Testing & standard development automation procedure of Android Apps via LLM-powered natural language translation (TSDAP)

Keywords: GUI Testing, Android Testing, Test Automation, Large Language Model, natural language translation

Abstract

GUI testing on the Android app is a critical process in ensuring software quality, but it often requires significant manual effort in both designing test cases and implementing test procedures. This manual dependency leads to inefficiencies, high maintenance costs, and limited scalability, especially in rapidly evolving user interfaces.

To address these challenges, several researches focus on test generation and automation and hope to build a system to create test cases with only Android apps as input. However, there’s still a limit in current research.

Instead of focusing on test generation, this research presents a new approach to GUI testing using the Large Language Model for testers to automate their tests based on natural language instructions. The system accepts user-described step-by-step actions and expected outcomes in a conversational format, and automatically transforms them into executable test scripts with corresponding result verifications.

Unlike traditional image-based verification methods, the system interprets flexible descriptions of expected results, enabling it to operate robustly across different devices, screen sizes, and application versions. Experimental results demonstrate that the proposed approach significantly reduces the need for manual intervention while maintaining high accuracy in both test execution and result evaluation, offering a more adaptable and efficient solution for modern GUI testing tasks. [TODO: change experiments to real data]

Acknowledgement

I sincerely thank my advisor, Professor Yi-Yu Liu, for their invaluable guidance, encouragement, and support throughout my research. Their professional insight and patient mentorship have been fundamental to the completion of this thesis.

I am also deeply grateful to Heroic Faith Medical Science, where I conducted part of this research. In particular, I would like to thank Yuan-Ren Cheng, Chief Executive Officer at Heroic Faith Medical Science, for their support and recognition of the academic value of this work. Their encouragement allowed me to explore the practical relevance of my research in a real-world setting.

My sincere thanks also go to the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, for providing a supportive academic environment and the necessary resources throughout my graduate studies.

Last, I am truly thankful to my family and friends for their continuous support, understanding, and motivation throughout this journey.

Contents

Abstract

Acknowledgement

contents

list of figures

list of tables

introduction

background

motivation

contribution

dissertation organization

related work and preliminaries

academic research

Old-Fashioned GUI Test Automation

Support Development of GUI Testing

Random-Based Testing: Android Monkey

Model-Based Approaches

Limitation of Model-Based Approaches

Higher-Level Abstractions

Test Oracle Researches

Historical Research on GUI Testing Standards

Android Debug Bridge (ADB)

Appium

UI Automator

Artificial Intelligence in Test Automation

Large Language Models (LLMs) for Natural Language Translation

methodology

experiment and evaluation

conclusion

reference

LIST OF FIGURES

Figure 2.1: Operation process of Appium [11]

LIST OF TABLES

**Introduction**

**1.1 Background**

With rapid development in the software industry, graphic user interfaces have become an essential element when interacting with software systems. GUI testing is a process that verifies the functionality and correctness of an application's graphical user interface. Unlike traditional testing approaches that validate internal program logic or API responses, GUI testing targets the elements with which users directly interact such as windows, buttons, forms, icons, and menus. The objective is to ensure that user interactions lead to the expected system behaviors, both visually and functionally. GUI testing checks properties such as layout correctness, component visibility, input responsiveness, error message display, and compliance with usability standards. For instance, if a user clicks a button to submit a form, GUI testing validates that the button is enabled, the action is triggered correctly, and appropriate feedback is shown. This type of testing is essential for detecting defects that affect user experience, such as broken navigation paths, visual misalignments, or non-responsive elements, which are often not caught by backend or unit tests.

Graphical User Interface (GUI) testing also plays a critical role in mobile applications, ensuring the functionality and usability of modern software applications. However, with the proliferation of mobile applications, there’s an increasing demand for advanced functionality. Operations on mobile applications become more complex day by day and leads to inevitable introductions of bugs.

Traditionally, to create GUI test cases, test engineers need to design two major processes: end-to-end actions and test oracles, which means the final expected result and the assertion way. GUI testing processes often rely heavily on manual efforts when it comes to implementing test scripts and deciding on test oracles. These tasks are time-consuming, error-prone, and difficult to scale, especially when dealing with frequent GUI updates or multiple platform versions. For example, if a test engineer needs to design an automated GUI test on the setting page, he/she needs to manually navigate to the setting page first to confirm the action path to the setting page, then he/she has to clarify the coordination or element each action on. Last but not least, he/she will record the result, usually by screenshots, as a standard sample for test oracle. Although test cases eventually operate automatically, this procedure is prone to regression tests, as Atif M. Memon mentioned [1], the expected outputs used by oracles can become obsolete in regressions, and manual interpretation of expected results makes the testing process subjective and inconsistent.

