Science of Computer Programming 201 (2021) 102522

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Efficient testing of GUI applications by event sequence reduction

Jiwei Yan[a,c], Hao Zhou[a,c], Xi Deng[b,c], Ping Wang[b], Rongjie Yan[b], Jun Yan[a,b,c,*], Jian Zhang[b,c,**]

[a] *Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, China*
[b] *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China*
[c] *University of Chinese Academy of Sciences, China*

ARTICLE INFO

Article history:
Received 21 November 2019
Received in revised form 14 July 2020
Accepted 20 July 2020
Available online 12 August 2020

Keywords:
GUI testing
Test reduction
Record-and-replay
Android application
Monkey

ABSTRACT

Automatic event sequence generation tools are widely used for testing GUI applications. With these tools, developers can easily test the target GUI applications with a large number of events and collect a group of crash-triggering sequences in a short time. However, some efficiency-oriented tools generate low-level events randomly based on coordinates of the screen instead of widgets, which leads to many ineffective events that have no contribution to the test. Besides, the randomly generated sequences may repeatedly operate on the same widget or jump to the same window, which increases the complexity of sequences and makes it difficult to extract key events that can lead to crashes.

The sequence reduction technique can effectively help developers to understand the crashes and further improve the quality of code. In this paper, we propose a general model for the *event sequence reduction problem* on GUI applications. For better illustration, we take the random test generation tool Monkey as a concrete instance, which is widely used for testing Android applications, owing to its simplicity, effectiveness and good compatibility. To address the major drawbacks in original Monkey testing, in this paper, we attempt to enhance Monkey to support the sequence record-and-replay and propose a sequence reduction approach for Android apps, which helps the crash behavior comprehension and fault localization. By manually investigating the effectiveness of Monkey events, we find three types of ineffective events, including no-ops, single and combination of effect-free ones, and design nine reduction rules for them. To extract key events in one sequence for crash understanding, we analyze the state transition relation among events and propose a static GUI state hierarchy-tree-guided reduction approach. Additionally, we implement our approach in a tool CHARD to achieve event sequence reduction on real-world apps. We also design a semi-structured format to describe the actual behavior of events and improve the sequence comprehensibility.

We collect 890 sequences from 74 applications as our benchmark, including 740 basic sequences, each of which contains 1,000 events, and 150 longer ones, each of which contains 10,000 events. CHARD can quickly identify 41.3% events as ineffective ones in the collected sequences. For sequences that can be stably replayed, over 94% of the reduced sequences keep the same functionalities as the original ones. By removing ineffective events, CHARD can be used as a pre-process part of the traditional delta-debugging process and make significant speed up. To evaluate the effectiveness of the key event extraction approach, we pick eight buggy applications and collect 40 crash-triggering event sequences

* Corresponding author at: Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, China.
** Corresponding author at: State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China.
E-mail addresses: yanjun@ios.ac.cn (J. Yan), zj@ios.ac.cn (J. Zhang).

https://doi.org/10.1016/j.scico.2020.102522

generated by Monkey, the length of which varies from 19 to 2700. The results show that CHARD can successfully remove over 95.4% crash-irrelevant events in these crash-triggering sequences within around ten seconds, while the state-of-the-art delta-debugging tool removes 71.3% ones using over 27 hours, which indicates that CHARD can efficiently help the crash replay and sequence comprehension.

1. Introduction

Graphical user interface (GUI) applications are becoming increasingly prevalent nowadays, due to the widespread use of desktop computers, webs, smartphones, tablets, gaming devices, car consoles, etc. Most popular GUI applications tend to be feature-rich, which puts forward higher requirements for testing. However, due to some characteristics of GUI applications, such as the fragmentation of supported devices and the infinite event combinations, it is exhausting to test them manually. GUI applications are event-driven programs, they allow users to perform interactions with them through the manipulation of the graphical elements. Recently, heterogeneous automated testing tools for GUI applications have been developed by practitioners and researchers, which aim at generating high-quality test sequences to explore the state space of the applications.

However, there are three major limitations in existing GUI test generation tools: (1) some efficiency-oriented tools generate low-level events that are hardly human-readable, which makes developers exhausted in inspecting the randomly generated test sequence; (2) there are a large number of ineffective events in the original generated test sequences that do not influence the application behavior; (3) the randomly generated sequences may repeatedly operate on the same widget or jump to the same window, which increases the complexity of sequences and makes it difficult to extract key events. Thus, to simplify the original sequences and perform efficient testing, the low-level events should be better modeled, both the ineffective events and duplicate events should be simplified, and user-friendly reports should be generated to help the understanding of test sequences behaviors.

The basic idea to simplify an event sequence is to set a target in advance and reduce target-irrelevant events using various techniques. The dynamic reduction algorithm Delta-debugging (DD) [1] uses an iterative process to execute the reduction process repeatedly until obtaining a minimal subsequence. DD is a widely used technique and its variants can be used to adapt to the characteristics of different GUI platforms [2,3]. Nonetheless, DD-based techniques are very time-consuming since they usually require dynamically re-executing the reduced sequences in the iteration process. For example, in paper [2], the sequences of 500 events take around 24 hours to reduce on average.

In this paper, we propose an efficient sequence reduction approach based on the detection of event effectiveness. We concentrate on the following aspects: 1) what is the taxonomy and pattern of ineffective events; 2) what are the characteristics of the crash-triggering irrelevant events; 3) how to guarantee the correctness after sequence reduction; 4) how to improve the readability of sequences to help testers localize bugs. Among all of the GUI applications, Android applications are the most representative ones. The number of available Android applications in Google Play Store [4] reached 2.87 million in December 2019, after surpassing one million in July 2013 [5]. On Android platform, there are a great many automatic testing tools, using various strategies. According to the thorough empirical study [6] by Choudhary et al., the light-weight test generation tool Monkey [7] outperforms other complex techniques in the aspects of ease of usability, compatibility, code coverage, and fault detection ability. Therefore, we pick the combination of Android and Monkey as a concrete instance to illustrate the effectiveness of our sequence reduction approach.

First, we conduct an empirical study on 200 real sequences, each of which has 1,000 events, from 40 popular open-source applications. For these sequences, we manually infer the effectiveness of events according to the runtime information, such as the generated method logs and the change of activity or window before and after an event execution. After investigation, we summarize nine reduction rules of ineffective events, including *No-ops*, *Single* and *Combination of effect-free ones*. For effect-free combination, we use extended finite state machines (EFSMs) to conduct the reduction. Especially, for the nested combinations, we add a stack to the automaton to address them. Besides, we make a specific reduction for crash-triggering sequences because of their importance in testing. In crash-triggering sequences, many events are irrelevant to crashes and can be removed, for example, duplicate transitions to the same state can be merged. The large number of duplicate crash-irrelevant events hinders the key-event extraction process. However, directly removing one event may influence the execution of its neighbor events and cause the failure of sequence replay. Therefore, we build the GUI state hierarchy-tree to help understand the behavior of sequences and propose a tree-guided static reduction approach with three strategies: Short, Duplicate and Ineffective.

We implement our reduction approaches in a tool CHARD (CompreHensibility And ReDuction), which accepts original Monkey sequences, and outputs a simplified sequence as well as a user-friendly bug report that describes the behavior of the corresponding sequence. CHARD is evaluated with two datasets, of which the first contains 890 normal test sequences from 74 Android apps, and the other contains 40 crash-triggering test sequences from eight buggy apps. By ineffective event reduction on the first dataset, CHARD can successfully remove 41.3% ineffective events, and significantly speed up the traditional DD technique by pre-processing the input of it. On the second dataset, CHARD can remove over 95.4% events in the collected crash-triggering sequences within around ten seconds, while the state-of-the-art delta-debugging tool removes 71.3% ones using over 27 hours. In our experiments, all the reduced sequences keep the same functionalities as the original ones, i.e., can trigger the same crash. The evaluation results show that CHARD can reduce test sequences effectively, correctly and efficiently.

The main contributions of this work are summarized as follows:

- We propose a general model for reducing GUI event sequences and pick the combination of Android and Monkey as a concrete instance;
- We design nine reduction rules to shorten the Monkey sequence by removing ineffective events;
- We propose a hierarchy-tree guided reduction approach and three strategies to remove crash-irrelevant events in crash-triggering sequences;
- We design a semi-structured format to describe the behavior of event sequences;
- We implement the proposed techniques in a prototype tool CHARD and evaluate its effectiveness, correctness, and efficiency on real-world applications.

The work presented in this paper is based on our previous work [8] and has been extended significantly. In the previous work, we only give the model of Monkey sequence reduction on Android GUI system, while in this journal version, we generalize our model to the event sequence reduction on GUI system and illustrate it using a concrete instance. Previously, we study how to identify and reduce ineffective events in Monkey sequence, including both the normal and crash-triggering sequences, while keeping the same behavior. Considering the importance of the crash-triggering Monkey sequence for bug detection, in this journal version, we further study the problem of how to identify and remove the crash-irrelevant events in crash-triggering sequences while keeping the same crashes being replayed. In this work, we classify crash-triggering sequences into three types and propose a key-event-directed event reduction approach with corresponding strategies. We build a hierarchy-tree for each given sequence and make sequence reduction based on the tree. We also update the tool CHARD to extract key events in a crash-triggering sequence using three strategies. Our evaluation demonstrates the effectiveness of our extensions, which achieves a similar reduction rate compared with the state-of-the-art tool while using extremely less time than it. We introduce a new motivating example for clear illustration throughout the paper, reorganize the problem definition and add more experiments. Moreover, we add the discussion about the implementation and deployment cost as well as the threat to the validity of our approach.

The remainder of this paper is organized as follows. In Section 2, we use a motivating example to show the original Monkey testing sequences as well as the necessity to make reduction and comprehension. Then we give problem definitions in Section 3. The ineffective-event-directed and the key-event-directed reduction approaches will be introduced in Section 4 and 5, respectively. We also show the comprehension of sequence and give the event behavior description format in Section 6. The evaluation of our tool CHARD is demonstrated in Section 7, and the threat to the validity is introduced in Section 8. In the last two sections, we discuss related works and conclude our work.

2. Motivating example

We use a motivating example to show the original Monkey testing sequences and the importance of making test reduction. `WhoHasMyStuff` [9] is an application that helps people to keep track of their lent things. Fig. 1 gives four screens of a mutated `WhoHasMyStuff` and displays a set of window transitions to trigger crash. In this example, we only show the four events that are essential to the triggering of the application crash. Note that in the first window, we can tap on either displayed item (Event 1.1 or 1.2) to trigger the crash. However, the real crash-triggering sequences generated by Monkey contain many ineffective or crash-irrelevant events, which increases the difficulty of manual reviewing.

One crash triggering sequence of Monkey is listed in Fig. 2, which contains 147 Monkey events and is poorly human-readable. The first event is a `switch` event aiming to launch the main activity of the application under test. In this example, 800 ms is the intervening time, which is the natural boundary in the sequence and separates low-level events into different groups. Each group of low-level events corresponds to a high-level event. For example, the combination of an `ACTION_DOWN` and an `ACTION_UP` Monkey event denotes a `tap` event. In this sequence, the original events can be translated into 74 user-level events based on the natural boundaries (sleeping time gap). If both the coordinates of the down event and up event point to the same widget, we can find the target object of that event pair. However, in this original sequence, many events operate on the blank area or ineffective widgets, which is useless to the test. This original sequence also opens the same window and closes it repeatedly as well as operates on the same widget many times (refer to Fig. 11). With the original long event sequence, it is difficult to find out a concise event sequence that can replay the same crash and reveal the key reason for the application crash. To achieve the efficient testing of GUI applications, both the ineffective events and the crash-irrelevant events should be removed from the original sequence. Besides, a user-readable test sequence, which contains the detailed information of each event, should be reported for effective debugging.

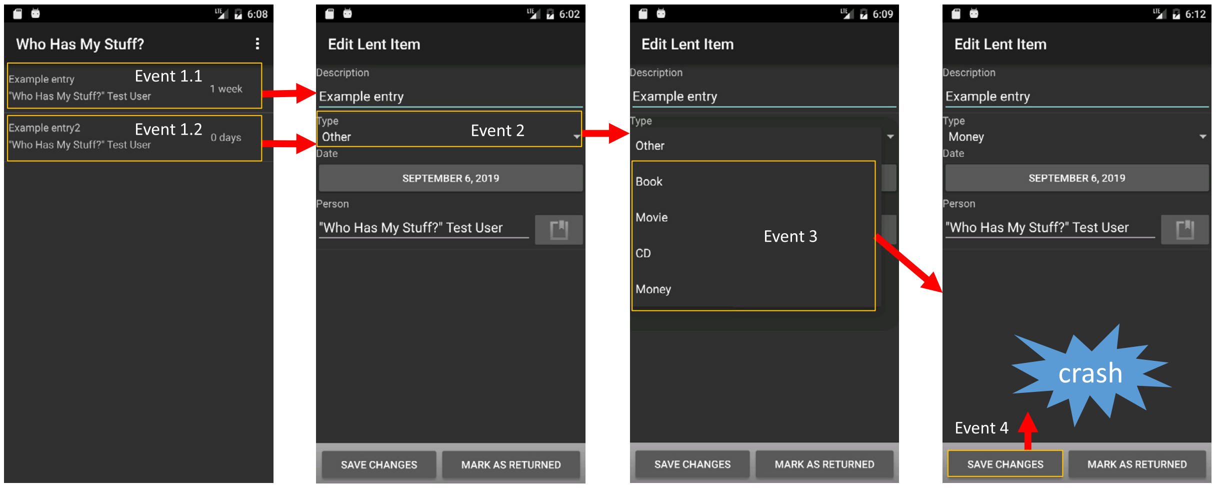
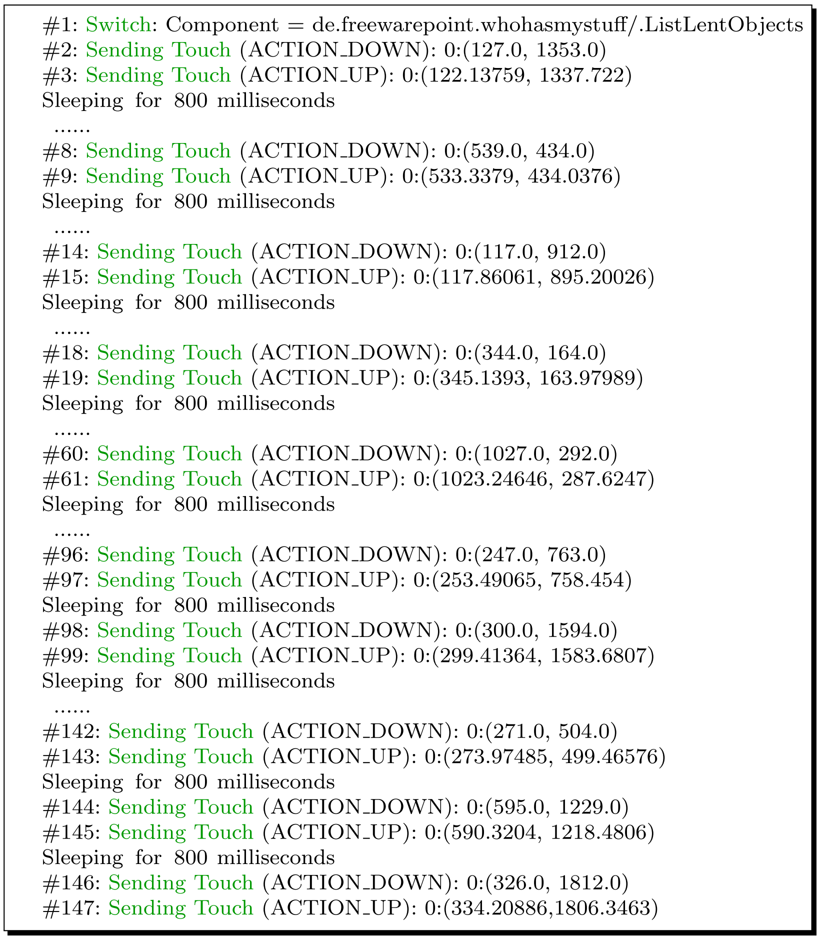


