Towards Effective Bug Reproduction for Mobile Applications

Xin Li†, Shengcheng Yu†, Lifan Sun, Yuexiao Liu, and Chunrong Fang*
State Key Laboratory for Novel Software Technology, Nanjing University, China
Shenzhen Research Institute of Nanjing University, China
† Equal contribution *Corresponding Author: fangchunrong@nju.edu.cn

Abstract—Bug reproduction is a critical task in software testing, as it helps developers to identify and fix bugs in the software. While some automated reproduction tools are designed to assist developers in reproducing bugs detected by automated testing tools, they are not entirely reliable. Thus, manual reproduction remains important. However, automated testing tools often generate testing results that are difficult for non-professional developers to understand, which complicates their efforts to reproduce bugs. In this paper, we propose REPASSISTOR, an approach that employs an interactive method to assist developers in reproducing bugs based on automated testing logs. REPASSISTOR is designed for reproducing bugs in mobile applications. It leverages deep learning (DL) and traditional computer vision (CV) techniques to analyze application screenshots, transforming automated testing logs into a graph representation. In this graph, edges represent test events while nodes represent application states. With this graph representation in place, REPASSISTOR is then able to monitor developers' actions and continuously track which node they are at in this graph in real-time. Based on this understanding, REPASSISTOR dynamically calculates and updates the optimal path to guide developers to reproduce bugs. This guidance is conveyed to the developers through an interactive method, enabling effective communication and assistance throughout the bug reproduction process. Our experiments demonstrate that REPASSISTOR improves the performance of developers in bug reproduction tasks.

Keywords–bug reproduction, conversational agent, software testing

1. INTRODUCTION

With the development of software technology, software is becoming increasingly powerful and correspondingly more complex [1]. To ensure its reliability, software testing plays a crucial role in software development [2], [3]. Initially, software testing was primarily conducted manually. However, given the increasing complexity of and the growing demands for software testing, manual efforts often face a shortage of manpower. As a result, automated testing tools have been developed to automate the process of software testing, demonstrating a concerted effort to ensure quality [4], [5], [6], [7].

Bugs discovered by automated testing tools need to be reproduced to confirm and address them. Similarly, manual reproduction may face a lack of manpower given the vast amount of testing done by automated testing tools. As a response, many automated reproduction tools [8], [9], [10], [11] have been developed to reproduce known bugs. These automated reproduction tools can analyze testing logs or testing reports and reproduce some of the bugs identified within the application.

However, automated reproduction tools are not completely reliable. They typically rely on specific environments to reproduce bugs. If a bug is tied to complex system environments, the automated tools may not be able to reproduce it. Furthermore, automated reproduction tools are also fragile and may fail to accurately reproduce bugs if there are changes in system configurations or widgets. For example, in rapidly changing UI scenarios (*e.g.,* news applications), automated reproduction tools struggle to recognize changes of the UI, hence finding it challenging to identify widgets and reproduce them accurately. In cases where the bugs are complex, automated reproduction tools may have difficulty determining if a bug has been successfully reproduced. During such instances, manual confirmation is often required.

This brings us to the necessity and advantages of manual reproduction. Manual reproduction not only avoids the problems of automated reproduction tools but also carries other benefits. Manual reproduction enables developers to confirm the existence of bugs reported by automated testing tools and gain a better insight into the problem [12]. Additionally, manual reproduction aids developers in writing more comprehensive bug reports. When bugs are fixed, verifying the effectiveness of those fixes also requires manual reproduction.

However, despite these advantages, reproducing bugs manually can be challenging for developers. Automated testing tools often include information that can be challenging for humans to comprehend, such as coordinates and complex file structures like XML. These unreadable data can make it difficult for developers to understand the testing results. Additionally, it requires not only basic testing skills but also a comprehensive familiarity with the specific software system [13], which is necessary to identify the environment where bugs occur.

Our study aims to address these issues related to manual reproduction. We propose a new tool, REPASSISTOR, to assist developers in reproducing bugs that are already discovered and documented by automated testing tools(*e.g.,* monkey [14]). REPASSISTOR is a bug reproduction guidance tool developed for Android and based on a conversational agent (*i.e.,* chatbot). Automated testing tools generate testing logs during the testing process. REPASSISTOR can analyze these logs, extracting GUI images and test events from them. These GUI images contain vital information about the application's page transaction structure, which is crucial for developers during reproduction.

DOI 10.1109/DSA59317.2023.00024

RepAssistor processes the extracted information and uses DL and traditional CV techniques to merge similar pages. By directly dealing with images through DL and traditional CV, RepAssistor avoid solving the application's underlying UI files or code. These merged pages, combined with the action timestamps from the logs, form a graph data structure. These action timestamps denote the sequence of actions, thus establishing transition relationships between the merged pages. RepAssistor reconstructs the testing process from this data structure, building a page transition model of the application. Then, RepAssistor monitors the developers' actions and obtains their current user interface in real time. RepAssistor compares the developers' current user interface with the generated model to determine their precise position within the model. Subsequently, RepAssistor calculates the optimal action sequence to trigger the bug and conveys these actions in a user-friendly way through an interactive communication approach, guiding the developers in reproducing the bug. This approach helps to solve issues stemming from a lack of familiarity with the application of the developers. Furthermore, RepAssistor takes into account potential unintended situations, such as developers not following guidance instructions as expected and performing incorrect actions. Developers may intentionally or unintentionally deviate from the provided guidance. In these cases, RepAssistor applies appropriate measures guide developers back to the correct interface.

Inspired by previous research on the use of conversational agents in helping write testing reports [15], [16], RepAssistor extends the focus to the aspect of human developers. When it comes to software testing, interactive methods are often used to assist in writing testing reports or performing similar simple tasks. In contrast, RepAssistor innovatively applies them to guide developers in reproducing bugs, particularly those found by automated testing tools. Developers who conduct the testing may not necessarily be very familiar with the application, and they might also lack relevant testing experience. Unlike only providing analysis reports, RepAssistor's conversational agent can give real-time feedback on testing information, correct errors during the testing process, and guide developers towards more effective testing. As such, RepAssistor can serve as a bridge between automated testing tools and developers, preventing situations where the capabilities of automated testing tools surpass the ability to reproduce bugs of developers. RepAssistor also differs from other automated bug reproduction tools that actively reproduce bugs based on known information. Instead, it delegates bug reproduction to developers while leveraging an interactive method to improve their efficiency and abilities.

