Augmenting Automated Mobile UI Testing with Semantic-aware Test Input Generation: A Cost-effective Approach with LLMs

Rui-Fan Yang
Guohua Memorial Middle School
Foshan, China
ly-niko@qq.com

Yuan-Rui Li
Shenzhen Experimental School
Shenzhen, China
leoli@6leo6.com

Shu-Wei Wu
Shenzhen Experimental School
Shenzhen, China
2183220676@qq.com

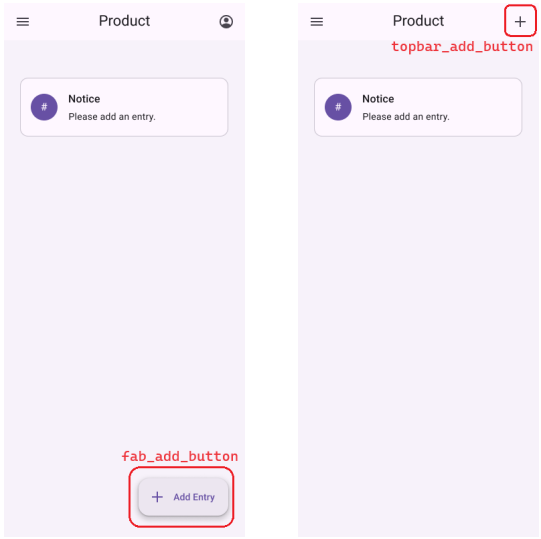
***Abstract*—UI testing for mobile applications is a critical process for identifying software defects. Crafting meaningful test cases for UI testing remains a significant challenge. As the complexity of application development increases, manually creating test cases for every conceivable scenario becomes impractical, thus elevating the importance of automated testing. Although automated testing tools are extensively utilized to swiftly generate test cases, they often fail to detect intricate issues that necessitate a sequence of logical events to uncover. Our Large Language Model (LLM)-guided approach prioritizes the generation of semantically-aware input sequences, enabling the efficient creation of practical test cases within a reasonable timeframe. We assessed our method using a collection of real-world applications and benchmarked it against commonly-used tool[1], which demonstrated that our tool attained a 79.2% increase in detected activities and a 57.6% higher success rate per activity through the analysis of semantic information and internal feature logic.**

***Index Terms*—Mobile UI Testing, Large Language Model, Test Input Generation, Automated GUI Testing**

I. Introduction

With the proliferation of mobile applications and their widespread use in daily life, the quality requirements for mobile applications have been greatly increased, making the search for potential bugs and security issues a focal point of research. However, due to the increasingly diverse and complex features of mobile applications, traditional manual testing methods have become inefficient and unable to meet industrial requirements, thus making automated UI testing a hot research topic. [2][1][3][4][5] Automated testing is designed to improve code coverage and test a more diverse set of application functions. However, existing automated tests often struggle to effectively navigate to specific pages associated with particular features for comprehensive testing due to challenges in leveraging important content such as object names provided by developers and in-app text, and their association with application logic.

To achieve the purpose of testing specific features, the method of feature-based UI testing has been proposed. This approach usually involves developers manually writing test cases to implement navigation between pages of the Application Under Test (AUT). For example, the operation sequence in the figure 1 implements navigation from one activity to another, thereby enabling better testing of the settings interface functions. However, with the rapid iteration of mobile applications, the fixed operation sequences between pages are highly uncertain. For instance, changes to the application interface elements in the updated application may lead to the invalidation of the original automated test cases. Therefore, existing feature-based UI testing still requires a significant amount of human labor to write and maintain test cases.



(a) In old version, the element's resource-id is 'fab_add_button'

(b) In new version, the element is no longer a floating action button, and the resource-id has also changed

Fig. 1. Different UI design cause old test script fail to work because locating the element is failed

The underlying issue we address stems from two primary factors: (1) The challenge of effectively interpreting the semantic content of page text. For instance, when logging into an application, conventional tools may repeatedly fail because they do not comprehend the significance of labels such as "username" and "password" adjacent to the input fields, considering them merely generic inputs. Though these labels could be hardcoded, the vastness and variability of modern applica-

tions make it unfeasible to anticipate all text in every interface, especially with dynamically generated content like daily news. (2) The inability to execute a coherent sequence of events. Such sequences are essential for tasks like adjusting a setting in an application's submenu, which necessitates navigating to that submenu first, then to the settings page, and eventually modifying the setting. The well-known random testing tool, Monkey, often fails at these tasks due to its random nature, making it incapable of systematically executing the required action sequences.

Despite advances in random testing and machine learning-based testing, neither approach has successfully emulated the human-like cognitive processes for recognizing user interface layouts and generating events. However, the logical structure inherent in real-world applications' activities, layouts, and especially texts, is not easily deciphered or exploited by most existing tools.

To bridge this gap, we propose our solution, a language model-based tool designed to enhance UI testing by emulating human cognitive patterns. Our tool takes the application under test (AUT), the target activity name, and an optional pre-collected path as inputs. It processes all interactive elements at each testing stage and translates them to natural language descriptions, thus integrating a language model to enrich mobile app UI testing. Our results demonstrate that compared to traditional testing methods, the language model approach significantly boosts efficiency and success rates while more effectively revealing potential flaws, attesting to its effectiveness and superiority. This offers a new strategy for mobile app UI testing that synthesizes human-like cognitive processes.