Fig. 1. Screens of App `WhoHasMyStuff`.



```
#1: Switch: Component = de.freewarepoint.whohasmystuff/.ListLentObjects
#2: Sending Touch (ACTION_DOWN): 0:(127.0, 1353.0)
#3: Sending Touch (ACTION_UP): 0:(122.13759, 1337.722)
Sleeping for 800 milliseconds
 ......
#8: Sending Touch (ACTION_DOWN): 0:(539.0, 434.0)
#9: Sending Touch (ACTION_UP): 0:(533.3379, 434.0376)
Sleeping for 800 milliseconds
 ......
#14: Sending Touch (ACTION_DOWN): 0:(117.0, 912.0)
#15: Sending Touch (ACTION_UP): 0:(117.86061, 895.20026)
Sleeping for 800 milliseconds
 ......
#18: Sending Touch (ACTION_DOWN): 0:(344.0, 164.0)
#19: Sending Touch (ACTION_UP): 0:(345.1393, 163.97989)
Sleeping for 800 milliseconds
 ......
#60: Sending Touch (ACTION_DOWN): 0:(1027.0, 292.0)
#61: Sending Touch (ACTION_UP): 0:(1023.24646, 287.6247)
Sleeping for 800 milliseconds
 ......
#96: Sending Touch (ACTION_DOWN): 0:(247.0, 763.0)
#97: Sending Touch (ACTION_UP): 0:(253.49065, 758.454)
Sleeping for 800 milliseconds
#98: Sending Touch (ACTION_DOWN): 0:(300.0, 1594.0)
#99: Sending Touch (ACTION_UP): 0:(299.41364, 1583.6807)
Sleeping for 800 milliseconds
 ......
#142: Sending Touch (ACTION_DOWN): 0:(271.0, 504.0)
#143: Sending Touch (ACTION_UP): 0:(273.97485, 499.46576)
Sleeping for 800 milliseconds
#144: Sending Touch (ACTION_DOWN): 0:(595.0, 1229.0)
#145: Sending Touch (ACTION_UP): 0:(590.3204, 1218.4806)
Sleeping for 800 milliseconds
#146: Sending Touch (ACTION_DOWN): 0:(326.0, 1812.0)
#147: Sending Touch (ACTION_UP): 0:(334.20886,1806.3463)
```

Fig. 2. A Monkey sequence on App `WhoHasMyStuff` (147 Monkey events, 74 atom events).

3. Problem definition

In this section, we first introduce the general event sequence reduction model of GUI systems and then instantiate the event sequence reduction problem on the *Android* system and the test generation tool *Monkey*.

3.1. General problem

Because of the event-driven characteristic of GUI applications, the interactions between the users and a specific application form a sequence of user events. In this paper, we omit the interaction among multiple GUI applications and regard each GUI application as well as its runtime environment as a single GUI system.

Definition 3.1 *(Graphical user interface system, GUI system).* A GUI system $\mathcal{G}$ is a 6-tuple $\langle app, GW, GIS, GO, GI, ges\rangle$, where

- app is the identifier of a GUI application;
- GW is a set of windows. Each window $gw \in GW$ is a triple $\langle id, WGT, gwl\rangle$, in which id is the identifier of gw, WGT contains a set of GUI widgets displayed on gw, and gwl denotes the layout structure of widgets in WGT. Each element $wgt \in WGT$ denotes a widget with a group of attributes, including type, location, visibility, supported operations, etc.;
- GIS is a set of internal statuses. Each gis indicates the abstraction of an internal status, which could be fine-grained or coarse-grained, e.g., the assignments of all variables, or the trace of the executed methods;
- GO is a set of supported user operations, e.g., click, press, longclick etc.;
- GI is a set of interactions. Each $gi \in GI$ is a 6-tuple $\langle gw_1, gis_1, go, wgt, gw_2, gis_2\rangle$, where $gw_1, gw_2 \in GW$, $gis_1, gis_2 \in GIS$, $go \in GO$ and $wgt \in gw_1.WGT$. When the operation go is performed on widget wgt, the GUI window transits from gw_1 to gw_2 and the internal status transforms from gis_1 to gis_2;
- ges indicates a specific external environment that the app runs in, which includes the platform version, screen size, and type of the device. For a given GUI application app, different ges may lead to different window set GW.

In the above definition, the user interactions can make the combination of a window and an internal status (gw_1, gis_1) transfer to another combination (gw_2, gis_2). We formalize these combinations as application statuses in Definition 3.2.

Definition 3.2 *(Application status).* An application status as in GUI System γ can be described as a pair (gw, gis), where $gw \in \gamma.GW$ is the window information and $gis \in \gamma.GIS$ denotes the internal status.

The user interaction can change an application status from source $src = (gw_1, gis_1)$ to destination $des = (gw_2, gis_2)$. If $gw_1 = gw_2$ and $gis_1 = gis_2$, we can take the application statuses src and des as equivalent ones. Then, the interactions can be separated into two types: the *transition interaction* and the *normal interaction*, where the transition interaction leads to a status transition ($src \neq des$), and the normal one keeps the application status ($src = des$).

To test these GUI systems, input generation tools generate events to interact with the GUI system and traverse its status space by adopting different event generation strategies. There are two classes of events: the high-level user events and the low-level executable events. The high-level user event is similar to user's real operation, for example, click the button whose text is "`submit`" or whose id is "`mButton`", which requires to obtain the layout of windows and parses widgets on-the-fly. Each high-level user event is equivalent to a user interaction in its GUI system. The definitions of *High-level User Event* and *User Event Sequence* are given as follows:

Definition 3.3 *(High-level user event).* A high-level user event E on a GUI system γ is a 6-tuple, $E(\gamma) = \langle op, obj, src, des, chg_w, chg_i\rangle$, where

- op denotes the operation type of interaction, $op \in \gamma.GO$;
- obj is the unique identity of the object, which may be the identity of a GUI widget $wgt \in src.gw.WGT$ or a system key value, to be operated. Note that obj is null if the event hits no widget or the key value is invalid;
- src and des are the application statuses before and after the operation is performed, each of which can be represented as a pair $\langle gw, gis\rangle$, where $gw \in \gamma.GW$ and $gs \in \gamma.GIS$;
- chg_w is a boolean variable, which denotes changes of windows of γ after executing op on obj. If $src.gw \neq des.gw$, we have chg_w = True, or else, chg_w = False;
- chg_i is a boolean variable, which denotes changes of internal statuses of γ after executing op on obj. If $src.gis \neq des.gis$, we have chg_i = True, or else, chg_i = False.

Definition 3.4 *(User event sequence).* A user event sequence

$$\mathcal{U}(\gamma) = E(\gamma)_1 \cdots E(\gamma)_m \quad (m > 0)$$

for a GUI system γ is a sequence composed of m separated user events, which will be executed in order. In sequence $\mathcal{U}(\gamma)$, each user event is a user-level atomic operation that can not be split. The *src* of the first event $E(\gamma)_1$ in $\mathcal{U}(\gamma)$ is the initial application status before the application is launched, and the *des* of $E(\gamma)_i$ equals to the *src* of its next event, i.e., $E(\gamma)_i.des = E(\gamma)_{i+1}.src$.

The low-level executable events do not bind to a specific widget, instead, they relate to the coordinates on the screen, i.e., sequences do not need to bind with a GUI system during the generation phase. Therefore, the test sequences composing low-level executable events can be generated offline. We then give the definitions of *Low-level Executable Event* and *Executable Event Sequence*.

Definition 3.5 *(Low-level executable event).* A low-level executable event e is a type of event that describes movement in terms of a basic action code and a set of axis values (c, V):

- the basic action code c specifies the state change that occurs, such as a pointer going down or up;
- the axis values in set V describe the position and the movement properties.

Definition 3.6 *(Executable event sequence).* An executable event sequence $\mathcal{E}$ consisting n low-level executable events can be formalized as:

$$\mathcal{E} = e_1 \cdots e_n \quad (n > 0)$$

During the record-and-replay, each sequence will be executed multiple times. In this occasion, an executable event sequence $\mathcal{E}$ should bind to the application under test. Unlike high-level user event, a single low-level executable event may not relate to a widget. Thus, we combine these low-level events into groups, each of which corresponds to an atom user event. After executing $\mathcal{E}$ on a specific GUI system γ, we can analyze the points-to relationship between the event combinations and GUI widgets and bind them. Finally, the sequence $\mathcal{E}$ is turned to be a transformed event sequence $\mathcal{U}_E(\gamma)$. The definition of transformed event sequence is shown as follows, in which the function *bind*(γ, events) binds a combination of low-level executable events with a GUI widget in the GUI system γ and transforms the event combination to a user event.

Definition 3.7 *(Transformed event sequence).* A transformed event sequence is combined by k high-level user events, which are transformed from n low-level events in the executable event sequence $\mathcal{E}$, according to the dynamic execution information of $\mathcal{E}$ after executing on a GUI system γ.

$$\mathcal{U}_E(\gamma) = bind(\gamma, e_1 \cdots e_i)^1 \cdots bind(\gamma, e_j \cdots e_n)^k = E(\gamma)_1 \cdots E(\gamma)_k \quad (1 \le i < j \le n, 0 < k \le n)$$

During the investigation, we find that the transformed user event sequence contains a large number of ineffective events and some events with the same functionality are executed repeatedly, which increases the test efforts. Therefore, our work aims to solve the **Event Sequence Reduction Problem** for an efficient testing. We use the symbol $\rightarrow$ to denote the transformation or reduction process of event sequences, and use the function *Reduced*() to represent the event sequence reduction approach. If the test generation tool generates high-level user events, we make reduction $\mathcal{U}(\gamma) \rightarrow Reduced(\mathcal{U}(\gamma))$. If the test generation tool generates low-level executable events, we make the following reduction: $\mathcal{E} \rightarrow \mathcal{U}_E(\gamma) \rightarrow Reduced(\mathcal{U}_E(\gamma))$.

3.2. The instantiation on Android and Monkey

In this paper, we consider an instantiation of the event sequence reduction problem under a specific domain: the *Android* system as well as the test generation tool *Monkey*, for a clear illustration. The Android GUI system is an instance of the GUI System. In an Android GUI system, we use the package name of the Android app to denote the *app* information. Each Android application is composed of multiple components, in which activity is the key component that displays a UI interface that users can interact with. Therefore, each window relates to a specific activity, and we use that activity name to represent the window id. The layout structure of widgets is a tree, which is built using a hierarchy of `View` and `ViewGroup` objects, in which a `View` corresponds to a GUI widget, whereas a `ViewGroup` is a container that defines the layout structure for `View` and other `ViewGroup` objects. The operation set in the Android GUI system is the set of user operations supported by the Android framework.

In the `WhoHasMyStuff` application (refer to Section 2), for instance, each window in Fig. 1 is composed of the name of activity component, the layout result returned by the "`adb shell uiautomator dump`" command and the widget set obtained by enumerating all the widgets in the layout file. The internal status indicates an executed trace of Android callback methods, which can be obtained by method-level instrumentation. The operation set is a group of user operations that could be operated on the available widgets in this app. The set of interactions contains all the Android window transitions, e.g., the window transfer from the first screen to the second screen after Event 1.1. And the external status indicates environment that the application under test runs in, including the platform version, screen size, and type of the experimental Android device.

