GUI Test Migration via Abstraction and Concretization

YAKUN ZHANG, Key Lab of HCST (PKU), MOE; SCS Peking University, China
CHEN LIU, Key Lab of HCST (PKU), MOE; SCS Peking University, China
XIAOFEI XIE, Singapore Management University, Singapore
YUN LIN, Shanghai Jiao Tong University, China
JIN SONG DONG, National University of Singapore, Singapore
DAN HAO, Key Lab of HCST (PKU), MOE; SCS Peking University, China
LU ZHANG*, Key Lab of HCST (PKU), MOE; SCS Peking University, China

GUI test migration aims to produce test cases with events and assertions to test specific functionalities of a target app. Existing migration approaches typically focus on the widget-mapping paradigm that maps widgets from source apps to target apps. However, since different apps may implement the same functionality in different ways, direct mapping may result in incomplete or buggy test cases, thus significantly impacting the effectiveness of testing the target functionality and the practical applicability of migration approaches.

In this paper, we propose a new migration paradigm (i.e., the abstraction-concretization paradigm) that first abstracts the test logic for the target functionality and then utilizes this logic to generate the concrete GUI test case. Furthermore, we introduce *MACdroid*, the first approach that migrates GUI test cases based on this paradigm. Specifically, we propose an abstraction technique that utilizes source test cases from source apps targeting the same functionality to extract a general test logic for that functionality. Then, we propose a concretization technique that utilizes the general test logic to guide an LLM in generating the corresponding GUI test case (including events and assertions) for the target app. We evaluate MACdroid on two widely-used datasets (including 31 apps, 34 functionalities, and 123 test cases). On the FrUITeR dataset, the test cases generated by MACdroid successfully test 64% of the target functionalities, improving the baselines by 191%. On the Lin dataset, MACdroid successfully tests 75% of the target functionalities, outperforming the baselines by 42%. These results underscore the effectiveness of MACdroid in GUI test migration.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*.

Additional Key Words and Phrases: Test migration, Functional GUI testing, Large language model

1 INTRODUCTION

Graphical User Interface (GUI) testing is common for testing functionalities of mobile apps [22, 56, 75]. A GUI test case is composed of some ordered *events* and *assertions* [25, 52]. These events are designed to probe

*Corresponding author

Authors' addresses: Yakun Zhang, zhangyakun@stu.pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS Peking University, Beijing, China; Chen Liu, cissieliu@stu.pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS Peking University, Beijing, China; Xiaofei Xie, xfxie@smu.edu.sg, Singapore Management University, Singapore, Singapore; Yun Lin, lin_yun@sjtu.edu.cn, Shanghai Jiao Tong University, Shanghai, China; Jin Song Dong, dcsdjs@nus.edu.sg, National University of Singapore, Singapore, Singapore; Dan Hao, haodan@pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS Peking University, Beijing, China; Lu Zhang, zhanglucs@pku.edu.cn, Key Lab of HCST (PKU), MOE; SCS Peking University, Beijing, China.

ACM 1557-7392/2025/4-ART
https://doi.org/10.1145/3726525

the functionalities of GUI *widgets*. These assertions verify whether the outcomes of the events align with developers' expectations. Developers typically develop multiple functionalities (e.g., sign-in) within an app. The target functionality of a GUI test case refers to the specific functionality that the test case is designed to verify. Automatic generation of GUI test cases is challenging due to the limited understanding of the specific functionality in target apps. Consequently, GUI test cases are still predominantly crafted manually, which is time-consuming and labor-intensive [23, 31, 54, 66]. To reduce the manual effort in writing GUI test cases, several migration approaches [25, 52, 61, 84] have been proposed. These approaches migrate GUI test cases from a source app to a target app by mapping widgets that are semantically similar.

Despite advancements in migration approaches, the migrated test cases often remain incomplete or contain bugs [86], making them challenging to directly use in real-world scenarios. This issue arises because existing migration approaches follow the *widget-mapping paradigm*, which involves mapping widgets from source test cases (i.e., test cases for source apps) to target test cases (i.e., test cases for target apps). However, different apps may implement the same functionality in different ways. Source widgets (i.e., widgets in the source apps) may not be similar to target widgets (i.e., widgets in the target apps). As a result, test cases generated based on the widget-mapping paradigm might lack some necessary widgets, leading to an incomplete test of the target functionality. For example, the sign-in functionality in some apps might require an email and a password, while other apps might require a phone number and a password. Directly migrating individual test cases may not completely test the target functionality of the target app.

Considering the significant disparity between test cases generated by existing migration approaches [25, 52, 61, 84] and their target functionalities, it is crucial to propose a new migration paradigm capable of generating high-quality GUI test cases. Inspired by existing research [25, 52], we observe that although different apps may have variations in their specific implementation of the same functionality, the underlying logics for the target functionality are typically similar. In other words, test cases targeting the same functionality tend to follow similar test logics. For instance, the typical test logic for the sign-in functionality of a shopping app involves navigating to the sign-in state, completing all the required fields, clicking the sign-in related button, and verifying the correct display of user information. This test logic abstracts away the implementation details of specific mobile apps (e.g., using email or phone number) while preserving the core concept of the testing process, making it an effective guide for generating test cases related to the target functionality.

Based on the preceding observation, we propose a new migration paradigm, the *abstraction-concretization paradigm*, which first abstracts a general test logic of the target functionality from multiple source test cases, and then uses this logic to guide the generation of concrete GUI test case for the target app. In the abstraction phase, we eliminate the app-specific details, focusing solely on the general test logic of the target functionality. In the concretization phase, we apply the general test logic to generate the concrete target events and assertions. Additionally, the emergence of large language models [6, 7, 13](LLMs) brings new opportunities for GUI test migration. With sophisticated semantic understanding and reasoning capabilities [30], LLMs have the potential to understand target functionality and tackle the complexities associated with test logic abstraction and event/assertion generation, thereby enhancing the effectiveness of GUI test migration.

Specifically, we introduce a two-stage approach named MACdroid (i.e., **M**igrating GUI test cases via **A**bstraction and **C**oncretization). First, we propose an **abstraction technique** that utilizes source test cases to extract a *general test logic*. Initially, we extract an individual test logic as a sequence of structured test steps for each source test case. These test steps retain only functionality-related information, facilitating adaptation to new apps. Subsequently, we integrate the individual test logics from multiple source test cases into a sequence of structured test steps, forming a general test logic. The general test logic summarized from multiple test cases provides a comprehensive perspective on the target functionality, facilitating complete testing of the target functionality. Second, we propose a **concretization technique** that utilizes the extracted general test logic to guide an LLM in the step-by-step concretization of the target test case. Initially, we identify a set of privileged events and

assertions derived from source test cases, which may be relevant to testing the functionality of the target app. To improve accuracy, we design a priority strategy that selects events and assertions from this privileged set, rather than directly selecting from all operable widgets in the target app. This restriction narrows the candidate set, enhancing the accuracy of event and assertion selection. Moreover, we design a validation mechanism that identifies and repairs potential inaccuracies by comparing the general test logic and the output of the LLM, ensuring the accuracy and reliability of the generated test case.

We conduct a comprehensive evaluation to analyze the effectiveness of MACdroid using 31 real-world apps, 34 functionalities, and 123 test cases from the FrUITeR dataset [10] and the Lin dataset [14]. We compare MACdroid with the state-of-the-art migration approach TEMdroid [84] and the state-of-the-art generation approach AutoDroid [78] on these datasets. On the FrUITeR dataset, the test cases generated by MACdroid successfully test 64% of the target functionalities, representing a 191% improvement over the baselines. On the Lin dataset, MACdroid successfully tests 75% of the target functionalities, outperforming the baselines by 42%. We also evaluate the effectiveness of MACdroid's main techniques for the GUI test migration. Overall, these results demonstrate that MACdroid is effective in GUI test migration for industrial apps.

This paper makes the following main contributions:

- We propose a new migration paradigm (i.e., the abstraction-concretization paradigm), and introduce MACdroid, the first approach that follows this paradigm to migrate GUI test cases.
- We propose a novel technique for automatically abstracting a general test logic from source test cases for the target functionality.
- We propose a novel technique for concretizing GUI test cases that focuses on accurately selecting events and assertions for the target apps.
- We conduct an empirical evaluation using real-world apps, demonstrating the effectiveness of MACdroid. The source code of MACdroid is publicly available [15].

2 ILLUSTRATIVE EXAMPLE

Figure 1 depicts the test process for the functionality of "Add and remove an item" in a To-do app based on the test case from the Lin dataset [14]. This test case is designed to validate whether a user can successfully add a to-do item in the target app and subsequently remove it after finishing this item. We use this example to illustrate the motivation of this paper. We also use this example to illustrate the methodology of MACdroid in the following sections.

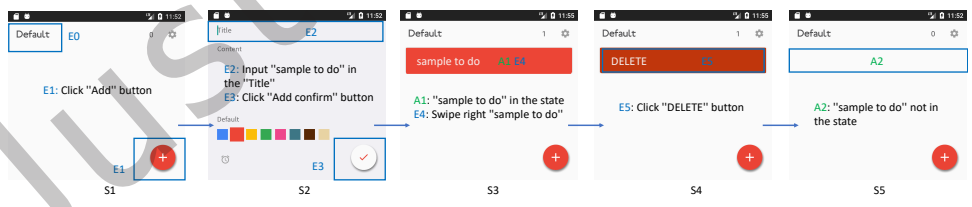

Fig. 1. The test process of "Add and remove an item" in a To-do app

Specifically, associated with five GUI states (i.e., S1 to S5), the test case includes five events (i.e., E1 to E5) and two assertions (i.e., A1 and A2). The test process is that a user clicks the "Add" button (E1), inputs "sample to do" in the "Title" box (E2), and clicks the "Add confirm" button (E3). One assertion (A1) checks for successfully adding one item by verifying that "sample to do" appears in the new state (S3). The user then swipes right on the "sample to do" item (E4) and clicks the "DELETE" button (E5). The other assertion (A2) checks for successfully removing this item by verifying that "sample to do" no longer appears in the new state (S5).

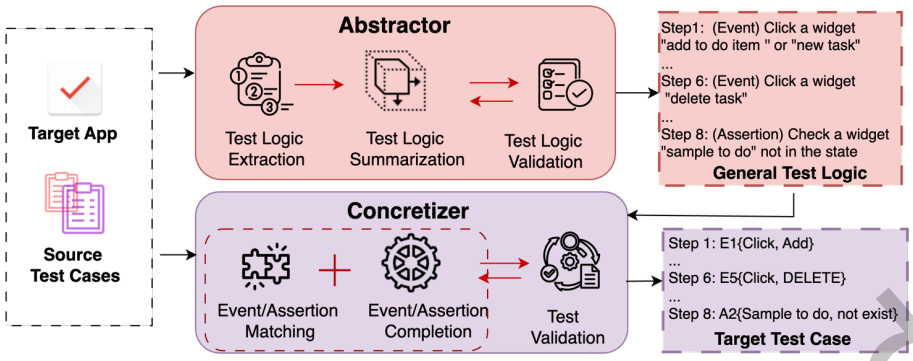

Fig. 2. Overview of MACdroid

We evaluate the effectiveness of existing approaches on the preceding example. Specifically, we select the state-of-the-art test case migration approach, TEMdroid [84], and the state-of-the-art functional test generation approach, AutoDroid [78]. However, neither approach can fully test the target functionality. This limitation arises because the specific implementations of the same functionality vary across apps, making it difficult to comprehensively test the target app's functionality by directly migrating or generating test cases. Furthermore, incorrect migrations or generations result in erroneous events and assertions, preventing these test cases from successfully testing the target functionality.

