, pp. 756–767.
[26] 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.
[27] S. Liu, Y. Zhou, L. Ji, T. Han, T. Chen, Enhancing test reuse with gui events deduplication and adaptive semantic matching, Sci. Comput. Program. 232 (2024) 103052.
[28] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S.A. Khan, Y. Brun, N. Medvidović, K. Moran, Avgust: automating usage-based test generation from videos of app executions, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA, 2022, pp. 421–433.
[29] 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.
[30] H. Jeong, S.P. Mason, A.L. Barab'Asi, Z.N. Oltvai, Lethality and centrality in protein networks, Nature 411 (2001).
[31] X. Liu, J. Bollen, M.L. Nelson, H. Van de Sompel, Co-authorship networks in the digital library research community, Inf. Process. Manag. 41 (6) (2005) 1462–1480.
[32] K. Faust, Centrality in affiliation networks, Soc. Netw. 19 (2) (1997) 157–191.
[33] L.C. Freeman, Centrality in social networks conceptual clarification, Soc. Netw. 1 (3) (1978) 215–239.
[34] L. Katz, A new status index derived from sociometric analysis, Psychometrika 18 (1) (1953) 39–43.
[35] M. Marchiori, V. Latora, Harmony in the small-world, Phys. A, Stat. Mech. Appl. 285 (3–4) (2000) 539–546.
[36] L.C. Freeman, A set of measures of centrality based on betweenness, Sociometry 40 (1) (1977) 35–41.
[37] D. Zou, Y. Wu, S. Yang, A. Chauhan, W. Yang, J. Zhong, S. Dou, H. Jin, Intdroid: Android malware detection based on api intimacy analysis, ACM Trans. Softw. Eng. Methodol. 30 (3) (2021) 1–32.
[38] Avgust's public repository, https://doi.org/10.5281/zenodo.7036218.
[39] Android uiautomator test tool, https://github.com/xiaocong/uiautomator.
[40] J. Devlin, M.W. Chang, K. Lee, K. Toutanova, Bert: Pre-Training of Deep Bidirectional Transformers for Language Understanding, 2018.
[41] F-droid, https://f-droid.org/.
[42] A. Rau, J. Hotzkow, A. Zeller, Transferring tests across web applications, in: International Conference on Web Engineering, Springer, 2018, pp. 50–64.
[43] 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.
[44] 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.
[45] J.-W. Lin, S. Malek, Gui test transfer from web to Android, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), 2022, pp. 1–11.
[46] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Sov. Phys. Dokl. 10 (1965) 707–710.
[47] Ui/application exerciser monkey, http://developer.android.com/tools/help/monkey.html.
[48] 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.
[49] T. Su, Y. Yan, J. Wang, J. Sun, Y. Xiong, G. Pu, K. Wang, Z. Su, Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs, Proc. ACM Program. Lang. 5 (OOPSLA) (2021) 1–31.
[50] J.-W. Lin, N. Salehnamadi, S. Malek, Route: roads not taken in ui testing, ACM Trans. Softw. Eng. Methodol. 32 (3) (apr 2023), https://doi.org/10.1145/3571851.