Android testers usually send user-level events to check the functional correctness of the target app. However, for the most widely used testing tool Monkey [7], the generated test cases are low-level executable event sequences. For the `WhoHas-MyStuff` application (refer to Section 2), an example of a low-level executable event is shown in Fig. 2. There are totally 22 types of Monkey events, which are declared in the `MotionEvent` class [10] in Android SDK. The Monkey event focuses on the movement of action. However, most of these events can not be executed separately. For example, it is impossible to execute an `ACTION_DOWN` event alone but leave out the `ACTION_UP` event. Indeed, these random test generation tools, like Monkey, do not generate low-level events directly. They usually target high-level events, which are atomic ones but do not bind to any widget, in the event generation process and then transform them into low-level ones for the convenience of event execution. Thus, we can summarize the transformation rules defined in the source code of Monkey and automatically restore each combination of low-level events to a high-level one. For Monkey event sequence $\mathcal{M}$, we can extract the n low-level Monkey events into k event combinations based on the natural boundaries (sleeping time gap). For example, in Fig. 2, the event with id #8 and #9 could be combined to an atom user event `Tap`(539.0, 434.0, 0). Besides, a user event sequence should bind to a target application for reduction and replaying. During the execution of $\mathcal{M}$ on a specific Android GUI system γ, we collect the activity and window information of each event. Their points-to relationship can be calculated after the execution. Then the atom event `Tap`(539.0, 434.0, 0) will be transformed to a high-level user event, i.e., #8,9 is a touch event that clicks on `ListView` with the id “list” and text “Add Entries” on the `ListLentObjects` activity, causing the launch of `AddObject` activity. Finally, we can turn sequence $\mathcal{M}$ to a transformed Monkey event sequence $\mathcal{U}_M(\gamma)$.

Considering that Monkey generates low-level executable events, the **Monkey Sequence Reduction Problem** can be defined as $\mathcal{M} \to \mathcal{U}_M(\gamma) \to \mathit{Reduced}(\mathcal{U}_M(\gamma))$. In the following reduction procedure, for non-crash-triggering sequences, we try to delete ineffective events; for crash-triggering sequences, we aim to remove the crash-irrelevant events. The key challenges are how to identify ineffective events and crash-irrelevant events correctly, which will be addressed in the following two sections.

4. Ineffective-event-directed reduction

The ineffective event can be a single event or a combination of multiple events, the execution of which does not change the original application status. We define the ineffective-event-directed reduction problem as follows:

Definition 4.1 *(Ineffective-event-directed reduction, IEDR).* The IEDR aims to optimize the user event sequence $\mathcal{U}_M(\gamma)$ into a refined one. We use symbol $\mathcal{U}_M^E(\gamma)$ to denote the reduced sequence after removing ineffective events.

$$\mathcal{M} \to \mathcal{U}_M(\gamma) \to \mathcal{U}_M^E(\gamma)$$

4.1. Empirical study on ineffective events

In order to investigate the properties of the test sequences generated by Monkey, we conduct an empirical study on 40 apps, which are randomly collected from F-Droid [11]. To achieve the record-and-replay of Monkey sequences, we enhance Monkey into Monkey$_{RR}$ to automatically record scripts, which can be replayed. In this part, we use Monkey$_{RR}$ to generate test sequences for each application and reduce them manually. For each app, we generate five sequences, each of which contains 1,000 low-level Monkey events. Each sequence can be transformed into around 400 high-level user events. The sleeping time gap is set as 3 seconds for the convenience of manual observation, which takes around 20 minutes to record the execution of one sequence. The analysis for each sequence, including the recording, manual reduction, and correctness verifying process, takes around 1-2 hours, and the empirical study totally costs one person-month. After reduction, we successfully remove around 66% events in the original sequences on the collected apps.

According to our investigation, we find three types of ineffective events that should be removed, including no-ops, single effect-free, and combination effect-free events. The distribution of each type of event on single application is shown in Fig. 3. And Fig. 4 presents the distribution of the ratio of events on all applications using box-plots, in which the median is demonstrated by the solid line and the average value is labeled by the X mark symbol. We also summarize the types and examples of ineffective events and list them in Table 1.

4.1.1. No-ops events

A no-ops event E is an event that does not link to any widget or operates on a widget with an unacceptable type of event. According to the Android response mechanism, these events are not intercepted and the Android system will not respond to them. The approach to identify a no-ops event is to check whether the application’s status changes after the event execution. If the application status remains unchanged, the performed event will be defined as a no-ops event.

The ratio of no-ops events ranges from 1% to 80%, and covers about 35% events on average. Some applications have high ratio of no-ops events. For example, application `AKA` is an application that needs to download sources firstly, without which the operational area on the home page is limited. But the downloading operation is often refused by the non-responsive

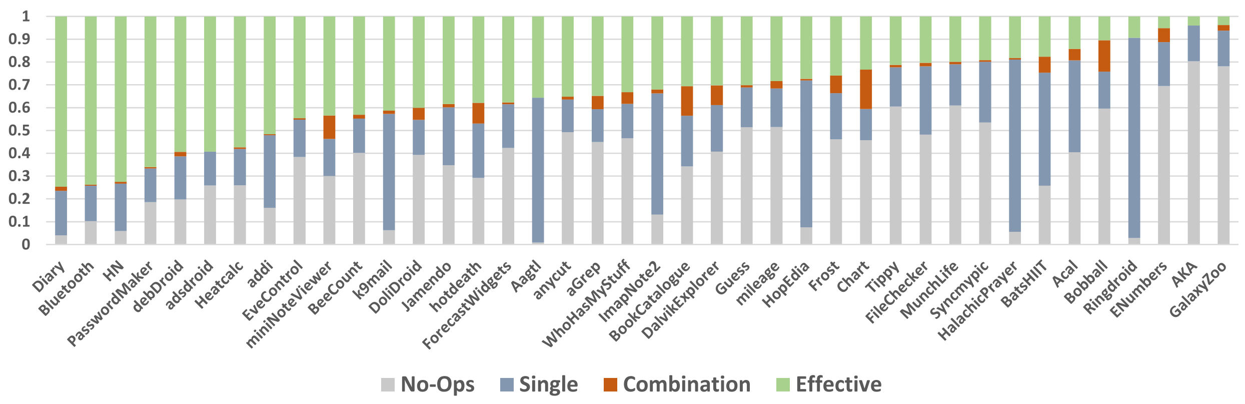


Fig. 3. The distribution of each type of event on a single application.

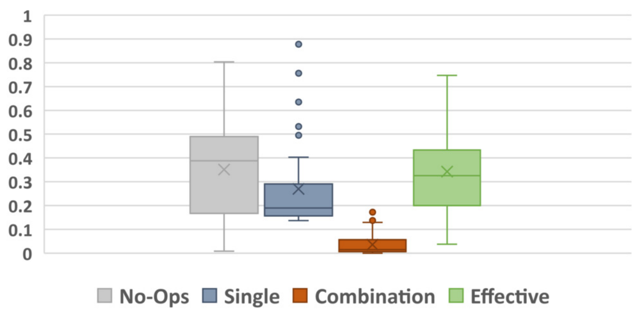


Fig. 4. The distribution of the ratio of events on all applications. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 1
Type and example of ineffective events.

Type	Name	Example
No-ops Event	No-ops Event	press on blank area without widget
Single Effect-free Event	Rejected Event	press `HOME` button press `BACK` button on the `MainActivity`
	Optional Event	press `VOLUME-UP` button, `VOLUME-DOWN` button or `CALL` button
Effect-free Combination	Contrary Pair	select one item of the checkbox and then disable it open an activity and go back without operation open a dialog window and cancel it without operation
	Last Effective	select an item in a group of radios, and select another immediately repeatedly click a text box with no input
	Text Slide	open a window of long text, slide the text without log and exit
	Hidden Menu	click window to open hidden menu bar, select one option to open a window, then close it

server, which causes the extremely low operational rate 19.6%. On the contrary, the number of no-ops events is small in some apps. For example, application `Aagtl` is a map application with a full-screen map, on which 99% events are operable.

4.1.2. Redundant events

After filtering out the no-ops events, we find that there are many other events that can be further reduced without sacrificing the effectiveness of the whole test sequence. We say one or more events $E(\gamma)_i \cdots E(\gamma)_j$ $(j \geq i)$ are redundant if the single event or the whole combination has no contribution to testing. The ratio of redundant events ranges from 15% to 88%, which covers about 31% events on average.

By investigation, we classify the redundant events into two categories according to the number of user events: single effect-free events $(i = j)$ and effect-free combinations $(i < j)$. If the execution of one user event is useless to the testing, this event can be seen as a single effect-free event. We find there are two types of single effect-free events: rejected event and optional event. A rejected event is an event that is abandoned in Monkey. To test a specific app, the test command should be configured with parameter `-p`, which only allows testing in the package of the application under test. Thus, activities out of the specific packages are rejected to be opened by Monkey. For example, pressing the `HOME` button on any page or triggering the `BACK` button on the `MainActivity` are both rejected events since they will lead to jumping to a target beyond the scope of testing. Optional events denote a set of configurable system events that are redundant to the application under test. For instance, the operations related to volume buttons are irrelevant for the target app `WhoHasmyStuff`, which can be reduced. For apps that overwrite the functions of the volume button, e.g., e-book uses volume buttons to turn pages, these operations can be seen as effective ones.

For a combination of multiple user events, if the last one counteracts the effect of the former ones, the whole combination can be seen as an effect-free combination, which is called “Contrary Pair”. For example, clicking on the check-box twice will rollback the application status. Even though any single event is effective, the combination of them is redundant. If the last event can replace the former ones, the combination except for the last event can be seen as an effect-free combination, which is called “Last Effective”. For example, if widgets in a radio group are selected multiple times, only the last one is effective for testing, so we only need to keep the last event. Sometimes, a combination of events leads to a window transition and then rollbacks its status. We call it the “Text Slide” pattern. For example, opening a window for text display, browsing it without internal status change, and then closing the window. Many apps hide the menu widget to avoid the complex and messy UI, which adds one more operation before selecting the menu. For random test generation tools, more logic-related steps mean more ineffective events. For example, clicking on the screen to open the menu display UI, and closing it immediately because of the failure in the menu item selection, we call it a “Hidden Menu” pattern.

4.2. Reduction rules for ineffective events

In this section, we define nine reduction rules for three types of ineffective events that need to be removed: no-ops events, single effect-free events, and effect-free combinations. Every pattern has a corresponding rule for reduction.

4.2.1. No-ops events

A No-ops event is a type of event that is not intercepted and handled by the Android system. We use the following pattern to identify no-ops events: $E \vDash obj = \texttt{null}, chg_w = \text{False}, chg_i = \text{False}$.

For each event in the sequence, we analyze its object, triggered methods and the change of application status. If one event does not link to any widget, does not trigger any callback methods in the application under test and does not lead to a state transition, we label it as no-ops one and remove it from the original sequence.

4.2.2. Single effect-free events

Single effect-free events include rejected events and optional events. For the two types of rejected events, we define two patterns to identify them. For optional events, one way to identify whether the events are effective is to enable users to configure them. The default patterns for single effect-free events are listed as follows:

- $E \vDash tp \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}, obj = \texttt{HOME}, chg_w = \text{False}$.
- $E \vDash tp \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}, obj = \texttt{BACK}, src.gw.id = \texttt{MainActivity}, chg_w = \text{False}$.
- $E \vDash tp \in \{\texttt{SysOp}, \texttt{Nav}, \texttt{MajorNav}\}, obj \in \{\texttt{CALL}, \texttt{VOLUME\_UP}, \texttt{VOLUME\_DOWN}\}, chg_w = \text{False}, chg_i = \text{False}$.

If an event satisfies any item in these patterns, it is a single effect-free event and will be removed.

4.2.3. Effect-free combinations

Effect-free combinations contain more than one effective single events. We use extended finite state machine (EFSM) [12] to describe the reduction rules for each type of the combinations in Table 1.

Definition 4.2 *(Extended finite state machine).* The extended finite state machine is a 6-tuple $M = \langle Q, q_0, \Sigma, V, C, \Lambda\rangle$, where

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- Σ is a finite set of operations;
- V is a set of variables;
- C is a finite set of conditions;
- Λ is a finite set of transitions. Each transition $t \in \Lambda$ is a 5-tuple $\langle q, q', o_1, c, O_2\rangle$, where $q \in Q$ and $q' \in Q$ are two states, $o_1 \in \Sigma$ is an operation and $c \in C$ is a condition. After operation o_1 is executed, if the condition c is satisfied, operations in set $O_2 \subseteq \Sigma$ will be executed. The transition can be represented as $q \xrightarrow{o_1[g]/O_2} q'$, in which all parts above the arrow are optional.

We use the five extended finite state machines in Figs. 5–9 to conduct reduction for four effect-free combinations. Taking the pattern *Contrary Pair* as an example, $Q = \{S_0, S_1, S_2\}$, $I = \{S_0\}$. We define three variables according to the progress of sequence reduction: $V = \{i, s, e\}$, in which variable i records the position of the event to be analyzed. Variables s and e denote two positions in the original sequence. Six operations are used in this EFSM: $\Sigma = \{$`init(x)`, `read(x)`, `move(x)`, `start(x)`, `end(y)` and `delete(x,y)`$\}$. Operation `init(x)` initializes the cursor variable $x = 0$, which points to the

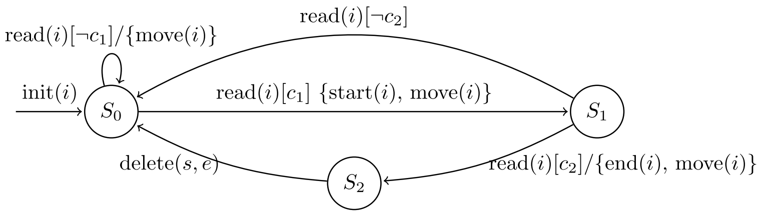

read(i)[$\neg c_2$]
read(i)[$\neg c_1$]/{move(i)}
init(i)
S_0
read(i)[c_1] {start(i), move(i)}
S_1
delete(s, e)
S_2
read(i)[c_2]/{end(i), move(i)}

Fig. 5. EFSM for contrary pair.

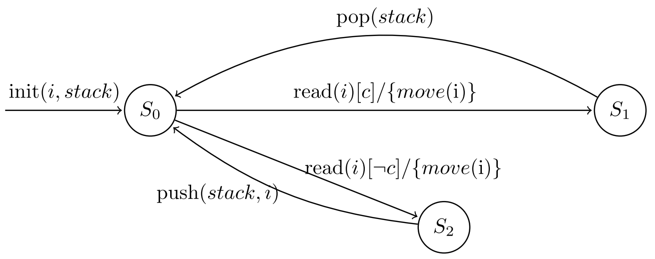


Fig. 6. EFSM for nested contrary pair.