The results obtained by existing approaches cannot be directly used in industry and still require manual modification. Given the substantial disparity between the test cases generated by existing approaches and their corresponding target functionalities, it is imperative to propose a new migration paradigm capable of generating high-quality GUI test cases.

3 MACDROID

Given a target app and a set of source test cases that test the same functionality in source apps as inputs, MACdroid (whose workflow is shown in Figure 2) generates a target test case to test the target functionality based on two components. The first component is the **Abstractor component**. This component aims to extract a general test logic based on multiple source test cases (see Section 3.1). The second component is the **Concretizer component**. This component aims to concrete target test cases according to test generation (including Event/Assertion Matching and Event/Assertion Completion) and test validation (see Section 3.2).

3.1 Abstractor

Given the source test cases for the target functionality, the Abstractor component (see Figure 2) aims to extract a general Test Logic (abbr. TL) for this functionality. The variability among these source test cases presents a challenge in extracting the test logic for the target functionality. To address this challenge, Abstractor includes three modules: *TL Extraction*, *TL Summarization*, and *TL Validation*.

In the **TL Extraction** module, MACdroid extracts a sequence of structured test steps from each source test case, referred to as an individual test logic (see Section 3.1.1). This step-based format facilitates easier comprehension by LLMs [77]. Note that, during this process, app-specific details are isolated to emphasize only functionality-related information, thereby enhancing adaptability across different apps.

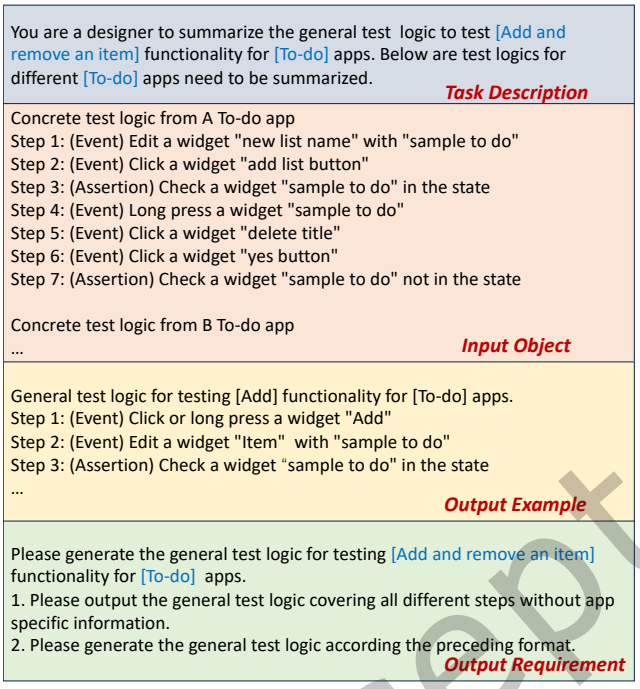

Fig. 3. A prompt example used by TL Summarization

After extraction, the individual test logic for different source test cases may vary. This is because different apps may implement the same functionality differently, the individual test logic for each source test case may be different and may only cover partial test steps for the target functionality. The **TL Summarization** module addresses this problem by creating a general and comprehensive test logic suitable for diverse apps. Since LLMs have demonstrated the capabilities to understand and summarize information, this step adopts an LLM to summarize a general and comprehensive test logic based on commonalities and differences among the individual logics (see Section 3.1.2). Furthermore, recognizing potential inaccuracies in LLMs' outputs (e.g., hallucination issues [63, 80] and comprehension biases [29, 39]), the **TL Validation** module employs a rule-based method to ensure the quality of the summarized test logic (see Section 3.1.3). Finally, Abstractor outputs a refined, general, and comprehensive test logic for the target functionality, which could be applied to various apps.

3.1.1 TL Extraction. We have two considerations for extracting the test logic from each source test case. First, standardizing the input with a consistent format and terminology makes it easier for an LLM to understand the specific task and eliminate ambiguities arising from diverse expressions [32, 41, 83]. Thus, for source test cases from different test frameworks (e.g., Appium [3] and Selenium [16]) and different programming languages (e.g., Python and Java), we represent the extracted test logic in a unified step-based structure. Second, source test cases may include app-specific information (e.g., specific app environment or interaction methods). To enhance the generalizability of the extracted test logic to new apps, we extract only the essential information from the test cases and remove the app-specific information.

Based on the preceding two considerations, we represent a test logic using a sequence of ordered test steps {S_{e1}, S_{a1},...}. There are two types (i.e., event and assertion) of test steps. A test step with an event type S_e is represented

as a tuple with four elements (e, w, a, v), where e represents the event type; w represents a widget; a represents an action to the widget; and v represents an optional input value. A test step with an assertion type S_a is represented as a tuple with three elements (a, w, c), where a represents the assertion type; w represents a widget; and c represents a condition to check the widget.

The "input object" of Figure 3 provides an example of using this module to extract an individual test logic from a source test case. Specifically, MACdroid divides the source test case into a sequence of events and assertions based on keywords (e.g., "gui", "assertion"). For each **event**, MACdroid extracts the widget, the action (e.g., click, edit, swipe, scroll, and long-press), and the optional input values. A widget typically has several attributes. To represent a widget accurately and concisely, we only use the *text*, *content-desc*, and *resource-id* attributes. After extracting these items, MACdroid structures each event as the template "*(Event) [Action] a widget [Widget] with [Value]*" (e.g., "Step 1" of "Input Object" in Figure 3). For each **assertion**, MACdroid extracts both the widget and its associated condition. After extracting these items, MACdroid structures each assertion as the template "*(Assertion) Check a widget [Widget] [Condition]*" (e.g., "Step 3" of "Input Object" in Figure 3).

Table 1. An introduction of prompt template used in MACdroid

Part	Aim
Task description	Provide the overview goal of corresponding module.
Input Object	Provide the input information to be processed by LLMs.
Output example	Provide an example with the expected output formats.
Output requirement	Provide the requirements and considerations for the output results.

3.1.2 TL Summarization. This module aims to summarize a general test logic from the extracted individual test logics of multiple source test cases. Considering the powerful comprehension and summarization capabilities of LLMs, as well as their adaptability to diverse and complex real-world scenarios, we guide LLMs in summarizing test logics rather than relying on manual summaries or rule-based techniques. This approach enables LLMs to capture the general logics of testing functionalities across a broader range of scenarios with full automation, thereby eliminating the need for manual intervention. To ensure that LLMs have a deep understanding of the logic summarization task, we design a clear and structured guidance mechanism. Specifically, we design a prompt with four parts, i.e., task description, input object, output example, and output requirement to interact with an LLM. The aims of these four parts are illustrated in Table 1. Figure 3 is an example of the prompt input of the TL Summarization module.

Task description. This part provides an overview guidance for the LLM to understand the aim of this module, which is summarizing multiple individual test logics into a general test logic. For different test cases, we only change the functionality to be tested and the category of the target app, which are highlighted in blue (see Figure 3). Note that, the functionality to be tested can be extracted from the function name of source test cases. The category of the target app can be extracted from the app location (e.g., Google Play Store [12] or F-Droid [8]).

Input object. This part displays the individual test logic extracted from each source test case by the TL Extraction module, which will be summarized.

Output example. One-shot prompting, which enables LLMs to acquire task-specific input-output formats, has demonstrated superior performance compared to zero-shot setups [19, 28]. Thus, we use one-shot prompting in this prompt design. To illustrate various types with shorter texts, we design an example with all the different output formats.

Table 2. An introduction of validation rules

Rule	Aim	Component
Irrelevant step	Verify whether the general test logic contains any irrelevant test steps.	Abstractor
Missing step	Verify whether the general test logic lacks any necessary test steps.	
Ambiguous action	Verify whether the actions in the general test logic align with the required actions.	
Incorrect type	Verify whether the matched types align with the types of the specific test steps.	Concretizer
Irrelevant matching	Verify whether the matched events and assertions belong to the privileged set.	
Completion checking	Verify whether the generated events and assertions of the specific test steps are complete.	
Incorrect format	Verify whether the output test cases align with the required formats.	

Output requirement. This part outlines two output requirements that guide the LLM in effectively summarizing the general test logic. Specifically, the first requirement aims to enhance the comprehensiveness and the generalizability of the general test logic. Note that, a single test step can be implemented in multiple ways (e.g., through different actions). For example, in different apps, deleting an item can be implemented in various actions, such as swiping or clicking the item. In this situation, the LLM should use "or" to align these various implementations within one test step. The second requirement focuses on improving the quality of the general test logic by including all important information while remaining concise. Specifically, this requirement ensures that the outputs of the LLM follow the event structure (i.e., "*(Event) [Action] a widget [Widget] with [Value]*") and the assertion structure (i.e., "*(Assertion) Check a widget [Widget] [Condition]*") as outlined in Section 3.1.1. This structured template makes the general test logic semantically clear within the word limit of LLMs inputs [6, 13].

3.1.3 TL Validation. The process of summarizing multiple individual test logics into a single general test logic involves combining test steps that achieve the similar objectives and also adding test steps related to achieve different objectives. This process may result in the general test logic being longer than a single individual test logic but not excessively long, as these individual test logics target at testing the same functionality and share several common test steps. However, due to hallucination issues of LLMs, outputs of LLMs may be irrelevant or fabricated with the input data [63, 80] even if the output requirements are included. To address these potential issues and ensure the quality of the general test logic, we design three rules to identify common issues related to the TL Summarization module. Subsequently, for each issue, we generate corresponding feedback for the TL Summarization module to re-summarize the general test logic accordingly. The first three rows of Table 2 provide an overview of the validation rules.

Irrelevant step. The general test logic generated by the TL Summarization module may not only include a summary of the test steps from the source test cases but also create irrelevant test steps. We use the longest source test case as a *reference*, and use Max_ratio to measure whether the general test logic remains within a reasonable length. If the generated test logic is too long (i.e., too many test steps), it may contain redundancy and irrelevant steps.

Specifically, MACdroid calculates the ratio of the number of test steps in the general test logic to the number of test steps in the longest source test case. If this ratio exceeds Max_ratio, irrelevant steps may be introduced. To repair this issue, we design a feedback prompt as "*The number of your summarized test steps is more than the maximum number of test steps, which may introduce irrelevant test steps. Please re-summarize it*". This feedback prompt is then sent to the TL Summarization module for re-summarization.

Missing step. The general test logic generated by the TL Summarization module may lack necessary test steps, resulting in an incomplete summary. We use the shortest test case as a *reference* for validation. This is because general test logic is derived by summarizing the individual test logics from multiple test cases. The process

involves merging similar steps and supplementing distinct ones. Therefore, it is rare for the general test logic to be shorter than the corresponding shortest test case.

Specifically, MACdroid compares the number of test steps in the general test logic. If the generated general test logic is shorter than the shortest test case, that means the general test logic may lack some necessary test steps. To repair this issue, we design a feedback prompt to the TL Summarization module for re-summarization. The feedback prompt is "*The number of your summarized test steps is less than the minimum number of test steps, which may miss some necessary test steps. Please re-summarize it.*".