To address the challenges in GUI testing, most of the existing researches focus on building fully automated tools. They require only system under test (SUT) but not human interference, to ease the problem of manual interpretation. Many are designed to detect crash bugs [2], [3], [4], [5], while others focus on detecting non-crash functional bugs [6], [7].

**1.2 Motivation**

Although there are many researches related to test automation in GUI testing, comparing researches [8] and [9] shows that these approaches were not mature enough to detect most of the bugs.

Due to the immaturity of test automation research, there should be an approach between fully automated tools and fully manual approaches. In this paper, we proposed a Testing & standard development automation procedure for Android Apps via LLM-powered natural language translation (TSDAP), an approach that inputs with human test instructions and system under test, and automatically implements the process and samples for test oracles. With the introduction of human test instructions, we can avoid the problem of low test case quality and coverage. To overcome the inconsistency of manual tests, we delegated the missions of operating GUI and asserting test results to the Large Language Model (LLM), providing an ability to detect specific objects on GUI instead of a whole screenshot.

**1.3 Contribution**

We have empirically evaluated TSDAP using the enterprise-level Android app as the benchmark. With 15 verbal test cases and 100 test steps as input, TSDAP has automatically transformed texts into actual test actions and assertions, resulting in 13 successful test cases and 90 successful test steps. [TODO: change experiments to real data]

The technical contribution of this paper are as follows:

* We present TSDAP, a GUI testing automation via LLM-powered natural language translation.
* We empirically evaluate TSDAP’s ability to build actionable actual test steps from verbal inputs.
* We provide a replication package of TSDAP that includes its public implementation.

**1.4 Dissertation Organization**

The rest of the paper is organized as follows. Section II presents related work and preliminaries as well as tools used to approach test automation. Section III Framework, and Section IV example, Section V evaluation, Section VI result, VII discussion VIII discusses threats to validity. Finally, Section IX concludes.

**related work and preliminaries**

In this chapter, we explore the foundational concepts, tools, and research that support the development of this thesis. Specifically, we provide a comprehensive overview of traditional GUI test automation techniques, notable advancements in GUI testing research, and the tools commonly used for Android GUI automation. Furthermore, we discuss recent developments in Large Language Models (LLMs), which form the backbone of the proposed testing methodology.

**2.1 Academic Research**

Graphical User Interface (GUI) testing has been a longstanding challenge in software engineering. Unlike backend logic or API layers, GUIs are designed primarily for human interaction, which introduces substantial variability and complexity. As mobile applications have become ubiquitous, academic research has increasingly focused on automating the testing of Android applications to reduce manual effort and improve reliability.

GUI testing research can be categorized into three main stages: traditional manual and script-based approaches, model-based testing, and recent innovations that leverage AI and natural language processing. Each of these approaches addresses different limitations of the previous generation, such as scalability, test maintenance, and adaptability to UI changes.

**2.1.1** **Old-Fashioned GUI Test Automation**

Traditionally, GUI test automation relied heavily on record-and-playback tools or manual scripting of test steps using specific coordinates or object locators. These early methods suffered from poor maintainability, as even minor UI changes would often cause test scripts to break. Additionally, these approaches lacked abstraction, making it difficult to reuse test logic or apply it across different platforms.

Based on observation and hands-on experience of GUI testing development on Android Apps, traditional GUI testing typically follows a structured process that can be divided into five main steps. First, **the testing objectives are defined**, clarifying the functionalities and behaviors to be verified through the tests. Second, **the testing paths are designed** based on the application's GUI structure and user workflows, aiming to cover relevant scenarios. Third, **the testing paths are implemented**, which will involve scripting interactions or configuring test tools to simulate user actions. Fourth, **the test samples are validated** to ensure that the input data and test scenarios accurately reflect real usage and are suitable for uncovering potential issues. Finally, **the tests are executed to verify stability**, where repeated runs help confirm the consistency and robustness of the GUI under various conditions.