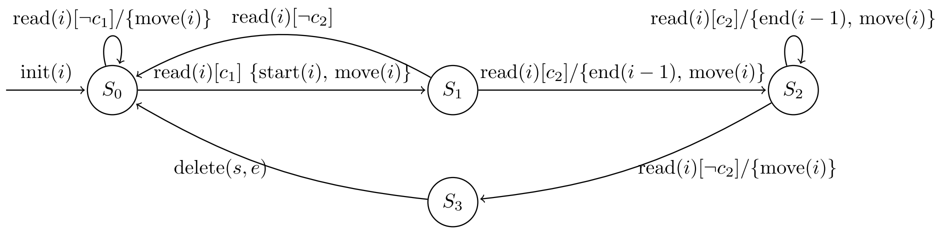

read(i)[$\neg c_1$]/{move(i)}
read(i)[$\neg c_2$]
read(i)[c_2]/{end($i-1$), move(i)}
init(i)
S_0
read(i)[c_1] {start(i), move(i)}
S_1
read(i)[c_2]/{end($i-1$), move(i)}
S_2
delete(s, e)
read(i)[$\neg c_2$]/{move(i)}
S_3

Fig. 7. EFSM for last effective.

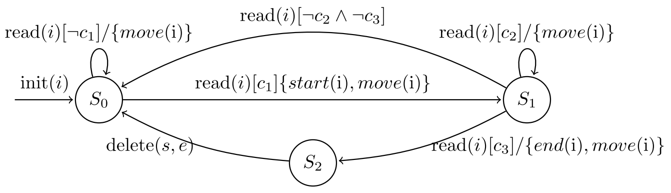

read(i)[$\neg c_2 \wedge \neg c_3$]
read(i)[$\neg c_1$]/{$move$(i)}
read(i)[c_2]/{$move$(i)}
init(i)
S_0
read(i)[c_1]{$start$(i), $move$(i)}
S_1
delete(s, e)
S_2
read(i)[c_3]/{end(i), $move$(i)}

Fig. 8. EFSM for text slide.

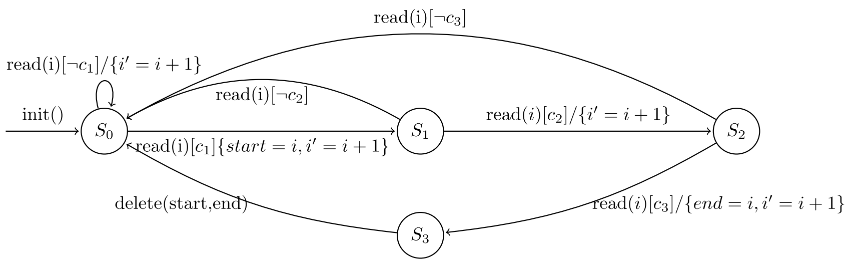

read(i)[$\neg c_3$]
read(i)[$\neg c_1$]/{$i' = i + 1$}
read(i)[$\neg c_2$]
init()
S_0
S_1
read(i)[c_2]/{$i' = i + 1$}
S_2
read(i)[c_1]{$start = i, i' = i + 1$}
delete(start,end)
read(i)[c_3]/{$end = i, i' = i + 1$}
S_3

Fig. 9. EFSM for hidden menu.

first event in the test sequence. Operation `read(x)` inputs the event E that is pointed by the cursor variable x to the EFSM. Operation `move(x)` increases the variable $x = x + 1$ to move the cursor to the next event. Operations `start(x)` and `end(y)` record the start and the end position, i.e., $s = x$, $e = y$. And operation `delete(x,y)` deletes the subsequence from event labeled x to event labeled y. There are two conditions c_1 and c_2, $C = \{c_1, c_2\}$, in which condition c_1 is satisfied if the current event meets the property of the first event in Contrary Pair, and c_2 is satisfied if the current event pairs with its former one. All the conditions for each property are kept in a collection in our tool implementation.

```
LaunchActivity(
de.freewarepoint.whohasmystuff,
de.freewarepoint.whohasmystuff.ListLentObjects) #1
Tap(539.0, 434.0, 0)   #8,9
UserWait(800)
Tap(117.0, 912.0, 0)   #14,15
UserWait(800)
Tap(344.0, 164.0, 0)   #18,19
 ......
Tap(326.0, 1812.0, 0)#146,147
UserWait(800)
```

Fig. 10. IEDR result on the motivating example (74 to 30 events).

In a few complex cases, the pattern Contrary Pair may be nested by itself. We use the idea of pushdown automata to deal with these cases. A pushdown automaton [13] is a finite state machine with an infinite stack. The stack helps to reduce the complexity of the algorithm to deal with the nested Contrary Pair. Thus, we declare a stack and define the operations of the stack on our extended finite state machine in Fig. 6. S_0 is the initial state. We first initiate variable $stack = \emptyset$ and $i = 0$. The condition c is satisfied if the top event of the stack pairs with the current event E, i.e., there is a contrary pair that should be removed. If condition c is satisfied, the EFSM will transfer to state S_1 and the top event in the stack will be popped up by operation `pop(stack)`. Otherwise, the EFSM will transfer to state S_2 and the current event with id i will be pushed into the stack by operation `push(stack, i)`. After running the automaton, the events in the stack are the remaining ones after removing the nested Contrary Pairs.

4.3. IEDR result of the motivating example

Fig. 10 gives the IEDR reduction result of the sequence in the motivating example, which is transformed into the format of Monkey script for the convenience of replaying. In this script, the first *LaunchActivity* event launches the application under test. The low-level executable events are combined, for example, the original events with id 8 and 9 are combined into a *Tap* event on the coordinate (539.0, 434.0). The sleeping time between each group of low-level executable events is transformed into a *UserWait* event. After removing ineffective events, the number of the atom events in the sequence is reduced from 74 to 30. For the motivating example, we can remove 59.5% events on average, and correctly replay the sequence after reduction.

5. Key-event-directed reduction

Software crashes represent unexpected program interrupts that manifest as software faults and can be dangerous in that their frequent occurrence diminishes user experience, can damage the reputation of a company, and potentially cause significant losses to stakeholders [14]. Therefore, among all sequences, the crash-triggering ones are of great importance for fault localization. When a crash is detected by GUI testing tools, testers usually replay the corresponding sequence to locate the key events leading to the crash. After adopting IEDR to simplify a crash-triggering sequence, the remaining sequence is still too complex for fault localization. We observed that many events are irrelevant to the triggering of crashes, for example, duplicate transitions to the same state could be merged. To further reduce a crash-triggering sequence and extract the key events, we define the key-event-directed reduction problem as follows:

Definition 5.1 *(Key-event-directed reduction, KEDR).* The KEDR aims to optimize the user event sequence $\mathcal{U}_M(\gamma)$ to a refined one, we use symbol $\mathcal{U}_M^K(\gamma)$ to denote the reduced sequence after removing crash irrelevant events.

$$\mathcal{M} \to \mathcal{U}_M(\gamma) \to \mathcal{U}_M^K(\gamma)$$

5.1. Key events of crash-triggering sequence

Key events in a crash-triggering sequence contain three parts: the crash-triggering event, which is the last event in sequences if crashes can terminate the program execution; the data-dependent events related to the crash-triggering event, which changes data and leads to conditions of crashes being satisfied; the control-dependent events related to the crash-triggering event, which help to reach the activity/window that the crash-triggering and data-dependent events belong to. To extract the control-dependent events in crash-triggering sequences, it is necessary to understand the control-flow relationship among events. Inspired by the tree-based dynamic DD [3], we use the GUI state hierarchy-tree to model sequences and guide the reduction, which is constructed according to the execution order and the state transition relationships among events.

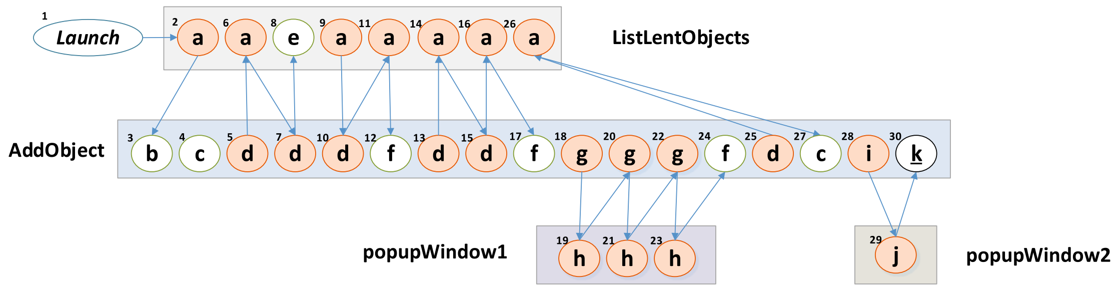


Fig. 11. GUI state hierarchy-tree of the sequence in the motivating example (30 events).

Definition 5.2 *(GUI state hierarchy-tree).* A GUI state hierarchy-tree of the user event sequence $\mathcal{U}$ is a triple $M = (E, N, T)$, where

- E is the set of user events in $\mathcal{U}$.
- N is a set of nodes in the tree. Each node $n \in N$ is a triple $\langle level, s, E_s \rangle$, where $level$ is the hierarchy level of a node, which is the shortest distance between the current node to the root node. s is a window status and E_s denotes a set of user events, where the source application status of each event in E_s is s, i.e., for each $e \in E_s$, $e.src.gw = s$.
- T is a set of transitions. For transition $\langle e_i, e_j \rangle \in T$, the execution of event $e_i \in E$ will lead to an activity transition to the node n, where $n.s = e_j.src.gw$, and e_j is the next event to be executed.

Fig. 11 gives the GUI State Hierarchy-Tree of the sequence in the motivating example, in which each rectangle is a node, representing a unique application status. Each circle in a node denotes an event. We use different colors to separate transition nodes and normal nodes. The text labeled beyond each state is the name of the corresponding application status, and the number beyond each event is the id of the event. We label equivalent events with the same character, i.e., all events labeled with "a" are the same. The lines represent transitions, e.g., the line from 7th event labeled with d in state `AddObject` to the 8th event labeled with e in `ListLentObjects` means the execution of 7th event will lead to a transition to state `ListLentObjects`, and the id of the next event to be executed is 8. This tree contains 30 user events, in which the 30th event labeled with $\underline{k}$ is the crash-triggering event.

The algorithm for tree construction is shown in Algorithm 1. In line 3, for each event in the given sequence, we get the corresponding node n by invoking method `getNodeFromTree` according to the application status it belongs to. If there is a node whose window status equals $e.s$ in the tree $\mathcal{H}$, that node will be returned, or else a new node is created. In lines 4-5, we calculate the level of n according to its distance to the entry node and append the current event into the event list of node n. In lines 7-10, we get the source and destination events to build a transition. Finally, the information about event e, node n and transition t will be updated to the hierarchy-tree $\mathcal{H}$.

Algorithm 1 Build hierarchy-tree.

```
Input: User Event Sequence $\mathcal{U}$
Output: Hierarchy-Tree $\mathcal{H}$
1: for $i$ : 1 to length($\mathcal{U}.events$) do
2:    Event $e$ = getEvent($\mathcal{U}$, $i$)
3:    Node $n$ = getNodeFromTree($\mathcal{H}, e.s$)
4:    $n.level$ = calculateDistance($\mathcal{H}, e.s$)
5:    $n.eventList$.append($e$)
6:    Transition $t = \varnothing$
7:    if $e$ is a transition event and $i + 1 \le$ length($\mathcal{U}$) then
8:       $t.source = e$
9:       $t.destination$ = getEvent($\mathcal{U}$, $i + 1$)
10:   end if
11:   $\mathcal{H}$.update($e, n, t$)
12: end for
```

5.2. Hierarchy-tree-based reduction

Crash-triggering sequences can also be categorized into three types according to the type of data-dependent events: 1) an empty set; 2) a set of non-repeating independent events; 3) others. Based on these types, we design three types of reduction strategies: *Short*, *Duplicate* and *Ineffective*. The crash-triggering event extraction process is shown in Algorithm 2. In line 1, we first build a hierarchy-tree t for the given sequence. If the data-dependent events do not exist, we only need to find out the events that lead to necessary GUI state transitions, i.e., the control-dependent events. Thus, we use the first *Short* reduction strategy to obtain the shortest path to reach the crash-triggering event. In lines 2-3, we get the last event

Algorithm 2 Crash-triggering event extraction algorithm.

```
Input: Crash-Triggering User Event Sequence $\mathcal{U}$
Output: Key Event Sequence $\mathcal{U}_s$
1: Tree $t$ = buildHierarchyTree($\mathcal{U}$)
2: Event $e$ = getLastEvent($\mathcal{U}$)
3: Sequence $\mathcal{U}_p$ = getShortestPath($t, e$)
4: if replay($\mathcal{U}_p$) = Success then
5:     $\mathcal{U}_s = \mathcal{U}_p$
6: else
7:     Tree $t_i$ = reduceIneffective($t$)
8:     $\mathcal{U}_i$ = transformTreetoSequence($t_i$)
9:     Tree $t_d$ = reduceDuplicate($t_i$)
10:    $\mathcal{U}_d$ = transformTreetoSequence($t_d$)
11: end if
12: $\mathcal{U}_s$ = ITE(replay($\mathcal{U}_d$) = Success, $\mathcal{U}_d$, $\mathcal{U}_i$)
13: return $\mathcal{U}_s$ and the corresponding bug report
```

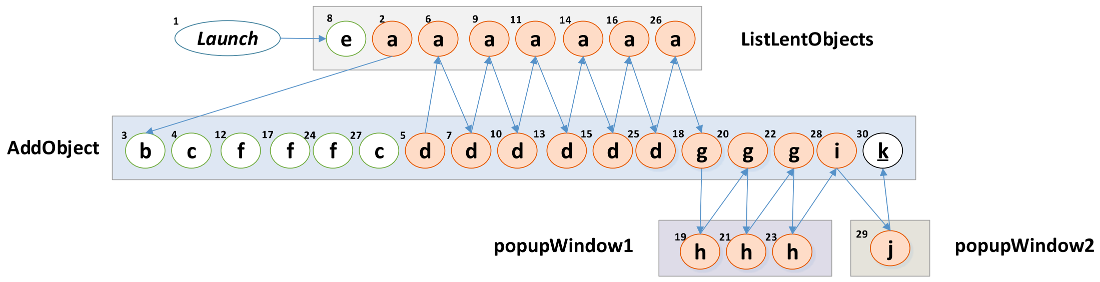


Fig. 12. Sorted GUI state hierarchy-tree of the sequence in motivating example (30 events).

in the given sequence and find the shortest path from the first event to reach it based on the tree t. If the data-dependent events are non-repeating independent events, we use the strategy *Duplicate* to remove both the ineffective and duplicate events to shorten the original sequence. Or else, we use the strategy *Ineffective*, which only identifies and removes ineffective events. The type of data-dependent events can not be detected before reduction, so we try these strategies in order and check whether the reduction success. In lines 4 and 12, we replay the shortest path $\mathcal{U}_p$ and detect whether the replay can trigger the same crash or not. Finally, we can get the key event sequence $\mathcal{U}_s$ and the corresponding bug report in the format mentioned in Section 6.