The conversational agent is the core of RepAssistor. It provides real time interaction, allowing developers to get immediate guidance and support. This helps developers locate and operate reproduction steps more efficiently. Additionally, the conversational agent's natural language understanding capability enables developers to communicate with the system in a familiar manner, lowering the learning curve and making it more accessible to non-professional developers.

RepAssistor's conversational agent is developed using a highly customizable conversational agent framework [17]. Compared to some traditional conversational agents [18], RepAssistor employs a more complex and customizable solution. We primarily focused on training the Natural Language Understanding (NLU) component to ensure the conversational agent fully understands the terminology of software testing and accurately conveys developers' intentions. Additionally, we customized its dialogue management component to provide reproduction guidance instructions, enabling it to effectively assist developers throughout the reproduction process.

In order to conduct our experiment with RepAssistor, we selected a diverse set of eight applications. To ensure the representativeness and diversity of our dataset, we utilized Monkey as the automated testing tool and generated 28 corresponding testing logs. Out of the eight applications, RepAssistor was successful in constructing accurate models for seven, though one application exhibited some inaccuracies in the test events. RepAssistor demonstrated a strong performance in assisting developers, successfully facilitating the complete reproduction of 21 out of the 28 testing logs (75%), while partially guiding the reproduction of the remaining logs. Furthermore, RepAssistor eliminates redundant test events, thus improving the overall speed and effectiveness of reproduction.

In summary, our study makes the following contributions:

- Design and develop RepAssistor, a testing result reproduction guidance tool based on a conversational agent.
- Conduct experiments on RepAssistor, demonstrating its effectiveness and accuracy in improving reproduction.

2. Motivation

There are many automated testing tools available, such as Monkey [14], Appium [19], *etc.*. Developers can use reproduction tools like ReCDroid [20] to reproduce the testing results of these automated tools. However, such tools don't always reproduce the results accurately. A variety of factors can influence the outcome when using these tools, including the system environment, specific app types, and compatibility issues, which could lead to unsatisfactory results.

In situations where automated reproduction fails, manual reproduction becomes necessary. However, Manual reproduction also faces various difficulties. When conducting manual reproduction, developers must locate the bug's position in the testing logs, try to reproduce the bug, understand it, and then fix it. Testing logs often contain a large amount of information and are difficult to comprehend, making the reproduction process challenging for developers.

2.1. Example of Appium

Appium [19] is an automated testing framework. Many automated testing tools are built on it [21], [22]. In one test, Appium's log records a tap test event on the 'settings' widget, highlighted by a red frame in Figure 1. The testing log records the widget's xpath (represents the widget's location in the source XML file), class, and id, which are machine-readable

but might not be easily comprehensible to developers. As a result, performing the operation can be quite difficult.

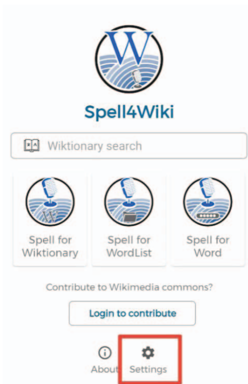

Figure 1: Example of Appium

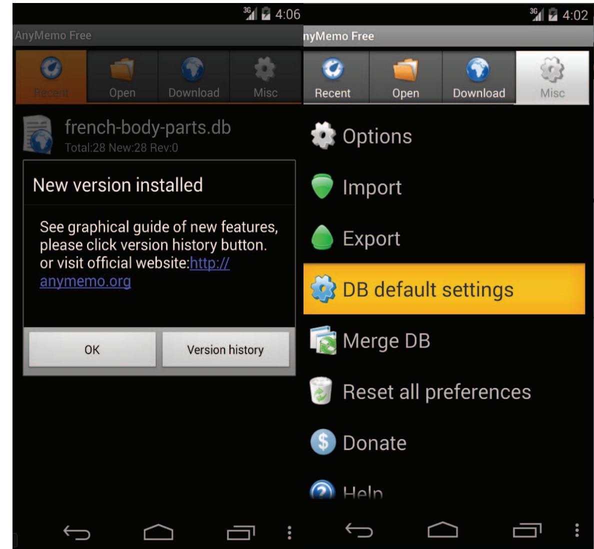

Figure 2: Example of ReCDroid

2.2. Example of ReCDroid

ReCDroid [20] is a representative tool for bug reproduction currently. ReCDroid uses artificial intelligence techniques to help reproduce different software issues effectively and accurately. However, it does have some limitations in practice. When the application interface contains confusing or ambiguous information, the UI search mechanism of ReCDroid fails to function correctly. For instance, we use ReCDroid to reproduce a bug that was triggered by switching from the app's start page to the misc page and clicking the notification button. ReCDroid, during the process, first automatically simulated the user behavior of switching from the app's launch page to the misc page, then it attempted to click the notification button. However, while ReCDroid managed to accomplish most steps successfully, it did not click on the correct target widget in the final test event, as shown in Figure 2. In other words, even though ReCDroid accurately reproduced most of the test events, it fell short in the final, critical click operation. While ReCDroid has improved bug reproduction efficiency to some extent, in some complex scenarios, we still need to rely on manual reproduction for the reliability.

3. Approach

We propose RepAssistor, a tool designed to assist software developers in reproducing bugs found by automated testing tools. RepAssistor simplifies the debugging process and enhances the user experience by the following key capabilities: (i) Constructing a model that uses information from automated testing logs to discern the necessary test events to reproduce the bug. (ii) Identifying the user's position within the model, determining the system's current state, then using algorithms to calculate the optimal actions needed to reproduce the bug. (iii) Guiding developers in reproducing the bug through an integrated interactive interface, which offers real-time support, instructions, and feedback on their actions.

RepAssistor extracts *Test Events* and *Exceptional State* from logs of automated testing to serve as a foundation for bug reproduction. Typically, exceptional state refers to uncaught exceptions in the software. However, in the context of RepAssistor, exceptional state specifically denotes an exception that is recorded in the log instead of in the software. It corresponds to a single bug, as only such an exception can be extracted from the log and can indicate a certain error. After obtaining the test events and exceptional state, RepAssistor transforms them into a specific model as a basis of the next guidance. To facilitate bug reproduction, a conversational agent is employed to guide users in exploring the test events accurately, helping to reproduce the exceptional state. At present, RepAssistor is developed based on the Android with Android Debug Bridge [23].

RepAssistor primarily consists of three processes: *model transformation*, *guidance generation*, and *reproduction guidance*. Model transformation tackles complex automated testing logs first. Automated testing logs contain various data such as steps, screenshots, exception records, and other pertinent information. To make the data more manageable and user-friendly, standardization is applied. Once the data is standardized, the transformed model is structured as a multilateral directed graph. In this graph, nodes represent application activities or the exceptional state, while edges signify test events. A path from the first page to the exceptional state represents a possible reproduction guidance, and the goal of RepAssistor is to navigate developers from application launch, through a series of operations to reach the final exceptional state. This graphical data structure simplifies navigation through the data, making path generation guidance easier. An illustration of a model is

shown in the Figure 3.

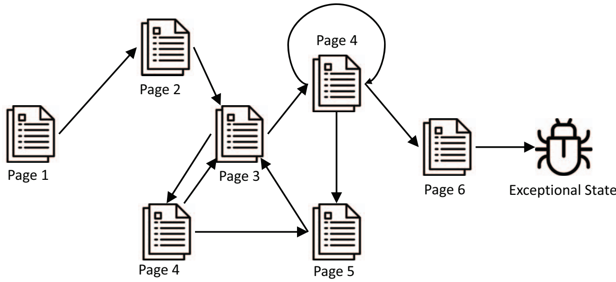

Figure 3: The multilateral directed graph for navigation.

REPASSISTOR employs a shortest-path strategy based on the transformed model. This involves monitoring the developer's application user interface in real time, estimating their position, and dynamically calculating the guide path for bug reproduction. The reproduction guidance is executed by an integrated, user-friendly interactive system that engages developers to provide precise and concise guidance. This interactive system not only simplifies the process but also enhances the overall user experience, making it easier for non-professional developers to follow the instructions for bug reproduction.
The entire operation process of REPASSISTOR can be viewed in Figure 4. In this section, we will delve into the development of REPASSISTOR in greater detail, elucidating its practical applications in the realm of software testing.

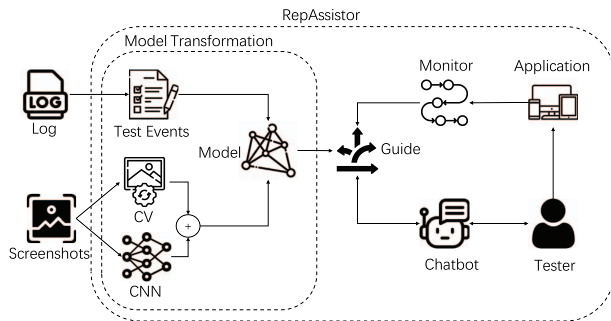

Figure 4: The process of the REPASSISTOR

3.1. Model Transformation

Model transformation contains three parts: *data standardization*, *page similarity calculation*, and *model building*. These phases work together to convert automated testing logs into a structured model. The structured model can be used to guide developers in reproducing bugs.
3.1.1. **Data Standardization**: The primary goal of data standardization is to process and normalize the automated testing log data, converting it into a format suitable for model transformation. Automated testing logs can be generated from multiple sources, including different testing tools and platforms. Data standardization is crucial for REPASSISTOR to adapt to logs generated by various automated testing tools. This adaptability allows REPASSISTOR to remain independent and decouples it from certain specific testing tools.
Testing logs should include test events, runtime screenshots (used for subsequent page merging), and exceptional state. These logs are then standardized into linked lists, where all nodes, except the last one, represent screenshots. The final node denotes the exceptional state, and the links signify test events. The linked list illustrates the transition relationships between user interfaces, allowing users to navigate between screenshots by following the test events until the exceptional state is reached. The structure of the linked list provides convenience for the model building, making it easier to work with the data in the next phases.
Additionally, data normalization includes the process of extracting widgets from application screenshots. During guidance, the specific content of the operation needs to be displayed to the developers, involving widgets. For example, in the case of clicking the "add" button, the specific style of the add button must be known in order to convey the detailed operation to the developer. During data normalization, the Canny algorithm [24] is used to detect the edges of widgets and extract them from the screenshots.
3.1.2. **Page Similarity Calculation**: Page similarity calculation involves identifying all application screenshots and assessing their similarity. The concept behind this method is frequently utilized for managing application interfaces, such as in the processing of crowdsourced testing reports [25], [26]. Accurate similarity measurements are vital for merging similar pages and reducing the complexity of the final model. The overall similarity can be calculated by the following methods.
Computer vision method. The traditional correlation matching method of traditional CV is employed to obtain matching coefficients between application screenshots [27]. These coefficients serve as a preliminary measure to determine the similarity of pages. This method is relatively fast and can provide a rough estimation of similarity. Let the matching score of the two screenshots be denoted as α.
Deep learning method. DL techniques are used to calculate the similarity between two application screenshots in our work. We utilize VGG16 [28], a classic deep convolutional neural network (CNN) model, to determine the similarity. The images are input into the CNN to obtain their embedding vectors, and the distance is represented by the Euclidean distance between these vectors. The similarity is expressed as 1 minus the distance. This approach provides a more accurate and robust similarity compared to the traditional CV method. Let the similarity obtained by DL be denoted as β.
3.1.3. **Model Building**: The model building process leverages the previously calculated similarity to evaluate the resemblance between the pages and perform the merging operation. The merging operation combines two pages into one, and all attributes of the original pages, such as screenshots and extracted widgets, are preserved. The merging criterion is that

one similarity value of the two screenshots must surpass a specific threshold, with similarity obtained by traditional CV being considered first. The specific merging method can be seen in (1). This logic allows the traditional CV method to short-circuit the formula, reducing the computational load. Through experimentation, we set the similarity threshold at 0.9 and 0.8 for traditional CV and DL. The merging operation folds two different nodes in the linked list into one, and the original links will be reconnected. Then, more nodes will be folded in. Continual merging operations will eventually form a graph, which is the target model. These threshold values ensure that only highly similar pages are merged, reducing the risk of merging unrelated pages.

$$Merging = \begin{cases} True, & \alpha > threshold_{CV} \\ True, & \beta > threshold_{DL} \\ False, & otherwise \end{cases} \quad (1)$$

For two pages meeting the merging requirement, a straightforward reconnection of the links in the linked list is executed to construct the model. This process results in a simplified model that retains the necessary information for reproducing the bugs, while eliminating redundant data. Merging similar pages also reduces the cognitive load on developers, as some actions that do not change the page are linked to the same page. Their priority is reduced and developers will avoid these low-priority test events. The algorithm is presented in Algorithm 1.

Algorithm 1: Model Transformer

```
Input: Automated Testing log log
Output: Transformed Model
1  extract actions A in log
2  extract screenshots S in log
3  extract error e in log
4  initiate linked list L
5  initiate first node node
6  L.add(node)
7  foreach screenshot ∈ S do
8  |   initiate page
9  |   page.screenshot ← screenshot
10 |   page.action←A.findAction(screenshot)
11 |   node.next←page
12 |   node←page
13 end
14 remove the first node in L
15 foreach n_i ∈ L do
16 |   foreach n_j ∈ L do
17 |   |   if n_i is similar to n_j then
18 |   |   |   mergePages(n_i, n_j)
19 |   |   end
20 |   end
21 end
22 node.next←e
23 return L
```

The final established model is a directed multigraph with (potentially) self-loops. This directed multigraph illustrates the transition relationships between activities of an application, enabling the application to be traversed from the initial page to the final exceptional state through a series of test events. This feature serves as the foundation for guiding users to reproduce bugs in subsequent stages. With the model, developers can follow a structured, guided approach to reproducing bugs.

3.2. Guidance Generation

Guidance generation is the core of REPASSISTOR, which generates guidance paths and updates them in real time. The paths are generated based on the developer's actions, they can efficiently guide them in reproducing exceptional states.

To achieve guidance generation's functionality, real time state monitoring and path generation are essential. Initially, the guidance generation starts in the standby state and generates a guidance path from the initial page to the exceptional state according to the shortest path first strategy. Afterward, it monitors the developer's actions, detects their position after each action, and updates the guidance path in real time. When developers struggle to reproduce the exceptional state, guidance generation displays the guidance path to developers by the conversational agent.

During the reproduction process, unexpected situations may occur. Only the pages reached and the actions performed during automated testing are recorded, which is obviously not comprehensive. This record is not complete and does not cover every possible scenario. In practice, developers may perform unknown test events and reach pages that were not accounted for during the automated testing phase. This can lead to gaps in the reproduction process.

We refer to these instances, where developers reach unrecorded pages, as *'Missing'*. This term indicates that developers operated widgets of the software that have not been captured in the automated testing process. These 'Missing' states can be due to a variety of reasons, such as unexpected user behaviors, overlooked scenarios in the automated testing process, or even changes of the software's environment that have not happened during testing. These issues are difficult to avoid.

Furthermore, once the guidance process begins, developers may misunderstand REPASSISTOR's guidance intention. This misunderstanding can lead to unexpected actions being performed, causing developers to depart from the planned guidance path. We refer to this state as *'Deviation'*. 'Deviation' represents developers deviating from the expected guidance path REPASSISTOR gave. These deviations can potentially lead to the failure of reproduction.

Based on the discussion above, this section will introduce guidance generation from three perspectives: guidance state switching, monitoring mechanism, and guidance methods.

3.2.1. **Guidance State Switching**: To guide developers efficiently and accurately in reproducing exceptional states, guidance generation has two states, including standby state and guiding state. In the standby state, developers explore the application freely, attempting to reproduce the exceptional state by analyzing test logs or based on their own experience.

There are two ways to enter the guiding state from the standby state: requesting guidance actively or reaching a 'Missing' state. The first way means developers recognize that they have difficulty reproducing the exceptional state and require additional assistance to navigate the process. The second way means that despite developers' confidence in reproducing the exceptional state on their own, they seem unable to do so. Both cases need to change the state to guiding. In the guiding state, users receive real time guidance prompts from guidance generation via the conversational agent. This interactive approach allows developers to follow step-by-step instructions, which can greatly increase the possibility of reproducing the exceptional state. Additionally, developers can report whether the guidance is successful, providing feedback for improving the guidance generation system. More user-friendly guidance can be provided by the mechanism of switching the state.

3.2.2. **Monitoring Mechanism**: Guidance generation needs to determine the developer's position in the model to update the guidance path and detect the developer's state. This is crucial for providing accurate and timely instructions to the developer. Guidance generation can read the developer's operation on the device as a trigger to update the current position. When a new action is performed, a new transaction to change position occurs. The logic for updating the position is similar to the page merging logic in model transformation, which is done by calculating similarity. The difference is that the thresholds are reduced, from 0.9 and 0.8 to 0.9 and 0.7. The adjusted thresholds allow for greater flexibility in recognizing the developer's current position within the model. If the current page's similarity to all pages in the model is below the threshold, mark the developer's state as 'Missing' instead of creating a new page. This monitoring method ensures the real time performance of guidance.

3.2.3. **Guidance Method**: REPASSISTOR adopts the most straightforward path-finding method, the shortest-path strategy, to attempt to generate the guidance path. This strategy is based on the principle of efficiency, aiming to minimize the number of test events a developer needs to reproduce the exceptional state. The starting point of this path is typically the initial page of the model, which represents the point where developers begin reproducing. The endpoint, on the other hand, is the exceptional state, with the connections between nodes representing test events. However, as mentioned earlier, developers may enter abnormal states, 'Deviation' and 'Missing'.

When a developer enters a 'Deviation' state, the guidance generation process recalculates the fastest way to return to the original guidance path. This recalculated path is then appended to the original guidance path, creating a new guidance path. This adaptive approach is designed to help developers recover from the abnormal state and continue their reproduction. It's a dynamic solution that solves the unpredictable state and provides a mechanism for correction.

In the case of a 'Missing' state, the guidance generation process instructs the developer to return to the last recorded position. This approach is designed to bring the developer back within the model's control, ensuring that they remain within the scope of the model. The full algorithm of the guidance method is detailed in Algorithm 2.