Test automation tools such as **Selenium** [10] and its mobile counterpart **Appium** [11] were originally designed based on this traditional paradigm. These tools require testers to define explicit interactions, such as clicking buttons or entering text into input fields, by specifying precise element locators (e.g., XPath or CSS selectors) or coordinates. While this approach is effective in controlled environments, it often results in brittle test scripts when applied to dynamic mobile UIs, leading to high maintenance overhead. Furthermore, the lack of standardized criteria for validating test samples introduces subjectivity, making the quality of testing highly dependent on the developer’s personal judgment. Additionally, traditional GUI testing methods that rely on screenshots as test samples are particularly vulnerable to UI regressions in Android applications, where frequent UI updates can cause tests to fail even if core functionality remains unchanged.

**2.1.2 Support Development of GUI Testing**

To mitigate the limitations of traditional methods, research began to focus on techniques to reduce manual effort, increase test coverage, and improve the reliability of test oracles. In this section, we review the evolution of such support mechanisms, from early random-based methods to more advanced model-based and higher-level approaches.

* **Random-Based Testing: Android Monkey**

To mitigate the limitations of manual scripting, **random-based testing** emerged as an alternative. The most well-known example is **Android Monkey** [12], a tool provided by the Android SDK that generates random streams of user events (e.g., touches, gestures, clicks) to stress test applications. While Monkey is simple and effective at revealing crashes, its **lack of awareness of app states and semantics** limits its effectiveness in systematic exploration and meaningful bug discovery. Moreover, it does not provide mechanisms for validating application behavior, thus offering limited support as a complete testing solution.

* **Model-Based Approaches**

To achieve more intelligent and automated GUI testing, **model-based approaches** have been actively explored. These methods typically build an abstract **GUI model** (such as a state-transition graph or event-flow model) representing the app’s navigation or UI state changes.

In the thesis Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End [13], Nguyen and Memon explore various model-based testing approaches to automate GUI software testing. Given the complexity and vast interaction sequences possible in modern GUIs, traditional manual testing proves inefficient. This research categorizes automated testing methods into different model types, including finite state machines (FSM), event flow graphs (EFG), pre-post condition models, probabilistic models, and hierarchical models. Each model provides a structured way to generate test cases, ensuring comprehensive coverage of GUI functionalities while reducing resource demands. By leveraging model-based approaches, the thesis demonstrates improved efficiency and robustness in GUI software testing, offering insights into optimizing test strategies for complex applications.

Sharma, Sabharwal, and Sibal [14] show an approach that treats a graphical user interface as an explicit state-transition model and then leverages a Genetic Algorithm (GA) to evolve high-coverage test sequences automatically. First, the GUI’s screens, widgets, and possible user actions are mapped onto a finite‐state machine: each state represents a particular screen or dialogue, and each transition corresponds to a user event (click, input, selection). Then, candidate test cases are encoded as chromosomes, which means ordered lists of transitions through this model, and their fitness is measured by how many new states, transitions, or fault-revealing behaviors they exercise.

Tools leverage static and dynamic analysis to construct models and systematically explore different execution paths. For example, **STOAT** [15], introduced by Ting Su et al., presents a novel model-based testing approach for GUI testing. Recognizing the complexities of mobile applications and the challenges of ensuring their correctness, the authors developed STOAT as a two-phase framework. Initially, the system reverse-engineers a stochastic model of an app’s GUI interactions using dynamic analysis and weighted UI exploration strategies, enhanced by static analysis. Following this, Gibbs sampling is employed to iteratively refine the model, guiding test generation to achieve high code and model coverage while exhibiting diverse interaction sequences. Notably, system-level events are injected randomly during testing to further improve effectiveness. Evaluations of real-world applications demonstrated STOAT’s ability to uncover significantly more unique crashes than existing testing tools, reinforcing its potential to enhance GUI testing methodologies.

Besides tools focusing on revealing crashes in the system, Jue Wang et al. [6] introduce a generic, automated oracle that detects arbitrary non-crashing bugs by analyzing behavioral deviations across equivalent GUI states. The approach clusters GUI execution traces, ensuring a balanced distribution of test inputs through calibrated random walks on mined GUI models. By systematically appending identical actions to comparable GUI layouts and clustering resulting behaviors, the method identifies small anomalous behavior clusters as potential defects. Implemented in the prototype tool **Odin**, the technique was evaluated on 17 real-world Android apps, revealing 28 non-crashing functional bugs, five of which were previously unknown and later confirmed by developers. Notably, 11 out of 28 bugs escaped detection by state-of-the-art tools like Genie, highlighting the effectiveness of behavior consistency analysis. This method enhances GUI testing by complementing existing coverage-guided oracles with deep-state behavioral consistency checks, making automated GUI testing more robust for mobile applications.