Based on the constructed hierarchy-tree, the relations among events are clear, so it is easy to find out the shortest path. And we can also remove the ineffective events by the approach mentioned in Section 4.2. The key challenge is how to make reduction using the *Duplicate* strategy, i.e., figure out duplicate events and remove them correctly. By comparing the equivalence among events, we can identify the duplicate ones. However, removing duplicate events in a straightforward way could lead to execution errors. During the manual record-reduce-replay process, we find that reducing different events affects the behavior of a sequence differently. If we directly delete an activity-launching event, the subsequent events will be triggered in the wrong activity. For example, if we remove event with id 9, which causes the transition from state `ListLentObjects` to state `AddObject`, the execution of event 10 will fail because of the change of state. To execute event 10, one way is to remove another transition from `AddObject` to `ListLentObjects`, e.g., event 7. But if we remove the event with id 7, the application status of event 8 also changes. The reduction must guarantee to keep the original execution context, i.e., application status, of each event correct.

To avoid the chain reaction due to event deletion, we first adopt a pre-process to sort the sequence, which keeps the structure of the original sequence. During sorting, we move the normal events in the front of the node it belongs to and cluster equivalent events together, while keeping the order of the last crash-triggering event. The sorted tree is displayed in Fig. 12. By sorting, we can remove transition events easier, e.g., deleting the pair $\langle 7, 9 \rangle$ directly. After that, we reduce the duplicate events from the root node to nodes at a deeper hierarchy level, until all nodes are analyzed.

The algorithm for reduction is shown in Algorithm 3. To remove duplicate events, we sort both the list of nodes in $\mathcal{H}$ according to their levels as well as the list of events in each node according to their types (normal or transition events) and equivalence. The equivalence can be obtained by comparing the attributes and behavior of events. We have several rules: 1) two normal events e_1, e_2 are the same if $e_1.src = e_2.src$, $e_1.tp = e_2.tp$ and $e_1.obj = e_2.obj$; 2) two transition events e_1, e_2 are the same if $e_1.src = e_2.src$, $e_1.des = e_2.des$ and $similarity(e_1.ms, e_2.ms) > \theta$, where the function $similarity$ calculates the ratio of the same method in the recorded method sequences and θ is the threshold; the reason to calculate similarity is that multiple events that operate on different objects can be seen as the same in user-level, for example, events 1.1 and 1.2 in the motivating example can be taken as the same one; 3) the crash-triggering event is not equivalent to any other event. After event sorting, we compare the equivalence among events in the same node. If multiple events are the same,

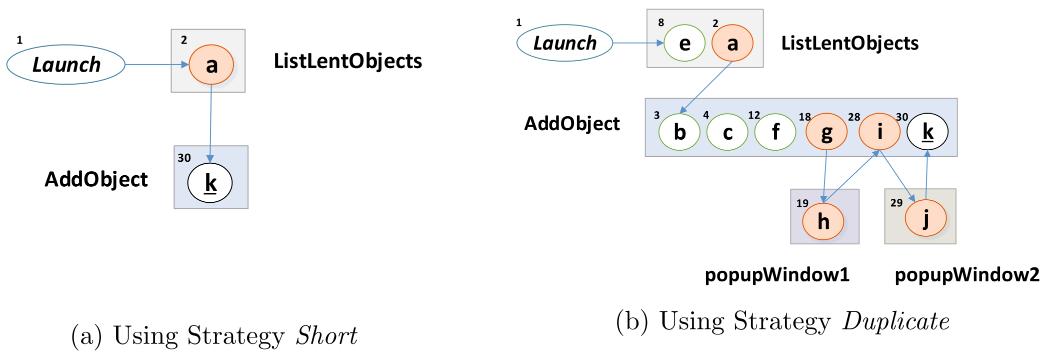


Fig. 13. KEDR results of the motivating example.

only the first one is kept and others will be removed. When removing one transition event, the previous event that leads to a pre-transition will be removed as well.

Algorithm 3 Hierarchy-tree reduction.

```
Input: Hierarchy-Tree $\mathcal{H}$
Output: Reduced Hierarchy-Tree $\mathcal{H}$
 1: sort the list of nodes in $\mathcal{H}$ according to their level
 2: for each node $n$ in $\mathcal{H}$ do
 3:     sort the list of events in $n$ according to their type and equivalence
 4: end for
 5: for each node $n$ in $\mathcal{H}$ do
 6:     $unReductedEvent$ = set()
 7:     for each event $e$ in n do
 8:         if findEquivalentEvent($unReductedEvent$, $e$) == false then
 9:             $unReductedEvent$.add($e$)
10:         else
11:             delete $e$ in $\mathcal{H}.n$
12:             if $e$ is a transition event then
13:                 find the previous event of $e_p$ of e and delete it in $\mathcal{H}$
14:             end if
15:         end if
16:     end for
17: end for
```

5.3. KEDR result of the motivating example

Fig. 13 gives the reduction results of the hierarchy-tree in the motiving example. For three different types of crash-triggering sequences, we have three strategies. Fig. 13(a) gives the reduction result using the *Short* strategy, which only retained three events. However, this sequence can not reproduce the same crash. The result of the *Duplicate* strategy that removes both ineffective and duplicate events is shown in Fig. 13(b). As we can see, 10 events are left. Although the order of the events left is different from the original sequence, the reduced sequence can still trigger the same crash as before. Suppose this simplified sequence also fails, we can only use strategy *Ineffective* to remove the ineffective events. In this case, the result is the same as the result in Fig. 10.

6. Event description format

To improve the readability of the reduced test sequence, we use a semi-structured format to describe an atom event. The format is defined as follows:

# ⟨*id*⟩ is a ⟨*type*⟩ event that ⟨*operates*⟩ object ⟨*obj*⟩, causing ⟨*result*⟩, whose level is ⟨*level*⟩.

The ⟨*id*⟩ represents the ids of a set of events in the original Monkey script. The ⟨*type*⟩ and ⟨*obj*⟩ are the type and object of the event. The ⟨*operates*⟩ describes the performed action on *obj*. The ⟨*result*⟩ describes the change of the application status, such as opening an activity or changing window status. And ⟨*level*⟩ denotes the significance of an event. We categorize events into the four significance levels, which is shown in Table 2. If an event satisfies more than one criterion, the higher level will be chosen.

Table 2
Criteria of event significance level.

Level	Description	Criteria
Essential	changes the application status	$E.op \in$ {Rotation, Appswitch, PinchZoom} $\vee E.chg_w =$ True
Major	triggers instrumented methods	$E.chg_i =$ True
Minor	operates widget or sends system key	$E.op \in$ {SysOp, MajorNav, Nav} $\vee E.obj \neq$ `null`
Trivial	is a no-ops event	other

#1 is a Appswitch event that opens the entry page of "de.freewarepoint.whohasmystuff", causing the launch of activity "/.ListLentObjects", whose level is Essential.
#8,9 is a Touch event that clicks on ListView with the id "list" and text "Add Entries", causing the launch of activity "/.AddObject", whose level is Essential.
#14,15 is a Touch event that clicks on EditText with id "personName" and text "Who Has My Stuff?", whose level is Major.
#18,19 is a Touch event that clicks on LinearLayout with text "Edit Lent Item", whose level is Minor.
#60,61 is a Touch event that clicks on EditText with id "add_description" and text "Add Entries", whose level is Major.
#96,97 is a Touch event that clicks on Button with the id "pickDate" and text "November 20, 2019", causing the launch of window "date" of activity "/.AddObject", whose level is Essential.
#98,99 is a Touch event that causes the close of window "date" of activity "/.AddObject", whose level is Essential.
#142,143 is a Touch event that clicks on Spinner with the id "type_spinner" and text "Other", causing the launch of window "popupWindow2" of activity "/.AddObject", whose level is Essential.
#144,145 is a Touch event that clicks on CheckedTextView with the id "text1" and text "Money", causing the close of window "popupWindow2" of activity "/.AddObject", whose level is Essential.
#146,147 is a Touch event that clicks on Button with the id "add_button" and text "Save changes", causing the crash of app, whose level is Essential.

Fig. 14. Semi-structured format bug report for Fig. 13(b).

The semi-structured format bug report for the sequence corresponding to the tree in Fig. 13(b) is shown in Fig. 14, which interprets the behavior of the remaining events. According to this description, developers that have knowledge of this application could find out the reason of application crash through tracking the behavior of events.

7. Evaluation

In this section, we give the details about the tool implementation and some experimental results to show the effectiveness, efficiency and usefulness of our approach.

7.1. Implementation

We implement a tool CHARD (CompreHensibility And ReDuction) aiming at removing ineffective events in normal sequences and extracting key events in crash-triggering ones. The overall design of our system is shown in Fig. 15. First, we use tool InsDal [15] to instrument all the callback methods of the application under test and update Monkey to Monkey$_{RR}$, which is a modified version of the original Monkey but supports record-and-replay. Then, we install both the instrumented apk and the modified Monkey$_{RR}$ on Android devices. In *Rule*, there are nine ineffective-event reduction rules, which can be configured by users in module *Configuration*. The module *Sequence Comprehension* analyzes the behavior of the events by GUI and log information and build models for sequences. In the *Event Sequence Reduction* module, both the ineffective-event-directed reduction (IEDR) and the key-event-directed reduction (KEDR) strategies are integrated. The outputs of the system are a reduced test script which can be reused via Monkey$_{RR}$ as well as a user-friendly simplified report about sequence execution.

7.2. Experimental setup

To evaluate the proposed approach, we set up the following three research questions:

- RQ1: (Remove Ineffective Events): How many ineffective events can be eliminated for various apps, and can the reduction keep the correctness? How can the IEDR approach combine with other sequence reduction approaches?
- RQ2: (Extract Key Events): To what extent can CHARD remove crash-irrelevant events and improve the efficiency of extracting key events in crash-triggering sequences?
- RQ3: (Cost Comparison): What are the deployment and implementation cost of our approach compared with the delta debugging technique?

To answer RQ1, we first randomly collect 80 popular applications from F-Droid, which involve 16 different categories. We pick open-source app market F-Droid considering the challenges in the practice of dynamic sequence reduction. For example,

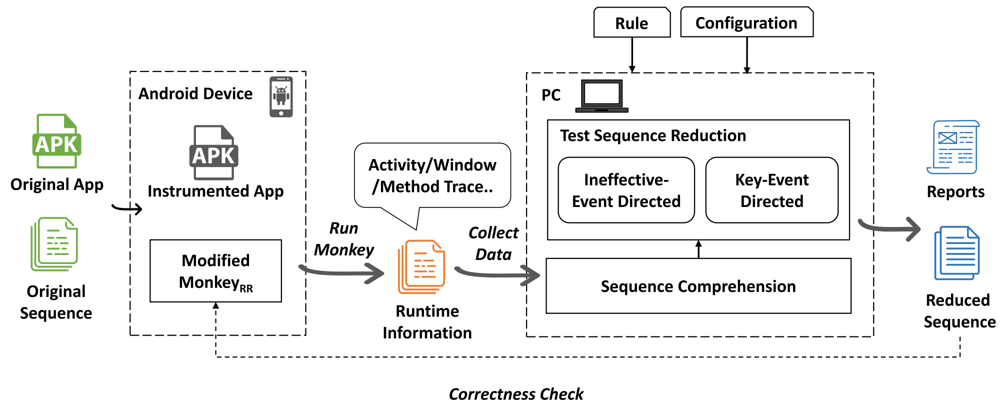


Fig. 15. Overview of our approach.