Algorithm 2:

```
Input: Guide Model model,User Monitor monitor
1  initiate GG as Guide Generation
2  initiate guide path
     path ← model.getPath(model.first,model.ES)
3  initiate position as model.first
4  while True do
5      monitor.acquireLock()
6      if monitor.getPosition() ≠ position then
7          position←monitor.getPosition()
8          if position not in model then
9              GG.state←MISSING
10         end
11         else if position deviated from path then
12             GG.state←DEVIATION
13             extraPath←model.getPath(position,path.first)
14         end
15         else
16             path←model.getPath(position,model.ES)
17         end
18     end
19     monitor.releaseLock()
20 end
```

This algorithm provides an approach to handling both normal and abnormal states. It guarantees the robustness and adaptability of REPASSISTOR, capable of handling a wide range of scenarios and ensuring that developers can accurately reproduce exceptional states.

3.3. Reproduction Guidance

In REPASSISTOR, the reproduction guidance process is responsible for directly interacting with users and controlling the entire guidance process. The complete process is shown in Figure 6. It can transform the real time guidance paths generated by guidance generation into a more readable form and display them to developers. It can also convey the developer's intentions for guidance generation. The core of reproduction guidance is a conversational agent. Some events such as sending guidance and asking for results are defined for the conversational agent. The triggering logic is managed by guide generation. We also added some dialogue rules in the testing domain, such as initiating guidance and conveying test results to the conversational agent. By the defined events and rules, we enabled the conversational agent to directly guide developers in reproducing the exceptional state.

After starting, the developer tests on their own without any instruction from reproduction guidance, while guidance generation enters the standby state. When developers actively request guidance or guidance generation determines that developers are unable to reproduce the exceptional state independently, reproduction guidance begins providing guidance. It presents

actions and the application's expected behavior to developers when guiding, where actions include the type of operation and the widget. There are two ways to guide developers: full path and step by step. Reproduction guidance first presents the full guidance path obtained from the guidance generation process to the developers, allowing them to have a basic understanding of the test events. Then, reproduction guidance obtains the developer's current position in real time from guidance generation to extract the details of the next action and informs the developer, as can be seen in Figure **??**. Reproduction guidance prompts the developer that he or she should perform test event 1 to reach page B. When the next test event can reach the exceptional state, reproduction guidance asks developers whether the reproduction was successful and records this reproduction as a complete guidance process.

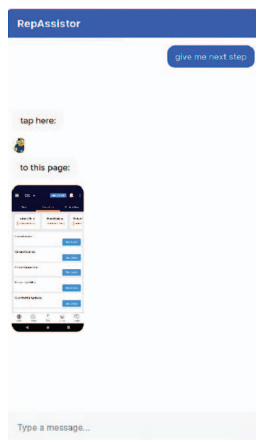

Figure 5: The real-time guide.

Similarly to guidance generation, developers may enter two abnormal states: 'Deviation' and 'Missing'. In the 'Deviation' state, the reproduction guidance plays a crucial role in navigating developers back to the planned guidance path. It does this by first alerting developers that they have deviated from the guidance path. This alert serves as a checkpoint, allowing developers to realize that they have deviated and need to recover. Then reproduction guidance re-routes developers with the new guidance path in a step by step manner, making it easy for developers to follow and return to the correct path. On the other hand, the 'Missing' state represents a more severe deviation, where developers have accessed unrecorded pages and are completely off the planned path. In this state, the reproduction guidance alerts developers that they may have completely deviated and sends them the last recorded page, asking them to return on their own. Once developers have returned to a recorded page, the reproduction guidance resumes its role in guiding them. It finds a new guidance path based on the current page and guides developers along this path, ensuring that they can continue the reproduction process. Through the carefully designed user-friendly conversational agent, reproduction guidance can serve as a bridge for communication between developers and REPASSISTOR and thus

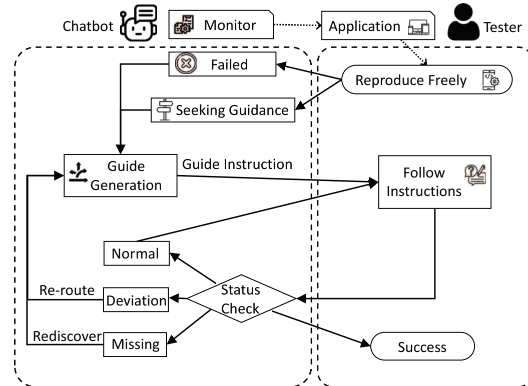

Figure 6: The flow of reproduction guidance.

complete the guidance process.

4. EXPERIMENTS

The accuracy of Model Transformation and the accuracy coupled with the efficiency of Guidance Generation are critical factors of REPASSISTOR. To evaluate these key aspects, we focus on the following points:
(i) Perceived usefulness and usability of the REPASSISTOR; (ii) Accuracy of REPASSISTOR in model transformation and guidance generation; (iii) Quality of the reproduction guidance.
To evaluate these attributes, we conducted experiments on various applications, aiming to answer the research questions:
RQ1: How accurate is the model transformation?
RQ2: How accurate is the reproduction guidance?
RQ3: What is the efficiency of the reproduction guidance?
To address the research questions, we selected a set of Android applications (refer to section 4.1) and employed Monkey [14] to collect logs and screenshots. These apps are widely used open-source applications in the area of automated testing, each corresponding to some logs that records bugs discovered through the testing scripts using Monkey strategy. Subsequently, we extracted the metadata of each application, such as the total number of exceptional states per app. We then identified relevant evaluation metrics and established a method to compute them using the experimental data (refer to Section 4.2). We applied REPASSISTOR to each app and collected the experimental data, including model and guidance information. Afterward, we analyzed the evaluation metrics and evaluated REPASSISTOR's behavior during the model transformation and guidance generation. The results of this evaluation, as well as our discussion, are detailed in Section 4.3.

4.1. Experimental Setup

To evaluate the effectiveness of REPASSISTOR, we chose eight Android apps, each representing a unique domain and possessing a variety of bugs. Automated test logs were procured by Monkey, which served as our automated testing tool. By

analyzing the meta information derived from Monkey's test logs and screenshots, we established the standard guidance path. Detailed records are presented in Table 1.
To guarantee the reliability of the datasets, we verified the meta records on both an Android virtual machine and a real device after obtaining the test logs.

Table 1. Original test log of apps and bug dataset

App	ID	Events	Last event
anki	1	18	tap
	2	29	tap
	3	32	tap
	4	311	back
budget watch	5	26	tap
	6	21	tap
	7	230	longtouch
	8	99	tap
myExpenses	9	31	tap
privacy friendly notes	10	43	tap
omninotes	11	11	tap
	12	19	tap
runnerup	13	1	tap
	14	8	back
	15	6	tap
	16	22	tap
	17	84	tap
	18	152	tap
passwordmaker	19	9	tap
	20	81	input
	21	46	tap
	22	7	input
	23	59	back
	24	3	longtouch
	25	6	tap
timber	26	119	back
	27	34	tap
	28	75	input

We processed each automated test log to obtain testing data, including steps, screenshots, exception records, and other relevant information. Once the data was collected, we transformed the data into multilateral directed graphs, which facilitated the guidance path generation. Subsequently, we calculated the shortest path and documented the paths. We then conducted an experiment for each bug, with the guidance provided by RepAssistor recorded in Table 2. As developers may seek help from a different starting point compared to the original test log, we noted whether the first test event was the same. By comparing the records from the original logs with those from RepAssistor, we evaluated the tool's performance and analyzed the research questions.

Table 2. Guidance from RepAssistor

App	ID	Guidance	Same start
anki	1	3	N
	2	5	N
	3	22	Y
	4	95	N
budget watch	5	9	Y
	6	7	Y
	7	3	Y
	8	3	Y
myExpenses	9	5	Y
privacy friendly notes	10	2	Y
omninotes	11	1	N
	12	1	Y
runnerup	13	1	Y
	14	1	Y
	15	2	Y
	16	3	Y
	17	8	Y
	18	2	Y
passwordmaker	19	2	Y
	20	1	Y
	21	2	Y
	22	1	Y
	23	3	Y
	24	1	Y
	25	1	Y
timber	26	1	Y
	27	1	Y
	28	1	Y

4.2. Experiments design

4.2.1. **RQ1: The accuracy of RepAssistor's model transformation.**: The accuracy of model transformation is fundamental to effective reproduction guidance. The model should contain complete information extracted from the automated testing logs, accurately documenting the details of each test event. To quantify the accuracy of the model transformation, we employ the metric of model completeness rate. This metric is calculated as the ratio of applications for which all models are complete to the total number of applications. This measure serves to provide an objective evaluation of the model's comprehensiveness and accuracy, ensuring the reliability of the reproduction guidance.

4.2.2. **RQ2: The accuracy of RepAssistor's reproduction guidance.**: RQ2 assesses whether the guidance accurately represents the correct sequence of test events and effectively directs developers to the desired exceptional state. We evaluate the accuracy of guidance via the guidance correctness rate. The guidance provided by RepAssistor is considered correct if it successfully navigates to the exceptional state and triggers the exceptional state. The guidance correctness rate quantifies the proportion of successful guidance and is calculated as the ratio of successful guidance instances to the total number of guidance attempts. This metric provides a valuable measure of the guidance's reliability in correctly leading developers to the target exceptional state.

4.2.3. **RQ3: The efficiency of RepAssistor's reproduction guidance.**: While RQ2 assesses the accuracy of the generated guidance, RQ3 primarily focuses on its efficiency. The efficiency of the guidance is assessed by comparing the length of the guidance path to the actual path derived from the original test log. We calculate the difference between the number of test events in the original test log and the number of test events in the guidance. This difference is then divided by the total number of test events in the original test log, providing an evaluation criterion for efficiency. The larger this value, the greater the efficiency of the guidance.

$$Efficiency = \frac{origin - guidance}{origin} \tag{2}$$

4.3. Results and analysis

we provide a comprehensive presentation and discussion of our evaluation results for each RQ.

4.3.1. **RQ1: The accuracy of RepAssistor's model transformation.**: We obtained a total of 28 models by constructing models for 8 apps. We then traversed each page of each model, observing for any blank pages or incorrect test events to determine the accuracy of the model. Through experiments, we found the app 'timber' contained unrecorded operation coordinates, while the models for the other applications were correct. Consequently, the model completeness rate was high, peaking at 87.5%. This high rate of completeness signifies that the model transformation exhibits a high degree of accuracy. This suggests that RepAssistor is highly reliable in transforming automated test logs into models which in turn enhances the quality of the reproduction guidance.

4.3.2. **RQ2: The accuracy of RepAssistor's reproduction guidance.**: RepAssistor successfully documented a total of 28 complete guidance instances. As demonstrated in Table 3, seven instances had triggering test events that were different from the original logs, resulting in a path correctness rate of 75%. Within the seven instances of incorrect guidance, five instances featured incorrect interfaces during the triggering of the exceptional state, one instance presented an incorrect operation, and one instance manifested both errors. Despite these discrepancies, the path correctness rate remains considerably high, suggesting that RepAssistor possesses a robust capability to parse automated testing logs accurately and provide precise guidance.

4.3.3. **RQ3: The efficiency of RepAssistor's reproduction guidance.**: In RQ3, we compared the number of test events in the original logs with those in the corresponding guidance to evaluate the efficiency of the guidance. We first eliminated cases with different starting points, resulting in 16 guidance instances that met this requirement. For these 16 instances, we employed equation (2) to calculate the efficiency. The results of this calculation are displayed in Table 4. The overall efficiency achieved was 76.78%, suggesting a high level of efficiency in the majority of the guidance instances. This implies that compared with the original test logs, RepAssistor can avoid some ineffective operations, thereby enhancing the efficiency of the bug reproduction process. These results indicate that RepAssistor is not only precise in its guidance but also efficient, resulting in quicker bug reproduction.

Table 3. Accuracy of reproduction guidance

App	ID	Same end	Validity
anki	1	Y	Y
	2	Y	Y
	3	Y	Y
	4	Y	Y
budget watch	5	Y	Y
	6	Y	Y
	7	Y	Y
	8	Y	Y
myExpenses	9	Y	Y
privacy friendly notes	10	Y	Y
omninotes	11	Y	Y
	12	Y	Y
runnerup	13	Y	Y
	14	Y	Y
	15	Y	Y
	16	Y	Y
	17	Y	Y
	18	Y	Y
passwordmaker	19	Y	N
	20	Y	N
	21	N	Y
	22	N	Y
	23	Y	Y
	24	N	Y
	25	Y	Y
timber	26	N	Y
	27	N	N
	28	N	Y

5. Discussion

Limitations. The current version of RepAssistor has certain limitations. Model transformation is entirely based on automated testing logs, which inherently maintain its limitations [29], [30], such as incorrect error records and incomplete reproduction actions. The error information in the original logs might be perplexing enough that RepAssistor cannot extract valuable information from it. This can be addressed by analyzing, correcting, and supplementing the testing logs, which have not been integrated into the current RepAssistor. RepAssistor also struggles to find effective paths to exceptional state when the application is too complex, or the guiding path is too long. Some applications are relatively complex, and the pages traversed to trigger a bug may be similar or interconnected (*e.g.,* a bug can only be properly triggered when navigating from a specific page). Under these circumstances, the effectiveness of the RepAssistor's guidance can obviously decrease. This is due to various factors, including the validity of application screenshot comparison,

Table 4. Efficiency

App	ID	Events	Guidance	Efficiency
anki	3	32	22	31.25%
budget watch	5	26	9	52.94%
	6	21	7	66.67%
	7	230	3	98.70%
	8	99	3	96.97%
myExpenses	9	31	5	83.87%
privacy friendly notes	10	43	2	95.35%
omninotes	12	19	1	94.74%
runnerup	13	1	1	0%
	14	8	1	87.5%
	15	6	2	66.67%
	16	22	3	86.36%
	17	84	8	90.48%
	18	152	2	98.68%
passwordmaker	23	59	3	94.91%
	25	6	1	83.33%
total		52.44	4.56	76.78%

and path selection strategy. When screenshots of application UI are quite similar, it becomes difficult to efficiently compare images [31], leading to different Activities being recognized as the same. This can reduce the effectiveness of the transformed model. The path selection strategy, which currently uses the simplest shortest path strategy, may also impact this process. Some exceptional states might require special paths to reach, and these special paths may be a subset of the automated testing logs. As a result, the shortest path may be ineffective for complex bug reproduction. To minimize this issue, it is necessary to analyze potential preceding events that lead to the bug and guide users to complete those events before proceeding with the final reproduction process. This may be considered in our future research.

Lastly, the application scope of REPASSISTOR is confined to bugs already discovered by automated testing tools. Bugs identified through other means are not addressable using REPASSISTOR, as the lack of testing logs. Moreover, some complex actions within applications, such as rotation, cannot be recorded and guided by REPASSISTOR.

Threats to validity. The primary threat to the validity of this study is the generalizability of the selected application and corresponding automated testing logs. Applications come in many types, some of which are extremely complex and unique, such as games. REPASSISTOR's capabilities have not been sufficiently confirmed for these special types of applications. Not only the applications but also the automated testing logs can threaten the validity. Most of the logs used in this study came from random testing methods like Monkey, which inherently has some limitations. It may generate many invalid actions, which can disrupt the guidance path, causing a decrease in reproduction guidance efficiency.

At the current stage, REPASSISTOR has not yet undergone user studies and comparative experiments with similar tools (*e.g.,* RecDroid) to verify its assistance to developers. This part can be supplemented in subsequent research.

Chatbots. Chatbots are now widely used in software engineering [32], [33], [34]. In the software testing domain, they mainly assist developers in understanding testing results, helping them complete simple testing tasks or writing testing reports [15] [16]. The main advantage of chatbots is their user-friendliness, which can compensate for developers' lack of relevant professional knowledge [35], [36].

6. RELATED WORK

The related work primarily focuses on automating bug reproduction, with the central theme being the reproduction of user bug reports or error reports. ReCDroid [20] analyzes bug reports to reproduce bugs. It uses NLP technology to extract key steps from reports. It then compares the target widget in the steps with the elements extracted from the application source file to confirm the targets of actions, thereby automating the reproduction based on the report. Yakusu [37] transforms user-reported bugs into test cases by extracting and identifying widgets from the mobile application's source code. It processes bug reproduction steps using NLP technology and matches widgets with UI elements through deep learning techniques. Finally, it generates a test case library using multiple strategies, attempting to reproduce the described bugs.

In addition to bug reports, studies are attempting to reproduce bugs using information from exception stacks or system information [38], [39]. CrashLocator [9] uses static analysis to expand the crash stack and attempts to find suspicious functions. The suspicious function is the key to reproducing the bug. All these methods extract useful content from ambiguous or complex information to automate bug reproduction. However, these methods have not considered the key factor, the developers' ability during bug reproduction.

Besides, GIFDroid [40] incorporates multimedia data into the research of bug reproduction. GIFDroid identifies keyframes in videos that reproduce bugs and compares them with the actual application in order to generate the reproduction steps. Similar to REPASSISTOR, GIFDroid employs image comparison techniques to find widgets and activities. However, it also overlooks the crucial role of developers.

Utilizing chatbots to assist in software testing is another focus of related research. BURT [15] employs chatbots to help developers refine their bug reports. Similar to REPASSISTOR, it first loads an application model. It offers potential actions for developers to choose according to the model. Based on the developer's selection, it recommends new candidates to assist developers in the completion of the bug report's main content and automatically generate the report. BURT's assisted report-writing approach simplifies developers' work. Similarly, FUSION [16] uses dynamic analysis to obtain application GUI widgets, assisting developers in completing error reports. However, the primary objective of these methods is to aid developers in reporting bugs rather than reproducing them.

They are complementary to REPASSISTOR and can be used together to improve reproduction ability.

There are a considerable number of related studies on automated testing tools [41], [42], [43], [44]. Their goal is to replace or assist manual testing and make the testing process more efficient. They usually target high coverage or more software bug detection by employing complex algorithms and models. These automated testing tools provide the foundation for REPASSISTOR. Many studies of them use GUI features for automated testing. The log generated by these works can easily be adapted to REPASSISTOR with data normalization.

When processing testing logs, REPASSISTOR employs DL and CV techniques for handling application images. Likely, Wang *et al.* [45] use deep learning technologies to process reports in their work. Yu *et al.* [25] applies deep learning techniques to individual components for feature extraction in test reports. Nguyen *et al.* [46] employed CV technology to analyze GUI screens and manipulate their widgets.

7. CONCLUSION

This paper introduces REPASSISTOR, a tool that aids developers in reproducing bugs based on automated test logs with an interactive method. REPASSISTOR first transforms the automated test logs into an application transition linked list. Then, utilizing DL and traditional CV techniques, REPASSISTOR analyzes the screenshots of the automated test logs to compute their similarity. REPASSISTOR merges corresponding pages based on similarity and collapses the linked list into a graph. By utilizing the conversational agent, REPASSISTOR monitors the developers' positions, provides guidance, and assists them in reproducing bugs. We also conducted an experiment on REPASSISTOR, demonstrating its effectiveness in enhancing developers' behavior of bug reproduction.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by the National Natural Science Foundation of China (61932012, 62141215, 62372228) and the Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

[1] J. D. Herbsleb, "Global software engineering: The future of socio-technical coordination," in *Future of Software Engineering*. IEEE, 2007, pp. 188–198.

[2] V. Garousi and M. V. Mäntylä, "A systematic literature review of literature reviews in software testing," *Information and Software Technology*, vol. 80, pp. 195–216, 2016.

[3] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, 2002.

[4] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," in *Future of Software Engineering Proceedings*, 2014, pp. 117–132.

[5] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering (FOSE'07)*. IEEE, 2007, pp. 85–103.

[6] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *The Eighth International Symposium On Software Reliability Engineering*, 1997, pp. 264–274.

[7] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] M. Soltani, A. Panichella, and A. Van Deursen, "Search-based crash reproduction and its impact on debugging," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1294–1317, 2018.

[9] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 204–214.

[10] T. Yu, T. S. Zaman, and C. Wang, "DESCRY: reproducing system-level concurrency failures," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 694–704. [Online]. Available: https://doi.org/10.1145/3106237.3106266

[11] N. Chen and S. Kim, "STAR: stack trace based automatic crash reproduction via symbolic execution," *IEEE Trans. Software Eng.*, vol. 41, no. 2, pp. 198–220, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2363469

[12] J. Bach and M. Bolton, "Rapid software testing," *Version (1.3. 2), wwvv. satisficc. com*, 2007.

[13] P. C. Jorgensen, *Software testing: a craftsman's approach*. CRC press, 2018.

[14] Google, UI/Application Exerciser Monkey, 2023, accessed: April 14, 2023. [Online]. Available: https://developer.android.com/studio/test/monkey

[15] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Toward interactive bug reporting for (android app) end-users," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 344–356.

[16] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Fusion: A tool for facilitating and augmenting android bug reporting," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 609–612.

[17] "Rasa," https://github.com/RasaHQ/rasa, 2023.

[18] Microsoft, "Microsoft conversational bot architecture," https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/ai/conversational-bot, 2023.

[19] Appium, 2023. [Online]. Available: http://appium.io/docs/en/2.0

[20] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 128–139.
[21] S. Singh, R. Gadgil, and A. Chudgor, "Automated testing of mobile applications using scripting technique: A study on appium," *International Journal of Current Engineering and Technology*, vol. 4, no. 5, pp. 3627–3630, 2014.
[22] F. Okezie, I. Odun-Ayo, and S. Bogle, "A critical analysis of software testing tools," in *Journal of Physics: Conference Series*, vol. 1378, no. 4, 2019, p. 042030.
[23] Google, "Android debug bridge," https://developer.android.com/studio/command-line/adb, 2023.
[24] J. Canny, "A computational approach to edge detection," *IEEE Transactions on pattern analysis and machine intelligence*, no. 6, pp. 679–698, 1986.
[25] S. Yu, C. Fang, Z. Cao, X. Wang, T. Li, and Z. Chen, "Prioritize crowdsourced test reports via deep screenshot understanding," in *IEEE/ACM 43rd International Conference on Software Engineering*, 2021, pp. 946–956.
[26] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st ACM international conference on Information and knowledge management*, 2012, pp. 852–861.
[27] J.-C. Yoo and T. H. Han, "Fast normalized cross-correlation," *Circuits, systems and signal processing*, vol. 28, pp. 819–843, 2009.
[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[29] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering*. IEEE, 2012, pp. 102–112.
[30] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye, and T. Xie, "Performance issue diagnosis for online service systems," in *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 2012, pp. 273–278.
[31] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain, "Content-based image retrieval at the end of the early years," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 12, pp. 1349–1380.
[32] C.-T. Lin, S.-P. Ma, and Y.-W. Huang, "Msabot: A chatbot framework for assisting in the development and operation of microservice-based systems," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 36–40.
[33] C. Tony, M. Balasubramanian, N. E. Díaz Ferreyra, and R. Scandariato, "Conversational devbots for secure programming: An empirical study on skf chatbot," in *International Conference on Evaluation and Assessment in Software Engineering*, 2022, pp. 276–281.
[34] N. Assavakamhaenghan, R. G. Kula, and K. Matsumoto, "Interactive chatbots for software engineering: A case study of code reviewer recommendation," in *2021 IEEE/ACIS 22nd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. IEEE, 2021, pp. 262–266.
[35] T. W. Bickmore and R. W. Picard, "Establishing and maintaining long-term human-computer relationships," *ACM Transactions on Computer-Human Interaction*, vol. 12, no. 2, pp. 293–327, 2005.
[36] M.-H. Huang and R. T. Rust, "Artificial intelligence in service," *Journal of service research*, vol. 21, no. 2, pp. 155–172, 2018.
[37] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 141–152.
[38] M. Soltani, A. Panichella, and A. Van Deursen, "A guided genetic algorithm for automated crash reproduction," in *2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 2017, pp. 209–220.
[39] Y. Cao, H. Zhang, and S. Ding, "Symcrash: Selective recording for reproducing crashes," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 791–802.
[40] S. Feng and C. Chen, "Gifdroid: automated replay of visual bug reports for android apps," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1045–1057.
[41] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.
[42] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.
[43] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.
[44] Y.-M. Baek and D.-H. Bae, "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*, 2016, pp. 238–249.
[45] J. Wang, M. Li, S. Wang, T. Menzies, and Q. Wang, "Images don't lie: Duplicate crowdtesting reports detection with screenshot information," *Information and Software Technology*, vol. 110, pp. 139–155, 2019.
[46] T. A. Nguyen and C. Csallner, "Reverse engineering mobile application user interfaces with remaui (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 248–259.