According to the above tools and research achievements, model-based testing has two key advantages:

1. It supports systematic coverage of the GUI space by prioritizing unexplored states or transitions.
2. It enables replayable test case generation since interactions are mapped to logical UI states rather than physical screen coordinates.

* **Limitation of Model-Based Approaches**

However, these methods often require sophisticated analysis techniques (e.g., activity lifecycle inference) and are still challenged by **semantic gaps**, they may know how to reach a screen but not whether the observed behavior matches the user’s intention or the app’s specification.

Choudhary et al. (2015) [8] performed a large-scale, head-to-head comparison of leading Android input generators which covered random-based strategies (Monkey, Dynodroid), model-based approaches (GUIRipper, A3E, SwiftHand, PUMA) and systematic methods (e.g., ACTEve’s). Running each tool on 68 real-world apps, the authors assess ease of use, cross-version compatibility, statement coverage, and fault-detection capability. Their results show that random injectors quickly trigger uncaught exceptions but waste effort on redundant events and lack stopping criteria, while model-based explorers systematically traverse GUI states yet miss non-UI internal transitions and suffer frequent restarts, and systematic techniques reach deep paths at the expense of heavy instrumentation and poor scalability. Although they can successfully trigger failures, there are two problems revealed in the results: the best performance of model-based approaches can barely reach 50% of statement coverage, and model-based approaches can’t actually achieve better coverage than random-based strategies. The study concludes that app behavior can be suitably exercised by generating only UI events, and it also suggests that we should consider **manually provided inputs** in GUI test automation (e.g., logins and passwords).

Su et al. (2021) [9] introduced the Themis benchmark, a ground‐truth suite of 52 reproducible crash bugs drawn from 20 widely used open‐source apps, and systematically comparing six state‐of‐the‐art testing approaches: Monkey’s [12] pure random event fuzzing, Ape and ComboDroid’s model‐based exploration, Humanoid’s deep‐learning‐driven UI prioritization, TimeMachine’s state‐snapshot‐and‐resume strategy, and Q-testing’s reinforcement‐learning-guided fuzzing. Despite each tool’s advances over prior work, their evaluation shows that 18 bugs (34.6 %) evade detection by any approach and that individual tools miss between 53.8 % and 71.2 % of crashes. Su et al. attribute these gaps to deficiencies in deep use‐case exploration, boundary‐value and invalid input generation, dynamic state abstraction, nuanced user interaction patterns, and cross‐app or system‐setting dependencies. Their results underscore a substantial disconnect between current automated GUI testing techniques and the complexity of real‐world bugs, pointing to the urgent need for more sophisticated exploration strategies, richer input‐modeling mechanisms, and robust state‐management capabilities in future research.

Based on the review research conducted, we can notice that there’s still a gigantic gap between state-of-art approaches’ performance and the ideal coverage to achieve. The reason for such a gap is that approaches lack rich enough input-modeling mechanisms, which leads to an important question: Can a hybrid approach of introducing human instruction and processing it automatically address the challenges in the GUI testing area?

* **Higher-Level Abstractions**

Recent efforts have shifted toward **higher-level abstractions** in GUI testing. For example, some works attempt to infer **user intents** or **semantic models** to guide testing. Others integrate **natural language processing** to bridge the gap between user-facing descriptions and executable test logic. This line of research provides a foundation for **human-in-the-loop** or **natural language-driven** testing workflows, where test scenarios are expressed in textual form and automatically converted into test scripts.

Liu et al. (2024) introduce GPTDroid [21], an LLM-based framework that recasts testing as an interactive Q&A loop: at each iteration, GPTDroid extracts structured GUI context (activities, widget texts, IDs, positions), prompts a pre-trained LLM to generate executable ADB/API commands, runs those commands, and feeds back the resulting app state. Crucially, it augments this loop with functionality-aware memory prompting, maintaining a record of tested functions, visited activities, and recent operations so the LLM can make global, function-driven exploration decisions rather than short-sighted clicks. On 93 Google Play apps, GPTDroid achieved a 32% lift in activity coverage, a 20% boost in code coverage, and detected 31% more bugs, and also discovered 53 new crashes, 35 of which were confirmed and fixed, outperforming ten state-of-the-art baselines.