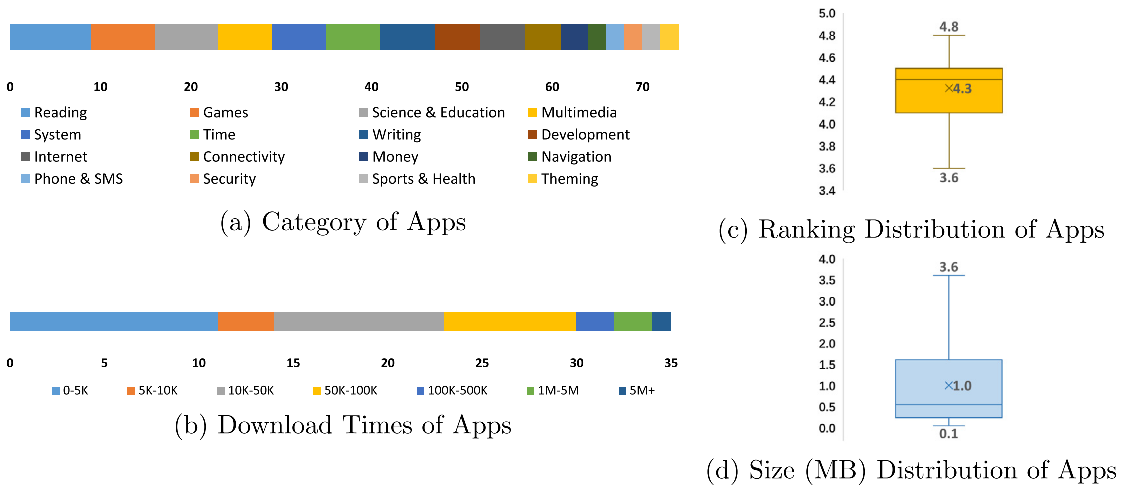


Fig. 16. Statistic information of the collected Apps.

the launching time of commercial apps is uncontrollable, whose conditions of occurrence and duration are not certain. Besides, the popping up of the app updating window and random advertisement influence the correctness of sequence replay. It is easy for corresponding developers to customize the setting of the app internally, while it is difficult for us to avoid these record-and-replay unfriendly characteristics. Therefore, we only pick open-source apps, which provide a relatively pure environment for record-and-replay, in order to avoid the indeterministic behaviors, even though our approach does not need the source code of apps. Six applications are filtered out because of the failure of de-compile or bad network connection. We give the statistic information about the rest 74 applications in Fig. 16. The average size of the collected applications is around 1 MB. The ratings of apps range from 3.6 to 4.8, and the average number of downloads is 50,000. After data set selection, we use Monkey$_{RR}$ to generate 10 sequences for each application with 1000 events. In addition, in order to check our approach could deal with long sequences, we randomly pick 15 stable applications and generate 10 long sequences for each application. Each sequence contains 10,000 events. In total, we obtain 890 test sequences.

To Answer RQ2, we evaluate to what extent CHARD can remove crash-irrelevant events and improve the efficiency of extracting key events in crash-triggering sequences. First, we reuse the eight applications provided by tool SimplyDroid. Due to the different devices and special characteristics of the app, four of the crashes can not be replayed (or stably replayed in three times) under our current evaluation environment. Thus, we use the left ones as well as another four apps with crashes on F-droid as our experimental instances. The statistic information of these applications and the crashes are listed in Table 3.

7.3. RQ1: remove ineffective events in normal sequences

We use CHARD to conduct experiments on collected test sequences with default configurations. The average rates of different types of ineffective events are shown in Fig. 17. As we can see, about 41.3% of events are ineffective ones, in which the no-ops event and the optional event take the most part. The results show that our approach has a good performance on sequence reduction. After the execution of CHARD, all sequences are interpreted to a set of user-friendly reports describing

Table 3
Information of Apps and crashes.

App	Category	Android version	Fault type	Exception type
SyncMyPic [16]	Photo synchronizer	4.4	Mutated	ArithmeticException
Tippy [17]	Tip calculator	4.4	Mutated	ArithmeticException
WeightChart [18]	Weight recorder	4.4	Real	ActivityNotFound
WhoHasMyStuff [9]	Item lending helper	4.4	Mutated	RuntimeException
Aeon's End [19]	Game setup randomizer	4.4	Mutated	NullPointerException
Dcipher [20]	Offline text encrypter	4.4	Real	RuntimeException
Prayer Times [21]	Tool for prayers	4.4	Real	RuntimeException
Transister [22]	Radio Player	4.4	Real	RuntimeException

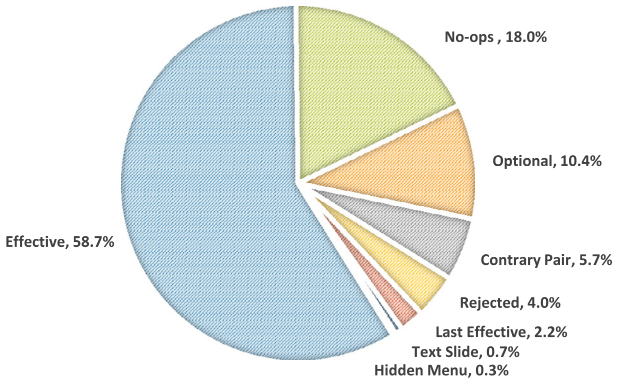


Fig. 17. Distribution of different event types.

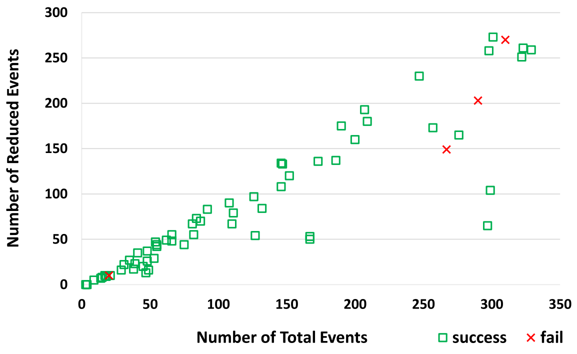
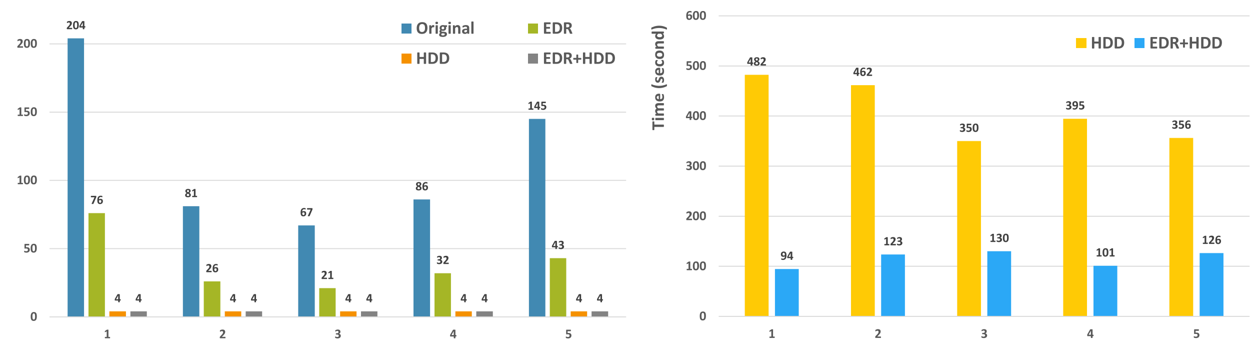


Fig. 18. Reduction rate and consistency of sequences.

the events in detail, which shows that 45% events are of level *Major* that trigger some functionalities of the app, 13% events are *Essential* that change the current window of the app, 24% events are *Minor* and 18% events are *Trivial*.

Then we measure whether the reduction by CHARD changes the execution result and leads to replay-failure. Due to the non-determinism of Android apps, executing the same sequence twice might have different results. To exclude these situations, we have to pick stable sequences to prove the correctness of our approach. We pick the stable sequences that can trigger crashes because it is difficult to set a criterion to check the consistency of two normal sequences. Among the 890 generated test sequences in our experiments, there are 132 sequences from 35 applications that abort due to error. According to the previous experiments [3], not all of the crashes are repeatable. Thus, we repeat each sequence three times and find 70 sequences that can trigger crashes in all three times. The 70 sequences with crashes are from 16 apps. We use CHARD to reduce them and replay the reduced script by Monkey. Then, we record the results of the execution and compare with the former results. The statistics are shown in Fig. 18. The 66 green dots represent the sequences that replay the former crash successfully. Conversely, 4 red dots indicate sequences where replay fails. These failures are related to the execution time. On the one hand, the attributes of some widgets are changeable with time, such as text and location. For example, `Acal` has ads to interfere in the layout. The changeable widgets in `BatHIIT` are used for timing. On the other hand, the time for network loading is uncertain, such as music player `Jamendo`. The event numbers in the sequences before and after reduction are also shown in Fig. 18. The average reduction rate is around 65.5%. In conclusion, our approach deletes a significant number of ineffective events and keeps consistence of sequences at the same time.



(a) Sequence Length (c) Execution Time

Fig. 19. Comparison between IEDR and HDD reduction.

Table 4
Number of events before and after reduction.

App name	Original	CHARD	Simply	App name	Original	CHARD	Simply
SyncMyPic	425	3	4	Tippy	98	3	3
	88	3	3		58	3	3
	153	3	3		88	3	3
	176	1	4		70	3	3
	461	3	8		33	3	3
WeightChart	63	3	4	WhoHas-MyStuff	204	17	4
	19	3	3		81	10	6
	29	3	3		67	8	4
	35	3	3		86	13	5
	78	3	7		145	11	4
Aeon's End	149	2	2	Dcipher	114	2	114
	123	2	2		122	2	122
	311	2	2		139	2	139
	79	2	2		150	2	150
	165	2	2		238	2	238
Prayer Times	168	13	147	Transister	222	2	2
	380	10	380		2700	2	1455
	90	11	84		986	2	3
	183	8	172		530	2	2
	160	12	160		199	2	2

The execution time of CHARD includes two parts: information collection as well as sequence reduction. The information collection time depends on the response time of the experimental device. For Android emulators, it takes 800 ms for one event execution and data collection, but with a rooted Android device, the time will be much shorter. Because we reduce sequence statically, the time for reduction is extremely short. In our experiment, CHARD takes an average of 1307 milliseconds to deal with 1000 events.

CHARD can be used as a pre-processor for other sequence reduction techniques. As we mentioned earlier, DD has been applied to perform event sequence reduction. Jiang et al. proposed a tool named SimplyDroid [3] to enhance the DD algorithm to simplify crash sequences generated by Monkey. The reduced scripts by DD are the shortest sequences to trigger the same crash. However, due to the iterative reduction progress, the running takes a long time. Our approach, which reduces test sequences based on event comprehension, could help to preprocess the test sequences and reduce the time of DD. Taking the motivating application WhoHasMyStuff as an example, we use SimplyDroid with the Hierarchical Delta Debugging (HDD) strategy to reduce sequences preprocessed by CHARD. Results about the lengths of sequences and the execution time before and after the pre-process are shown in Fig. 19. Because the IEDR strategy does not change the behavior of the whole sequence, we can get the same reduced sequence using HDD as well as using CHARD + HDD. But using CHARD as a pre-processor can significantly speed up the reduction process. In this example, the speedup ratio is over 3.6.

7.4. RQ2: extract key events in crash-triggering sequences

For each collected buggy app, we apply Monkey to generate five sequences that can successfully trigger the crashes. Then, we use CHARD, which combines the shortest path generation, the ineffective event based and the duplicate event based sequence reduction strategies. We compare our tool with the state-of-the-art tool SimplyDroid using HDD strategy. The results about the sequence length before and after reduction are shown in Table 4. In this table, we drop the first event in all the sequences, i.e., the application launching event. The column `Original` represents the original count of the user events. The columns `CHARD` and `Simply` represent the numbers of unreduced events using CHARD and SimplyDriod, respectively. By the correctness validation module, we find that all the reduced sequence can successfully trigger the same crashes as the original ones. The reduction rate of CHARD varies from 84.2% to 99.9%, and the number of the extracted key events ranges from 1 to 17. As a comparison, the reduction rate of SimplyDroid ranges from 0 to 99.7%, which fails to shorten the sequence of Dcipher and Prayer Times. After removing the ten sequences of these two apps, the reduction rate of SimplyDroid ranges from 46.1% to 99.7%. The results show that based on the analysis of the hierarchy-tree, CHARD can effectively extract the key events in crash-triggering sequences.

We also compare the time cost of using CHARD and SimplyDroid. The results are displayed in Fig. 20. Both CHARD and SimplyDroid need to replay the original sequence for sequence recording, the time of which is the same. We exclude the tree-construction time for both of them in the figure. For each reduction, we set the time limit as two hours. CHARD can finish all the reduction using almost the same time cost. For tool SimplyDroid, it does not terminate while reducing eleven sequences, so we manually finish the reduction process after two hours and record the result at that time. According to the results, SimplyDroid takes over 27 hours for all sequences using the hierarchical delta debugging strategy, while all the reduction by CHARD finishes within 10.2 seconds. We also measure the time that delta debugging needs to achieve the reduction given by CHARD. Excluding the apps that SimplyDroid fails on or does not terminate on, it takes SimplyDroid 4.8 h and 5.5 h to achieve the same reduction rate as the CHARD strategies *Ineffective* and *Short/Duplicate*. The experimental result indicates that CHARD can efficiently help the crash replay and the comprehension of the crash-triggering sequences.

The reduction rate of each strategy in CHARD is shown in Fig. 21, which contains all the 40 test cases for eight apps. Using strategy *Short*, almost 97% events will be removed, while using strategies *Duplicate* and *Ineffective*, we can delete 93% and 73% events in average, respectively. The results show that our reduction can help to remove a large number of events whatever strategy is taken. We replay the reduced sequences using different strategies, in which all the sequences adopting strategy *Ineffective* can trigger crashes successfully. For strategies *Short* and *Duplicate*, the success rate is 75% and 72.5%, respectively. Fig. 22 displays the distribution of the reduction rate using each strategy in a box-plots. According to the reduction order provided by CHARD, i.e., *Short* > *Duplicate* > *Ineffective*, the strategy that can reduce more events has higher priority. If both the strategies *Short* and *Duplicate* fail, the last strategy *Ineffective* can guarantee that the same crash will be successfully replayed.

As we can see, in the scenario of extracting key events in crash-triggering sequences, using either the strategy *Short* or *Duplicate* may lead to replay failure. There are several reasons. On the one hand, each strategy has its own application scenario. The strategy *Short* is suitable for finding the shortest path for the single crash-triggering event. However, when there are multiple key events and at least one key event is not in the shortest path, the preproduction fails. And the *Duplicate* strategy is suitable for extracting a set of non-repeating independent crash-triggering events. If the key events need to be executed repeatedly, e.g., click the button on the keyboard to form an extremely long input for the `EditText` widget, *Duplicate* will not work. On the other hand, the result of all the reduction strategies depends on the abstraction degree of the hierarchy-tree. The more fine-grained the state abstraction is, e.g., consider all the attributes of widgets, the more precise the event modeling will be. In our experiments, we use activity/window to separate different states, which may lead to infeasible simplified subsequences.