To evaluate the effectiveness of our tool, we construct a benchmark used to evaluate the capable for finding path to deep activities, consisting of test cases from 2 highly popular industrial Android apps that are widely used in previous work [6][7][8][9].

In general, this paper made these contributions:

- Introduces a novel problem regarding automated feature-based UI testing that leverages only the targeted activity name.
- Proposes a cost-effective methodology for automating UI testing, which entails utilizing Large Language Models (LLMs) to interpret the UI state and engender appropriate actions to navigate to specified activities.
- Presents an evaluation showcasing the efficacy of our approach in enhancing the accesibility of UI activities by 79.2% and achieving a 57.6% success rate in reaching individual activities across two widely-used industrial applications.

II. Background

The advancement of software engineering has significantly influenced mobile computing, especially concerning user interface (UI) systems. The Android platform, due to its widespread adoption, has emerged as a pivotal area for research and development, underlining the importance of an in-depth comprehension of its UI system. This section examines the fundamental concepts that underpin our research.

We begin by exploring the architecture of the Android UI system and the underlying principles that govern its operation, which are vital for app development and user engagement.

Additionally, we explore the transformative role of Large Language Models (LLMs) in UI design and testing within the Android ecosystem, shedding light on their sophisticated features that facilitate automation.

Finally, the section discusses the definitions of key terms to enhance the clarity of our discourse.

A. Android UI System

In the realm of the Android operating system, the UI framework [10] is a meticulously architected hierarchy, predominantly consisting of Activities and constituent UI components. Each activity a encapsulates a discrete screen outfitted with a user interface, delineated as an XML-encoded tree comprising various elements such as TextView, EditText, and Button, to name a few.

The inter-element relationships are articulated through a defined parent-child nexus, underscoring their stratified positioning within the UI schema. For example, a ViewGroup, serving as a container element, may encompass a multitude of TextView elements, signifying that these TextViews are arrayed at an equivalent hierarchical echelon.

Every individual element w is endowed with an array of attributes. These attributes are instrumental in dictating user interactivity capabilities (e.g., scrollable) and in defining the visible text content. The latter is integral for the conveyance of accessibility information or for aligning with specific content identifiers within the resource ID framework.

B. Large Language Model

Large Language Models (LLMs) are advanced natural language processing tools characterized by their vast parameter space and extensive capabilities. These models employ deep learning methodologies[11] to internalize grammatical rules, semantic nuances, and contextual understanding through rigorous training on substantial data sets, supported by significant computational resources.

In recent times, LLMs have achieved remarkable progress, showcasing their versatility across a spectrum of disciplines. Among the notable instances is the ChatGPT series by OpenAI, which incorporates a Generative Pretrained Transformer (GPT)[12][13] architecture. Trained on exceptionally large data corpora, these models integrate methods like Reinforcement Learning from Human Feedback (RLHF)[14] and boast hundreds of billions of parameters. As a result, they are proficient in generating coherent text, decoding and responding to inquiries, executing translation tasks, and composing summaries, among other functions.

By providing natural language prompts and steering LLMs through Prompt Engineering, we can devise potential operational sequences to assess core functionalities within software applications. To this end, we furnish the LLM with a

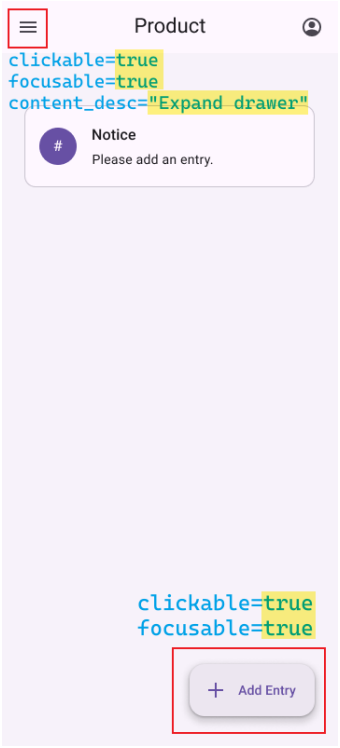


Fig. 2. This figure shows several attributes indicating whether the element can be interacted and semantic information in the attributes. Note that the "Add Entry" button at the right bottom corner doesn't have any semantic-meaningful attributes because of *element wrapper*, which separate actionable elements and text elements apart.

specific testing challenge, direct it to formulate suitable input sequences, and apply these sequences to the Application Under Test (AUT). This technique permits the evaluation of various app features in a manner that closely resembles real-world user interactions.

C. Android UI Testing

Android UI Testing is designed to generate test cases derived from the application's hierarchy, thereby facilitating a more comprehensive exploration with minimal human intervention, uncovering bugs, and performing tests on the Application Under Test (AUT) with enhanced efficiency and quality. Numerous methods are available for the automated testing of mobile applications, including random testing with tools like Monkey, leveraging deep learning with artificial intelligence, and employing algorithms for more effective testing strategies. Presently, conventional testing methodologies predominantly rely on random events or manually crafted test cases, which are often inefficient and labor-intensive, yielding less than optimal outcomes.

D. Definitions

UI State. Each UI state, denoted as s, is associated with an activity a and comprises a multitude of elements. Within each state s_1, users can interact with specific elements such as buttons, text boxes, and scrollable containers. These interactive elements are represented as W_{s_1}.

