Synthesis-Based Enhancement for GUI Test Case Migration

Yakun Zhang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhangyakun@stu.pku.edu.cn

Qihao Zhu
DeepSeek-AI
Beijing, China
zhuqh@pku.edu.cn

Jiwei Yan
Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
Beijing, China
yanjiwei@otcaix.iscas.ac.cn

Chen Liu
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
cissieliu@stu.pku.edu.cn

Wenjie Zhang
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhang_wen_jie@pku.edu.cn

Yifan Zhao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhaoyifan@stu.pku.edu.cn

Dan Hao
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
haodan@pku.edu.cn

Lu Zhang*
Key Lab of HCST (PKU), MOE; SCS
Peking University
Beijing, China
zhanglucs@pku.edu.cn

Abstract

GUI test case migration is the process of migrating GUI test cases from a source app to a target app for a specific functionality. However, test cases obtained via existing migration approaches can hardly be directly used to test target functionalities and typically require additional manual modifications. This problem may significantly impact the effectiveness of testing target functionalities and the practical applicability of migration approaches.

In this paper, we propose *MigratePro*, the first approach to enhancing GUI test case migration via **synthesizing a new test case based on multiple test cases for the same functionality** migrated from various source apps to the target app. The aim of MigratePro is to produce functional test cases with less human intervention. Specifically, given multiple migrated test cases for the same functionality in the target app, MigratePro first combines all the GUI states related to these migrated test cases into an overall state-sequence. Then, MigratePro organizes events and assertions from migrated test cases according to the overall state-sequence and endeavors to remove the should-be-removed events and assertions, while also incorporating some connection events in order to make the should-be-included events and assertions executable. Our evaluation on 30 apps, 34 functionalities, and 127 test cases shows that MigratePro improves the capability of three representative migration approaches (i.e., Craftdroid, AppFlow, ATM), successfully improving testing the target functionalities by 86%, 333%, and 300%, respectively. These results underscore the generalizability of MigratePro for effectively enhancing migration approaches.

CCS Concepts

• **Software and its engineering** → *Software testing and debugging*.

Keywords

Test migration, GUI testing, Synthesis

ACM Reference Format:
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 (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680327

1 Introduction

GUI (Graphical User Interface) testing is an important way to test the functionalities of mobile apps [13, 47]. A functional GUI test case consists of one or more ordered *events* and *assertions* [14, 30]. The events aim to explore a functionality of GUI *widgets*, and the assertions aim to check whether the outcomes of these events satisfy the expectations of users. During the execution of events and assertions in a test case, a series of GUI *states* are correspondingly triggered. Developers typically develop multiple functionalities (e.g., sign-in) for an app. In this paper, we define the target functionality of a test case as the functionality that the test case aims to check. In general, constructing GUI test cases manually is labor-intensive and time-consuming [19, 31, 39]. However, automated GUI test generation approaches [10, 17, 28, 35, 45] can hardly generate oracles. To reduce the cost of manually writing GUI test cases, several migration approaches [14, 22, 30, 32, 36, 37, 53] have been proposed

*Corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680327

to migrate GUI test cases from a source app to a target app that shares the same functionalities.

Despite advancements in migration approaches, using migrated test cases to test functionalities still encounters a significant **problem**. The aim of test case migration is to successfully test functionalities in the target app, e.g., the sign-in functionality. However, our preliminary study, which focuses on evaluating the effectiveness of migration approaches (detailed in Section 2), indicates that at least 57% of the migrated test cases generated by the representative migration approaches [22, 30] are incomplete or buggy, thus cannot be directly used for functional testing. Therefore, users may need additional manual modifications to meet their expectations [54]. This problem negatively impacts the effectiveness of migration approaches in real-world scenarios.

Considering the significant disparity between the test cases generated by existing migration approaches [14, 22, 30, 32, 36, 37] and their target functionalities, it is crucial to provide a unified approach to enhance these approaches. Two intuitions from our preliminary study may help address this problem. First, combining migrated test cases may be promising as this process may provide more ***should-be-included events and assertions***, which are required to test the target functionality. Second, this process may also introduce more ***should-be-removed events and assertions***, which do not contribute to the testing of the target functionality and may even hinder the execution of the generated test case.

Based on the two intuitions, we propose *MigratePro*, the first approach to enhancing test case migration by synthesizing new test cases from the migrated test cases, which are generated by migration approaches, to successfully test the target functionalities.

Given a target app and multiple migrated test cases for one functionality, we design a **two-stage generation strategy** for MigratePro to synthesize a new test case for the functionality. First, in the **stage of test case generation**, MigratePro collects the GUI states that include all the widgets from the migrated test cases. This way enables MigratePro to collect the states related to the target functionality. Then, it combines these states into a state graph and extracts the overall state-sequence from the state graph for testing the target functionality. Furthermore, MigratePro organizes the events and assertions from the migrated test cases according to the overall state-sequence, and forms a base test case. Second, in the **stage of test case adjustment**, the base test case is adjusted by removing the *should-be-removed events and assertions*, and also incorporating the ***connection events***, which are crucial for bridging the sequentially-executed events and assertions, to increase the executability for the *should-be-included events and assertions* in the sequence. MigratePro deems the adjusted sequence as the new test case synthesized for testing the target functionality.

We conduct a comprehensive evaluation to analyze the effectiveness of MigratePro in enhancing test case migration using 30 real-world apps, 34 functionalities, and 127 test cases. MigratePro is applied to enhance the migrated test cases from three representative migration approaches, i.e., Craftdroid [30], AppFlow [22], and ATM [14], respectively. Compared to these three approaches, MigratePro substantially increases the number of test cases that can successfully test the target functionalities, with improvements of 86%, 333%, and 300%, respectively. MigratePro also enhances the test cases that align with the corresponding ground-truth test cases (which are written by developers), by 28%, 217%, and 100%, respectively. Additionally, we also compare Craftdroid enhanced by MigratePro with two recent migration approaches, i.e., TRASM [32] and Adaptdroid [37], to evaluate the effectiveness of the enhanced results by MigratePro. The results show that Craftdroid enhanced by MigratePro outperforms these two approaches by more than 45% in availability and 31% in coverage. Furthermore, we evaluate the generalizability of MigratePro on five new apps. Compared to the original Craftdroid, using MigratePro shows a substantial improvement of 329% in availability and 100% in coverage. Overall, these results demonstrate that MigratePro is effective and generalizable to enhance test case migration for industry apps.

This paper makes the following main contributions:

- This paper presents a preliminary study that investigates the effectiveness of three representative test migration approaches in industry apps. This study complements prior studies on test case migration and points out the gaps between the test cases generated by existing migration approaches and their target functionalities.
- This paper introduces MigratePro, the first approach to enhancing test case migration, aiming to synthesize new test cases that can successfully test target functionalities for different migration approaches. MigratePro bridges the gap between the migrated test cases from existing migration approaches and usable functional test cases.
- We conduct an empirical evaluation on real-world apps to validate the effectiveness of MigratePro. The evaluation results demonstrate that MigratePro has good effectiveness and generalizability for enhancing test case migration.

2 Preliminary Study

Existing evaluations of migration approaches [14, 22, 30, 32, 37] mainly focus on evaluating precision and recall of events and assertions in the migrated test cases, but ignore the effectiveness of these test cases for testing the target functionalities in real-world scenarios. However, high precision and recall of event migration may not necessarily result in high effectiveness of these migrated test cases. For example, a migration approach correctly migrates seven out of ten events in a migrated test case, resulting in a recall of 70% for event migration. Despite this high recall rate, the migrated test case still fails to test the target functionality as the three missed events may be essential for testing the functionality.

To investigate the effectiveness of existing migration approaches in real-world scenarios, we undertake a preliminary study. Our investigation aims to answer the research question: *What is the effectiveness of existing migration approaches in real-world scenarios?*

2.1 Experimental Setup

In test case migration, two datasets are predominantly used, i.e., the Lin dataset[1] [6] and the FrUITeR dataset [7]. We use them as our experimental objects. For a comprehensive assessment, we employ all migration approaches that have the complete migration results on these two datasets. Thus, we utilize Craftdroid [30], ATM [14], and AppFlow [22] as the objects of our study.

[1]The dataset provided by Lin et al. [30] has not been named by the authors. For the sake of clarity, we refer to this dataset as the "Lin dataset" in this paper.

Table 1: Evaluation of representative migration approaches.

Dataset	Lin	FrUITeR	
Approach	Craftdroid	AppFlow	ATM
$\mathbf{Type_{suc-1}}$	29% (49)	6% (13)	2% (4)
$\mathbf{Type_{suc-2}}$	14% (24)	0% (1)	2% (5)
$\mathbf{Type_{unsuc}}$	57% (95)	94% (208)	96% (213)
Total	168	222	222

To accurately calculate the ratio of the migrated test cases generated by migration approaches that successfully test the target functionalities, we first calculate the ratio of the test cases that align with the ground-truth test cases. However, test cases that successfully test one functionality are not necessarily unique. Consequently, for those test cases that do not exactly match the ground-truth, we need to further investigate whether they still successfully test their target functionalities. Accordingly, we classify the migrated test cases into three types.

- $\mathbf{Type_{suc-1}}$: The migrated test cases are identical to these ground-truth test cases for the target functionalities.
- $\mathbf{Type_{suc-2}}$: The migrated test cases can fully execute and successfully test the same functionalities as the ground-truth test cases, but they are somehow different.
- $\mathbf{Type_{unsuc}}$: The migrated test cases cannot successfully test the target functionalities.

Evaluation process. To classify the migrated test cases generated by the migration approaches, we design an evaluation strategy with a combination of algorithmic check and manual check.

Algorithm check. First, we identify the $\mathbf{Type_{suc-1}}$ test cases by comparing the migrated test cases and the corresponding ground-truth. Second, we execute the remaining migrated test cases and keep the executable subsequences. Third, we identify the $\mathbf{Type_{suc-2}}$ test cases based on the functionalities they test. These test cases may not only include 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. In some cases, some events in the ground-truth test cases may be replaced by additional events with the same effects. A manual check is adopted to verify these additional events and assertions. The remaining test cases are classified as $\mathbf{Type_{unsuc}}$.

Manual check. We invite three volunteers[2] to participate in this study, who have industrial experience in Android programming ranging from 3 to 5 years. For each manual check, we provide the volunteers with the migrated test cases, the events to be checked, the target app, and the corresponding ground-truth test case. Each volunteer independently checks the events. In cases of disagreement, the volunteers discuss until they reach a consensus.

2.2 Experimental Result

For *Craftdroid*, using the 168 migrated test cases that Craftdroid generated on the Lin dataset, we count the numbers of migrated test cases of the three types based on the definition in Section 2.1. Our evaluation shows that 29% test cases belong to $\mathbf{Type_{suc-1}}$ and 14% test cases belong to $\mathbf{Type_{suc-2}}$. This means that only 43% of test cases successfully test the target functionalities. 57% test cases belong to $\mathbf{Type_{unsuc}}$, meaning that they cannot successfully test the target functionalities. As for *AppFlow* and *ATM*, the ratios of $\mathbf{Type_{unsuc}}$ test cases (94% and 96%) are notably high.

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

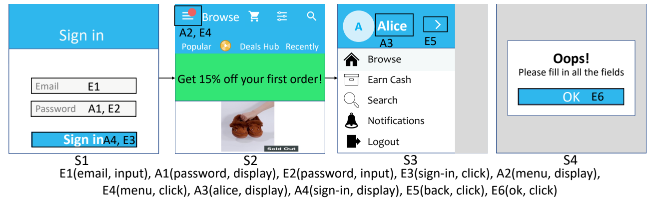

Figure 1: Example of a sign-in functionality.

This assessment indicates that existing migration approaches produce a high proportion of migrated test cases that fail to successfully test the target functionalities. Therefore, it is crucial to improve the effectiveness of existing migration approaches.

To improve the effectiveness of existing migration approaches, a potential idea is to combine individual migrated test cases for a target functionality into one test case. There are two intuitions. First, combining individual test cases may facilitate to enhance the effectiveness of the migration approaches due to covering more **should-be-included events and assertions**. Second, this process may also cover more **should-be-removed events and assertions**.

> **Answer to preliminary study:** The effectiveness of existing migration approaches still needs improvement. Combining individual test cases may cover more should-be-included events and should-be-removed events and assertions.

3 MigratePro

3.1 Running Example

We use an example (see Figure 1) to illustrate MigratePro's approach for enhancing test cases. This example demonstrates a sign-in functionality for a shopping app in the Lin dataset [6]. We tailor off some widgets for ease of presentation.

In Figure 1, each event (e.g., E1) corresponds to a widget and an action. Similarly, each assertion (e.g., A1) corresponds to a widget and a condition. The ground-truth test case for this functionality is: {E1, A1, E2, E3, A2, E4, A3}. Briefly, it involves inputting an email (E1) and a password (E2), clicking the sign-in button (E3), and then clicking the menu widget (E4), with assertions (A1, A2, A3) checking the progress through the sign-in state (S1), the main state (S2), and the account state (S3), respectively.

Craftdroid, utilizing source apps, migrates test cases for the specific functionality of the target app. We use {A4, E3, A1, E2, E1, E3, A2, A3}, {A4, A1, E2, E1, E3, E5, A3}, and {E6, E1} as the inputs to illustrate MigratePro's approach in the following sections.

3.2 Approach Overview

MigratePro is constructed to synthesize a new test case for a target app by combining multiple migrated test cases for the same functionality. These migrated test cases are generated by a migration approach that migrates different source test cases from their

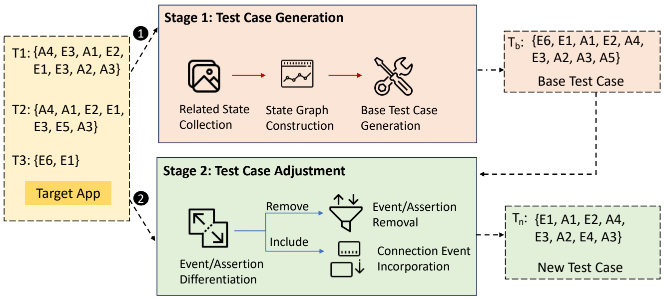

Figure 2: Workflow of MigratePro.

corresponding source apps to the target app. Note that, we do not require the migrated test cases to be fully executable.

We design a **two-stage generation strategy** for MigratePro to synthesize new test cases, as depicted in Figure 2. First, in the stage of **test case generation**, MigratePro collects a sequence of states related to the target functionalities for the target app. Based on the collected states, MigratePro constructs a functionality-specific state graph. Furthermore, MigratePro generates a base test case by organizing the events and assertions in the migrated test cases according to the state graph (see Section 3.3). Second, in the stage of **test case adjustment**, given the base test case from the first stage, MigratePro removes the *should-be-removed events and assertions*. It also incorporates some connection events in order that the *should-be-included events and assertions* are executable and the new test case successfully tests the target functionality (see Section 3.4).

This two-stage generation strategy is based on the observation that successfully executing a test case to test a specific functionality for a target app involves triggering states in a specific order. The events and assertions need to be executed in corresponding states.

3.3 Test Case Generation

The inputs of the stage of test case generation comprise a target app and a group of migrated test cases for one target functionality. The output is a base test case for this functionality. This stage is subdivided into three key phases, i.e., *related state collection*, *state graph construction*, and *base test case generation* (see Figure 2). First, MigratePro collects a sequence of states that include all widgets in the migrated test cases. These states are related to the target functionality. Second, MigratePro uses the collected states to construct the functionality-specific state graph. Third, MigratePro organizes the events and assertions of the migrated test cases based on the state graph, and generates a base test case for testing the target functionality. Note that, MigratePro does not collect all states of the target app but only the states related to target functionality.

1. Related state collection. MigratePro extracts the widget of each event or assertion in the migrated test cases, and obtains a collection of widgets related to the target functionality. Then, MigratePro uses **two steps** to collect **two parts of states** including these widgets. The final states are the *combination* of these states. Corresponding algorithm is available in our repository [9].

First, MigratePro collects **execution-specific states** by executing the migrated test cases. Specifically, MigratePro executes all the migrated test cases on the target app to get the executable subsequences, and selects the longest one. When executing the longest executable subsequence, MigratePro collects states containing *widgets in the migrated test cases*. Since the event order of a migrated test case follows that of its corresponding source test case (thus probably consistent with the event order for testing the target functionality), an executable subsequence of a migrated test case is typically able to test part of the target functionality and provide the related states. The longest one may provide the most related states.

Second, MigratePro collects **exploration-specific states** by exploring the target app. Specifically, for the *remaining widgets in the migrated test cases* that are not present in the preceding collected states, MigratePro uses a greedy-random strategy to explore the target app, starting from the end-point of the longest executable subsequence. Then, MigratePro randomly selects a widget in the current state, and generates an appropriate event (e.g., generating a click event for a clickable widget). If the triggered state contains the remaining widgets in the migrated test cases, MigratePro continues random exploration in this state. Otherwise, MigratePro returns to the previous state to explore. When the collected states contain all the remaining widgets, or the exploration time exceeds *max_time*, the exploration stops.

Note that, the effectiveness of the greedy-random strategy depends on two main factors. First, the greedy-random strategy only needs to explore a relatively small number of states. While there may be a large number of states in the target app, only those states relevant to the target functionality but are not collected in the longest executable subsequence need to be explored. Second, the migration results are able to constrain the search space. The longest subsequence provides the basis and exploration direction for the target functionality. By starting exploration based on this subsequence, random exploration within a smaller search space may be able to efficiently collect related states.

2. State graph construction. MigratePro constructs a functionality-specific state graph based on the collected states with three steps.

First, for each migrated test case, MigratePro generates an individual state-sequence. It matches each widget from the test case with its corresponding state by checking the widget's `resource-id` in the collected states. Then, MigratePro generates a state-sequence following the widget order in this test case.

Second, MigratePro utilizes the state-sequences to construct a directed graph. In the graph, nodes represent states, while edges represent the relations between states in the state-sequences. The weight of a node denotes the frequency of a state across various state-sequences. Likewise, the weight of an edge denotes the frequency of a specific state transition within these state-sequences.

Third, the directed graph may contain cycles. To generate an overall state-sequence for testing a target functionality, it is necessary to break cycles. Specifically, MigratePro employs a depth-first search algorithm to detect cycles within the directed graph, and retains those edges that align with the longest executable subsequence. For those edges that are not aligned with the longest executable subsequence, MigratePro removes the edge with the smallest edge weight within each cycle. The rationale behind this is that edges with lower frequency may imply less relevance for testing the intended functionality. MigratePro repeats this process until the directed graph is transformed into a directed acyclic graph.

Note that, it may not be appropriate to directly use the graph obtained through related state collection as the functionality-specific state graph. Since the state collection involves random exploration, the state transitions identified through random exploration may not reflect the state transitions for testing the target functionality.

3. Base test case generation. This phase aims to generate a base test case for testing the target functionality. To generate the base test case, MigratePro organizes the events and assertions from the migrated test cases according to the overall state-sequence. The migrated test cases provide the events and assertions related to the target functionality; while the overall state-sequence provides the testing order for the target functionality. The details of the two steps for generating the base test case are as follows.

First, MigratePro generates an overall state-sequence by employing a topological sorting algorithm to organize the states in the directed acyclic graph. During each iteration, MigratePro identifies nodes with an in-degree of zero. When multiple nodes meet this criterion, MigratePro prioritizes those nodes with higher weights.

Second, MigratePro organizes the events and assertions from the migrated test cases according to the states that the widgets belong to. For events and assertions associated with widgets in the same states, MigratePro organizes them according to the widget locations, following a top-left to bottom-right trajectory [42, 50]. Otherwise, MigratePro organizes them following the order of states in the overall state-sequence.

Note that, the generated base test case may not always successfully test the target functionality, as it may contain some should-be-removed events and assertions and may lack some should-be-included ones, which requires further adjustments (see Section 3.4).

Applying the first stage to the example. We illustrate the process of this stage using the running example depicted in Figure 1. First, MigratePro executes three migrated test cases to obtain the longest executable sequence. During this process, MigratePro collects two related states (i.e., S1 and S2). Among the widgets in the migrated test cases, only three widgets (i.e., the widgets of A3, E5, and E6) are not present in S1 and S2. Then, MigratePro explores the target app from the endpoint of the longest executable sequence (i.e., from the S2 state) with a greedy-random strategy to identify states including these three widgets. Thus, MigratePro collects four related states (i.e., S1 to S4). Second, MigratePro generates three state-sequences for the three migrated test cases. The three state-sequences are {S1, S2, S3}, {S1, S3}, {S4, S1}. Based on the three state-sequences, MigratePro constructs a functionality-specific state graph. The generated state graph is a directed acyclic graph. Third, MigratePro utilizes a topological sorting algorithm to generate the overall state-sequence. The generated overall state-sequence is {S4, S1, S2, S3}. Furthermore, MigratePro organizes all the events and assertions from the migrated test cases based on the overall state-sequence to generate the base test case. The generated base test case is {E6, E1, A1, E2, A4, E3, A2, A3, E5}.

3.4 Test Case Adjustment

In the stage of test case adjustment, the inputs include a target app, multiple migrated test cases, and a base test case generated in the first stage. The output is a new test case that includes a sequence of ordered events and assertions to test the target functionality. This stage is divided into three key phases, i.e., *event/assertion differentiation*, *event/assertion removal*, and *connection event incorporation* (see Figure 2). Specifically, MigratePro starts to differentiate the events and assertions in the base test case between **should-be-included** and **should-be-removed** ones according to their execution and frequency information in the migrated test cases. Following this differentiation, MigratePro removes the identified should-be-removed events/assertions from the base test case. It also incorporates connection events into the base test case in order to make the identified should-be-included events/assertions executable.

1. Event/Assertion differentiation. This phase is to identify which events and assertions in the base test case should be included and which should be removed. Specifically, for each event or assertion (denoted as e_b) in the base test case, MigratePro launches the target app and attempts to execute e_b. If e_b can be executed, MigratePro continues to execute the next one. However, if e_b cannot be executed (e.g., the widget of e_b is not present in the current state), MigratePro focuses on its frequency. If the frequency of e_b in the migrated test cases surpasses a $frequency\_threshold$, it is deemed as a should-be-included event or assertion but is non-executable. However, if the frequency of e_b in the migrated test cases is less than $frequency\_threshold$, it is deemed as should-be-removed.

2. Event/Assertion removal. For events and assertions identified as should-be-removed, MigratePro removes them from the corresponding base test case.

3. Connection event incorporation. For events and assertions identified as should-be-included but not executable, MigratePro dynamically explores the target app to identify connection events, thus making these should-be-included events and assertions executable. Corresponding algorithm is available in our repository [9].

Specifically, MigratePro launches the target app. For each identified should-be-included event or assertion (denoted as e_i) but not executable in the base test case, MigratePro first generates a sequence of events (denoted as E) for widgets in the current state (denoted as s) according to the widgets' attributes. Then, MigratePro randomly selects one event of E (denoted as e_s) in state s to execute, and determines whether e_i can be executed after executing e_s. If e_i can be executed, MigratePro appends e_s and e_i in the new test case. We denote e_s as the **connection event** for e_i. Otherwise, MigratePro backtracks to state s, and executes another event of E. If all the events of E have been executed but still do not identify the connection events, MigratePro randomly selects one event of E to transition to a new state and tries to identify the connection events in this state. There are two termination conditions for this process. First, all the should-be-included events and assertions in the base test case are executable. Second, when the entire process's running time reaches a max_time and the base test case still contains some non-executable events and assertions, MigratePro removes them to make the sequence executable. As MigratePro selects events randomly, it repeats the entire process with a $repeat\_num$, and chooses the shortest sequence as the new test case.

Applying the second stage to the example. We illustrate the process of this stage using the running example depicted in Figure 1. First, MigratePro executes the base test case to differentiate the should-be-removed and should-be-included events and assertions. MigratePro identifies two events (i.e., E6 and E5) and one assertion

Table 2: Statistics of Benchmark Apps.

Dataset	Category	App	Functionality	Test	Event	Assertion	Ave_Size
Lin	Browser	5	2	10	32	20	4M
	To-Do	5	2	10	39	15	2M
	Shopping	4	2	8	49	26	25M
	Mail	4	2	8	33	27	11M
	Calculator	5	2	10	33	10	2M
	Total	**23**	**10**	**46**	**186**	**98**	**8M**
FrUITeR	News	5	12	42	112	-	22M
	Shopping	5	12	39	207	-	20M
	Total	**10**	**24**	**81**	**319**	**-**	**21M**

(i.e., A3) that cannot be executed sequentially. Second, E6 and E5 are identified as should-be-removed events due to their low occurrences and thus removed. Third, A3, despite being non-executable but of high-occurrence, is identified as a should-be-included assertion. MigratePro identifies that when clicking the menu widget (i.e., E4), the should-be-included assertion (i.e., A3) can be executed. Hence, the connection event (i.e., E4) is inserted into the base test case just prior to A3. Finally, the new test case synthesized by MigratePro is {E1, A1, E2, A4, E3, A2, E4, A3}, which not only executes completely but also successfully tests the target functionality.

4 Evaluation

Our evaluation aims to study the following research questions.

RQ1: *How effective and efficient is MigratePro in enhancing test case migration for different migration approaches?*

RQ2: *How effective is MigratePro in enhancing test case migration for different app categories?*

RQ3: *How do different techniques in MigratePro affect the effectiveness of MigratePro?*

RQ4: *How competitive are the enhanced results by MigratePro compared with related approaches?*

4.1 Experimental Setup

Experimental objects. There are already some datasets [1, 4, 6, 7] for evaluating test case migration. We mainly select the Lin dataset [6] and the FrUITeR dataset [7] as our objects. The Lin dataset is the most widely-used migration dataset for evaluating previous migration approaches [30, 32, 36]. The FrUITeR dataset [7] includes the most functionalities to be tested among the migration datasets [1, 4, 6, 7]. 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 events and assertions, while those in the FrUITeR dataset contain only events. For app selection and test case selection, we consider all the installable apps and executable test cases provided by the Lin dataset and the FrUITeR dataset as our experimental objects[3]. Table 2 presents the basic statistics of our experimental objects. Detailed information is available in our repository [9].

Baseline approaches. To evaluate the enhancement ability of MigratePro in test case migration, we select different migration approaches. For a comprehensive assessment, we employ all migration approaches that have the complete migration results on these two datasets. Thus, we utilize Craftdroid [30], ATM [14], and AppFlow [22]. The inputs of MigratePro are all the migrated test cases by one migration approach for the target functionality.

To evaluate MigratePro's enhanced results, we further compare the enhanced results by MigratePro with two recent migration approaches (i.e., TRASM [32] and Adaptdroid [37]).

MigratePro can also be viewed as a GUI test repair approach that repairs the migrated test cases generated by migration approaches to successfully test the target functionalities. Among existing repair approaches [18, 27, 40, 49], only GUIDER [49] provides a binary file, while the others [18, 27, 40] remain closed source without even binary versions. Unfortunately, the binary file of GUIDER is not executable. Despite reaching out to the authors of GUIDER for assistance, we are informed that the source code has been lost. Thus, we are not able to compare these approaches with MigratePro.

Evaluation process. To evaluate the effectiveness of MigratePro and the migration approaches, we classify the generated test cases into three types (i.e., $\mathbf{Type_{suc-1}}$, $\mathbf{Type_{suc-2}}$, and $\mathbf{Type_{unsuc}}$) based on whether they successfully test the target functionalities. These types are consistent with those outlined in Section 2.1.

Compared with the evaluation process described in Section 2.1, this evaluation process includes two additional steps. First, before initiating the evaluation, we randomize the generated test cases from MigratePro and migration approaches. This step ensures no undue bias in favor of MigratePro. Second, after all test cases have been classified into $\mathbf{Type_{suc-1}}$, $\mathbf{Type_{suc-2}}$, and $\mathbf{Type_{unsuc}}$, we handle $\mathbf{Type_{unsuc}}$ test cases differently. In practice, users often have to manually modify test cases to align them with the target functionalities. The number of modification steps measures the effectiveness of $\mathbf{Type_{unsuc}}$ test cases. We engage the same three volunteers who have participated in the evaluation process to manually modify the generated test cases. If there is any disagreement among the volunteers, they discuss the issue until reaching a consensus.

Evaluation metrics. The aim of migration approaches is to generate test cases that successfully test the target functionalities. To evaluate the effectiveness of MigratePro and the compared migration approaches, we design three metrics: availability, coverage, and reduction.

Availability. This metric is calculated as the ratio of generated test cases ($Test_{suc}$) that successfully test the target functionalities to the total number of generated test cases ($Test_t$). This metric measures to what extent the generated test cases are able to accurately accomplish the tasks they are designed for.

$$Availability = Test_{suc} \ / \ Test_t \quad (1)$$

Coverage. This metric is calculated as the ratio of functionalities ($Func_{suc}$) successfully tested by the generated test cases to the total number of target functionalities ($Func_t$). This metric measures how many functionalities can be covered by the generated test cases.

$$Coverage = Func_{suc} \ / \ Func_t \quad (2)$$

Reduction. This metric is calculated as the ratio of steps saved by employing the generated test case compared to writing a new test case from scratch. We follow the methodology in the FrUITeR study [54]. Specifically, based on the Levenshein distance [26], we first compute the number of steps ($Step_m$) needed to modify a generated test case to test the target functionality. In this context, a

[3]The Lin dataset and the FrUITeR dataset share 3 common apps, leading to 30 distinct popular industrial applications in our experimental objects.

step in the Levenshtein distance refers to an insertion, a modification, and a deletion of an event or an assertion. We then compute the number of steps ($Step_t$) required to write the test case to successfully test the target functionalities from scratch. This metric measures the potential effectiveness of $\mathbf{Type_{unsuc}}$ test cases.

$$Reduction = (Step_t - Step_m) \,/\, Step_t \quad (3)$$

Among the preceding metrics, *availability* and *reduction* are test-case-level metrics, providing individual scores for each test case. These metrics are robust to evaluate the effectiveness of both MigratePro and the selected migration approaches.

However, *coverage* is a functionality-level metric. For a migration approach that typically generates multiple test cases for one functionality, if one of these test cases successfully tests the functionality, this functionality is considered covered. In contrast, MigratePro generates a single test case per functionality. Thus, the functionality coverage in MigratePro is solely dependent on this single test case. This makes the coverage metric potentially more favorable to the compared migration approaches but more stringent to MigratePro.

Implementation and parameter selection. We implement MigratePro in Python to support Android [3] apps. We use UIAutomator [11] and adb [2] tools to dump GUI states. The experiments are based on a Pixel 3 Emulator running Android 6.0. Some apps require installation in this setup, but MigratePro can adapt to others.

MigratePro needs to set three parameters (i.e., $frequency\_threshold$, $repeat\_num$, and max_time). We randomly select 10% of the total apps in the Lin dataset as a validation set for parameter selection. Following ATM [14] and Craftdroid [30], we set the candidate parameters for $frequency\_threshold$, $repeat\_num$, and max_time to {0.2, 0.5, 0.8}, {1, 2, 3}, and {3, 5, 7}, respectively. After experiments on the validation set, we observe that setting $frequency\_threshold$ to 0.5, $repeat\_num$ to 3, and max_time to 5 minutes yields the best effectiveness. Thus, all the experiments utilize this setting.

4.2 RQ1: Overall Effectiveness and Efficiency

To assess the effectiveness and the efficiency of MigratePro in enhancing test case migration, we conduct evaluations across different apps and migration approaches. Specifically, we evaluate MigratePro to enhance Craftdroid [30] on the Lin dataset, and to enhance AppFlow [22] and ATM [14] on the FrUITeR dataset.

Note that, we do not evaluate the migration approaches on a combined dataset for two reasons. First, the Lin dataset includes both events with complex actions (e.g., swipe-right) and assertions. However, the FrUITeR dataset includes only events with simple actions (e.g., click) without assertions. Second, AppFlow cannot migrate assertions, and ATM cannot migrate complex actions and assertions in the Lin dataset.

Enhancement results. Table 3 shows the test results that use MigratePro to enhance Craftdroid (i.e., the column denoted as Cra. → Cra.+Mig.), to enhance AppFlow (i.e., the column denoted as App. → App.+Mig.), and to enhance ATM (i.e., the column denoted as ATM → ATM+Mig.).

Craftdroid enhanced by MigratePro, achieves an 80% availability rate, implying that 80% of the new test cases (including both the $\mathbf{Type_{suc-1}}$ and $\mathbf{Type_{suc-2}}$ cases) successfully test the target functionalities. This rate is identical to its functional coverage, which means that the new test cases synthesized by MigratePro cover 80% of the total functionalities. Compared to the original Craftdroid, using MigratePro shows a substantial improvement of 86% in availability and 27% in coverage. Furthermore, 37% ($\mathbf{Type_{suc-1}}$) of the new test cases synthesized by MigratePro align with the ground-truth test cases. In addition, the remaining 20% ($\mathbf{Type_{unsuc}}$) of the new test cases that do not completely test the target functionalities are also beneficial for users. Compared to crafting the target test cases from scratch, utilizing the $\mathbf{Type_{unsuc}}$ test cases synthesized by MigratePro can potentially save 52% of human effort.

Table 3: Evaluation of MigratePro.

Dataset	Lin Dataset	FrUITeR Dataset	
Approach	**Cra. → Cra.+Mig.**	**App. → App.+Mig.**	**ATM → ATM+Mig.**
$Type_{suc-1}$	29% → 37% (↗ 28%)	6% → 19% (↗ 217%)	2% → 4% (↗ 100%)
$Type_{suc-2}$	14% → 43% (↗ 207%)	0% → 7% (↗ ∞)	2% → 12% (↗ 500%)
$Type_{unsuc}$	57% → 20% (↘ 65%)	94% → 74% (↘ 21%)	96% → 84% (↘ 13%)
Availability	43% → 80% (↗ 86%)	6% → 26% (↗ 333%)	4% → 16% (↗ 300%)
Coverage	63% → 80% (↗ 27%)	15% → 26% (↗ 73%)	10% → 16% (↗ 60%)
Reduction	53% → 52% (↘ 2%)	18% → 27% (↗ 50%)	4% → 9% (↗ 125%)

AppFlow and *ATM*, after enhancement with MigratePro, show substantial availability improvement of 333% and 300%, respectively. Additionally, their coverage improves by 73% and 60%, respectively.

Note that, MigratePro only synthesizes one test case for each functionality, so the availability score is equal to its coverage score. In contrast, existing migration approaches obtain multiple test cases for a single functionality of a target app based on multiple source apps. For example, among the 10 different functionalities for 23 apps in the Lin dataset (see Table 2), Craftdroid generates 168 migrated test cases (see Table 1). Thus, Craftdroid's availability is different from its coverage. Additionally, the average agreement score for volunteers to modify the $\mathbf{Type_{unsuc}}$ test cases is 91% in this experiment, as calculated using Krippendorff's alpha [43].

Result analysis. Availability improvement. Availability is a pivotal metric for measuring the effectiveness of migration approaches. A high availability implies that the migration approaches can successfully generate a greater number of test cases to test the target functionalities. This minimizes the need for human intervention in modifying or crafting new test cases. Conversely, a low availability implies that fewer generated test cases are capable of successfully testing the target functionalities, demanding increased human intervention. Notably, MigratePro considerably improves the availability scores for Craftdroid, AppFlow, and ATM by 86%, 333%, and 300%, respectively. Such notable enhancements indicate that test cases synthesized through MigratePro are more complete and require less human intervention.

Coverage improvement. Based on the analysis of "evaluation metrics" in Section 4.1, the coverage metric presents a more stringent requirement to MigratePro. However, MigratePro still manages to enhance the coverage rate by 27%, 73%, and 60% compared to the original Craftdroid, AppFlow, and ATM, respectively.

Efficiency study. We calculate the runtime information of MigratePro. The average running time for MigratePro per test case is 13 minutes on the FrUITeR dataset and 8 minutes on the Lin dataset. Note that, the migration approach Craftdroid [30] reportedly takes about 89 minutes per test case. Therefore, the time overhead for

Table 4: Evaluation of MigratePro by category.

Dataset (Approach)	Category	Cra./App.→ Cra./App.+Mig. Ava.	Cov.
Lin (Cra.)	Browser	83% → 100% (↗ 20%)	100% → 100% (→ 0%)
	To-Do	25% → 80% (↗ 220%)	40% → 80% (↗ 100%)
	Shopping	0% → 25% (↗ ∞)	0% → 25% (↗ ∞)
	Mail	63% → 88% (↗ 40%)	88% → 88% (→ 0%)
	Calculator	38% → 100% (↗ 163%)	80% → 100% (↗ 25%)
FrUITeR (App.)	News	7% → 26% (↗ 271%)	17% → 26% (↗ 53%)
	Shopping	6% → 26% (↗ 333%)	13% → 26% (↗ 100%)

Table 5: Contribution of each stage in MigratePro.

Approach	Cra.+Mig.		App.+Mig.	
Metric	Ava.	Cov.	Ava.	Cov.
Full stages	80%	80%	26%	26%
w/o test case generation (w/o first stage)	59%	72%	14%	22%
w/o test case adjustment (w/o second stage)	57%	57%	21%	21%

enhancing test cases with MigratePro is not significantly prominent compared to the original test case migration.

Answer to RQ1: MigratePro's ability to effectively enhance test cases across various migration approaches, combined with its acceptable efficiency, makes it a valuable approach.

4.3 RQ2: Effectiveness by Category

To investigate the enhancement of MigratePro across app categories, we conduct a statistical analysis of its effectiveness in enhancing Craftdroid, AppFlow, and ATM for different app categories.

Enhancement results by category. Table 4 presents the availability (denoted as "Ava.") and coverage (denoted as "Cov.") of the test cases enhanced by MigratePro (denoted as "Mig.") for Craftdroid (denoted as "Cra.") or AppFlow (denoted as "App.") by app category. The initial five rows denote Craftdroid's related results, while the last two rows denote AppFlow's related results. Note that, the results of ATM and enhanced by MigratePro share similar patterns with those of AppFlow. Due to space limit, we make these results related to ATM available in our repository [9].

The results of Table 4 reveal two observations. First, employing MigratePro to enhance Craftdroid and AppFlow notably improves both availability and coverage across various app categories. Second, the availability and coverage of MigratePro vary across different app categories. These variations may be related to the quality of the test cases migrated by the migration approaches and the complexity of the app categories.

Result analysis. Each row of Table 4 presents the effectiveness of a selected migration approach and the enhancements by MigratePro under a specific app category. We classify the variations across categories for MigratePro into three groups.

- First, the migration approach often generates test cases that can successfully test the target functionality.
- Second, the migration approach often fails to generate test cases that successfully test the target functionalities, although the target app design is not complex.
- Third, the migration approach often fails to generate effective test cases, and the design of the target apps is complex.

Group analysis. The first group includes the Browser category and the Mail category. Since the selected migration approaches already generate a large number of effective test cases, there is little room for MigratePro to further enhance these test cases. The second group includes the Calculator category and the To-do category. For example, a typical calculator app has a main state with a few actionable widgets to fill in the number and select the operation. Such a less complex app design facilitates test case enhancement as there are few states to explore and few widgets to incorporate. In these scenarios, MigratePro can significantly improve the test cases generated by these migration approaches. The third group includes the Shopping category and the News category. These apps typically include multiple states and widgets. The functionalities in such apps involve a series of events, each interacting with different widgets across various states. For instance, the sign-in functionality in a shopping app typically involves transitions from the initial state to the sign-in state, filling in all the required information, and finally, confirming the successful completion the sign-in process. Such complexity in app design increases the difficulty in migrating test cases and further enhancing these migrated test cases.

Dataset analysis. We observe that both original AppFlow and AppFlow enhanced by MigratePro do not perform as well on the FrUITeR dataset as original Craftdroid and Craftdroid enhanced by MigratePro do on the Lin dataset. This discrepancy arises from the fact that the Shopping category and the News category in the FrUITeR dataset are of higher complexity than the Browser category and the Calculator category in the Lin dataset. This complexity, which includes multiple GUI states with various widgets in each state, presents notable challenges for both test case migration and enhancement. Furthermore, the test cases migrated by AppFlow contain fewer should-be-included events and assertions but more should-be-removed events and assertions, which further complicates the enhancement of the migrated test cases. Nevertheless, even in these challenging scenarios, MigratePro has managed to significantly enhance the migrated test cases obtained with AppFlow. For original AppFlow (see Table 4), only 6% of shopping test cases and 7% of news test cases are able to test the target functionalities, but after enhancement by MigratePro, 26% of the synthesized test cases could successfully test the target functionalities without any manual modifications.

Answer to RQ2: MigratePro has good generalizability across different app categories and is particularly effective in situations where migration approaches struggle to generate test cases that successfully test the target functionalities, although the target app design is not complex.

4.4 RQ3: Ablation Study

We evaluate the contributions of MigratePro's two stages using migrated test cases generated from Craftdroid and AppFlow.

Experimental setting. We design two ablation studies. First, to evaluate the effectiveness of the generation techniques, we remove the first stage from MigratePro. Without this stage, MigratePro cannot combine migrated test cases and generate a base test case for the second stage. Instead, we use each migrated test case directly as input for the second stage. Second, to evaluate the effectiveness of the adjustment techniques, we remove the second stage from MigratePro. Without this stage, MigratePro cannot adjust the base test cases generated in the first stage. Thus, the base test cases are treated as the final test cases.

Results for ablation study. Table 5 presents the availability (denoted as "Ava.") and coverage (denoted as "Cov.") of the ablation study. The first row displays the effectiveness of MigratePro. The subsequent rows illustrate the effectiveness of MigratePro's ablation stages. It indicates that the removal of any stage results in a substantial decrease for MigratePro to enhance test cases. For instance, regarding the test cases migrated from Craftdroid, removing the first stage and the second stage leads to a decrease in availability metrics from 80% to 59% and 57%, respectively.

While this experiment helps analyze the factors for MigratePro's effectiveness in enhancing test case migration, there is room for further analysis. One potential direction is to compare the impact of using state graphs constructed through the *state graph construction* phase with those constructed during the *related state collection* phase on the overall effectiveness of MigratePro. Another potential direction is to analyze the factors that influence MigratePro's collection of execution-based and exploration-based states during the *related state collection* phase.

> **Answer to RQ3**: MigratePro's two stages significantly contribute to enhancing migrated test cases.

4.5 RQ4: Comparison with Related Approaches

To evaluate MigratePro's enhanced results, we compare Craftdroid enhanced by MigratePro with two recent migration approaches, i.e., TRASM [32] and Adaptdroid [37], on the Lin dataset.

Experimental setting. *TRASM* [32] only publishes the migration results within three categories (i.e., Browser, To-Do, and Shopping) on the Lin dataset. By following the evaluation process outlined in Section 4.1, we obtain the results of TRASM in testing the target functionalities for the three categories.

Adaptdroid [37] has published the source code but does not evaluate on the Lin dataset. We execute Adaptdroid to get its migrated test cases on the Lin dataset. To ensure the correct execution, we engage in multiple rounds of communication with the authors of Adaptdroid. Furthermore, we follow the evaluation process outlined in Section 4.1 to assess Adaptdroid. Note that, Adaptdroid has some limitations in supporting events and assertions. Specifically, it cannot support some complex events in test cases (e.g., "swipe_right" to a widget and "long_press" to a widget). It also cannot support the assertions at the top or in the middle of test cases. Consequently, we filter out test cases from the Lin dataset that Adaptdroid does not support, resulting in the removal of all test cases from the Shopping and Mail categories with only 13 test cases retained.

Result analysis. Table 6 presents the availability (i.e., "Ava.") and coverage (i.e., "Cov.") of TRASM, Adaptdroid, and Craftdroid enhanced by MigratePro (i.e., "Cra.+Mig.") on the Lin dataset. The results show that Craftdroid enhanced by MigratePro demonstrates greater effectiveness, outperforming TRASM and Adaptdroid with a 45% higher availability and 31% higher coverage. However, without enhancement by MigratePro, Craftdroid's migration effectiveness

Table 6: Comparison with TRASM and Adaptdroid.

Approach	TRASM		Adaptdroid		Cra.+Mig.	
Metric	**Ava.**	**Cov.**	**Ava.**	**Cov.**	**Ava.**	**Cov.**
Browser	80%	80%	33%	67%	100%	100%
To-Do	58%	70%	50%	75%	80%	80%
Shopping	9%	25%	-	-	25%	25%
Mail	-	-	-	-	88%	88%
Tip	-	-	17%	33%	100%	100%
Average	**55%**	**61%**	**33%**	**54%**	**80%**	**80%**

(see Table 4) in the To-Do and Shopping category is inferior to TRASM. This further highlights MigratePro's capability and the necessity of enhancement for migrated test cases.

Note that, TRASM and Adaptdroid are less effective compared to Craftdroid enhanced by MigratePro because they target different goals. Unlike MigratePro, which aims to enhance migrated test cases to successfully test target functionalities, TRASM and Adaptdroid aim to improve the accuracy of event/assertion migration.

> **Answer to RQ4**: The enhanced results by MigratePro demonstrate strong effectiveness in generating GUI test cases for target functionalities when compared to the related approaches.

4.6 Threats to Validity

A possible threat to external validity is the generalizability to other mobile apps and migration approaches. To mitigate this threat, we use the most app categories and functionalities compared with related work. Moreover, we also evaluate MigratePro using three distinct migration approaches [14, 22, 30]. Using the same datasets for both the preliminary study and the evaluation may somewhat impact MigratePro's generalizability negatively. However, in the preliminary study, we only conduct high-level statistical analysis to identify the migration problems. To further evaluate MigratePro's generalizability, we perform an additional study using five new apps (see Section 5).

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 and 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 test cases. These metrics offer a reliable and objective basis for evaluating test cases generated by migration approaches and those enhanced by MigratePro.

5 Generalizability Study

To further evaluate the generalizability of MigratePro, we conduct a study to evaluate MigratePro in enhancing test case migration using five new target apps in Google Play Store [8]. In this study, we select Craftdroid [30] as the migration approach since it can migrate both events and assertions. Note that, the inputs of MigratePro are

Table 7: Evaluation of MigratePro on New Apps.

App	Cra. → Cra.+Mig.		
	Ava.	Cov.	Red.
Web Browser	25% → 50% (↗ 100%)	50% → 50% (→ 0%)	58% → 14% (↘ 76%)
Done	20% → 50% (↗ 150%)	50% → 50% (→ 0%)	61% → 75% (↗ 23%)
Ubuy	0% → 50% (↗ ∞)	0% → 50% (↗ ∞)	45% → 38% (↘ 16%)
Pro Mail	25% → 100% (↗ 300%)	50% → 100% (↗ 100%)	38% → 100% (↗ 163%)
Tip Calculator	0% → 50% (↗ ∞)	0% → 50% (↗ ∞)	80% → 75% (↘ 6%)
Average	**14% → 60% (↗ 329%)**	**30% → 60% (↗ 100%)**	**58% → 51% (↘ 12%)**

the target app and its migrated test cases for a target app. We use the migrated test cases generated by Craftdroid on five new target apps as the inputs of MigratePro.

Experimental objects. To migrate test cases of apps from the Lin dataset to apps in Google Play Store [8], we use three steps to identify the target apps that belong to the same categories as those in the Lin dataset. First, for each app in the Lin dataset, we search for the top ten similar apps in Google Play Store. Second, we select the target app for each category with the highest number of appearances in the app search. In case multiple apps have the same appearances, we select the target app with the highest number of downloads. Using this app selection strategy, we obtain five target apps, each corresponding to one category.

Evaluation process. To evaluate the effectiveness of MigratePro and Craftdroid, we follow a similar evaluation process outlined in Section 4.1 and invite the same three volunteers. Note that, since the five new apps do not have ground-truth test cases, we cannot provide the volunteers with the ground-truth target test cases. Nevertheless, we provide the volunteers with the ground-truth source test cases for the same functionality, which can still help volunteers understand the target functionality. The evaluation metrics are availability, coverage, and reduction.

Result analysis. Table 7 presents the availability (denoted as "Ava."), coverage (denoted as "Cov."), and reduction (denoted as "Red.") of the test cases migrated by Craftdroid (denoted as "Cra.") and enhanced by MigratePro (denoted as "Mig.") on five new apps.

Craftdroid, enhanced by MigratePro, achieves an 60% in availability and 60% in coverage. Compared to the original Craftdroid, using MigratePro shows a substantial improvement of 329% in availability and 100% in coverage. For the new test cases that do not completely test the target functionalities, MigratePro potentially saves 51% of human effort. Furthermore, the app categories for these new target apps are different. From Table 7 we observe that MigratePro demonstrates significant improvement for different categories of apps. Overall, this study demonstrates the satisfactory generalizability of MigratePro in new apps.

6 Discussion

Assertion analysis. Assertions are crucial in functional test cases. We further analyze MigratePro's effectiveness to enhance assertion generation. 80% of the test cases generated by MigratePro based on the migrated test cases from Craftdroid contain fully effective assertions (see Table 3). We also analyze the overall precision and recall of assertions in test cases generated by MigratePro and Craftdroid, comparing them with the corresponding ground-truth test cases. MigratePro demonstrates an improvement in precision of 2% (87% vs. 89%) and recall of 13% (78% vs. 88%) compared with the original Craftdroid. This analysis demonstrates that MigratePro can effectively enhance assertion generation.

There are two reasons for MigratePro to effectively enhance assertion generation. First, test case migration facilitates the generation of relevant assertions for the target app. In GUI testing, each assertion corresponds to a *widget* and a *condition*. Test case migration from source to target apps enables the migration of target widgets for assertions, while conditions for assertions, typically consistent across similar functionalities, can be reused. Second, the techniques used in MigratePro's test case adjustment stage facilitate to adjust these assertions. MigratePro leverages frequency and execution information to identify and remove assertions that may be irrelevant or obstructive. In addition, the incorporation of connection events further helps those identified effective assertions to play the role in the generated test cases.

Effectiveness of related state collection. To further analyze the effectiveness of MigratePro's related state collection approach for the target functionalities, we check MigratePro's collected states based on the migrated test cases generated by Craftdroid, ATM, and AppFlow. Our analysis focuses on two metrics: the activity coverage for each functionality, and the success rate in collecting all related states for the target functionalities using MigratePro.

We have three observations through the analysis. First, exploration of each functionality within an app typically requires traversing a relatively small space. The average exploration for *each functionality* only requires exploring 3% of the total activities[4] for a target app. Second, executing the longest executable subsequence is effective in collecting related states, which enables collecting all the related states for 63% of the total number of target functionalities. Third, a combined approach of executing the longest executable subsequence and random exploration further enhances the state collection. This combined approach enables collecting all the related states for 94% of the total number of target functionalities. These results underscore the effectiveness of MigratePro's state collection approach.

Identifying and mapping functionalities. When the source and target apps *belong to the same category*, it is not hard to identify and map functionalities of the test cases. First, when writing functional test cases, developers typically specify the corresponding functionalities within the function names (e.g., testSignIn) to enhance the readability. Second, apps belonging to the same category often share similar functionalities. In fact, existing migration approaches [14, 22, 30, 32, 37] are developed *under this scenario*.

However, there may be some challenges when apps *belong to different categories*. Specifically, it is challenging to identify the functionalities of test cases when function names have ambiguous semantics; to map test cases that have different expressions but test the same functionalities; and to map apps that share the same functionalities but belong to different app categories.

Some potential solutions may address these challenges. Two potential solutions that may aid functionality mapping are using broader sources (e.g., function names, code comments, and widget information) of a test case and employing advanced matching models [5, 15, 33]. Additionally, another potential solution that may aid

[4]An activity is a collection of states for the same purpose designed by the developers.

app mapping is analyzing the functionalities of apps included in the app introductions from Google Play store [8].

The number of migrated test cases on MigratePro. We conduct a statistical analysis to evaluate the influence of the number of migrated test cases that generated by Craftdroid, ATM, and AppFlow on MigratePro. While MigratePro inputs all migrated test cases generated by a migration approach, its effectiveness is influenced by the number of migrated test cases. The results show that as the number of migrated test cases per functionality increases, the effectiveness of MigratePro gradually improves. This also demonstrates the rationality of MigratePro's intuition in combining multiple migrated test cases. Due to space limit, we make the detailed results available in our repository [9].

Integrating MigratePro with migration approaches. The inputs of MigratePro are multiple migrated test cases for the target app. We require the migration approaches to migrate multiple test cases simultaneously. Leveraging intermediate information during the migration process may enhance the efficiency of MigratePro. For instance, Craftdroid obtains the states corresponding to each migrated widget during the migration process. By modifying the Craftdroid code, we may effectively obtain the executable subsequence of each migrated test case. After the migration process, we may identify the longest executable subsequence by comparing the executable subsequences of the migrated test cases. Therefore, we can directly select the states corresponding to this sequence without re-executing the migrated test cases.

7 Related Work

GUI Test case migration. There are three approaches [41, 46, 52] aiming to migrate test cases across different platforms within the same apps. Testmig [41] employs static analysis to guide event exploration and map similar events from iOS to Android. MAPIT [46] executes bi-directional test migration between Android and iOS using dynamic analysis. LIRAT [52] employs computer vision techniques to map similar events across Android and iOS.

There are six approaches [14, 22, 30, 32, 37, 53] focusing on migrating test cases across different apps. AppFlow [22] utilizes a trained multi-classifier to identify widget labels (e.g., registration widget) of both source widgets (i.e., the widgets in the source app) and target widgets (i.e., the widgets in the target app). If a source widget and a target widget share the same label, AppFlow considers them as matched widgets. ATM [14] leverages word embeddings from word2vec [38] fine-tuned with app manuals to represent widgets. It further manually defines a matching function to match widgets. In contrast to ATM, Craftdroid [30] and TRASM [32] use word embeddings from a standard word2vec [38]. Adaptdroid [37] uses word embeddings from Word Mover's Distance [24]. TEMdroid [53] trains a learning-based matching model to match widgets. After identifying matched widgets, these migration approaches generate events and assertions based on the matched widgets, and then form migrated test cases.

There are two key differences between existing migration approaches and MigratePro. First, they are oriented to different problems. Although both approaches have the broad goal of enabling test case migration, existing migration approaches focus on mapping from source test case to target test case, whereas MigratePro aims to enhance the migrated test cases that cannot successfully test the target functionalities. Second, the core ideas of these approaches are different. Existing migration approaches generate a test case based on a single source test case. In contrast, MigratePro synthesizes a new test case from multiple migrated test cases.

GUI test case repair. Existing repair approaches [18, 27, 40, 49] aim to repair the outdated test cases between different versions of the same apps. Atom [27] and CHATEM [18] utilize the accurate behavior model to replace outdated events. In contrast, METER [40] and Yoon et al. [51] leverage computer vision techniques to detect deviations of events from the different app versions, and construct repairs to reduce the deviations. GUIDER [49] is a modified version of METER with structural information.

MigratePro can also be viewed as a repair approach that repairs the migrated test cases to successfully test the target functionalities. However, there are two key differences between existing repair approaches and MigratePro. First, the purpose of existing repair approaches is to make the repaired tests executable, whereas the purpose of MigratePro is not only to make the repaired tests executable, but also to successfully test the target functionalities. Second, existing repair approaches repair a target test case based on a single test case. In contrast, MigratePro synthesizes a new test case based on all the migrated test cases for the same functionality.

GUI test case generation. GUI test case generation is to detect bugs [48]. According to exploration strategies, these approaches can be classified into four categories, i.e., random approaches [10, 34], model-based approaches [13, 21, 25, 28, 45], systematic approaches [12, 20, 35], and learning approaches [16, 23, 29, 44]. MigratePro can also be viewed as a GUI test case generation approach that leverages the generated test cases to test a target app. While these approaches are proficient in generating events, they often struggle to generate oracle information. In contrast, MigratePro can automatically generate both events and assertions in case the migrated test cases contain them.

8 Conclusion

In this paper, we have proposed MigratePro, the first approach designed to synthesize new test cases to successfully test the target functionalities based on the migrated test cases. We have evaluated the effectiveness of MigratePro on 30 real-world apps, 34 functionalities, 127 test cases, and three representative migration approaches. Our experimental results demonstrate the effectiveness of MigratePro in enhancing test case migration.

9 Data Availability

The artifact of MigratePro is in a repository [9].

Acknowledgements

We thank the anonymous ISSTA reviewers for their valuable feedback and the insightful comments provided by Zhiyong Zhou on the preliminary study. Lu Zhang was partially supported by National Natural Science Foundation of China under Grant No.62232003. Dan Hao was partially supported by National Natural Science Foundation of China under Grant No.62372005. Jiwei Yan was partially supported by National Natural Science Foundation of China under Grant No.62102405 and Grant No.62132020.

References

[1] 2024. *Adaptdroid dataset.* Retrieved July 1, 2024 from https://drive.google.com/drive/folders/1NVxoYQZRa5ZFwbnq2QJ4_xGJ-VZci_oX/

[2] 2024. *Android debug bridge (adb).* Retrieved July 1, 2024 from https://developer.android.com/tools/adb

[3] 2024. *Android official developing documents.* Retrieved July 1, 2024 from https://developer.android.com/docs

[4] 2024. *ATM dataset.* Retrieved July 1, 2024 from https://sites.google.com/view/apptestmigrator/

[5] 2024. *ChatGPT- A large language model for OpenAI.* Retrieved July 1, 2024 from https://chat.openai.com/auth/login

[6] 2024. *Craftdroid dataset: a dataset to evaluate the effectiveness of test case migration tools within categories.* Retrieved July 1, 2024 from https://github.com/seal-hub/CraftDroid

[7] 2024. *Fruiter dataset: a dataset to evaluate the effectiveness of test case migration tools within categories.* Retrieved July 1, 2024 from https://felicitia.github.io/FrUITeR/

[8] 2024. *Google Play store.* Retrieved July 1, 2024 from https://play.google.com/store/games

[9] 2024. *Source code and extra materials for MigratePro.* Retrieved July 1, 2024 from https://github.com/YakZhang/MigratePro

[10] 2024. *UI/Application Exerciser Monkey.* Retrieved July 1, 2024 from https://developer.android.com/studio/test/monkey

[11] 2024. *UIAutomator API.* Retrieved July 1, 2024 from https://developer.android.com/training/testing/ui-automator

[12] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.* 1–11. https://doi.org/10.1145/2393596.2393666

[13] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* 238–249. https://doi.org/10.1145/2970276.2970313

[14] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 54–65. https://doi.org/10.1109/ASE.2019.00016

[15] Luca Bertinetto, Jack Valmadre, Joao F Henriques, Andrea Vedaldi, and Philip HS Torr. 2016. Fully-convolutional siamese networks for object tracking. In *European conference on computer vision.* Springer, 850–865. https://doi.org/10.1007/978-3-319-48881-3_56

[16] Nataniel P Borges Jr, Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems.* 133–143. https://doi.org/10.1145/3197231.3197243

[17] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test.* 93–96. https://doi.org/10.1145/3387903.3389308

[18] Nana Chang, Linzhang Wang, Yu Pei, Subrota K Mondal, and Xuandong Li. 2018. Change-based test script maintenance for android apps. In *2018 ieee international conference on software quality, reliability and security (qrs).* IEEE, 215–225. https://doi.org/10.1109/QRS.2018.00035

[19] 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 *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE).* IEEE, 271–282. https://doi.org/10.1109/ISSRE.2019.00035

[20] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* 419–429. https://doi.org/10.1145/3238147.3238225

[21] 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 *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 103–114.

[22] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 269–282. https://doi.org/10.1145/3236024.3236055

[23] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-based exploration of android applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST).* 105–115. https://doi.org/10.1109/ICST.2018.00020

[24] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger. 2015. From word embeddings to document distances. In *International conference on machine learning.* PMLR, 957–966.

[25] Duling Lai and Julia Rubin. 2019. Goal-driven exploration for android applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 115–127. https://doi.org/10.1109/ASE.2019.00021

[26] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[27] Xiao Li, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li. 2017. ATOM: Automatic maintenance of GUI test scripts for evolving mobile applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 161–171. https://doi.org/10.1109/ICST.2017.22

[28] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[29] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A deep learning-based approach to automated black-box android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 1070–1073. https://doi.org/10.1109/ASE.2019.00104

[30] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 42–53. https://doi.org/10.1109/ASE.2019.00015

[31] 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. https://doi.org/10.1109/ICSME.2017.47

[32] 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). https://doi.org/10.1109/QRS57517.2022.00076

[33] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[34] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 224–234. https://doi.org/10.1145/2491411.2491450

[35] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 94–105. https://doi.org/10.1145/2931037.2931054

[36] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. 2021. Semantic matching of GUI events for test reuse: are we there yet?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 177–190. https://doi.org/10.1145/3460319.3464827

[37] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2023. An evolutionary approach to adapt tests across mobile apps. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST).* IEEE, 70–79. https://doi.org/10.1109/AST52587.2021.00016

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[39] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 153–164. https://doi.org/10.1145/3395363.3397354

[40] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-guided test script repair for mobile apps. *IEEE Transactions on Software Engineering* 48, 3 (2020), 910–929. https://doi.org/10.1109/TSE.2020.3007664

[41] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. Testmig: Migrating gui test cases from ios to android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 284–295. https://doi.org/10.1145/3293882.3330575

[42] Jeremy Reimer. 2005. A History of the GUI. *Ars Technica* 5 (2005), 1–17.

[43] Bizhan Shabankhani, J Yazdani Charati, Keihan Shabankhani, and S Kaviani Cherati. 2020. Survey of agreement between raters for nominal data using Krippendorff's alpha. *Arch Pharma Pract* 10, S1 (2020), 160–164.

[44] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 12–22. https://doi.org/10.1145/3092703.3092709

[45] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 245–256. https://doi.org/10.1145/3106237.3106298

[46] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 756–767. https://doi.org/10.1109/ASE51524.2021.9678643

[47] Najam us Saqib and Sara Shahzad. 2018. Functionality, performance, and compatibility testing: A model based approach. In *2018 International Conference on Frontiers of Information Technology (FIT)*. IEEE, 170–175. https://doi.org/10.1109/FIT.2018.00037

[48] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1319–1331. https://doi.org/10.1145/3597926.3598138

[49] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. 2021. Guider: Gui structure and vision co-guided test script repair for android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 191–203. https://doi.org/10.1145/3597926.3598138

[50] Chee Kit Yee, Choo Seah Ling, Wong Seok Yee, and Wan Mohd Nazmee Wan Zainon. 2012. GUI design based on cognitive psychology: Theoretical, empirical and practical approaches. In *2012 8th International Conference on Computing Technology and Information Management (NCM and ICNIT)*, Vol. 2. IEEE, 836–841.

[51] Juyeon Yoon, Seungjoon Chung, Kihyuck Shin, Jinhan Kim, Shin Hong, and Shin Yoo. 2022. Repairing Fragile GUI Test Cases Using Word and Layout Embedding. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 291–301. https://doi.org/10.1109/ICST53961.2022.00038

[52] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. 2021. Layout and image recognition driving cross-platform automated mobile testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1561–1571. https://doi.org/10.1109/ASE.2019.00103

[53] 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. https://doi.org/10.1145/3597503.3623322

[54] 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 *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1190–1201.