We also measure the time saved in the manual debugging process by using event sequence reduction tools with the assistance of four postgraduates. Each tester obtains four apps with CHARD's result and four apps with SimplyDroid's result. Also, each app is delivered to four testers, two of whom have CHARD's result and the others have SimplyDroid's. The results about time to successfully replay the given test script and to find the minimal set of key events after script replaying are shown in Fig. 23. Three sequences fail to be replayed on the testers' emulator after a hard trying, two belong to SimplyDroid and one belongs to CHARD. So we exclude them. As we can see, the key events of most test cases can be extracted by testers in 100 seconds, and the average debugging time of CHARD and SimplyDroid are 38 and 232 seconds, respectively. The time of debugging is positively related to the length of the given sequence. For example, the debugging time for app *Prayer's time* and *Dcipher* using SimplyDroid is longer than using CHARD, and the debugging time for app *WhoHasMyStuff* using CHARD is longer than using SimplyDroid. One tester uses over 10 minutes to locate the crash-triggering key events in *WeightChart*, and reports that it is difficult to find out the key events manually without a user-friendly report.

7.5. RQ3: cost comparison

As we can see, both the tools CHARD and SimplyDroid perform reduction on Android GUI system and use the test sequences generated by tool Monkey. In this part, we compare the cost of our approach and delta debugging technique on these two instantiations, i.e., tool CHARD and SimplyDroid, about the cost of tool implementation and deployment.

7.5.1. Tool implementation cost

The first part that needs to be implemented is Monkey enhancement. Due to the characteristic of Monkey, it is difficult to make a sequence reduction based on the original Monkey. Therefore, both CHARD and SimplyDroid enhance Monkey to support record-and-replay and obtain the window information during recording. Considering the compatibility issue, the enhanced Monkey should be compiled for different Android versions according to the test requirement. Fortunately, this pre-process needs to be done only once. Besides, the testers should implement the reduction algorithm. Delta debugging is

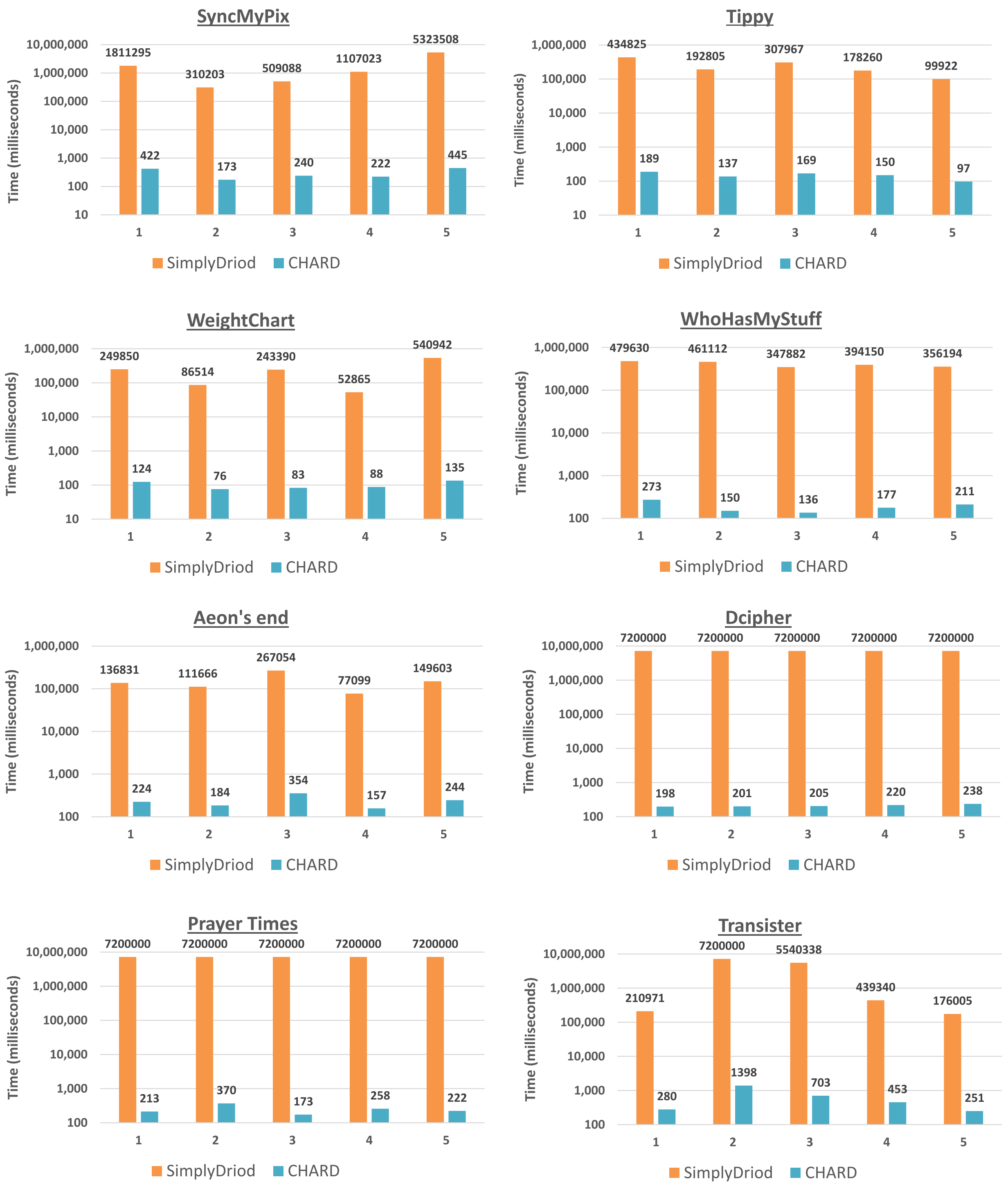


Fig. 20. Execution time (ms) of CHARD and SimplyDroid (HDD).

a light-weight technique, which is easy to apply. Existing statistic shows that a delta debugging implementation fits in less than 100 lines of code [23], where the core algorithm implementation (DeltaDebuggingReducer) takes 24 lines only. Further, to evaluate the difference of implementation cost between two tools, we measure their lines of code (LOC). The results show that CHARD contains 2879 lines of code and SimplyDroid contains 2054 (including four dynamic reduction strategies), which indicate that our approach introduces little technical debts in implementation.

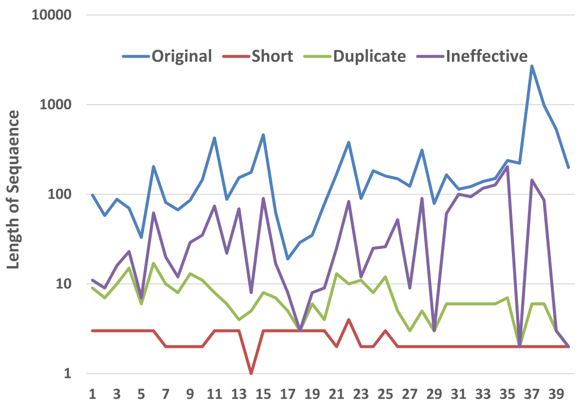


Fig. 21. Length of sequence after reduction.

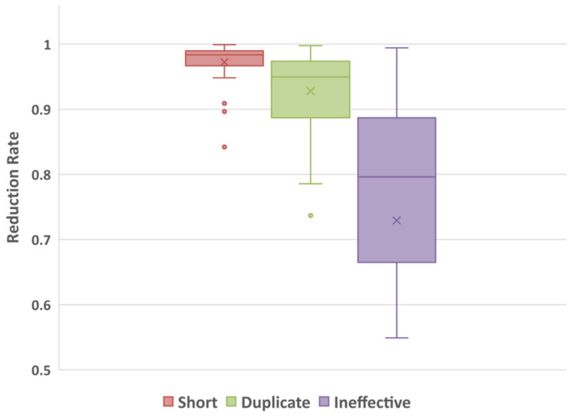


Fig. 22. Reduction rate of each strategy.

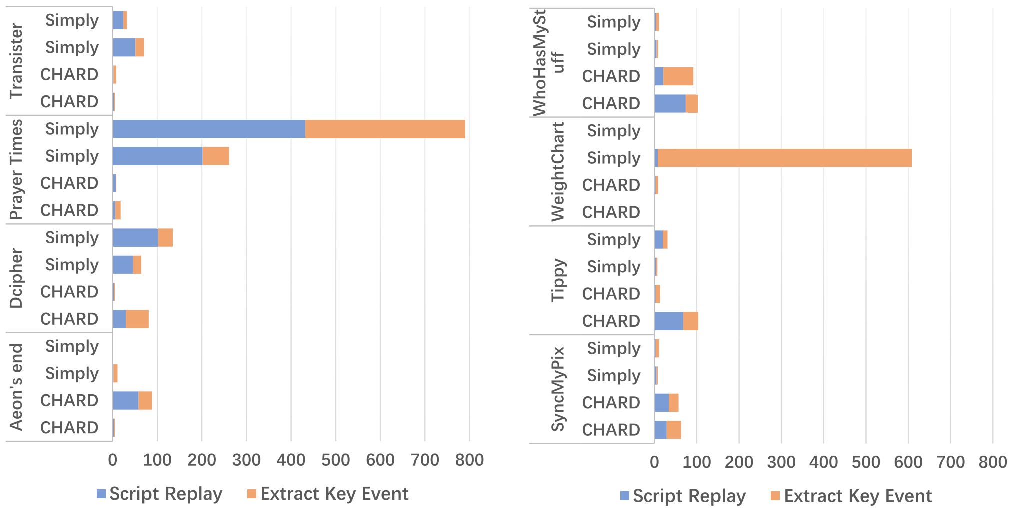


Fig. 23. Time of manual debugging by four testers.

7.5.2. Tool deployment cost

Before making a reduction, testers should prepare a set of crash-triggering Monkey sequences, which could be successfully replayed. Then CHARD and SimplyDroid are deployed to replay these sequences and collect runtime information. During this process, SimplyDroid only collects UI information, while CHARD collects both the UI and log information to represent the window and the internal status of the app. In our approach, we insert log statements at the method-level and detect their occurrence in log files. The occurrence of the instrumentation related logs can map the changes of internal status. Since the number of callback methods is limited in an app, the overhead of instrumentation is negligible. Based on the recorded information, CHARD and SimplyDroid make reduction automatically using their strategies, and then verify the reduction results by dynamic sequence replay.

8. Threats to validity

In this section, we give the external validity and internal validity of our approach.

8.1. External validity

The main external validity of our approach is the generalization of the experimental instances. In the experiments, we only pick 80 open-source apps, which provide a relatively pure environment for record-and-replay. The number of experimental subjects is limited and these selected sample subjects may not generalize to complex, frequently used, and closed-source commercial apps. However, it is reasonable to assume that CHARD could be used by the developers of commercial apps, who can exercise more control on testability.

Another external validity is about the reliability of record-and-replay. The generated event sequence should be successfully recorded-and-replayed before reduction. If the original sequence can not re-trigger the crash before it is simplified, any reduction algorithm, including DD and CHARD, becomes ineffective. On the one hand, the success of sequence record-and-replay depends on the determinism of each user event in the sequence. For example, if one widget changes with time, e.g., some ad, or the program status depends on the value of a random variable, like the game *2048*, the correctness of the replay process can not be guaranteed. One solution is to pre-detect these nondeterministic widgets by static analysis and combine the dynamic instrumentation technique to keep the path execution result unchanged [24]. On the other hand, the initial status of the application also influences the success of record-and-replay, for the execution of the test sequences may change the global status of the app. Researchers usually assume that the sequence execution starts from the install state of applications to construct the same environment. However, reinstalling applications before each event sequence execution is costly. To minimize the number of restarts, testers may set an extremely long sequence until crashes are detected, which increases the difficulty of fault localization and makes the sequence reduction approach necessary.

8.2. Internal validity

The internal validity discusses the conditions under which the results of CHARD may not be reliable. In our approach, the internal validity involves the reliability of reduction.

If a sequence can be successfully replayed, there are still several prerequisites for its correct reduction. The ineffective-event-directed-reduction aims to remove all the ineffective events, which has two assumptions. First, any previous event has been handled by the Android system before the next event is executed, i.e., the effect of an event will not delay. Considering that an application may not be able to process a multitude of events in a given time frame, proper sleeping time after each event execution should be set to minimize the impact of this situation. Second, we assume that events meeting the requirements of effect-free combination patterns always can counteract each other according to the observations of empirical study. However, in a few cases, checking a box widget twice or open and close a window may influence the internal state of the application. Therefore, during the reduction process, we determine whether a combination influences the internal state also by checking the changes of UI windows as well as the execution logs. In our implementation, we use the method-level code instrumentation instead of the statement-level one to balance the instrument overhead and the accuracy of the information monitoring. And we use dynamic replay to verify the correctness of our reduction. In the future, it will be useful to pre-determine which events are reversible by static analysis before the record-and-replay. The key-event-directed-reduction aims to remove all crash-irrelevant events, which uses more aggressive assumptions to make more reduction. For example, one of the strategies assumes that the crash is triggered by a set of non-repeating independent crash-triggering events. To guarantee the correctness of reduction, we combine an adaptive strategy with the dynamic verification to pick the proper reduction strategy. Due to the efficiency of static reduction, this key-event-directed-reduction takes little effort and brings more benefits to make an in-depth reduction when the assumption is met. And the most conservative strategy *Ineffective* could deal with most of the situations.

9. Related work

Monkey testing. Recently, there are a large number of works focusing on the topic of GUI input generation of Android applications [25], which can be separated into several types: random testing [7,26,27], model-based [28–33], systematic testing [34–36], search-based testing [37–39] and others [40,41]. According to the results of several existing empirical studies [6,42], the random testing tool, Monkey, outperforms other state-of-the-art tools. Some works make use of Monkey to detect GUI bugs [43] and security bugs [44]. And some researchers develop new tools based on the random exploration strategies of Monkey. For example, tools Dynodroid [26] and PUMA [27] are based on Monkey. They use more sophisticated strategies and trigger effective events only by obtaining and analyzing the layout information.

Different from these tools, which are designed for taking the place of Monkey, in this paper, we improve Monkey from another aspect. We do not develop a new tool, for Monkey is still the most widely used tool in industrial settings [45]. Instead, we update Monkey without modifying the original framework but only add the support of record-and-replay ability. We further improve the usability of Monkey by analyzing the recorded data and making reduction of the original Monkey sequence. Moreover, we also generate user-friendly behavior reports or bug reports for each executed Monkey sequence.

Test reduction. Delta-debugging [46] algorithm is widely used for dynamic sequence reduction, which uses an iterative process to repeat the reduction until a minimal subsequence is obtained. For test reduction of Android apps, Clapp et al. [2] proposed the Non-Deterministic Delta Debugging Minimization algorithm, which handles the non-deterministic execution of Android input event sequence and tries to find a minimal event trace that can reach a given target Activity. Besides, Jiang et al. proposed the approach SimplyDroid [3], which enhances the original delta-debugging algorithm to simplify Android input event sequences while reproducing the same crash. The dynamic delta-debugging process is time-consuming, and our tool CHARD can be adopted as a pre-processing module of delta-debugging-based tools and achieve significant speed-up.

Program slicing [47] [48] [49] is another technique frequently used in sequence reduction. EFF [50] aims to reduce the event sequence for UNIX by using a dynamic slicing technique. JSTrace [51] is a tool that uses a novel dynamic slicing technique and reduces the event sequence of the web application, aiming to effectively cut down error reproduction time. AppDoctor [52] is a system that tests applications efficiently and effectively under various systems and user actions. It uses two slicing strategies, which require the dependence of actions, to help the reduction of event sequence. Our approach does not concentrate on the event dependencies, it only requires the transition relationship among events, which is easy to obtain.

Besides the event sequence reduction, there are also some works that concentrate on the reduction of test suites. For example, McMaster et al. [53] proposed a GUI test suite reduction technique for reducing the number of test cases in a test suite. Another tool DetReduce [54] minimizes the test suites for regression testing. It reduces both the number of test cases as well as the size of each test case by removing some common forms of redundancies. DetReduce makes reduction in a single test case to improve the effectiveness of the whole test suite, while our approach only concentrates on improving the effectiveness of a single event sequence.

Crash reproduction. For Android apps, the key challenge of crash-directed reduction is how to simplify the test sequence, while keeping the correct reproduction of the crash. Tool SimplyDroid [3] achieves the crash reproduction by using the enhanced delta-debugging technique. It constructs the hierarchy-tree of applications and uses several heuristics to dynamically reduce the tree, which performs well when the crash-triggering event does not depend on any other events. When there are event dependencies, the dynamic iteration is time-consuming. In our approach, we are inspired by the idea of hierarchy-tree based reduction. But we make the reduction statically by understanding the behavior of events and give different strategies for different types of crash-triggering sequences, which is more efficient. Another tool ECHO [54] aims at removing bug irrelevant events by exploiting the differential behavior between the GUI states collected and finding the shortest path from the initial to the error state. Similar to SimplyDroid, this approach does not work if the crash-triggering event relies on other events. In addition to reproducing crash from test sequences, there are also approaches which replay crashes using bug reports. CRASHSCOPE [55] tests applications by using systematic exploration and generates a human-readable report showing the steps to reproduce crashes. However, this work focuses on understanding test sequences but does not reduce them. Our tool CHARD not only removes unnecessary events but also generates user-friendly reports for sequence comprehension. A novel tool ReCDroid [56] uses a combination of natural language processing and dynamic GUI exploration to synthesize event sequences with the goal of reproducing the reported crash. Compared with CHARD, ReCDroid focuses on a different task.

10. Conclusion

Test reduction is an important ingredient for efficient debugging. In this paper, we propose a general model for GUI event sequence reduction problem, and give an instantiation of the model on Android platform as well as the widely used random testing tool Monkey. During the investigation, we find that many events of Monkey are ineffective ones. So we propose an ineffective-event-directed sequence reduction approach, which can effectively remove more than 40% events in the original sequence. Besides, we build a hierarchy-tree for each given sequence and propose a key-event-directed reduction approach to remove crash-irrelevant events, which achieves a similar reduction rate compared with the state-of-the-art tool, while using extremely less time. In addition, we design a semi-structured format to describe the behavior of events for improving the comprehensibility. The evaluation shows that CHARD can reduce ineffective events in normal sequences as well as extract key events in crash-triggering sequences effectively, correctly and efficiently. In the future, we plan to combine the technique of static analysis and dynamic execution to further analyze the behavior of each event in test sequences and guide the test reduction.

CRediT authorship contribution statement

Jiwei Yan: Conceptualization, Methodology, Writing - original draft. **Hao Zhou:** Data curation, Software. **Xi Deng:** Investigation. **Ping Wang:** Investigation. **Rongjie Yan:** Writing - review & editing. **Jun Yan:** Funding acquisition, Supervision. **Jian Zhang:** Funding acquisition, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported by National Natural Science Foundation of China (Grant No. 61672505), and Key Research Program of Frontier Sciences, CAS (Grant No. QYZDJSSW-JSC036). We are grateful to Yajun Zhu for proofreading.

References

[1] M. Hammoudi, B. Burg, G. Bae, G. Rothermel, On the use of delta debugging to reduce recordings and facilitate debugging of web applications, in: 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30–September 4, 2015, pp. 333–344.
[2] L. Clapp, O. Bastani, S. Anand, A. Aiken, Minimizing GUI event traces, in: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, pp. 422–434.
[3] B. Jiang, Y. Wu, T. Li, W.K. Chan, Simplydroid: efficient event sequence simplification for Android application, in: 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30–November 03, 2017, pp. 297–307.
[4] Google Play, The Google Play application market, https://play.google.com/store?hl=en, 2019.
[5] Statista, Number of available applications in the Google Play Store from December 2009 to December 2018, https://www.statista.com/statistics/266210/number-of-availableapplications-in-the-google-play-store/, 2019.
[6] S.R. Choudhary, A. Gorla, A. Orso, Automated test input generation for Android: are we there yet?, in: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015, pp. 429–440.
[7] Android Monkey, http://developer.android.com/tools/help/monkey.html, 2019.
[8] P. Wang, J. Yan, X. Deng, J. Yan, J. Zhang, Understanding ineffective events and reducing test sequences for Android applications, in: 2019 International Symposium on Theoretical Aspects of Software Engineering, TASE 2019, Guilin, China, July 29–31, 2019, pp. 264–272, in press.
[9] Who has my stuff, https://f-droid.org/en/packages/de.freewarepoint.whohasmystuff/, 2020.
[10] Motionevent, https://developer.android.com/reference/android/view/MotionEvent, 2019.
[11] F-droid, https://f-droid.org/, 2019.
[12] V.S. Alagar, K. Periyasamy, Specification of Software Systems, Graduate Texts in Computer Science, Springer, 1998.
[13] M. Sipser, Introduction to the Theory of Computation, PWS Publishing Company, 1997.
[14] Y. Gu, P. Ma, X. Jia, H. Jiang, J. Xuan, Progress on software crash research, Sci. Sin. Info. 49 (2019) 1383–1398.
[15] J. Liu, T. Wu, X. Deng, J. Yan, J. Zhang, Insdal: a safe and extensible instrumentation tool on dalvik byte-code for Android applications, in: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20–24, 2017, pp. 502–506.
[16] Syncmypix, https://github.com/nloko/SyncMyPix, 2020.
[17] Tippytipper, https://github.com/mandlar/tippytipper, 2020.
[18] Weightchart, https://github.com/bluezoot/weight-chart, 2020.
[19] Aeon's end, https://f-droid.org/en/packages/com.games.boardgames.aeonsend/, 2020.
[20] Dcipher, https://f-droid.org/en/packages/com.adityakamble49.dcipher/, 2020.
[21] Prayer, https://f-droid.org/en/packages/com.metinkale.prayer/, 2020.
[22] Transistor, https://f-droid.org/packages/org.y20k.transistor/, 2020.
[23] Delta-debugging cost, https://www.fuzzingbook.org/html/Reducer.html, 2020.
[24] O. Sahin, A. Aliyeva, H. Mathavan, A.K. Coskun, M. Egele, RANDR: record and replay for Android applications via targeted runtime instrumentation, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, 2019, pp. 128–138, https://doi.org/10.1109/ASE.2019.00022.
[25] T. Wu, X. Deng, J. Yan, J. Zhang, Analyses for specific defects in Android applications: a survey, Front. Comput. Sci. 13 (2019) 1210–1227.
[26] A. Machiry, R. Tahiliani, M. Naik, Dynodroid: an input generation system for Android apps, in: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, August 18-26, 2013, pp. 224–234.
[27] C. Qian, L. Huang, M. Cao, J. Xie, H. So, PUMA: an improved realization of MODE for DOA estimation, IEEE Trans. Aerosp. Electron. Syst. 53 (2017) 2128–2139, https://doi.org/10.1109/TAES.2017.2683598.
[28] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, Z. Su, Guided, stochastic model-based GUI testing of Android apps, in: 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017, pp. 245–256.
[29] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, Z. Su, Practical GUI testing of Android applications via model abstraction and refinement, in: 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 269–280.
[30] Y.M. Baek, D. Bae, Automated model-based Android GUI testing using multi-level GUI comparison criteria, in: 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, pp. 238–249.
[31] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, J. Lu, Aimdroid: activity-insulated multi-level automated testing for Android applications, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17–22, 2017, pp. 103–114.
[32] W. Yang, M.R. Prasad, T. Xie, A grey-box approach for automated GUI-model generation of mobile applications, in: Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013, pp. 250–265.
[33] T. Azim, I. Neamtiu, Targeted and depth-first exploration for systematic testing of Android apps, in: 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, Part of SPLASH 2013, Indianapolis, IN, USA, October 26–31, 2013, pp. 641–660.
[34] S. Anand, M. Naik, M.J. Harrold, H. Yang, Automated concolic testing of smartphone apps, in: 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA, November 11–16, 2012, p. 59.
[35] C.S. Jensen, M.R. Prasad, A. Møller, Automated testing with targeted event sequence generation, in: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013, pp. 67–77.
[36] H. van der Merwe, B. van der Merwe, W. Visser, Execution and property specifications for jpf-Android, ACM SIGSOFT Softw. Eng. Notes 39 (2014) 1–5, https://doi.org/10.1145/2557833.2560576.

[37] D. Amalfitano, N. Amatucci, A.R. Fasolino, P. Tramontana, Agrippin: a novel search based testing technique for Android applications, in: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015, Bergamo, Italy, August 31–September 4, 2015, 2015, pp. 5–12.
[38] R. Mahmood, N. Mirzaei, S. Malek, Evodroid: segmented evolutionary testing of Android apps, in: 22th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE-22, Hong Kong, China, November 16–22, 2014, pp. 599–609.
[39] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for Android applications, in: A. Zeller, A. Roychoudhury (Eds.), Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18–20, 2016, ACM, 2016, pp. 94–105.
[40] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, 1977, pp. 238–252.
[41] C.M. Liang, N.D. Lane, N. Brouwers, L. Zhang, B.F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, F. Zhao, Caiipa: automated large-scale mobile app testing through contextual fuzzing, in: The 20th Annual International Conference on Mobile Computing and Networking, MobiCom'14, Maui, HI, USA, September 7–11, 2014, 2014, pp. 519–530.
[42] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, T. Xie, An empirical study of Android test generation tools in industrial cases, in: 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018, pp. 738–748.
[43] C. Hu, I. Neamtiu, Automating GUI testing for Android applications, in: 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23–24, 2011, pp. 77–83.
[44] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, A. Stavrou, A whitebox approach for automated security testing of Android applications on the cloud, in: 7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2–3, 2012, pp. 22–28.
[45] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, T. Xie, Automated test input generation for Android: are we really there yet in an industrial case?, in: 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13–18, 2016, pp. 987–992.
[46] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Trans. Softw. Eng. 28 (2002) 183–200, https://doi.org/10.1109/32.988498.
[47] T. Gyimóthy, Á. Beszédes, I. Forgács, An efficient relevant slicing method for debugging, in: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'99, Toulouse, France, September, 1999, pp. 303–321.
[48] M. Weiser, Programmers use slices when debugging, Commun. ACM 25 (1982) 446–452.
[49] X. Zhang, R. Gupta, Y. Zhang, Precise dynamic slicing algorithms, in: 25th International Conference on Software Engineering, Portland, Oregon, USA, May 3–10, 2003, 2003, pp. 319–329.
[50] X. Zhang, S. Tallam, R. Gupta, Dynamic slicing long running programs through execution fast forwarding, in: 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5–11, 2006, pp. 81–91.
[51] J. Wang, W. Dou, C. Gao, J. Wei, Jstrace: fast reproducing web application errors, J. Syst. Softw. 137 (2018) 448–462, https://doi.org/10.1016/j.jss.2017.06.038.
[52] G. Hu, X. Yuan, Y. Tang, J. Yang, Efficiently, effectively detecting mobile app bugs with appdoctor, in: 9th EuroSys Conference 2014, EuroSys 2014, Amsterdam, the Netherlands, April 13–16, 2014, pp. 18:1–18:15.
[53] S. McMaster, A.M. Memon, Call-stack coverage for GUI test suite reduction, IEEE Trans. Softw. Eng. 34 (2008) 99–115, https://doi.org/10.1109/TSE.2007.70756.
[54] Y. Sui, Y. Zhang, W. Zheng, M. Zhang, J. Xue, Event trace reduction for effective bug replay of Android apps via differential GUI state analysis, in: ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019, pp. 1095–1099.
[55] K. Moran, M.L. Vásquez, C. Bernal-Cárdenas, C. Vendome, D. Poshyvanyk, Automatically discovering, reporting and reproducing Android application crashes, in: 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11–15, 2016, pp. 33–44.
[56] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, W.G.J. Halfond, Recdroid: automatically reproducing Android application crashes from bug reports, in: 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25–31, 2019, pp. 128–139.