Command and Action. Commands, represented as c, are types of UI interactions that include *input*, *click*, and *back*, among others. The collection of all commands supported by our tool is denoted by C_{all}. A command c is considered applicable to an element w if $c \in C_w$; for instance, the *input* command should only be applicable to text boxes. Actions, denoted as e, are described by a triplet (w, c, v) where w signifies the targeted element, c the command executed, and v any additional parameters necessary for the action (e.g. the *input* command requires a textual argument representing the input text).

Activity Path. An activity path $p = (a_0, e_0, e_1, ..., a_1)$ traces the sequence of actions from a starting activity a_0 to a target activity a_1, through successive actions $(e_0, e_1, e_2, ...)$.

Descriptions. Each activity a and UI state s possesses a description, symbolized by d_a and d_s respectively, which articulates its functionality and content. The activity description d_a could be provided by the developer or generated by an LLM, based on all UI states previously encountered $s_1, s_2, ...$, within the activity a.

III. Preliminary Study

The automated mobile application testing tool Monkey is renowned for its ability to swiftly generate a vast array of random events with designated weights, which facilitates the random invocation of application functionalities and the transition into new activities. Monkey's prevalent use in automated application testing underscores its utility. Within this section, we examine the inefficiencies inherent in Monkey's testing approach, identify their underlying causes, and underscore the imperative to integrate Large Language Models (LLMs) with the aim of refining the process of automated mobile application testing.

A. Coverage of Monkey for Industrial Applications

Our application of Monkey for automated testing on two extensively used commercial Android applications has revealed that Monkey consistently achieves only a modest activity coverage rate over prolonged test durations.

In complex industrial applications, Monkey's capacity to thoroughly probe the application's functional Activities is not satisfactory, even with considerable testing time allotted. As depicted in Table II, Monkey achieved an average coverage of merely 42.1% of possible Activities, which poses a considerable limitation on the testing's effectiveness and thoroughness.

B. Root Cause Analysis of Activities and Edges Not Covered by Monkey

Our analysis indicates that Monkey is hampered by its inability to execute complex scenarios which necessitate a sequential chain of input events. Additionally, it fails to reauthenticate following a log out, as exemplified by the Spotify application. These limitations have been corroborated by previous studies [8].

For instance, as illustrated in Figure 3, to activate the `SubjectActivity` in the School Planner application, users must navigate through a series of steps including: reaching the corresponding activity, invoking the "Create" button at the UI's bottom right, selecting a Subject, entering the Subject's name, saving it, and ultimately accessing the Subject. Given Monkey's propensity to execute more frequent back button presses and inadvertent area taps, it often interrupts the operational flow, thereby precluding the generation of complete test scenarios.

IV. Approach

Our methodological approach is comprised of the following steps:

1) We infer the latent functionality of activities lacking explicit descriptions by analyzing their package and activity names, utilizing the Large Language Model (LLM) to generate provisional descriptions.
2) Where accessible, we engage the LLM to pinpoint the description that most closely aligns with our target activity (for instance, to navigate to the `.settings.ThemeSettingsActivity` interface for theme configuration, one would logically initiate at the `.settings.MainActivity` settings menu), thereby grounding our exploration in relevant context.
3) Prior to execution, we excise all textual data from the interface's hierarchical layout, submit it to the LLM to craft a concise state description reflective of the current context.
4) We meticulously structure both target and state descriptions to integrate them into the user prompt, amalgamating this with the UI state's elemental data, and subsequently relay it to the LLM for discernment and directive. We then interpret its guidance to interact with the Application Under Test (AUT) via the execution of appropriate actions.
5) This procedure is iteratively applied until the designated target activity is attained.

The algorithm is delineated through the accompanying pseudocode:

Our method is divided into several modules as follows:

A. Target Analysis: Finding a Starting Point

Target analysis involves describing the functionality of the target activity. Since semantic analysis of the content is crucial for finding the target path, we use a phrase of no more than ten words to indicate the functionality associated with an activity (e.g., the target description for `TimetableActivity` is "for managing time tables"), and we insert this description each time we search for elements. As the activity names carry significant semantic information (e.g., `ThemeChooserActivity` suggests that the activity is likely for theme settings and would be located within the "Settings" page), we also implement fallback strategies to generate descriptions based on activity names when no description is provided. To find the optimal starting point, we construct prompts based on the track of automatic testing tools like Monkey or descriptions of different activity paths obtained from previous explorations, formatted as "`[activity_name]`: It is `[activity_desc]`", where `[activity_name]` and `[activity_desc]` denote the name and description of the activity, respectively. When LLM selects the starting activity a_2, we filter out paths starting from the current state S_0 from all known paths to a_1, Pa_1, and randomly select three of the shortest paths using path length as a weight. For each path, we try to execute all operations $e_0, e_1, e_2...$, and if the final state s_2 belongs to a_2, we begin our exploration from there.