In recent work, Yoon et al. introduce DroidAgent [22], an LLM-based autonomous agent framework for Android GUI testing that advances beyond structural exploration to deliver intent-driven, human-readable test scenarios. DroidAgent decomposes testing into four collaborating agents—Planner, Actor, Observer, and Reflector—backed by short-term, long-term, and widget-specific memory, enabling it to autonomously set high-level goals (e.g., “send a friend request”), execute the necessary GUI interactions, self-critique its progress, and learn from each task’s outcome. In an empirical study on 15 open-source apps from the Themis benchmark, DROIDAGENT generated 547 unique tasks (85 % judged realistic) and achieved 61 % average activity coverage—outperforming not only traditional tools like Humanoid (51 %) and DroidBot but also prior LLM-based exploration GPTDroid [21], which relies on summarizing past states to choose individual actions and attains significantly lower coverage and no self-driven task planning. These results demonstrate that embedding autonomous, memory-augmented LLM agents into GUI testing can both deepen coverage and align test generation with developer-centric, use-case semantics, albeit with trade-offs in API latency, token-context limits, and occasional model hallucinations. While DroidAgent brings human-like, data-efficient testing via generating high-level semantic goals first, its actual activity coverage is 61%, which shows that without human instructions, there’s still a gap that can’t be conducted only by fully automatic test generation tools.

Focusing on non-crash functional defects, Liu et al. propose Trident [7], a vision-driven, multi-agent testing framework powered by a multimodal large language model (GPT-4V) that “sees” and reasons about app behavior beyond superficial visual anomalies. Trident’s Explorer agent captures screenshots and UI hierarchies, annotates actionable widgets, and generates interaction prompts; its Monitor agent abstracts high-level functionality from action sequences to guide exploration while respecting token limits; and its Detector agent employs a functionality-driven chain of thought and in-context examples to infer test oracles and pinpoint deviations between expected and observed GUI transitions. Evaluated on 590 non-crash bugs and three diverse datasets, Trident achieves up to a 112 % recall and 147 % precision improvement over the best baseline and uncovers 43 new defects on Google Play, 31 of which were confirmed and fixed by developers. Despite its effectiveness, Trident’s reliance on high-cost LLM API calls introduces latency and expense, token constraints limit long-running sessions, and its action set currently excludes complex gestures and cross-platform support, suggesting fertile ground for future optimization.

These evolutions align closely with the motivation of this thesis, which proposes leveraging **Large Language Models (LLMs)** to enable a natural language interface for GUI testing. By translating user instructions into structured UI actions and utilizing LLMs’ image reasoning capabilities as a semantic test oracle, our approach aims to enhance both **test generation** and **oracle validation**, further reducing the manual overhead in mobile app testing.

* **Test Oracle Researches**

Another important advancement was in the field of **test oracle**, where researchers explored ways to determine whether a GUI test has passed or failed. Traditional GUI tests often required hard-coded assertions, but newer approaches aimed to infer expected behaviors from specifications, usage patterns, or machine learning models.

Chang et al. [17] proposed the Sikuli Test, a computer-vision-based framework that shifts the test oracle from code-level assertions to image-based “visual assertions.” In Sikuli Test, testers write oracles using two primary APIs: assertExist(image\_or\_text) and assertNotExist(image\_or\_text), which leverage OpenCV template matching and built-in OCR to detect whether a given widget or text appears (or disappears) on the screen, without relying on internal component identifiers or fixed coordinates. This decoupling from implementation details greatly enhances script resilience against minor layout tweaks. Moreover, Sikuli Test’s record-and-replay “test-by-demonstration” captures both input events and periodic screenshots, automatically cropping impacted regions to generate corresponding visual assertions, thereby minimizing manual scripting effort. By embedding these vision-driven oracles within the familiar unit- and regression-testing workflows, the framework not only streamlines GUI automation but also enables test-driven development for applications whose correctness is best judged by their rendered appearance. Although this framework allowed users to input images as a test oracle, the manual burden still exists when it comes to generating the first input images. It also inspires the thesis to build a framework that allows an even simpler input, which is verbal instructions, to automate the GUI testing process.

**2.1.3 Historical Research on GUI Testing Standards** [TODO: not sure if we really need this paragraph]

Standardization in GUI testing has been a topic of research for decades. Some researches proposed domain-specific languages (DSLs) [16] for test specification, such as **Test Description Languages (TDLs)** that abstract away implementation details. These DSLs aimed to make test cases more readable, reusable, and less sensitive to UI changes.