Ambiguous action. The actions specified in the general test logic generated by the TL Summarization module may differ from the expected actions, increasing the difficulty for the LLM to understand the functionality in the subsequent stage. For instance, the click action can be described in various ways, such as "tap a screen" or "touch by finger", all of which are same semantics. However, these diverse descriptions increase the difficulty for LLMs. To mitigate this issue, we standardize the description of the click action to consistently use "click", as outlined in the "output example" (see Figure 3). This standardization reduces ambiguity, and enhances the accuracy and consistency of the LLM's understanding. To identify this issue, MACdroid compares the actions generated by the TL Summarization module with the expected actions outlined in the "output example". Specifically, for events, the expected actions include "click", "edit", "swipe", "scroll", and "long-press". For assertion, the expected action is "check". If MACdroid identifies discrepancies between the generated actions and the expected actions, it then generates a corresponding feedback prompt to the TL Summarization module. The feedback prompt is as follows: "*The [Step] does not include an action that appears in the output example. Please select one action in the output example to re-describe this step*".

3.2 Concretizer

The Concretizer component aims to generate a test case for the specified functionality of the target app, utilizing the general test logic and the source test cases. The test logic summarized by the Abstractor component is general, which cannot be directly utilized as a test case for the target app. To concretize an executable test case, MACdroid needs to concretize each test step in the general test logic with the specific events and assertions in the target app. However, it is challenging to select appropriate events and assertions due to the large number of candidates in the target app. For example, the BBC News [4] app includes more than 30 GUI states, and each state has an average of 60 operable widgets.

To address this challenge, we propose to first construct a *priority strategy* that guides the LLM to select events and assertions from a smaller set with higher priority, which is more relevant to the target functionality. If this selection fails, we then guide the LLM to select events and assertions from a larger set with lower priority. This priority strategy restricts the selection set, enhancing the accuracy of selecting events and assertions. Furthermore, considering potential inaccuracies using LLMs, we propose to validate the events and assertions selected by the LLM, thereby improving the effectiveness of the generated test cases to test the target functionality.

To implement the preceding idea, Concretizer involves three key modules as depicted in Figure 2: the *Event/Assertion Matching*, *Event/Assertion Completion*, and *Test Validation* module. Specifically, MACdroid utilizes one of existing migration approaches [25, 52, 84] to obtain a set of events and assertions derived from the source test cases. These events and assertions, referred to as *privileged events and assertions*, are relevant to test the functionality of the target app. Based on our designed priority strategy, MACdroid initially matches each test step in the general test logic with these selected events and assertions using the **Event/Assertion Matching** module (see Section 3.2.1). If the matching fails, MACdroid dynamically explores the target app to identify appropriate events and assertions using the **Event/Assertion Completion** module (see Section 3.2.2). Additionally, the **Test Validation** module (see Section 3.2.3) plays a crucial role in validating and correcting any errors that arise. These

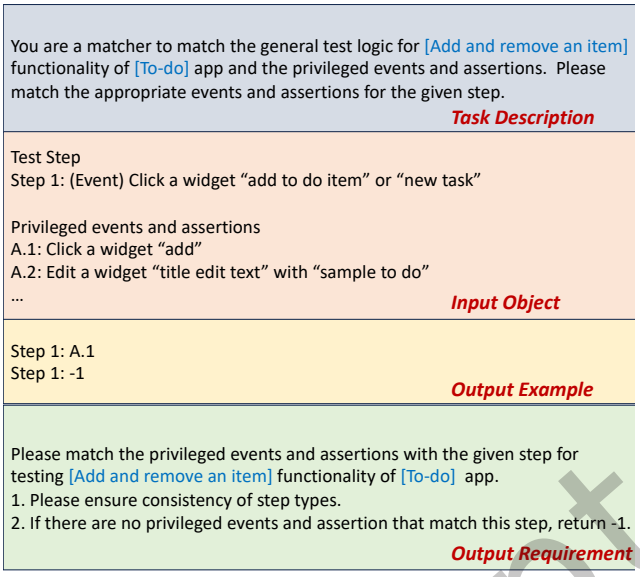

Fig. 4. A prompt example for Event/Assertion Matching

three modules collaboratively work to select appropriate events and assertions from the target app, ensuring the generated test cases are accurate and effective.

Note that, since existing migration approaches may generate incorrect events and assertions or omit essential ones [86] when migrating GUI test cases, we only treat the events and assertions generated by existing migration approaches as privileged but not as the ground-truth.

3.2.1 Event/Assertion Matching. This module aims to match the privileged events and assertions with test steps in the general test logic. By first selecting events and assertions from the small privileged set, rather than from the entire target app, we enhance the LLM to accurately select appropriate events and assertions. By selecting events and assertions based on each structured and concise test step, rather than a block of texts for the whole functionality, we aid the LLM in systematically deconstructing the target functionality and incrementally generating test cases.

Specifically, we utilize the LLM to select events and assertions for each test step in the general test logic. The prompt structure used in this module also follows four parts, as shown in Table 1. Figure 4 is an example for this module.

The main differences between the prompt design in the TL Summarization module and this module are the “input object” and “output requirement”. First, *input object* displays the test step to match and the privileged events and assertions that have not been matched by the preceding test steps. Second, *output requirement* emphasizes two requirements. Specifically, the first requirement emphasizes that not every test step needs to match an event or assertion of the privileged events and assertions because these privileged events and assertions may not fully cover the target functionality. Thus, the LLM should return an unmatched indicator (e.g., “-1” in our design) if no corresponding event or assertion is found. The second requirement emphasizes the need for type alignment. For example, a test step with the event type should only match events but not assertions, thus avoiding incorrect matching by the LLM.

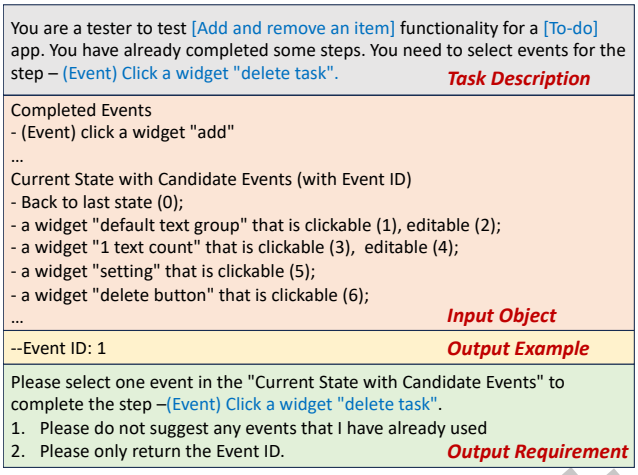

Fig. 5. A prompt example for Event/Assertion Completion

3.2.2 Event/Assertion Completion. This module aims to select events and assertions in the target app when the privileged events and assertions cannot be matched to a given test step. The key parts are *state description*, *event selection*, and *assertion generation*.

State description. To select events and assertions, the LLM needs to understand the semantics of GUI widgets and the actions that these widgets are capable of implementing in the target app. MACdroid converts the current GUI state of the target app into a natural language description to aid the LLM's understanding. For each widget, MACdroid lists the widget semantics and the associated actions. The widget semantics are represented by three key attributes, i.e., *text*, *content-desc*, and *resource-id*. For all widgets within a given state, MACdroid organizes them according to their spatial locations, adhering to a trajectory from the top-left to the bottom-right of the state. This sorting maintains a natural and intuitive flow in the state description. For instance, the "input object" of Figure 5 provides the state description for the "S4" state in Figure 1.

Event selection. Given a test step with event type (referred to as an event step) from the general test logic and the current state, MACdroid selects the appropriate events in the target app to complete this step with the LLM. Figure 5 is an example prompt for this part. The "Input Object" of this prompt not only provides a description of the current state but also displays the selected events from the preceding steps to avoid duplicated selection.

Specifically, MACdroid first generates a prompt (e.g., Figure 5) including an event step and a state description of the current state to the LLM. The LLM returns an event ID from the state description. MACdroid then executes the event corresponding to that event ID for updating the current GUI state and performs a test validation, as detailed in the Test Validation module (see Section 3.2.3). When the test validation passes, MACdroid incorporates the event into the GUI test case. Note that, if the LLM cannot select the appropriate events for the current test step after trying *Max_selection* events (i.e., the test validation cannot pass), MACdroid skips this test step. This discrepancy may arise because the extracted general test logic is designed to be broadly comprehensive for the functionality, aiming to cover various possible implementations. However, the specific implementation of the target app might not incorporate every step outlined in this general test logic.

Assertion generation. An assertion includes a widget and a condition. GUI testing primarily involves two types of conditions [25, 52]. The first type involves checking the presence of a widget in the current state (e.g., A1 in Figure 1), while the second type involves checking the disappearance of a widget in the current state that appears

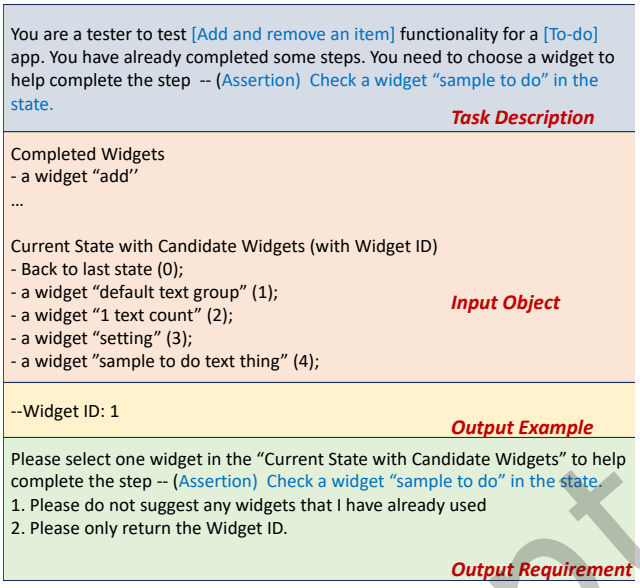

Fig. 6. A prompt example for widget selection

in a preceding state (e.g., A2 in Figure 1). As the widget for the second type of assertions does not appear in the current state, we cannot select assertions in the same way as event selection.

To address this problem, given a test step with the assertion type (referred to as an *assertion step*) from the general test logic and the current state, MACdroid first selects the appropriate widget in the target app and utilizes the widget to generate the corresponding assertion. Unlike event selection, this prompt removes the widget-associated actions, allowing the LLM to focus solely on the widget semantics. A related prompt is Figure 6.

Specifically, for the first type of condition, MACdroid inputs the current state and the current test step into the LLM. The LLM then selects the appropriate widget based on the step description. For the second type of condition, since the widget is not present in the current state but appears in a previous state during the generation of this GUI test case, MACdroid backtracks from the current state to the previous state to identify the widget. After MACdroid identifies the appropriate widget, it generates a corresponding assertion based on the widget and the condition. If the LLM cannot select the appropriate widgets for the current step after trying *Max_selection* widgets, MACdroid skips this step.

3.2.3 Test Validation. Both the Event/Assertion Matching module and Event/Assertion Completion module utilize LLMs, making their outputs susceptible to the inaccuracies inherent in LLMs. To address these issues, we design four rules to identify common issues in these modules and provide feedback to LLMs for repairing them. The overview introduction is shown in Table 2.

Validation on the Event/Assertion Matching module. We check the type and the content of the outputs in this module. There are two common issues (i.e., incorrect type and irrelevant matching) in this module.

Incorrect type. The LLM may incorrectly match assertions with an event step, or vice versa, leading to type faults. To identify this issue, MACdroid compares the type of each test step with the corresponding matched events/assertions. Upon identifying this issue, MACdroid generates a feedback prompt to the Matching module for repairing it. The feedback prompt is "*The type of [step] and the corresponding events/assertions are not aligned. Please re-match this step*".

Irrelevant matching. The outputs of the Matching module may include the events and assertions that do not appear in the privileged set. To identify this issue, MACdroid compares the outputs of the Matching module with the privileged events and assertions. Upon identifying this issue, MACdroid sends the corresponding feedback to the Matching module: "*The [Step] matches new events and assertions. Please re-match this step*".

Validation on the Event/Assertion Completion module. Below are two common issues in this module.

Completion checking. One challenge in using LLMs for GUI test migration is the difficulty in determining the completion of specific test steps due to the comprehension biases [78]. The LLM often continues to select events and assertions for a test step without termination, resulting in the generation of irrelevant events that deviate from the target functionality. To address this issue, we design a checking mechanism to validate whether the current test step has been completed after each event or assertion selection. The checking prompt is "*Based on [Events] or [Assertions] you generated for [Step], I would like to confirm if [Step] has been successfully completed. Please provide a response in just yes or no*". If this mechanism passes (i.e., a response of "yes"), MACdroid proceeds to select events and assertions for the next test step.

Incorrect format. The outputs generated by the Completion module do not adhere to the required formats, which may influence MACdroid to accurately locate the selected events and assertions. To identify this issue, we compare the outputs of the Completion module with the format of the "output example". The feedback is: "*The [Step] does not adhere to the required formats. Please re-generate this step with the provided format*".

4 EVALUATION

To evaluate the effectiveness of MACdroid, we aim to answer the following research questions:

RQ1: How effective is MACdroid compared with the baselines?

RQ2: How do MACdroid's main techniques affect the GUI test migration?

RQ3: How efficient is MACdroid compared with the baselines?

RQ4: How useful is MACdroid in new apps?

4.1 Experimental Setup

Experimental objects. We select two widely-used datasets for evaluating GUI test migration, which are the FrUITeR dataset [10] and the Lin dataset [14]. These datasets include a variety of complex industrial apps (e.g., ABC News [1] and Firefox Browser [9]), which may help to evaluate MACdroid in real-world scenarios. Both these two datasets provide apps along with test cases for target functionalities written by developers (i.e., the ground-truth test cases). The test cases in the Lin dataset contain both events and assertions, while those in the FrUITeR dataset contain only events. We consider all the installable apps and executable test cases provided by the two datasets as our experimental objects. For the entire evaluation experiments, we evaluate MACdroid and related approaches using 31 apps, 34 functionalities, and 123 test cases. Table 3 presents basic statistics of our experimental objects. Notably, different apps within the same category may share the same functionalities. For example, the five apps in the Browser category all share two same functionalities, which require 10 corresponding test cases.

Test case migration involves migrating source test cases from source apps to target apps within the same app category. The source apps and target apps are distinct. In the experimental setup of MACdroid, we employ an iterative methodology in which one app from a selected dataset is designated as the target app, while the remaining apps within the same category of the same dataset are utilized as source apps for migration. For example, there are five apps in the News category of the FrUITeR dataset (see Table 3). In our experiments, we iterate over each of the five apps, designating one as the target app, while the test cases from the other four apps are used as the source test cases and these four apps are used as the source apps. This iterative methodology enables a comprehensive evaluation

Table 3. Statistics of experimental objects

Dataset	Category	App	Functionality	Test	Event	Assertion	Ave_Size
FrUITeR	News	5	12	42	112	-	22M
	Shopping	5	12	39	207	-	20M
	Total	10	24	81	319	-	21M
Lin	Browser	5	2	10	32	20	4M
	To-Do	5	2	10	39	15	2M
	Shopping	4	2	8	49	26	25M
	Mail	2	2	4	14	12	6M
	Calculator	5	2	10	33	10	2M
	Total	21	10	42	167	83	7M

[Volunteer-1] Input a task of "sample to do" and select this item to remove it.

[Volunteer-2] Create a sample to do task in the to-do list by clicking the add button and inputting the sample to do within the app and then delete it.

[Volunteer-3] Click the add task button and fill the task title "sample to do". Then remove it from the task list.

Fig. 7. An illustration of AutoDroid inputs

Baseline approaches. There are two categories of approaches that can generate GUI test cases, i.e., migration approaches [25, 40, 52, 55, 84] and generation approaches [34, 74, 78, 79]. Existing migration approaches migrate source test cases to target apps based on widget mapping. Generation approaches require a manually crafted test logic as the input, and use LLMs to select appropriate events in the target app based on test logic. To comprehensively evaluate MACdroid, we employ representative approaches from both categories. Specifically, we compare MACdroid with TEMdroid [84] (the state-of-the-art migration approach) and AutoDroid [78] (the state-of-the-art generation approach).

Note that, generation approaches (including AutoDroid) require manually crafted test logics as inputs, but the FrUITeR and the Lin datasets do not provide this information. To compare MACdroid with AutoDroid on these two datasets, we invite volunteers to write the necessary test logics. We mitigate the potential influence of different writing styles on the effectiveness of AutoDroid by engaging three volunteers[1] with industrial experience in Android programming ranging from 3 to 5 years. For each volunteer, we provide the example descriptions from AutoDroid, the target apps, and the ground-truth test cases for the target functionalities. Each volunteer independently writes the descriptions for all the functionalities to be tested in the two datasets. Since AutoDroid cannot generate assertions, we instruct the volunteers to omit descriptions related to assertions. For example, AutoDroid utilizes three manually crafted test logics (see Figure 7) as the test logics of Figure 1.

Evaluation metrics. To evaluate MACdroid and the baselines, we design three metrics: executable-rate, success-rate, and perfect-rate.

Executable-rate. This metric is calculated as the ratio of migrated test cases that can be fully executed ($Test_{exe}$) to the total number of migrated test cases for the target functionalities ($Test_t$).

[1]None of the volunteers are co-authors of this paper.

$$Executable\text{-}rate = Test_{exe} \,/\, Test_t \tag{1}$$

Perfect-rate. This metric is calculated as the ratio of migrated test cases aligning with the ground-truth test cases ($Test_{gt}$) to the total number of migrated test cases for target functionalities ($Test_t$).

$$Perfect\text{-}rate = Test_{gt} \,/\, Test_t \tag{2}$$

Success-rate. This metric is calculated as the ratio of migrated test cases that successfully test the target functionalities ($Test_{suc}$) to the total number of migrated test cases for the target functionalities ($Test_t$). Note that, migrated test cases being fully executable is a prerequisite for successful testing the target functionalities. Additionally, migrated test cases that align with the ground-truth target test cases are a subset of test cases that successfully test the target functionalities [53, 55, 86]. This adheres to the purpose of test migration, which seeks to utilize migrated test cases as replacements for manually written target test cases.

$$Success\text{-}rate = Test_{suc} \,/\, Test_t \tag{3}$$

Test cases that successfully test one functionality are not necessarily unique. Consequently, for those test cases that are fully executable but do not align with the ground-truth, we adopt a manual check to further investigate whether these test cases still successfully test their target functionalities. These test cases may include not only all the events and assertions of the ground-truth test cases but also additional events and assertions that do not hinder the testing of the specific functionalities. Specifically, we invite the same three volunteers who have written the test logics for the two datasets (see "Baseline approaches" of Section 4.1) to help check these additional events and assertions. For each manual check, we provide the volunteers with the generated test case, the additional events or assertions to be checked, the target app, and the corresponding ground-truth test case. Each volunteer independently checks the events and assertions. In cases of disagreement, the volunteers discuss until they reach a consensus.

Note that, the goal of test case migration is to generate test cases that successfully test a functionality of a target app. However, as it is often not feasible to obtain all test cases that successfully test a specific functionality, existing migration approaches [25, 40, 86] typically rely on pre-existing test cases from the datasets that are designed to test the target functionality, referring to them as "*ground-truth test cases*". The "ground-truth test cases" serve as a *reference set*, representing a subset of test cases that successfully test the target functionalities. Since it is impossible to exhaustively capture all the ground-truth for the target functionality of a given app, we adopt a combined method that incorporates both automated and manual evaluation.

Parameter selection. MACdroid needs three parameters, which are Max_ratio used in the Abstractor component, $Max\_selection$ used in the Concretizer component, and Tem used for LLMs. The determinations of specific parameter values are described as follows.

First, Max_ratio measures whether the generated test logic remains within a reasonable length. If the generated test logic is too long, it may result in redundancy and irrelevant steps. To determine the value of Max_ratio, we compare the length of the general test logic to that of the longest source test case. Specifically, we set the minimum value of Max_ratio to 1, as general test logic typically encompasses or exceeds the scope of the individual source test cases. Additionally, we set the maximum value of Max_ratio to 2, as the individual test logics of source test cases targeting the same functionality exhibit substantial overlap, with only minor variations between them. Based on these considerations, we select Max_ratio values of 1, 1.5, and 2 as reasonable ranges.

Second, $Max\_selection$ defines the maximum number of attempts that the LLM makes to select candidate events for a given test step. The design of this parameter aims to balance testing effectiveness with cost, response time, and the inherent uncertainty of LLM outputs. We set the minimum number of $Max\_selection$ to 1, representing the simplest scenario where the LLM tries only once for each test step. It minimizes cost and response time but

may result in suboptimal selection. We set the maximum number of *Max_selection* to 3 to control costs and response time. Allowing more than three attempts would significantly increase the cost of LLM invocations and lead to longer response times, which could negatively impact user experience in commercial applications. By selecting 1, 2, and 3 as the range for *Max_selection*, we achieve a balanced trade-off between cost, efficiency, and the quality of the generated results.

Third, *Tem* is used to determine the appropriate temperature parameter for the LLM to generate higher-quality test cases. In our experiments, we select the GPT series models [6, 13], which have an official temperature parameter ranging from 0 to 2. At temperature 0, the LLM's output is completely deterministic, which leads to low variability. At temperature 2, the randomness of the outputs is maximized, leading to highly unpredictable results. We aim to generate test cases that are both stable and flexible. Therefore, the extremes of 0 and 2 do not meet our requirements. We select 0.4, 0.8, 1.2, and 1.6 as the candidate values for the temperature parameter.

In summary, the candidate parameters for *Max_ratio*, *Max_selection*, and *Tem* are {1, 1.5, 2}, {1, 2, 3}, and {0.4, 0.8, 1.2, 1.6}, respectively. We randomly select 10% of the total apps in the Lin dataset as a validation set for parameter selection. After conducting experiments with the validation set, we observe that setting *Max_ratio* to 1.5, *Max_selection* to 3, and *Tem* to 0.4 yields the best effectiveness. Thus, all the experiments utilize this configuration.

Common setting. We implement MACdroid in Python to support Android [2] apps. The experiments are based on a Pixel 3 Emulator running Android 6.0. Some apps require installation in this setup, but MACdroid can adapt to others. For the LLM that MACdroid utilizes in evaluation, we select two widely-used models: GPT-3.5 [6] (i.e., the "gpt-3.5-turbo-0613" model used in this evaluation) and GPT-4.0 [13] (i.e., the "gpt-4-0613" model used in this evaluation) to compare the effectiveness of utilizing different LLMs.

To assess the effectiveness of MACdroid, we conduct evaluations across different apps, baselines, and LLMs. Specifically, we evaluate MACdroid, TEMdroid [84], and AutoDroid [78] on the FrUITeR dataset and the Lin dataset, respectively. Both MACdroid and AutoDroid need to interact with an LLM, for which we evaluate the effectiveness of these approaches using two different LLMs, i.e., GPT-3.5 [6] and GPT-4.0 [13]. Due to budget constraints, we evaluate the effectiveness of MACdroid and AutoDroid on the two full datasets using GPT-3.5. When using GPT-4.0, we randomly select half apps in each app category of the two datasets. Considering the inherent randomness of LLMs, we run each experiment three times and report the average results.

4.2 RQ1: Effectiveness

We evaluate the effectiveness of MACdroid, and compare it with two baselines (i.e., TEMdroid [84], and AutoDroid [78]) using the Executable-rate, Perfect-rate, and Success-rate on the FrUITeR dataset and the Lin dataset, respectively. Additionally, we calculate the statistical significance using the Mann-Whitney U-Test [64] and effect size using the Cohen's d [24] between MACdroid and the baseline approaches.

Effectiveness results. Table 4 shows the overall effectiveness of the test cases generated by MACdroid, TEMdroid, and AutoDroid on the FrUITeR dataset and Lin dataset, using both GPT-3.5 and GPT-4.0 as the LLMs. For the results related to GPT-3.5, we further analyze the effectiveness of these approaches across different app categories (see Table 5) and provide the significance and effect sizes of these approaches by category (see Table 6). We count the executable-rate (denoted as "Exec."), perfect-rate (denoted as "Perf."), and success-rate (denoted as "Suc.") of MACdroid and baselines. We also separately count the impact of the test logics written by the three volunteers (denoted as "Vol.") on the effectiveness of AutoDroid.

Effectiveness on the FrUITeR dataset. All the test cases generated by MACdroid are fully executable (i.e., 100%). When utilizing GPT-3.5 as the LLM, 64% of the test cases generated by MACdroid successfully test the target functionalities (i.e., success-rate), and 18% of them align with the corresponding ground-truth (i.e., perfect-rate). These results surpass TEMdroid and AutoDroid by over 191% in success-rate and 29% in perfect-rate. Note that,

Table 4. Overall effectiveness of MACdroid and the baselines

Dataset	Approach	LLM	Vol.	Exec.	Perf.	Suc.
FrUITeR	MACdroid	GPT-3.5	-	**100%**	**18%**	**64%**
		GPT-4.0	-	100%	16%	57%
	AutoDroid	GPT-3.5	V-1	100%	1%	7%
			V-2	100%	2%	9%
			V-3	100%	1%	6%
		GPT-4.0	V-1	100%	3%	11%
			V-2	100%	5%	13%
			V-3	100%	5%	12%
	TEMdroid	-	-	64%	14%	22%
Lin	MACdroid	GPT-3.5	-	**100%**	**57%**	**75%**
		GPT-4.0	-	100%	68%	77%
	TEMdroid	-	-	77%	46%	53%
	MACdroid*	GPT-3.5	-	**100%**	**61%**	**86%**
		GPT-4.0	-	100%	68%	89%
	AutoDroid	GPT-3.5	V-1	100%	2%	16%
			V-2	100%	3%	13%
			V-3	100%	2%	18%
		GPT-4.0	V-1	100%	8%	18%
			V-2	100%	14%	27%
			V-3	100%	14%	39%

Table 5. Effectiveness of MACdroid and the baselines by category

Dataset	Category	MACdroid			TEMdroid			AutoDroid		
		Exec.	Perf.	Suc.	Exec.	Perf.	Suc.	Exec.	Perf.	Suc.
FrUITeR	News	100%	21%	71%	42%	18%	19%	100%	1%	6%
	Shopping	100%	15%	58%	79%	12%	23%	100%	2%	8%
Lin	Browser	100%	55%	100%	85%	85%	85%	100%	0%	22%
	To-Do	100%	50%	70%	58%	28%	28%	100%	1%	9%
	Shopping	100%	13%	25%	54%	4%	17%	100%	0%	0%
	Mail	100%	88%	88%	100%	100%	100%	100%	0%	8%
	Calculator	100%	90%	90%	100%	45%	65%	100%	9%	32%

Table 6. Significant difference and Effect size of MACdroid and the baselines

Dataset	Category	MACdroid vs TEMdroid		MACdroid vs AutoDroid	
		Significance	Effect size	Significance	Effect size
FrUITeR	News	1.97E-19	0.49	2.31E-51	0.49
	Shopping	2.87E-10	0.34	7.24E-29	0.36
Lin	Browser	0.03	0.13	9.44E-14	0.58
	To-Do	4.65E-03	0.36	1.59E-11	0.46
	Shopping	0.49	0.07	1.36E-05	0.18
	Mails	0.45	-0.12	5.89E-07	0.55
	Calculator	0.04	0.18	2.49E-07	0.41

when GPT-4.0 is used, the test cases generated by MACdroid also show a slight improvement in the success-rate (i.e., 57% in Table 4) compared to using GPT-3.5, which achieves a 54% success-rate in the same *half* of the total apps.

Effectiveness on the Lin dataset. When using GPT-3.5 as the LLM, 75% of the test cases generated by MACdroid successfully test the target functionalities, and 57% of them align with the ground-truth test cases. These results outperform TEMdroid by 42% in success-rate and 24% in perfect-rate. We also separately calculate the accuracy of generated assertions by MACdroid and TEMdroid at the case level. MACdroid achieves the assertion accuracy of 83%, compared to 75% for TEMdroid.

The test cases in the Lin dataset include both events and assertions, but AutoDroid cannot generate assertions. To fairly compare MACdroid with AutoDroid on the Lin dataset, we evaluate the test cases generated by MACdroid and AutoDroid without considering the generated assertions. In this scenario, MACdroid (denoted as "MACdroid*") outperforms AutoDroid by more than 378% in success-rate and 1933% in perfect-rate. Additionally, switching from GPT-3.5 to GPT-4.0 also slightly improves the effectiveness of MACdroid.

Statistical analysis. Table 6 presents the significant differences and effect sizes between MACdroid and TEMdroid, as well as between MACdroid and AutoDroid. In this table, red numbers indicate statistically significant differences or large effect sizes, while black numbers indicate no significant difference or not large effect sizes. As can be seen, MACdroid shows statistically significant differences compared to both TEMdroid and AutoDroid, with large effect sizes.

Note that, all results from the FrUITeR dataset exhibit statistical significance and large effect sizes, whereas not all results from the Lin dataset show similar trends. The reason for this phenomenon lies in the difference in the number of test cases per category in the two datasets. Specifically, the FrUITeR dataset contains an average of 41 different test cases for each category, while the Lin dataset contains only an average of 8 test cases per category. As statistical analysis requires a sufficient number of samples, the small number of test cases per category for the Lin dataset are difficult to show significant differences.

Result analysis. We have several findings from Table 4.

First, compared to TEMdroid, MACdroid significantly improves the success-rate (e.g., 22% vs. 64% on the FrUITeR dataset). Existing migration approaches only migrate a test case to the target app, but the same functionality can be implemented differently across various apps. As a result, the migrated test case may only partially test the functionality of the target app. In contrast, MACdroid extracts the general test logic from multiple test cases to provide a comprehensive perspective of the target functionality, which is conducive to completely testing the target functionality.

Second, the test cases generated by TEMdroid may not be fully executable (see the column of "Exec."). This is because existing migration approaches migrate test cases based on widget mapping, but the migrated events and assertions may miss some connection events in the target app, making the generated test case not fully executable. In contrast, both MACdroid and AutoDroid generate test cases by incrementally selecting and executing events within the target app, thereby ensuring that all connection events are retained and the generated test cases are fully executable.

Third, compared to AutoDroid, MACdroid significantly improves the perfect-rate (e.g., 3% vs. 61% on the Lin dataset). Existing generation approaches struggle to generate perfect test cases because the provided test logics may be vague and ambiguous, making it challenging for the LLM to determine whether the functionalities have been fully tested, thus including irrelevant events. In contrast, MACdroid splits a whole test logic into a sequence of structured test steps, guides the LLM to generate test cases step-by-step, and verifies the completion of each step. This approach effectively reduces the generation of irrelevant events and assertions, facilitating the generation of perfect test cases.

Fourth, even when AutoDroid utilizes GPT-4.0, it still does not perform as well as MACdroid utilizing GPT-3.5. This situation indicates that simply upgrading the LLM does not substantially enhance the generated test cases. Instead, providing the LLM with high-quality test logics, restricting the input candidates for the LLM, and repairing potential errors are important for improvement.

Fifth, the effectiveness of generated test cases for AutoDroid varies depending on the test logics written by different volunteers. For example, the success-rates of AutoDroid on the FrUITeR dataset are 7%, 9%, and 6%, respectively when utilizing the descriptions from the three volunteers. This variability underscores the impact of human factors on the robustness of existing generation approaches. Instead, the general test logic extracted by MACdroid not only automates the generation of test logics but also ensures stable quality for the LLM to generate test cases.

Sixth, according to Table 5 we can observe that MACdroid achieves high effectiveness across different app categories and outperforms the baselines, demonstrating the robustness of MACdroid.

Seventh, the accuracy of assertion generation is also evaluated at the test case level, as the test case serves as the fundamental unit for functional testing. The accuracy is defined as the condition where all assertions generated for a test case are completely consistent with the assertions in the ground-truth test cases. Assertions play a crucial role in functional testing [33, 43]. Among the test cases generated by MACdroid, 83% of them include assertions that successfully meet the requirements for testing the corresponding functionalities. This result indicates that MACdroid is capable of generating high-quality assertions effectively and demonstrates strong effectiveness in test case migration compared to related approaches.

Failure Analysis. To understand the weaknesses of MACdroid, we manually analyze all failure test cases and identify three main reasons. We select three examples (see Figure 8) to introduce the failure reasons of MACdroid and also to provide the potential solutions to address these failures.

First, a limited number of source test cases may not cover all the necessary test steps for the target functionality, leading to the generated test case not fully testing this functionality. For example, the state (a) of Figure 8 shows a registration functionality. MACdroid does not generate the necessary event (i.e., E1) because the source test cases do not include steps related to confirmation. Increasing the diversity of source test cases, while leveraging the general knowledge of LLMs to supplement necessary steps for general test logic, may potentially help address this problem.

Second, the Event/Assertion Matching module may incorrectly match events and assertions to the corresponding test steps, resulting in the generated test cases containing incorrect events and assertions. For example, states (b) and (c) of Figure 8 illustrate a functionality designed to test the terms. The state (b) represents a part of the source test case, while the state (c) shows the corresponding part of the target test case. Given the E2 event from the source test case, the Event/Assertion Matching module incorrectly matches E4 event instead of the

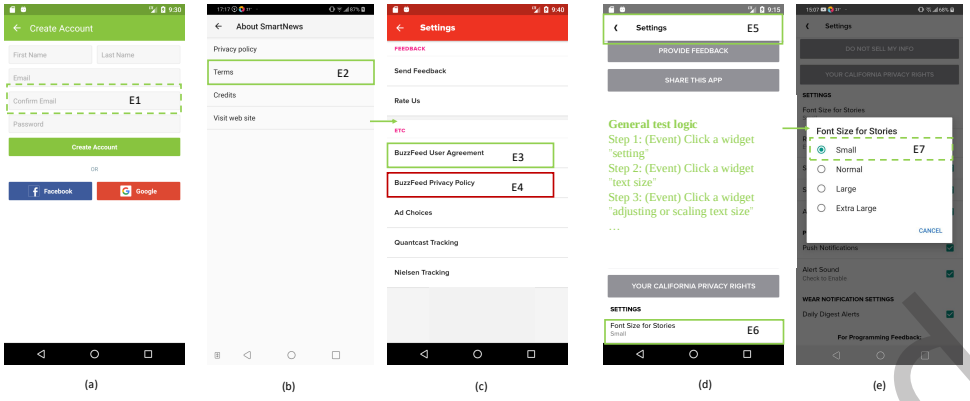

Fig. 8. Failure case study

correct event, i.e., E3. This happens because the content descriptions of E4 (i.e., privacy policy) and E2 (i.e., terms) are related, both falling under the broader agreement category, making the matching process prone to error. Incorporating the state information into the matching process and utilizing more accurate matching algorithms may help improve the effectiveness of the Event/Assertion matching module.

Third, the Event/Assertion Completion module may also select incorrect events and assertions or do not select any events and assertions, impacting the generated test cases to successfully test the target functionality. We illustrate this issue using states (d) and (e) in Figure 8, which depict the functionality of changing text size. In this case, the Event/Assertion Completion module follows a general test logic (highlighted in green in the state (d)) to generate events E5 and E6 for the first two steps of the general test logic. However, the module determines that no events are suitable for Step 3, leading to the omission of an event selection for this step. As a result, the E7 event is missing from the final generated test case. Enhancing the Completion module with more self-learning capabilities, and ensuring it fully understands the semantics behind each step of the general test logic, could help mitigate this issue.

> **Answer to RQ1:** MACdroid is effective and substantially outperforms the baselines in GUI test migration.

4.3 RQ2: Main Techniques

We evaluate the contributions of the main techniques used in MACdroid based on metric evaluation and statistical analysis on the FrUITeR dataset. The LLM used is GPT-3.5.

Experimental setting. There are four experiments. First, we evaluate the effectiveness of Abstractor in MACdroid. Specifically, we modify AutoDroid by replacing its manually crafted test logics with the extracted general test logics by Abstractor, denoted as "AutoDroid (with Abstractor)". We then compare the effectiveness of the original AutoDroid with "AutoDroid (with Abstractor)". Second, we investigate the effectiveness of Concretizer in MACdroid. Specifically, we compare the test cases generated by "AutoDroid (with Abstractor)" with those generated by the original MACdroid, as they both use the same test logics but different generation approaches. Third, we investigate the effectiveness of the validation modules used in the Abstractor component and the Concretizer component of MACdroid. Specifically, we compare the test cases generated by MACdroid without these validation modules (denoted as "MACdroid (without Validation)") with those generated by the original MACdroid. Fourth, we investigate the robustness of MACdroid when using different sources of privileged

Table 7. Effectiveness of main techniques in MACdroid

Approach	Executable	Perfect	Success
MACdroid	**100%**	**18%**	**64%**
AutoDroid	100%	2%	9%
AutoDroid (with Abstractor)	100%	9%	27%
MACdroid (without Validation)	100%	4%	31%
MACdroid (with AppFlow)	100%	15%	59%

Table 8. Significant difference and Effect size of MACdroid and the main techniques

Approach	Significance	Effect size
AutoDroid (with Abstractor) vs AutoDroid	5.79E-12	0.12
MACdroid vs AutoDroid (with Abstractor)	4.67E-21	0.37
MACdroid vs MACdroid (without Validation)	1.21E-07	0.29
MACdroid vs MACdroid (with AppFlow)	0.38	0.04

events and assertions (i.e., different migration approaches). Specifically, we select AppFlow [40], a representative migration approach that has been extensively compared in numerous studies [61, 84, 86]. Then, we compare the test cases generated by the original MACdroid that uses privileged events and assertions from TEMdroid [84] with the results generated by MACdroid (denoted as "MACdroid (with AppFlow)") that uses privileged events and assertions from AppFlow [40].

Effectiveness results. Table 7 presents the executable-rate, perfect-rate, and success-rate of this evaluation. First, by comparing the second row with the third row, we observe that using the general test logics generated by the Abstractor component significantly improves the effectiveness of AutoDroid (e.g., 27% vs. 9% in the success-rate). This result indicates that the general test logics generated by the Abstractor component provide better guidance for the LLM compared to manually crafted test logics, making the LLM easier to understand the functionalities being tested. Second, by comparing the first row with the third row, we observe that even with the general test logics, AutoDroid does not perform as well as MACdroid (e.g., 27% vs. 64% in the success-rate). This result suggests that the priority strategy, along with the test validation mechanisms, are also crucial for generating high-quality test cases. Third, a comparison between the first and fourth rows reveals that the LLM tends to generate some inaccurate results, highlighting the importance of the validation modules used in the Abstractor component and the Concretizer component (e.g., 64% vs. 31% in the success-rate). Fourth, a comparison between the first and fifth rows indicates that MACdroid is reliable and robust when employing various sources of privileged events and assertions (e.g., 64% vs. 59% in the success-rate).

Statistical analysis. Table 8 presents the significant differences and effect sizes for the following four comparisons, which are AutoDroid (with Abstractor) versus the original AutoDroid, the original MACdroid versus AutoDroid (with Abstractor), the original MACdroid versus MACdroid (without Validation), and the original MACdroid versus MACdroid (with AppFlow).

In Table 8, the red numbers indicate significant differences or large effect sizes, while the black numbers represent no significant differences or not large effect sizes. The statistical results show that the Abstractor component, Concretizer component, and the validation module are crucial in MACdroid. When these modules are removed, significant differences are observed compared to the original version. All results in the fourth row

Table 9. Efficiency of MACdroid and the baselines

Dataset	MACdroid		TEMdroid		AutoDroid	
	Runtime	Token	Runtime	Token	Runtime	Token
FrUITeR	5.1	6378	9.6	18	5.5	6496
Lin	4.7	8256	8.8	14	4.5	9425

are black, indicating no significant differences when changing different privileged events and assertions. This result also demonstrates that the MACdroid approach exhibits high reliability and robustness, not relying on specific sources of privileged events and assertions.

Answer to RQ2: MACdroid's main techniques substantially contribute to GUI test migration.

4.4 RQ3: Efficiency

To assess the efficiency of MACdroid, we evaluate the execution time and the token usage of MACdroid, TEMdroid [84], and AutoDroid [78] on the FrUITeR dataset and the Lin dataset, respectively.

Efficiency results. Table 9 shows the average runtime and token-usage per test case for MACdroid, TEMdroid, and AutoDroid. The average runtime for MACdroid per test case is 5.1 minutes on the FrUITeR dataset and 4.7 minutes on the Lin dataset. These results are faster than TEMdroid and comparable to those of AutoDroid. The average token usage for MACdroid per test case is 6378 on the FrUITeR dataset and 8256 on the Lin dataset, which is smaller than that of AutoDroid. Note that, TEMdroid fine-tunes its own model based on BERT [5] instead of utilizing LLMs, resulting in the lowest token usage.

The time cost of MACdroid is primarily influenced by its two components, i.e., Abstractor and Concretizer. For a target functionality, MACdroid spends less than 0.5 minutes on average when generating a general test logic using the Abstractor component. The majority of the time is consumed by the Concretizer component in generating test cases. This is because Abstractor interacts with the LLM fewer times than Concretizer does. Ideally, the Abstractor component only needs to interact with the LLM twice (once to generate the test logic and the other to validate it). However, the Concretizer component requires multiple interactions with the LLM to select events for each test step, including matching, completion, and validation. Reducing the number of interactions between the Concretizer component and the LLM would significantly improve MACdroid's efficiency.

Answer to RQ3: MACdroid's efficiency is comparable to the baselines.

4.5 RQ4: Usefulness

To further evaluate the usefulness of MACdroid, we conduct a study evaluating MACdroid in new apps. The LLM used in this evaluation is GPT-3.5.

Experimental objects. We leverage a new dataset [17], referred to as the TEM dataset in this study, which was developed as part of the research on TEMdroid [84]. The TEM dataset incorporates all the apps and test cases from the Lin dataset [14] as source apps and source test cases. These source test cases include both events and assertions, which enables the evaluation of MACdroid's effectiveness in generating comprehensive test cases. The TEM dataset includes five new target apps, which are popular apps in the Google Play Store [12], with each app belonging to a distinct category in the Lin dataset. Note that, the TEM dataset does not provide the ground-truth test cases for the target apps.

Table 10. Results of the usefulness study in MACdroid

App	Executable	Success
Web Browser	100%	67%
Done	100%	83%
Fivemiles	100%	50%
Pro Mail	100%	83%
Tip Calculator	100%	83%
Average	**100%**	**73%**

Evaluation process. To assess the effectiveness of MACdroid, we adopt a similar evaluation process as described in Section 4.1 and engage the same three volunteers. Since the TEM dataset does not include ground-truth test cases, we are unable to provide the volunteers with the ground-truth target test cases or evaluate the perfect-rate metric. To address this issue, we provide the volunteers with ground-truth source test cases for the same functionality, helping them gain a clearer understanding of the target functionality. Thus, the evaluation metrics for this study are executable-rate and success-rate.

Usefulness results. Table 10 presents the executable-rate and success-rate of the test cases migrated by MACdroid on the TEM dataset. In this study, MACdroid achieves an executable-rate of 100% and a success-rate of 73%. From Table 10 we observe that MACdroid demonstrates significant effectiveness across different app categories, highlighting its satisfactory usefulness in new apps.

Answer to RQ4: MACdroid shows satisfactory usefulness in new apps.

4.6 Threats to Validity

A possible threat to external validity is the generalizability to other datasets. To mitigate this threat, we use the largest number of apps and app categories compared with related work. Moreover, we use all the popular datasets in test case migration, i.e., the Lin dataset [52] and the FrUITeR dataset [10], as evaluation benchmarks. These datasets include a variety of complex industrial apps (e.g., ABC News [1] and Firefox Browser [9]), which may help to evaluate MACdroid and the baselines in real-world scenarios.

A possible threat to internal validity is the possible mistakes involved in our implementation and experiments. To mitigate this threat, we manually inspect our results and analyze the test cases that fail to test the target functionalities. We also publish our implementation and experimental data, which welcome external validation. As for the human evaluation, we invite three experienced developers and provide them with a clear evaluation process.

A possible threat to construct validity is about evaluation metrics. To mitigate this threat, we carefully design three metrics aiming at validating the effectiveness of the generated test cases. These metrics offer a reliable and objective basis for assessing the quality of the test cases generated by both the baselines and MACdroid.

5 DISCUSSION

Collecting and retrieving source test cases. Migration approaches typically leverage source test cases as input to migrate them into target apps. These approaches reduce the cost of manually writing GUI test cases for the target apps and enable the reuse of existing GUI test cases. MACdroid is also a migration approach that

significantly outperforms existing approaches (see Section 4.2). However, applying migration approaches to industry still needs to consider the collection and retrieval of source test cases.

Collection. In real-world scenarios, there are many similar apps and corresponding GUI test cases [40, 62, 86]. For example, app stores (e.g., Google Play Store [12] and F-Droid [8]) organize similar apps into the same categories (e.g., shopping and news). Additionally, open-source communities (e.g., GitHub [11]) host numerous mobile apps with test cases. Leveraging these open-source test cases and existing datasets [10, 14], collecting apps and GUI test cases becomes a straightforward process.

Retrieval. These apps and test cases can be organized by app category and the specific functionalities within each category. Cluster-based algorithms (e.g., K-Means [46]) can be used to group test cases based on functionality names and test case annotations. Each cluster contains test cases corresponding to specific functionalities within the same category. Given a target app, its category, and the functionality name to be tested, the closest cluster in the collected dataset can be identified, and all test cases associated with that cluster can be retrieved.

Impact and solution of the quality of source test cases. The quality of source test cases impacts the effectiveness of MACdroid. Below we first detail analyze the impact of the high-quality source test cases and the low-quality ones on the effectiveness of MACdroid. We further analyze the potential solution to the test case selection.

Impact. High-quality source test cases, which are representative of testing a target functionality with minimal app-specific information (e.g., environment-specific details), may enhance MACdroid's effectiveness. Such test cases effectively encapsulate the core logical information required to test a target functionality while minimizing interference from irrelevant app-specific details. As a result, the general test logic derived from high-quality source test cases accurately reflects the essence of the target functionality and can be seamlessly instantiated for the target app.

In contrast, low-quality source test cases, which lack representativeness or include excessive app-specific details, may lead to incomplete or noisy test logic generation. This deficiency may lead to inconsistencies between the generated test logic and the actual requirements for testing the target app. Consequently, the test cases derived from such logic may fail to fully test the target functionality, either by only partially testing it or containing numerous errors, and thus still require manual modification.

Solution. There are two ways to potentially improve the quality of source test cases. First, the selection of source test cases could be diversified. By selecting test cases from different scenarios and across various mobile apps that test the same functionality, we may obtain a broader and more accurate representation of how the functionality should be tested. Second, filtering out low-quality test cases is also essential. By filtering out test cases that have incomplete test logics or rely heavily on app-specific information, we can reduce interference from irrelevant details, thereby facilitating the accurate generation of the general test logic and the concrete test case.

Branch coverage of MACdroid and the baselines. Branch coverage [36, 38, 67] is a commonly-used code coverage metric. Compared to method coverage or block coverage, branch coverage imposes a higher standard and requires higher qualities for test cases.

In Section 4, we evaluate the effectiveness of MACdroid, TEMdroid, and AutoDroid from three perspectives. These perspectives test whether the test cases are fully executable test cases (i.e., executable-rate); whether the test cases successfully test the target functionality (i.e., success-rate); and whether the test cases align with the ground-truth test cases (i.e., perfect-rate). To provide a comprehensive understanding of these approaches, we also discuss the branch coverage of the test cases generated by MACdroid, TEMdroid, and AutoDroid compared with that of the ground-truth test cases. Specifically, we design a new metric, *coverage-capability*. For a given app, this metric is calculated as the ratio of common covered branches ($Cover_{common}$) between the generated test

Table 11. Branch Coverage of MACdroid and the baselines

Dataset	Category	App	Coverage number				Coverage capability		
			GT.	MAC.	TEM.	Auto.	MAC.	TEM.	Auto.
FrUITeR	News	ABC News	21775	24498	14072	16725	83%	60%	53%
		BuzzFeed	22146	19299	29640	19202	79%	84%	78%
		Fox News	27120	26496	13309	20521	94%	47%	51%
		Reuters News	10272	10508	10219	16492	99%	82%	82%
	Shopping	Etsy	12561	11896	7249	8621	94%	54%	64%
		Geek	13860	12273	21481	10654	86%	77%	76%
		Wish	9055	8969	8903	8903	94%	93%	93%
	Average		16684	16277	14982	14445	**89%**	**67%**	**67%**
Lin	Browser	Lightning	14294	13962	5344	5061	97%	37%	35%
		Privacy Browser	3400	3447	3378	2749	96%	99%	81%
		FOSS Browser	1432	1432	1432	827	100%	100%	58%
		Firefox	13217	13157	13324	2186	99%	98%	16%
	To-Do	Minimal	2554	11539	2501	1543	98%	98%	49%
		Clear List	1723	1724	1732	1703	100%	100%	99%
		To-Do	3211	11856	2631	3376	91%	82%	91%
		Simply Do	50	50	36	42	100%	72%	80%
		Shopping List	3144	10944	2136	2688	88%	62%	58%
	Shopping	Geek	9897	7236	6979	6917	71%	68%	66%
		Yelp	18291	17265	10634	10591	92%	57%	57%
		Etsy	10663	11238	9009	9366	98%	82%	84%
		Wish	9053	17854	5049	5130	98%	56%	56%
	Mail	K-9 Mail	3355	3216	3355	1433	95%	100%	43%
		Fast Email	2584	2586	2584	1592	99%	100%	60%
	Calculator	Tip Calculator	212	212	190	190	100%	89%	89%
		Simple Tip	1773	1626	1625	1810	92%	92%	100%
		Tip Plus	2619	2598	2438	2601	99%	93%	99%
		Free Tip	1521	1450	1412	1507	95%	93%	99%
	Average		5421	7021	3989	3227	**94%**	**72%**	**57%**

cases and the ground-truth test cases, to the total covered branches ($Cover_{gt}$) by the ground-truth test cases. We use the WALLMANUER [21] tool to calculate the branch coverage for each app.

Note that, coverage-capability and the metrics in Section 4 are evaluated from different perspective. In this way, the result trends of MACdroid, TEMdroid, and AutoDroid presented here and in Section 4 may be related but different.

$$Coverage\text{-}capability = Cover_{common} \,/\, Cover_{gt} \tag{4}$$

Table 11 shows the branch coverage results for MACdroid (denoted as MAC.), TEMdroid (denoted as TEM.), and AutoDroid (denoted as Auto.) across all successfully *instrumented* apps from the Lin and FrUITeR datasets. For each approach, we report the number of covered branches and the coverage-capability. We also report the number of branches covered by the ground-truth (denoted as GT.) test cases for reference.

Specifically, on the FrUITeR dataset, 89% of the branches covered by the ground-truth test cases can be covered by MACdroid, while TEMdroid and AutoDroid can only cover 67% and 67%, respectively. The results on the Lin dataset are similar, with MACdroid covering 94% of the branches covered by the ground-truth test cases, while TEMdroid and AutoDroid only cover 72% and 57%, respectively. These results indicate that MACdroid achieves the closest match to the ground-truth in terms of branch coverage, outperforming TEMdroid and AutoDroid. Note that, the FrUITeR and Lin datasets used in this study are among the most widely-used datasets in test case migration. Both datasets include industry-level apps (e.g., ABC News [1] and Firefox [9]), making them effectively evaluate test case migration approaches in real-world environments.

6 RELATED WORK

GUI test migration. Several approaches [25, 40, 52, 55, 62, 84] have been proposed for migrating GUI test cases between different apps. AppFlow [40] utilizes a trained multi-classifier to identify widget labels, considering widgets with the same widget labels as mapped widgets. ATM [25], Craftdroid [52], TRASM [55], and Adaptdroid [62] leverage different word embeddings [47, 65] to represent words in widgets and employ a manually defined matching function to map widgets. TEMdroid [84] is the first approach that trains a matching model for widget mapping. After mapping widgets, these migration approaches generate the corresponding events and assertions for the target apps. MigratePro [85] is a related approach for GUI test migration that seeks to improve migration techniques through test case synthesis.

The key difference between existing migration approaches and MACdroid lies in the distinct paradigms they follow for GUI test migration. Existing migration approaches follow the widget-mapping paradigm, which maps widgets from source test cases to target test cases. In contrast, MACdroid is based on the abstraction-concretization paradigm, which first abstracts the general test logics for the target functionalities and then uses these logics to guide the LLM to concretize the target test cases. Compared with existing migration approaches, the test cases generated by MACdroid show significant improvements (see Section 4.2).

Functional GUI test generation. Several approaches [34, 42, 44, 49, 74, 78, 79] aim to generate GUI test cases for apps in different systems. Among them, four approaches [44, 49, 78, 79] target the Android system. Li et al. [49] input manually crafted test logics and manually selected candidate app screenshots for the target functionality and utilized a matching model to select events appearing in the screenshots. However, this approach only supports one action (i.e., click), limiting its applicability in real-world scenarios. FARLEAD-Android [44] requires users to provide formal specifications as inputs, which poses a significant challenge for user adoption. DroidBot-GPT [79] utilizes LLMs to select events based on manually crafted test logics. AutoDroid [78], an advanced version of DroidBot-GPT, includes an offline stage to understand the state relationships. For the iOS system, AXNav [74] and ILvuUI [42] input test logic and app screenshots. They further employ vision-based LLMs to select events. AssistGUI [34] is a GUI test generation approach specifically designed for the Windows system. Note that, the source code and tools for FARLEAD-Android, AXNav, ILvuUI, and AssistGUI are not available.

The key difference between existing GUI test generation approaches and MACdroid lies in how the test logics for the target functionalities are obtained. These approaches rely on manually crafted test logics, making the process time-consuming and impractical for large-scale apps. In contrast, MACdroid extracts test logics from source test cases, effectively replacing the need for manually crafted test logics.

Bug detection for mobile apps. According to exploration strategies, bug detection related approaches for mobile apps can be classified into four categories, i.e., random testing approaches [18, 59, 72, 73], model-based approaches [22, 37, 48, 50, 57, 70, 71, 76, 81], systematic testing approaches [20, 35, 60], and learning-based approaches [26, 27, 45, 51, 58, 68, 69, 82]. Specifically, Monkey [18], a widely-used approach, employs random exploration to uncover bugs. AIMDROID [37] and Stoat [70] leverage static analysis and dynamic exploration to construct a model of the app under test, and subsequently detect bugs based on this model. SCENTEST [81] adopts a different modeling approach by collecting extensive test reports and generating event knowledge graphs to guide its exploration. These event knowledge graphs integrate user testing information, representing a valuable effort to leverage user-provided insights for enhancing automated exploration and identifying complex bugs. SynthesiSE [35] is a concolic execution approach for Android applications. Unlike traditional methods, which rely on manually written models for the Android framework, SynthesiSE dynamically infers expressions representing Android models during execution. RoScript [68] and ROBOTEST [82] focus on embedded systems. They use robotic arms with computer vision-based algorithms to detect bugs across embedded systems. V2S [26] aids bug detection by automatically translating video recordings of Android app usages into replayable test cases.

The key distinction between MACdroid and existing bug detection approaches lies in their different purposes. Existing approaches primarily focus on GUI exploration and bug detection but lack the capability to effectively emulate user behavior for testing individual functionalities. Therefore, to validate user behavior during the execution of specific functionalities, industry practitioners often rely on extensive manual effort to design and execute tests for individual functionalities, ensuring the correctness of individual functionalities. In contrast, MACdroid generates test cases based on specific test logics for individual functionalities, enabling it to emulate real-world user interactions with each functionality more accurately.

7 CONCLUSION

GUI test migration aims to produce test cases for specific functionalities of a target app. Existing migration approaches follow the widget-mapping paradigm. However, test cases produced using this paradigm are often incomplete or contain bugs, making them difficult to use directly for testing target functionalities and require additional manual modifications. In this paper, we have proposed a new paradigm for GUI test migration (i.e., abstraction-concretization paradigm). We then proposed MACdroid, the first approach that follows this paradigm to migrate GUI test cases. We have evaluated the effectiveness of MACdroid on 31 real-world apps, 34 functionalities, and 123 test cases, and compared it with two state-of-the-art approaches, TEMdroid and AutoDroid. Our experimental results demonstrate the effectiveness of MACdroid in GUI test migration.

ACKNOWLEDGEMENTS

We thank the anonymous TOSEM reviewers for their valuable feedback and the insightful comments provided by Zhiyong Zhou. This work was supported by the National Key Research and Development Program of China under Grant No. 2023YFB4503803, the National Natural Science Foundation of China under Grant No.62372005, the Young Elite Scientists Sponsorship Program by CAST (Doctoral Student Special Plan), and the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (Proposal ID: T2EP20223-0043; Project ID: MOE-000613-00). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] 2024. *ABC News - Breaking News, Latest News and Videos.* https://abcnews.go.com/
[2] 2024. *Android | Do More With Google on Android Phones & Devices.* https://www.android.com/
[3] 2024. *Appium: automtate test for apps.* https://appium.io/
[4] 2024. *BBC News apk.* https://play.google.com/store/apps/details?id=bbc.mobile.news.ww&hl=zh&gl=US

[5] 2024. *BERT base uncased.* https://huggingface.co/bert-base-uncased
[6] 2024. *ChatGPT.* https://chat.openai.com/
[7] 2024. *Claude AI.* https://claude.ai/login?returnTo=%2F%3F
[8] 2024. *F-Droid: free and open source Android app repository.* https://f-droid.org/
[9] 2024. *Firefox browser: fast, private and secure Web browser.* https://play.google.com/store/apps/details?id=org.mozilla.firefox
[10] 2024. *FrUITeR dataset.* https://felicitia.github.io/FrUITeR
[11] 2024. *GitHub: Let's build from here.* https://github.com/
[12] 2024. *Google Play store.* https://play.google.com/store/games
[13] 2024. *GPT-4, a large multimodal model.* https://openai.com/research/gpt-4
[14] 2024. *Lin dataset.* https://github.com/seal-hub/CraftDroid
[15] 2024. *The MACdroid project.* https://sites.google.com/view/macdroid-test?usp=sharing
[16] 2024. *Selenium automates browsers.* https://www.selenium.dev/
[17] 2024. *TEMdroid dataset: popular apps in Google Play Store.* https://github.com/YakZhang/TEMdroid/tree/main/Dataset/Usefulness_study
[18] 2024. *UI/application exerciser Monkey.* https://developer.Android.com/studio/test/monkey
[19] Kabir Ahuja, Rishav Hada, Millicent Ochieng, Prachi Jain, Harshita Diddee, Samuel Maina, Tanuja Ganu, Sameer Segal, Maxamed Axmed, Kalika Bali, et al. 2023. Mega: Multilingual evaluation of generative ai. *arXiv preprint arXiv:2303.12528* (2023).
[20] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *FSE*. 1–11.
[21] Michael Auer, Iván Arcuschin Moreno, and Gordon Fraser. 2024. WallMauer: Robust Code Coverage Instrumentation for Android Apps. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. 34–44.
[22] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *ASE*. 238–249.
[23] Yude Bai, Sen Chen, Zhenchang Xing, and Xiaohong Li. 2023. ArgusDroid: detecting Android malware variants by mining permission-API knowledge graph. *SCIS* 66, 9 (2023), 1–19.
[24] Lee A Becker. 2000. Effect size (ES). (2000).
[25] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *ASE*. 54–65.
[26] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 309–321.
[27] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *MOBILESoft*. 133–143.
[28] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[29] Guiming Hardy Chen, Shunian Chen, Ziche Liu, Feng Jiang, and Benyou Wang. 2024. Humans or LLMs as the Judge? A Study on Judgement Biases. *arXiv preprint arXiv:2402.10669* (2024).
[30] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep learning-based software engineering: progress, challenges, and opportunities. *Science China Information Sciences* 68 (2025), 11102. https://doi.org/10.1007/s11432-023-4127-5
[31] Felix Dobslaw, Robert Feldt, David Michaëlsson, Patrik Haar, Francisco Gomes de Oliveira Neto, and Richard Torkar. 2019. Estimating return on investment for GUI test automation frameworks. In *ISSRE*. 271–282.
[32] Sidong Feng and Chunyang Chen. 2024. Prompting Is All You Need: Automated Android Bug Replay with Large Language Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[33] Harry D Foster, Adam C Krolnik, and David J Lacey. 2004. *Assertion-based design.* Springer Science & Business Media.
[34] Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, et al. 2023. ASSISTGUI: Task-Oriented Desktop Graphical User Interface Automation. *arXiv preprint arXiv:2312.13108* (2023).
[35] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *ASE*. 419–429.
[36] Giovanni Grano, Timofey V Titov, Sebastiano Panichella, and Harald C Gall. 2019. Branch coverage prediction in automated testing. *Journal of Software: Evolution and Process* 31, 9 (2019), e2158.
[37] Tianxiao Gu, Chun Cao, Tianchi Liu, Chengnian Sun, Jing Deng, Xiaoxing Ma, and Jian Lü. 2017. AimDroid: activity-insulated multi-level automated testing for Android applications. In *ICSME*. 103–114.
[38] Neelam Gupta, Aditya P Mathur, and Mary Lou Soffa. 2000. Generating test data for branch coverage. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE, 219–227.

[39] Tianyang Han, Qing Lian, Rui Pan, Renjie Pi, Jipeng Zhang, Shizhe Diao, Yong Lin, and Tong Zhang. 2024. The Instinctive Bias: Spurious Images lead to Hallucination in MLLMs. *arXiv preprint arXiv:2402.03757* (2024).
[40] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *ESEC/FSE*. 269–282.
[41] Zhongjian Hu, Peng Yang, Yuanshuang Jiang, and Zijian Bai. 2024. Prompting large language model with context and pre-answer for knowledge-based VQA. *Pattern Recognition* 151 (2024), 110399.
[42] Yue Jiang, Eldon Schoop, Amanda Swearngin, and Jeffrey Nichols. 2023. ILuvUI: Instruction-tuned LangUage-Vision modeling of UIs from Machine Conversations. *arXiv preprint arXiv:2310.04869* (2023).
[43] Bogdan Korel and Ali M Al-Yami. 1996. Assertion-oriented automated test data generation. In *Proceedings of IEEE 18th International Conference on Software Engineering*. IEEE, 71–80.
[44] Yavuz Koroglu and Alper Sen. 2021. Functional test generation from UI test scenarios using reinforcement learning for android applications. *Software Testing, Verification and Reliability* 31, 3 (2021), e1752.
[45] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of Android applications. In *ICST*. 105–115.
[46] K Krishna and M Narasimha Murty. 1999. Genetic K-means algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 29, 3 (1999), 433–439.
[47] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *ICML*. 957–966.
[48] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for Android applications. In *ASE*. 115–127.
[49] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. *arXiv preprint arXiv:2005.03776* (2020).
[50] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight UI-guided test input generator for Android. In *ICSE-C*. 23–26.
[51] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *ASE*. 1070–1073.
[52] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *ASE*. 42–53.
[53] Jun-Wei Lin and Sam Malek. 2022. GUI test transfer from Web to Android. In *ICST*. 1–11.
[54] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622.
[55] Shuqi Liu, Yu Zhou, Tingting Han, and Taolue Chen. 2023. Test reuse based on adaptive semantic matching across Android mobile applications. *arXiv preprint arXiv:2301.00530* (2023).
[56] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. 2023. LegoDroid: flexible Android app decomposition and instant installation. *SCIS* 66, 4 (2023), 142103.
[57] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided bug crush: Assist manual gui testing of android apps via hint moves. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–14.
[58] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Nighthawk: Fully automated localizing ui display issues via visual understanding. *IEEE Transactions on Software Engineering* 49, 1 (2022), 403–418.
[59] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *ESEC/FSE*. 224–234.
[60] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. 94–105.
[61] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of GUI events for test reuse: are we there yet?. In *ISSTA*. 177–190.
[62] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2023. An evolutionary approach to adapt tests across mobile apps. In *AST*. 70–79.
[63] Ariana Martino, Michael Iannelli, and Coleen Truong. 2023. Knowledge injection to counter large language model (llm) hallucination. In *European Semantic Web Conference*. Springer, 182–185.
[64] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini encyclopedia of psychology* (2010), 1–1.
[65] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*. 3111–3119.
[66] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*. 153–164.
[67] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.
[68] Ju Qian, Zhengyu Shang, Shuoyan Yan, Yan Wang, and Lin Chen. 2020. Roscript: a visual script driven truly non-intrusive robotic testing system for touch screen applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 297–308.

[69] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *ISSTA*. 12–22.
[70] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *ESEC/FSE*. 245–256.
[71] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31.
[72] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1088–1100.
[73] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and finding system setting-related defects in android apps. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2941–2963.
[74] Maryam Taeb, Amanda Swearngin, Eldon School, Ruijia Cheng, Yue Jiang, and Jeffrey Nichols. 2023. Axnav: Replaying accessibility tests from natural language. *arXiv preprint arXiv:2310.02424* (2023).
[75] Najam us Saqib and Sara Shahzad. 2018. Functionality, performance, and compatibility testing: a model based approach. In *FIT*. 170–175.
[76] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 434–446.
[77] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
[78] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android *(MobiCom '24)*. Association for Computing Machinery, Washington D.C., DC, USA. https://doi.org/10.1145/3636534.3649379
[79] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI Automation for Android. *arXiv preprint arXiv:2304.07061* (2023).
[80] Jia-Yu Yao, Kun-Peng Ning, Zhen-Hui Liu, Mu-Nan Ning, and Li Yuan. 2023. Llm lies: Hallucinations are not bugs, but features as adversarial examples. *arXiv preprint arXiv:2310.01469* (2023).
[81] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Zimin Ding, Zhenyu Chen, and Zhendong Su. 2024. Practical, Automated Scenario-based Mobile App Testing. *IEEE Transactions on Software Engineering* (2024).
[82] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Practical Non-Intrusive GUI Exploration Testing with Visual-based Robotic Arms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[83] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2024. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems* 36 (2024).
[84] Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. 2024. Learning-based Widget Matching for Migrating GUI Test Cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[85] Yakun Zhang, Qihao Zhu, Jiwei Yan, Chen Liu, Wenjie Zhang, Yifan Zhao, Dan Hao, and Lu Zhang. 2024. Synthesis-Based Enhancement for GUI Test Case Migration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 869–881.
[86] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *ESEC/FSE*. 1190–1201.