Algorithm 1 Design Overview of Our Tool

Input: The target activity a_1
Output: The activity path p_{a_1} if success

```
actions ← []
known_activities ← load_knowledge()
goal_desc ← llm_ask(a_1)
a_start ← llm_ask(goal_description, known_activities)
go_to_best_start_activity(a_start)
start_acivity ← get_current_activity()
start_time ← time()
while get_current_activity() ≠ a_1 do
    if actions.length > max_step then
        return []
    end if
    if time() − start_time > max_time then
        return []
    end if
    s_i ← get_current_status()
    d_{s_i} ← llm_ask(s_i)
    W_s ← get_elements(s_i)
    W_s ← filter_elements(W_s)
    elements_descs ← get_descs(W_s)
    elements_commands ← get_commands(W_s)
    llm_input ← [d_{s_i}, elements_descs,
          elements_commands, goal_desc, actions]
    llm_output ← llm_ask(llm_input)
    element ← llm_output[0]
    command ← llm_output[1]
    extra ← llm_output[2]
    if is_valid(llm_output) then
        continue
    end if
    execute_command(command, extra)
    action_desc ← llm_ask(element, command, extra)
    actions.append(element, command,
          extra, action_desc)
    handle_action(element, command, extra, s_i)
end while
if get_current_activity() = a_1 then
    path ← start_activity + actions + a_1
    save_knowledge(path)
    return actions
end if
```

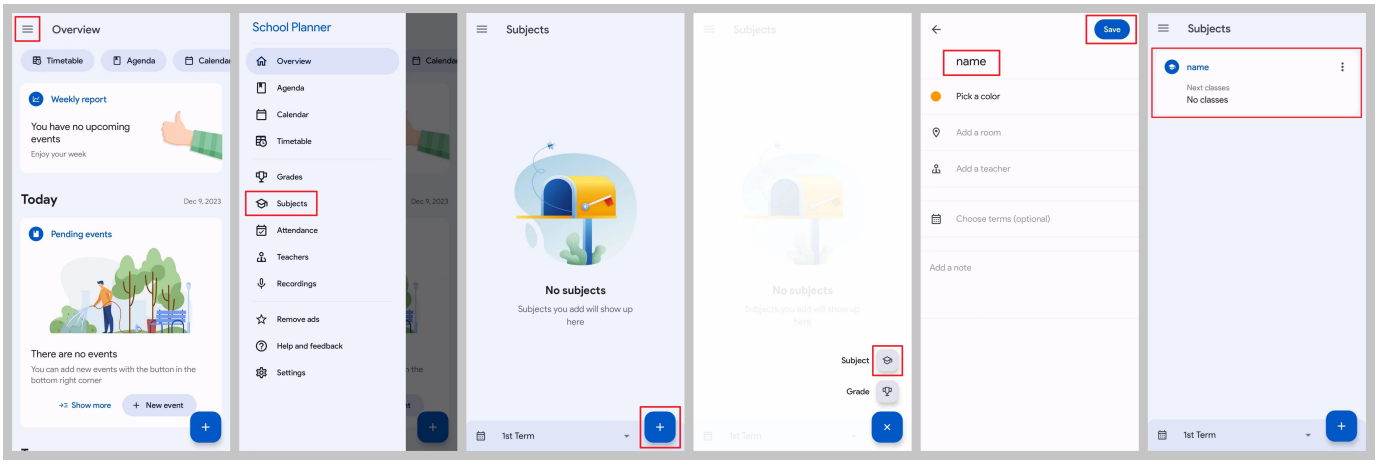


Fig. 3. Demonstration of achieving *SubjectActivity* with a sequence of actions

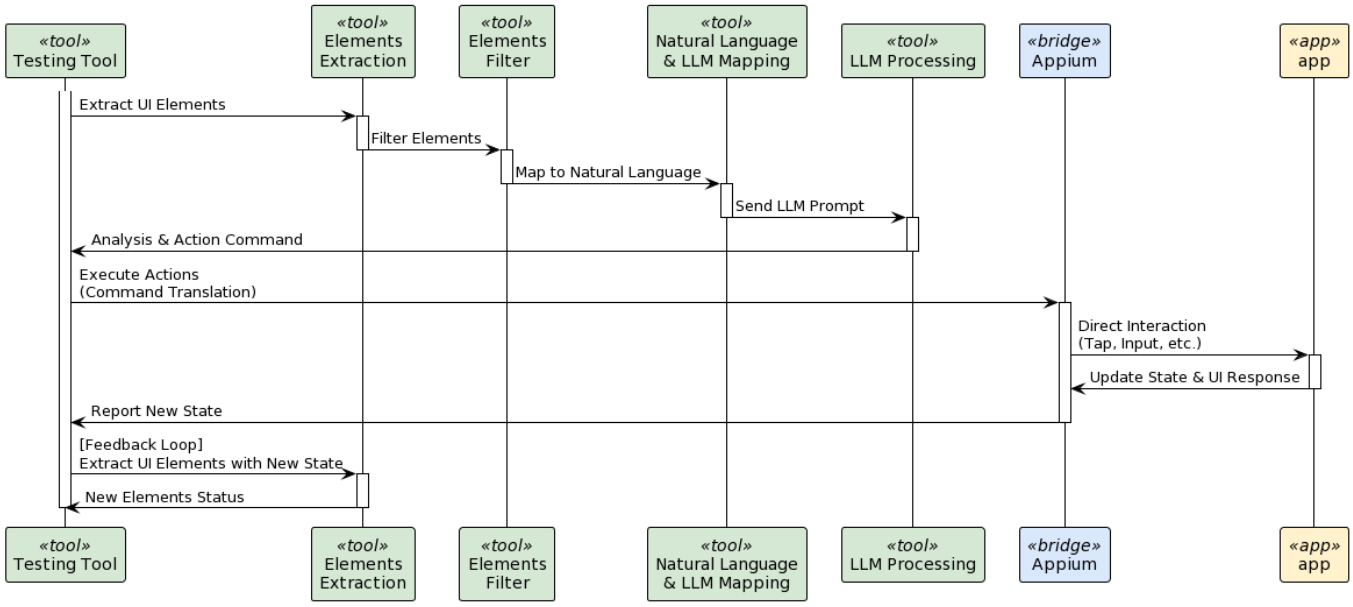


Fig. 4. Architecture of our tool

If none of the paths lead to the goal, we ignore a_2 and start directly from s_0.

B. Page Description

For each UI state s_{new} reached, we extract all elements' text information (text, content-description, resource-id attributes), remove duplicate and irrelevant content (text lengths ¡= 1), and truncate overly long content. We then send this to LLM separated by newline characters to obtain the description $d_{s_{new}}$ for s_{new}.

C. Element Extraction

For every new status s_{new} discovered, we use Appium to obtain UI interface information represented in XML format. We traverse each element, determine whether commands like click or scroll are supported based on its XML attributes such as clickable, scrollable, etc., and decide if "click" $\in A_w$. Due to limitations of automated testing frameworks, we identify text input boxes by the class attribute, such as `EditText`, indicating the need for text input. When an element supports commands A_w, we add it to the key elements w_s for that page. The entire process can be represented by the following code:

For each unselected element that carries semantic information (at least one of the text, content-desc, resource-id attributes is non-empty), if its class attribute indicates it is a wrapper for its child elements, we transfer its semantic information to all child elements. Otherwise, we incrementally merge its text information into its parent element's `child_texts` attribute until it is either merged into key elements w_s or reaches the top of the XML tree.

Specially, for elements with class `android.widget.RelativeLayout` and `android.view.ViewGroup`, we pass their attrbutes

Algorithm 2 Initialize element list

Input: Current UI state s_{new}
Output: Important element list $W_{s_{new}}$

```
W_{s_new} ← ∅
for all w ∈ s_new do
    A_w ← ∅
    for all c ∈ C_all do
        if c is available for w then
            C_w ← C_w ∪ a
        end if
    end for
    if C_w! = ∅ then
        W_{s_new} ← W_{s_new} ∪ w
    end if
end for
```

like `clickable` to their children, as they are only a wrapper without actual effect and semantic information.

We format key elements w belonging to w_s into natural language for LLM processing using the template:

```
[index]. ([commands]) "[text]",
"[resource-id]", "[classname]", "child
texts: [[child_texts]]"
```

Here, `[index]` is the element number starting from 0, `[commands]` is a space-separated list of supported commands for the element, `[text]`, `[resource-id]`, `[classname]` are the corresponding XML attribute values for the element (ignored if empty), and `[child_texts]` is the semantic information passed from its child nodes. Further, we provide all commands available on the current page in the form of function calls and require their use in execution. Specifically, the function call for input accepts two mandatory parameters: `element_index` and `value`. LLM returns a JSON format like `{"element_index": 0, "value": "$PASSWORD$"}` which we parse and pass into the input function to complete the action of inputting to the element with index 0. In particular, `back` is a parameter-less function used to trigger the "back" event.

D. Element Filter

This component analyzes the effect of each (element, command) pair on reaching a new state based on historical actions executed. It uses this analysis to prevent ineffective interactions and reduce repetitive interactions.

E. LLM Interaction

To better enable LLM to accomplish tasks, we have made several optimizations on top of the natural language description. Specifically, our measures are as follows:

1) System prompt. We use a system prompt to indicate to ChatGPT that it is involved in application testing and to request adherence to format requirements.
2) For each erroneous behavior, we attach a user prompt during retries to guide it to make the correct choice.
3) The application of function calls. As direct interaction with ChatGPT can be challenging for returning JSON format, and returned JSON may have format or parameter validity issues, function calls have been specially optimized by OpenAI. By using function calls and restricting parameter ranges with enums, we obtain more efficient execution results and improve operational efficiency.

V. Evaluation

In order to study the feasibility of using LLM for automated testing of mobile applications and the efficiency of our tool, we conducted the following two experiments:

RQ1. *How effective is our approach in generating test inputs for Android UI testing?*

RQ2. *How can LLM follow required response format?*

To answer RQ1, we tester the effectiveness of our tool by comparing it with Monkey on 2 popular industrial apps. To answer RQ2, we recorded the response of LLM and analyzed the response format.

A. Experiment Setup

Selection of Conversational LLM. Due to the function call capability of OpenAI's GPT series models, which have a significant advantage in accurately providing reply in machine-readable JSON format, our tool is primarily tailored for GPT series LLMs. Taking the cost into consideration, we have chosen the `gpt-3.5-turbo-1106` model, which performs exceptionally well on conversational tasks and is relatively cheaper, approximately one-tenth the cost of `gpt-4-turbo`, suitable for our experimental needs. We used the official Python API provided by OpenAI, and conducted experiments with a temperature of 0.15 to enhance the stability of the results.

Benchmark Construction. For our evaluation dataset, we utilized two applications widely used in real life, which have also been employed in previous research. For each application, we manually extracted all the Activities that real users can reach, which were explored by both Monkey and our tool. For Monkey, we used the default command-line arguments and controlled the runtime to be over three hours, obtaining the reachability information of Activities through logs. As for our tool, we started with the Activities that Monkey failed to reach, conducting up to three tests for each activity. We did not provide extra out-of-scope information such as manually written descriptions of target activities. Each test had a maximum duration of 30 minutes and a maximum number of valid events limited to 30. As shown in table I, most of the applications have been installed over 100 million times, indicating their high popularity and quality. Furthermore, these applications have a complex page hierarchy structure, making them representative in assessing the effectiveness of UI path testing generation methods.

Evaluation Metrics. For each application, we first manually analyze to obtain all Activities that are accessible in the logged-in state, denoted as A_{all}. During the Monkey testing process, we analyze the covered Activities, referred to as

TABLE I
APPS USED FOR EVALUATION

App	Version	Approximate Download Count
Quizlet	8.6	10m+
Spotify	8.8.88.397	1b+

TABLE II
ANALYSIS OF MONKEY'S COVERAGE ON INDUSTRIAL APPS

App	Covered	Count	Reason for not covered
Quizlet	✓	21	/
Quizlet	×	9	Failed to perform consequence of action
Quizlet	×	8	Pre-requirements not satisfied
Quizlet	×	1	Scroll required
Quizlet	×	1	File input required
Spotify	✓	3	/
Spotify	×	14	Logged out immediately

A_{monkey}. While testing with our tool, for each target activity, we record the tuple $(a, step, llm_error)$, where a represents the target activity, $step$ stands for the minimum number of actions executed among all the paths leading to this activity from the main page after logging in, and llm_error indicates how many of the operations returned by the LLM are invalid (such as: calling non-existent commands, not providing sufficient parameters, specifying non-existent elements, specifying elements that do not support the command, or refusal to execute an action).

Test platform. All experiments are conducted on the official Android x64 emulators running Android 11.0 on a server with four AMD EPYC 7H12 64-Core Processors. Each emulator is allocated with 4 dedicated CPU cores, 2 GB RAM, and 2 GB internal storage. Emulator data are stored on an in-memory disk for minimal mutual influences caused by disk I/O bottlenecks. Hardware graphics acceleration is also enabled with two Nvidia GeForce RTX 3090 Graphics Cards to ensure the responsiveness of emulators.

B. RQ1. How effective is our approach in generating test inputs for Android UI testing?

For Monkey, Table II shows the number of Activities covered, the Activities not covered, and the corresponding analysis of the reasons. The covered Activities here exclude those that are inaccessible to normal users under the login state and those Activities that are redirected to other Activities immediately after reaching them. For each activity that is not covered, Table III shows the coverage and the required steps by our tool.

Overall, our results are as shown in Table IV. The results indicate that on the Activities with more complex paths that Monkey did not cover, our tool demonstrated better performance. On average, our tool covered 79.2% more Activities than Monkey and had a 57.6% success rate across all target Activities in the evaluation.

C. RQ2. How can LLM follow required response format?

By analyzing the proportion of errors in all LLM-related actions that we requested JSON format (i.e., function call), we derive the LLM error rate shown in Table V. The results show that the average error rate of LLM stays at a relatively low level of 13.1%, and although there is still room for improvement, it already meets the experimental requirements.

TABLE III
ACTIVITY COVERAGE OF OUR TOOL

App	Activity	Covered	Steps
Quizlet	AchievementsActivity	✓	2
Quizlet	ClassCreationActivity	✓	2
Quizlet	CoursesActivity	✓	19
Quizlet	QuestionDetailActivity	×	/
Quizlet	AddSetToFolderActivity	✓	25
Quizlet	FolderActivity	✓	24
Quizlet	JoinContentToFolderActivity	×	/
Quizlet	EditSetDetailsActivity	✓	30
Quizlet	EditSetLanguageSelectionActivity	✓	11
Quizlet	EditSetPermissionSelectionActivity	✓	17
Quizlet	MatchSettingsActivity	×	/
Quizlet	StudyPathActivity	×	/
Quizlet	AccountSettingsActivity	✓	3
Quizlet	ChangeEmailActivity	×	/
Quizlet	ChangePasswordActivity	✓	11
Quizlet	ChangeProfileImageActivity	✓	2
Quizlet	ChangeUsernameActivity	×	/
Quizlet	CropImageActivity	×	/
Quizlet	NightThemePickerActivity	✓	9
Spotify	AllboardingActivity	✓	17
Spotify	AuthorizationActivity	×	/
Spotify	PremiumSignupActivity	✓	2
Spotify	LyricsFullscreenPageActivity	✓	3
Spotify	RatingsActivity	×	/
Spotify	ReportWebViewActivity	✓	20
Spotify	ScannablesActivity	✓	2
Spotify	StorageDeleteCacheActivity	×	/
Spotify	StorageRemoveDownloadsActivity	×	/
Spotify	SoundEffectsWarningActivity	×	/
Spotify	MoveCacheConfirmationActivity	×	/
Spotify	SuperbirdSetupActivity	×	/
Spotify	PageActivity	✓	/
Spotify	TrackCreditsActivity	✓	/

TABLE IV
OVERALL RESULT COMPARISON

App	Monkey	Ours	Not covered	Coverage increment	Success rate
Quizlet	21	12	7	57.1%	63.2%
Spotify	3	7	7	233.3%	50.0%
Total	24	19	14	79.2%	57.6%

NOTES: $Monkey$, $Ours$ denotes count of activity covered by respective tool. Specifically, the numbers in $Ours$ column only include activities not covered by Monkey. $Notcovered$ denotes count of activity not covered by any, $Coverageincrement$ denotes the extra activity achieved by our tool compared to Monkey, $SuccessRate$ denotes success rate of all activities tested by our tool.

TABLE V
WRONG LLM REPLY AND TOTAL ACTIONS

App	Wrong actions	Total actions	Error rate
Quizlet	156	1112	14.0%
Spotify	106	888	11.9%
Total	262	2000	13.1%

VI. Discussion

A. Threats to Validity

External threats mainly stem from the complexity inherent in the UI architectures of different applications. To mitigate these threats, we selected a variety of commercial applications that are widely used in real life for testing. In particular, because the effectiveness of our method relies heavily on the abstraction of the UI architecture, the applications we tested covered a range of UI frameworks, including Jetpack Compose recommended by Google, and those applications with traditional manually-written XML structures. Effective UI descriptions were extracted from different applications.

Internal threats primarily depend on the randomness inherent in Large Language Models (LLMs) and the precision of our method in extracting, summarizing, and defining UI elements, as well as the potential for over-filtering or under-filtering during the process of filtering out ineffective operations. For this reason, we set the temperature of the LLM to a lower value of 0.15, and through sufficient iterations of experiments, we obtained results with universality. We examined the correlation between element descriptions and their functions in applications with different UI architecture patterns, as well as the relevance of operations chosen by the LLM to the targets, in order to derive algorithms that are universally applicable.

B. Limitations

Our study is confined to the use of user interface (UI) hierarchy and text label information, with negligible or no reliance on visual information. Tools for converting graphical content into textual descriptions or other informative forms have not been employed in our research. Preliminary testing on large multimodal models, such as GPT-4 Vision[15], indicates that further optimization could improve the accuracy of our tool. Nevertheless, the cost associated with GPT-4 Vision is directly proportional to the resolution and size of the images, and the quality of the output is significantly dependent on these attributes of the input images.

VII. Related Work

A. Automated Mobile Application Testing

Automated mobile app testing aims to generate test cases based on the application hierarchy, exploring the app more comprehensively with less human intervention, discovering bugs, and conducting tests on AUT more efficiently and qualitatively. There are many avenues for automated testing of mobile applications, such as using Monkey for random testing, deep learning[11] with artificial intelligence, or utilizing algorithms for effective testing. Currently, the mainstream methods of testing involve using random events or manually created test cases; both these approaches are time-consuming and labor-intensive with subpar results.

B. Machine-Learning Based UI Testing

In order to discover more valuable activity interfaces or events, prevent inefficient exploration, and expand the amount of functionality and code covered in the exploration process, various machine learning techniques have been applied to UI testing [16] [17] [18]. These methods primarily focus on evaluating the value of each activity or event, thereby using machine learning as a basis for subsequent decision-making. However, this approach focuses on the operation itself without considering its semantics, making it difficult to conduct effective analysis at the semantic level.

C. Semantic UI Testing

In order to better analyze semantic information such as text and attributes in applications, some research employs algorithms like Transformer and word vector operations for analysis. This incorporates application semantics into automated testing, enabling the use of lexical similarity and sentence similarity in UI element analysis and event analysis. [19] [20] [21]

D. Large Language Models

Large Language Models are natural language processing models with a large number of parameters and capabilities. These models use deep learning techniques[11] to learn features such as grammar, semantics, and context by training on massive amounts of data and powerful computational resources.

In recent years, large language models have made significant breakthroughs and demonstrated their powerful abilities in various fields. One of the most famous large language model series is OpenAI's ChatGPT, which is a Generative Pretrained Transformer (GPT) [12] [13] trained on extremely large datasets with several techniques, i.e. Reinforcement Learning from Human Feedback (RLHF) [14]. These models have hundreds of billions of parameters, enabling them to generate high-quality text, understand and answer questions, perform translation tasks, generate summaries, etc.

By using natural language input and guiding LLM through Prompt Engineering to generate possible sequences of operations, it is possible to test the key functionalities of an application. Specifically, we provide LLM with the task we want to test, guide it to provide possible test input sequences, and attempt to achieve this task on AUT using this sequence. Throughout this process, we can test different features of the application in a way closer to how actual users may operate the application.

E. Prompt Engineering

Prompt engineering[22] is a novel concept that focuses on creating and optimizing prompts to effectively use language models (LLMs)[13] in different applications and research areas. This skill helps us understand the potential and limitations of LLMs.

Researchers use prompt engineering to improve the performance of LLMs in various tasks, such as answering questions

or solving arithmetic problems. Developers also utilize it to create robust and efficient methods for interacting with LLMs and other tools.

Prompt engineering goes beyond just crafting prompts; it involves a wide range of skills that are valuable for working with, developing, and understanding LLM capabilities. By using these techniques, we can better guide LMs to assist us in exploring AUTs and evaluating generated tests.[21]

VIII. Conclusion and Future Work

We have proposed a feasible solution for automated exploration of Android applications based on the reasoning capabilities of large language models. These models facilitate the testing process by conducting comprehensive reasoning centered around the objective and incorporating contextual information such as historical records. We have implemented this method in our open-source solution and tested it on real applications. The results show that this method significantly improves deep code coverage compared to traditional random testing methods, offering new insights into the application of LLMs in automated mobile application testing.

In the future, we plan to utilize the multimodal inputs of emerging large language models ([15, GPT-4V], [23, CogVLM]) to better identify the elements within the AUT, thus conducting tests in a way that more closely resembles human thinking. Currently, we have attempted to incorporate OpenAI's newly launched GPT-4V (*gpt-4-vision-preview*) model into our tool for state summarization and action selection, and have achieved better results than with the purely textual *gpt-3.5-turbo*. However, the cost of using *GPT-4V* to accurately analyze a screenshot at $0.02 per analysis is not cost-effective, so we have not made it our final solution.

Furthermore, given the cross-platform nature of Appium, we also plan to attempt to apply this method to other platforms, such as iOS, Windows, and the Web.

Although the tools we build have leveraged function call feature[24] of OpenAI's models, this project can also be run on other LLMs. The core functionality of our research does not rely on any particular large language model, and it only requires adjustments in the parsing of the LLM output to fit into other models. Moreover, there are related efforts that enable popular open-source models such as Meta's Llama to support Function Call features.

Despite the fact that we have only implemented automated testing based on targeted Activities, the potential applications of our method are not limited to that. For instance, by determining whether the page content corresponds to a specific function through LLMs or other methods, our research could also accomplish more precise UI testing targeted at specific functionalities.

Acknowledgment

This work is supported by Tencent Sustainable Social Value Organization.

References

[1] *UI/Application Exerciser Monkey*, [Accessed 20-Jul-2023]. [Online]. Available: https://developer.android.com/studio/test/other-testing-tools/monkey.

[2] Y. Arnatovich and L. Wang, "A systematic literature review of automated techniques for functional gui testing of mobile applications," Dec. 2018.

[3] C. D. Ngo, F. Pastore, and L. Briand, "Automated, Cost-effective, and Update-driven App Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, 61:1–61:51, Jul. 2022, ISSN: 1049-331X. DOI: 10.1145/3502297. [Online]. Available: https://dl.acm.org/doi/10.1145/3502297 (visited on 07/21/2023).

[4] J.-W. Lin, N. Salehnamadi, and S. Malek, "Route: Roads Not Taken in UI Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, 71:1–71:25, Apr. 2023, ISSN: 1049-331X. DOI: 10.1145/3571851. [Online]. Available: https://dl.acm.org/doi/10.1145/3571851 (visited on 07/21/2023).

[5] T. Gu, C. Sun, X. Ma, *et al.*, "Practical GUI testing of Android applications via model abstraction and refinement," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, Montreal, Quebec, Canada: IEEE Press, May 2019, pp. 269–280. DOI: 10.1109/ICSE.2019.00042. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00042 (visited on 07/20/2023).

[6] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel Testing of Android Apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Oct. 2020, pp. 481–492.

[7] W. Wang, D. Li, W. Yang, *et al.*, "An Empirical Study of Android Test Generation Tools in Industrial Cases," in *ASE*, 2018.

[8] W. Wang, W. Yang, T. Xu, and T. Xie, "Vet: Identifying and avoiding ui exploration tarpits," in *ESEC/FSE*, 2021.

[9] W. Wang, W. Lam, and T. Xie, "An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools," in *ISSTA*, 2021.

[10] *Layouts in views*. [Online]. Available: https://developer.android.com/develop/ui/views/layout/declaring-layout.

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *CCS*, 2017.

[12] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2017. arXiv: 1706.03762 [cs.CL].

[13] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020. arXiv: 2005.14165 [cs.CL].

[14] L. Ouyang, J. Wu, X. Jiang, *et al.*, *Training language models to follow instructions with human feedback*, 2022. arXiv: 2203.02155 [cs.CL].

[15] Openai. "GPT-4V(ision) System Card." (2023), [Online]. Available: https://cdn.openai.com/papers/GPTV_System_Card.pdf.

[16] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep Reinforcement Learning for Black-box Testing of Android Apps," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, 65:1–65:29, Jul. 2022, ISSN: 1049-331X. DOI: 10.1145/3502868. [Online]. Available: https://dl.acm.org/doi/10.1145/3502868 (visited on 07/21/2023).

[17] F. YazdaniBanafsheDaragh and S. Malek, "Deep GUI: Black-box GUI input generation with deep learning," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21, Melbourne, Australia: IEEE Press, Jun. 2022, pp. 905–916, ISBN: 978-1-66540-337-5. DOI: 10.1109/ASE51524.2021.9678778. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678778 (visited on 07/20/2023).

[18] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020, New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 153–164, ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397354. [Online]. Available: https://doi.org/10.1145/3395363.3397354 (visited on 07/20/2023).

[19] T. J.-J. Li, L. Popowski, T. Mitchell, and B. A. Myers, "Screen2Vec: Semantic Embedding of GUI Screens and GUI Components," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21, New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–15, ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445049. [Online]. Available: https://dl.acm.org/doi/10.1145/3411764.3445049 (visited on 07/20/2023).

[20] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems*, vol. 26, Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html (visited on 07/21/2023).

[21] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, *Mapping Natural Language Instructions to Mobile UI Action Sequences*, arXiv:2005.03776 [cs], Jun. 2020. DOI: 10.48550/arXiv.2005.03776. [Online]. Available: http://arxiv.org/abs/2005.03776 (visited on 07/21/2023).

[22] J. Wei, X. Wang, D. Schuurmans, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[23] W. Wang, Q. Lv, W. Yu, *et al.*, "Cogvlm: Visual expert for large language models," en-US, [Online]. Available: https://github.com/THUDM/CogVLM/blob/main/assets/cogvlm-paper.pdf.

[24] *Function calling and other api updates*. [Online]. Available: https://openai.com/blog/function-calling-and-other-api-updates.