Another research direction focused on defining metrics for GUI test quality [8] [9], such as **UI coverage**, **event coverage**, and **interaction diversity**. These metrics laid the groundwork for benchmarking test tools and strategies, enabling more empirical evaluation of testing effectiveness.

**2.2** **Android Debug Bridge (ADB)**

The **Android Debug Bridge (ADB)** [20] is a versatile command-line tool that facilitates communication between a development machine and an Android device or emulator. ADB is an essential tool for Android developers and testers, offering a wide range of functionalities including:

* Installing and uninstalling applications
* Capturing device logs (logcat)
* Executing shell commands
* Simulating user input events (e.g., tap, swipe, text input)

In the context of GUI testing, ADB provides low-level access to UI operations that may not be exposed through higher-level frameworks like Appium. For example, testers can use ADB to simulate input events directly, capture screenshots, or automate sequences of actions without relying on application instrumentation. This flexibility makes ADB a valuable component in hybrid test automation strategies, especially when integrating with LLM-powered tools that convert high-level test descriptions into executable commands.

**2.3 Appium**

Appium [11] is one of the most popular open-source tools for automating mobile applications. Built on the philosophy of "write once, run anywhere," Appium enables cross-platform test automation of native, hybrid, and mobile web apps for Android and iOS.

Appium supports multiple programming languages, including Java, Python, and JavaScript, and relies on the **WebDriver** protocol, which facilitates interaction with the UI elements of an application. For Android specifically, Appium communicates with the **UI Automator** framework to perform actions such as clicking, swiping, or retrieving element properties.

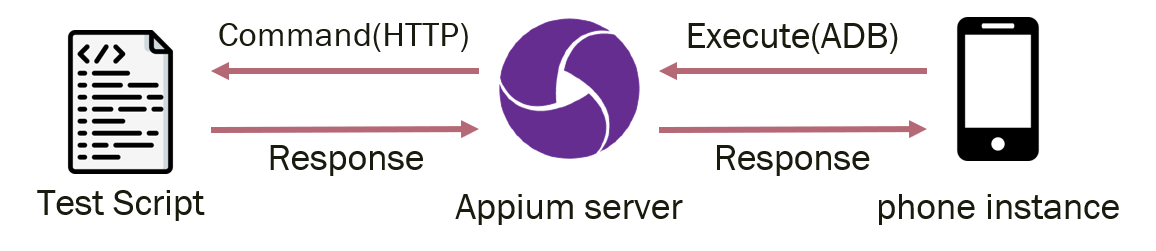
 One of the key advantages of Appium is that it does not require modification or recompilation of the application under test. As shown in Figure 2.1, Appium provides an HTTP server as a bridge between actual phone instances and test scripts, allowing scripts in different languages to be executed on distinct versions or platforms (e.g. IOS or Android) of phones. This black-box approach makes it well-suited for testing production builds. However, test development using Appium can still be cumbersome, especially for non-developers or testers unfamiliar with scripting. This has led to recent efforts to simplify test creation using higher-level abstractions, including natural language interfaces.

Figure 2.1: Operation process of Appium [11]

**2.4 UI Automator**

**UI Automator** is an Android testing framework provided by Google that allows for the automation of user interface interactions across multiple apps. Unlike **Espresso**, which is confined to a single application context, UI Automator can interact with system-level components such as the notification shade, settings menu, or launcher.

UI Automator exposes a rich API for querying and interacting with UI elements, using selectors based on text, content descriptions, resource IDs, and other attributes. It also supports synchronization mechanisms to ensure that interactions occur only when the UI is in a stable state.

Due to its comprehensive access to the Android UI hierarchy, UI Automator is often used as the underlying engine for tools like Appium. In the context of this thesis, UI Automator serves as a crucial execution backend for translating natural language test steps into concrete UI actions and is mainly used to achieve a low-latency scroll action.

**2.5 Artificial Intelligence in Test Automation**

Artificial Intelligence (AI) has been increasingly adopted in software testing to address the challenges of scalability, robustness, and efficiency. In the context of GUI testing, AI models are employed not only to reduce manual labor but also to enhance the adaptability of testing frameworks in dynamic environments.

AI introduces a new paradigm by enabling systems to understand, interpret, and interact with graphical user interfaces more intelligently. For instance, computer vision models can detect and classify UI components from screenshots, identify layout changes, and support semantic element recognition even in the absence of stable element IDs. Such models can enhance test execution by dynamically locating UI elements based on visual features or context rather than hard-coded locators.

Additionally, AI models can be integrated into the test operation process. Instead of relying solely on pixel-perfect image comparisons or hard-coded coordinate values, AI can be used to evaluate elements that can be interacted with.

One such innovation is ScreenAI [18], a vision-language model designed to analyze UI elements and infographics with a high degree of accuracy. ScreenAI enhances traditional GUI testing methodologies by leveraging a unique screen annotation task, allowing it to identify UI elements' types, locations, and relationships. With its ability to generate structured representations of screens, ScreenAI enables automated question-answering, UI navigation, and summarization tasks, reducing reliance on manual testing. By incorporating advanced vision-language modeling, including PaLI architecture and Pix2Struct’s flexible patching strategy, ScreenAI achieves state-of-the-art performance on multiple benchmarks related to UI comprehension. This approach not only improves test coverage but also streamlines UI testing workflows, ensuring applications function correctly and efficiently across diverse interfaces.

Another tool is OmniParser [19], a newly introduced vision-based screen parsing method, that plays a crucial role in advancing the capability of accurately identifying interactable elements and associating intended actions with specific regions on the interface. By effectively identifying interactable icons and understanding their semantic functions within a UI, OmniParser enables AI models like GPT-4V to generate highly accurate action predictions. This approach significantly improves action grounding, ensuring that AI-driven GUI testing can operate across multiple operating systems and applications, such as web browsers, productivity software, and mobile platforms.

While Large Language Models (LLMs) focus on natural language understanding and translation, which will be discussed in Section 2.6, the broader scope of AI in test automation encompasses visual recognition, state validation, and even test case prioritization based on learned heuristics.

By leveraging AI in these aspects, modern test automation systems can become more robust, less brittle, and significantly more adaptive to changes in UI layout, component rendering, or system responses.

**2.6 Large Language Models (LLMs) for Natural Language Translation**

Large Language Models (LLMs), such as OpenAI's GPT series and Google's PaLM, have revolutionized natural language processing by demonstrating unprecedented capabilities in understanding and generating human language. These models are trained on vast corpora of text and fine-tuned for various downstream tasks including translation, summarization, code generation, and question-answering.

In the domain of software testing, LLMs offer a novel opportunity to bridge the gap between human-readable test intents and machine-executable test scripts. By leveraging LLMs, it is now possible to translate natural language descriptions of test scenarios into structured automation scripts that interact with mobile applications.

Recent research has explored the application of LLMs in generating test cases, identifying bugs, and even suggesting fixes. The key advantage of LLMs lies in their ability to understand context, infer user intent, and generalize across domains. This makes them ideal for democratizing test automation, enabling stakeholders without programming expertise to contribute to the testing process.

In this thesis, we build on this potential by integrating an LLM-based translation layer that converts natural language test instructions into executable Android UI test scripts. This approach aims to reduce the entry barrier to mobile testing and improve test coverage by empowering a wider range of contributors.

References

[1] GUI Testing: Pitfalls and Process

[2] Using GUI Ripping for Automated Testing of Android Applications

[3] GUI Testing Using Random Event-Based Test Cases

[4] A Distinctive Genetic Approach for Test-Suite Optimization

[5] RobotBT: Behavior-Tree-Based Test-Case Specification for the Robot Framework

[6] Detecting Non-crashing Functional Bugs in Android Apps via Deep-State Differential Analysis

[7] Seeing is Believing: Vision-driven Non-crash Functional Bug Detection for Mobile Apps

[8] Automated Test Input Generation for Android: Are We There Yet?

[9] Benchmarking Automated GUI Testing for Android against Real-World Bugs

[10] Selenium

[11] Appium

[12] Monkey

[13] Advances in Automated Model-Based System Testing of Software Applications with a GUI Front-End

[14] A Survey on Software Testing Techniques using Genetic Algorithm

[15] Guided, Stochastic Model-Based GUI Testing of Android Apps

[16] Build an Auto Testing Framework Based on Selenium and FitNesse

[17] GUI Testing Using Computer Vision

[18] ScreenAI

[19] OmniParser

[20] Android Debug Bridge (adb)

[21] Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions

[22] Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents