The Journal of Systems & Software 183 (2022) 111085

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

User behavior pattern mining and reuse across similar Android apps

Qun Mao, Weiwei Wang *, Feng You, Ruilian Zhao, Zheng Li

College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, PR China

ARTICLE INFO

Article history:
Received 10 January 2021
Received in revised form 7 June 2021
Accepted 2 September 2021
Available online 22 September 2021

Keywords:
Android apps
Behavior pattern reuse
Semantic-based event fuzzy matching
GUI model

ABSTRACT

Nowadays, Android apps have penetrated all aspects of our lives. Despite their popularity, understanding their behaviors is still a challenging task. Considering that many Android apps are in the same category and share similar workflows, in this paper, we propose a user behavior pattern mining and reuse approach across similar Android apps, thereby reducing the cost of understanding new apps. Particularly, for a specific new app, to figure out its typical behaviors, the behavior patterns that refer to the frequently-occurring workflows can be obtained from another similar app and transferred to this app. Moreover, to reuse the behavior patterns on this app, a semantic-based event fuzzy matching strategy and continuous workflow generation strategy are raised to generate workflows for this app. To evaluate our approach's effectiveness and rationality, we conduct a series of experiments on 25 Android apps in five categories. Furthermore, the experimental results show that 88.3% of behavior patterns can be completely reused on similar apps, and the generated workflows cover 89.1% of the top 20% of important states.

1. Introduction

In recent years, we have witnessed the rapid development of the mobile Internet and reached a consensus that mobile platforms have dominated the future trend of personal computing and internet working. As Android is the most used mobile platform, a large number of Android apps are developed and launched. Despite their popularity, comprehending the dynamic behavior of an Android app is still challenging in practice. Currently, the most direct way to have insight into a new app's behaviors is to interact with it according to authentic human experience, but it is tedious and costly.

Intuitively, similar apps in the same domain share similar behaviors, as reported by Kudo et al. (2019) and Mao et al. (2017b). Rather than depending on manual labor, the known behaviors from other similar apps also make it possible to get started with new apps quickly. More specifically, migrating or reusing the existing apps' behaviors to new apps can help quickly figure out new apps' behaviors. However, there has been a lack of effort exploiting the benefits of behavior reuse across similar Android apps.

At present, a software system's behavior is usually captured through dynamic exploration and expressed by *traces* (Lorenzoli et al., 2008; Brooks and Memon, 2007; Wang et al., 2018). The traces record the system's workflows, consisting of concrete events sequences triggered by users or crawling. To further analyze and understand the typical behavior of a system, mining generic behavior *patterns* from the traces, which denote the frequently-occurring workflows of the system, is put forward. In contrast with *traces*, *patterns* involved in a system stand for behaviors that are more likely to be triggered. Since similar apps share similar behaviors, the behavior *patterns* can also be transferred from one app to similar apps. More importantly, behavior patterns reuse can support the automatic creation of tutorials or usage guides, or automatic assistance to the user.

Therefore, this paper proposes a behavior pattern mining and reuse approach across similar Android apps, which mines behavior *patterns* from existing *traces* of similar apps and reproduces the patterns to new apps, thereby helping users understand unfamiliar apps promptly. Although this approach is theoretically feasible, we still face several technical challenges both in behavior pattern mining and reusing. Concretely, in pattern mining, the complexity of traces brings difficulties to sequential pattern mining. In pattern reusing, transferring the event sequences in behavior patterns to the new apps presents two challenges. One is that different apps usually vary in GUI, affecting the event mapping across Android apps. The other is that subtle difference still exists in the workflow of similar Android apps, leading to the discontinuity of the mapped events.

To address these issues, the semantic of events and the GUI model of apps are taken into account in this paper in order to achieve behavior pattern reuse across apps. More specifically, since GUI widgets that bind similar events may be syntactically different but semantically similar, the semantics of GUI widgets are considered in the paper in order to realize accurate event

* Corresponding author.
E-mail address: wangww@mail.buct.edu.cn (W. Wang).

https://doi.org/10.1016/j.jss.2021.111085
0164-1212/

mapping across apps. Besides, the attributes of two matched events may not correspond one to one. Thus, a fuzzy matching strategy for GUI events is necessary. Moreover, the GUI model of an application provides a way to link the discontinuous events. So, it is used to guide the generation of feasible and successive workflows for new apps during pattern reuse.

To the best of our knowledge, it is the first to provide an automated behavior pattern reuse approach across Android apps. The primary contributions of this paper are as follows:

1. A user behavior pattern mining and reuse approach across similar Android apps is proposed, which can reproduce the typical workflows (patterns) from known apps for other unfamiliar apps, reducing the cost of apps comprehension.
2. A semantic-based event fuzzy mapping strategy and a continuous workflow generation strategy are raised to generate workflows for the new apps during behavior pattern reuse.
3. An empirical evaluation of 25 open-source Android apps demonstrates that our behavior pattern mining and reuse approach is feasible and effective. Moreover, the generated workflows cover most of the important behaviors in the new apps.

The remainder of this paper is organized as follows. Section 2 introduces a case study to illustrate the behavior pattern reuse process as well as the challenges. Section 3 describes the details of our proposed approach. Section 4 analyzes the empirical evaluation results and threats to validity. Section 5 depicts the related work. Finally, Section 6 concludes our work in this paper.

2. Case study

In this section, we conduct a case study on two similar Android apps to illustrate the process of behavior pattern reuse. Meanwhile, the problems encountered in the process are also pointed out in this part.

Suppose the shopping list app *ShoppingList*2 is a new app that users want to know, which can be regarded as the target app. To quickly figure out the commonly used functionalities of this app, users can take advantage of another similar app (e.g., *ShoppingList*1) analyzed before. *ShoppingList*1 can be taken as the source app. Assume that behavior patterns of *ShoppingList*1 are mined from traces in advance. Take one behavior pattern indicating sorting shopping items as shown in Fig. 1 as an example. Our goal is to reuse this pattern to the target app *ShoppingList*2, thereby knowing the similar functionality on *ShoppingList*2.

The behavior pattern of the source app in Fig. 1 describes that a user wants to sort the shopping items he/she added before. In more detail, firstly, he/she has to enter the main activity, e.g. activity a, and click on the shopping list "Fruit" (e_1) to reach the *shoppinglist* activity b. Then, click on "Sort" element (e_2), arriving at activity c. At last, check the "ascending" event (e_3) in activity c and click on the "OKAY" button (e_4) to sort shopping items in ascending alphabetical order, leading the sorting result in activity e. The behavior pattern is represented as $\langle e_1, e_2, e_3, e_4\rangle$.

In order to facilitate the understanding of the reuse process of behavior patterns, we give the final generated workflow for the target app *ShoppingList*2 in advance. The event sequence involved in the target workflow is shown in Fig. 2. Concretely, to sort shopping items, the user should start at the main activity, e.g. activity a', and click on the shopping list "Fruit" (e'_1) to enter the *shoppinglist* activity b'. Then, click on "More options" (e'_0) arriving at activity c'. Then, the "Sort..." event in the activity c' is triggered (e'_2), and the application transfers to the activity d'. Finally, click on "A-Z" (e'_3) in activity d', receiving the sort result in the activity e'. The generated workflow is represented as $\langle e'_1, e'_0, e'_2, e'_3\rangle$.

Even though the workflows for item sorting on the source app and the target app are similar, a direct copy of the behavior pattern from the source app to the target app is not feasible due to two reasons: (1) The appearance of GUI widgets of different apps is different, leading the difficulty of creating event mapping relations between apps. For instance, the widget's text of e_3 in the source app is "ascending", while that of e'_3 in the target app is "A-Z". (2) There are still subtle differences between the workflows of different apps, causing the matched event sequence on the target app not continuous and executable. Take the two workflows in Figs. 1 and 2 as example, for e_1, e_2 and e_3 of the source app in Fig. 1, e'_1, e'_2 and e'_3 of the target app in Fig. 2 are supposed to match them, respectively. Because both event e_1 and e'_1 are to enter the *shoppinglist* activity, e_2 and e'_2 aim to provide the sorting menu selection, while e_3 and e'_3 are to sort the shopping items in ascending alphabetical order. And no event is found to match e_4 of the source app on the target app. Thus, event sequence $\langle e'_1, e'_2, e'_3\rangle$ on the target app is obtained after creating event mapping relations. However, it is not an executable event sequence since e'_0 is missing between e'_1 and e'_2.

As the example shows, to transfer the behavior pattern $\langle e_1, e_2, e_3, e_4\rangle$ of the source app to the target workflow $\langle e'_1, e'_0, e'_2, e'_3\rangle$, the above two challenges must be solved. For the first challenge in event mapping, from e_3 and e'_3, we can see that although the widgets' texts are syntactically different, they are similar in semantics. In other words, events achieving similar functionality are alike in the semantics of the widgets they bind. So the semantic of events should be considered when matching events across apps. Besides, the semantic of two similar events may come from different attributes of widgets. For example, the widget of e_2 has the content description "sort" while that of e'_2 has the text "sort". Intuitively, all attributes of a widget can represent its semantics, and they are consistent in semantics. Thus, the attributes of widgets from one application can cross-compare with those of widgets from a similar application. For the second challenge in pattern reuse, the GUI model of an application is widely used to supplement events when migrating event sequences across different platforms (Qin et al., 2019) and applications (Behrang and Orso, 2019) . Inspired by them, in this paper, the GUI model of the target app is taken into account to support feasible workflow generation.

3. Behavior pattern mining and reuse approach

It can be observed that behavior patterns reuse across Android apps can facilitate automatic apps comprehension to users or automatic creation of tutorials or usage guides. Nevertheless, how to automatically create an event mapping relation between similar apps and link all the mapped events together to form a continuous workflow for the target app is still a complex problem.

As mentioned above, events achieving similar functions are alike in their semantics, and the GUI model can chain events when migrating event sequences across apps. For this reason, this paper proposes a semantic-based event fuzzy matching strategy and pattern-oriented continuous workflow generation strategy to support behavior patterns reuse across Android apps.

Fig. 3 shows the overview of our approach, which consists of three modules: *User Behavior Pattern Miner*, *GUI Model Builder* and *User Behavior Pattern Reuser*. *User Behavior Pattern Miner* aims to mine user behavior patterns from traces obtained during users' interaction with the source app. Moreover, a sequential pattern mining algorithm is utilized to deal with traces and draw out behavior patterns. To assist in reusing user behavior patterns, *GUI Model Builder* dynamically builds a GUI model for the target app by depth-first exploration. *User Behavior Pattern Reuser* takes advantage of the source app's behavior patterns and the GUI model of the target app to generate continuous and executable workflows for the target app. In the rest of this section, we will describe each module in more detail.

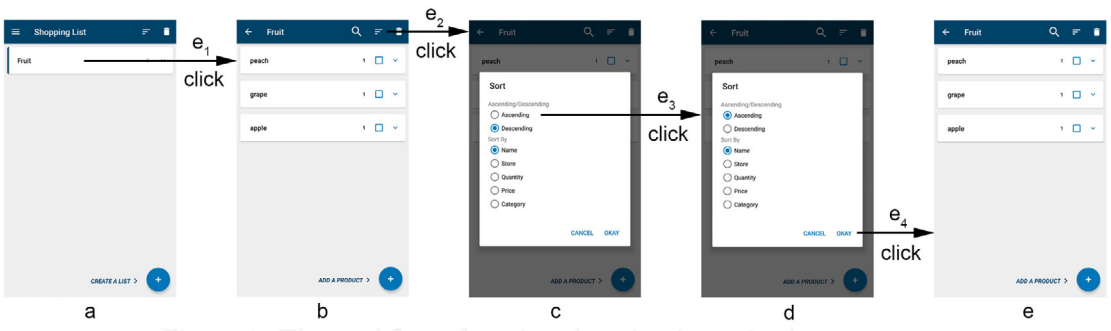

Fig. 1. The workflow of sorting shopping items in the source app *ShoppingList* 1.

Fig. 2. The workflow of sorting shopping items in the target app *ShoppingList* 2.

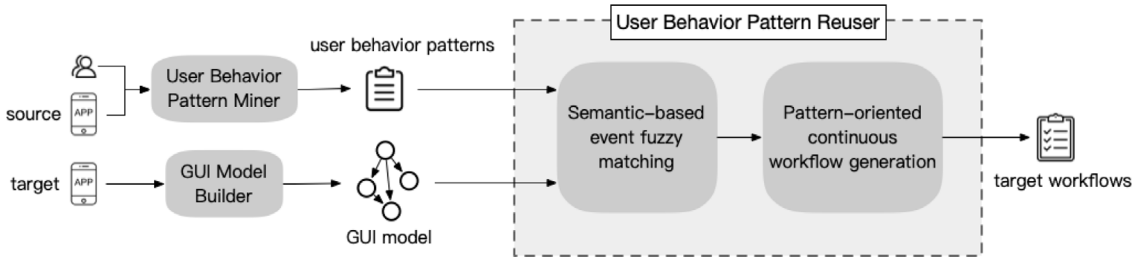

Fig. 3. Overview of our approach.

3.1. User behavior pattern miner

Before mining behavior patterns from traces, we first give the formal representation of traces and behavior patterns of Android apps reported in this paper. On this foundation, the traces collection and pattern mining methods are introduced.

3.1.1. Definitions of traces and behavior patterns

In this paper, Android apps' behaviors are captured as traces when users interact with the apps. The trace involves events and states that a user-triggered. Concretely, a user trace represents an access process performed by one user on a certain app, which can be defined as follows:

Definition 1 (*Trace*)**.** A user trace is composed of states and associated events triggered by users. That is, trace $T = \langle te_1, te_2, \ldots, te_n \rangle$, and a trace element te_i maps to a triple (s_i, e_i, s_{i+1}), where e_i is the i_{th} event recorded, and s_i and s_{i+1} are the app states before and after e_i triggered, respectively. In more detail, event e is a triplet $(type, widget, input)$, where *type* is the event type (In this work, we classify events into three types, namely *clickable*, *long-clickable* and *editable*), *widget* is the GUI widget on which e triggered and *input* is the text value enters by users. State s consists of the activity name and the UI hierarchy of the activity page.

User behavior patterns represent the frequent workflows mined from the concrete traces. To quantify the frequency of workflows, we first introduce a concept of **support** commonly used in patterns mining.

Definition 2 (*Support*)**.** A sequence $\alpha = \langle te_i, \ldots, te_j, \ldots, te_k \rangle$ is supposed to be a sub-sequence of trace T only if $i \geq 1$ and $k \leq n$, denoted as $\alpha \subset T$. Given a trace set $TS = \{T_1, T_2, \ldots, T_m\}$, the absolute support of the sequence α in TS is the number of traces in TS which contains α. And the relative support of α is the absolute support divided by |TS|, where |TS| refers to the total number of traces in TS.

Based on the definition of **support**, behavior patterns of Android apps can be defined as below:

Definition 3 (*Behavior Pattern*)**.** Given a minimum support *min_sup*, the behavior pattern is a frequent sequence α in trace set TS satisfying $support(\alpha) \geq min\_sup$.

3.1.2. Behavior patterns mining

Based on the definition of behavior patterns, this part discusses the behavior patterns mining approach. There are three steps for mining patterns, including traces collection, traces preprocessing, and patterns mining. The details are as below.

Traces collection is the foundation of behavior pattern mining in Android apps. Appium (Singh et al., 2014) is widely used to record and replay complex user actions. In this work, it is utilized to record users' interactions with the source app. Nevertheless, only the events can be captured, while the state information (i.e., activity) cannot be gotten by Appium. To record the states that users trigger, we record the test scripts of Appium and insert probes into test scripts before and after the triggering event. After that, the instrumented test scripts are replayed to obtain the triggered states and events of the source app, forming a trace set.

In order to facilitate the sequential pattern mining, trace elements in the trace set are preprocessed in advance. The trace preprocessing is to recognize similar trace elements and give them the same representation. In more detail, given two trace elements te_i and te_j, whether they are similar is determined by gradually comparing the event type, widget attributes, activity name and UI hierarchy of state. If te_i and te_j are recognized as similar trace elements, they will be assigned a same label. After all trace elements in the trace set are preprocessed, the trace set is transferred to a labeled trace set.

The user trace is essentially an event sequence. Thus, sequential pattern mining can be applied to identify the frequent workflow based on the labeled trace set. *CloFAST* (Fumarola et al., 2016) is a novel closed sequential pattern mining algorithm, which combines a new data representation of the sequence dataset with an one-step technique to fast count the support of sequences and efficiently mine closed sequential patterns. A closed sequential pattern refers to a sub-sequence with support greater than *min_sup*, and it is not a sub-sequence of any other sequential patterns with the same support. Thus, this paper chooses *CloFAST* to mine user behavior patterns from the labeled trace set. Besides, there may be an inclusion relation between two event sequences mined from traces. That is, one event sequence is a sub-sequence of another one when they are with different support. In this situation, it is unnecessary to keep both of them. Since the longest sequences represents a more complete user behavior pattern, this paper filters out the event sequences contained by other sequences and reserves the longest sequences as the behavior patterns for subsequent pattern reuse.

3.2. GUI model builder

As mentioned above, when reusing behavior patterns from the source app to the target app, the slight difference between applications may cause event sequences of behavior patterns to break in the target app. To alleviate this issue, this paper employs the target app's GUI model to make discrete mapped events linked together.

Since behavior patterns reuse across apps concern the reproduction of events and states, the GUI model of target apps only needs to focus on events and states. Hence, the GUI model of an Android app can be defined as follows:

Definition 4 (*GUI Model*)**.** Essentially, the GUI model is a Finite State Machine (FSM) which can be represented by a tuple (S, s_0, F, E, T). S is the set of states which refers to activity pages of the app, $s_0 \in S$ is the initial state and $F \subset S$ is the set of final states. E is the set of events. T is the set of transitions, and each transition $t \in T$ takes the form of (s_s, e, s_t), where $s_s, s_t \in S$ are the source and target state and $e \in E$ is the event.

According to this definition, the GUI model of target apps is constructed through dynamic exploration automatically. In more detail, the target app starts with the initial state s_0. In the current state, all available events are extracted and stored in a stack. Then, the event at the top of the stack is popped and triggered. If a new state is reached, a new state s_i and a new transition t_j associated with the event and the states are created and added to the GUI model. Meanwhile, the acquirable events on the state s_i are identified and pushed into the stack. Otherwise, only a new transition t_k associated with the event and the existing states is created and added to the model. Repeat the process above until all events in the stack are triggered.

Notably, by observing Android apps' behavior, it can be found that all input widgets on a GUI are logically associated. Moreover, most editable events binding the input widgets need to be followed by a clickable event to jump to another state. In this situation, the editable events should be combined with the clickable event so as to explore more states and reduce self-loops states as well as the unnecessary transitions. Thus, in this paper, *composite events* are raised to organize such events above, thereby compressing the state space of the GUI model. In contrast with *composite events*, *atomic events* refer to single clickable events and long-clickable events.

The discrimination strategy for *composite events* and *atomic events* is as below. At first, all input widgets are distinguished and extracted from the state. Then, the context information related to the input widgets is obtained. That is, the data type and semantic information, such as "username," "password", "mobile phone number", and other text information, are identified for the input widgets. Then, according to the keywords associated with clickable events, such as "submit", "confirm", and "register", search the relevant click widget related to these input widgets. Finally, the corresponding *composite events* are distinguished according to the input widgets and click the widget. Moreover, all events except *composite events* are recognized as *atomic events*.

3.3. User behavior pattern reuser

Based on the behavior patterns mined from the source app and GUI model of the target app, the *User Behavior Pattern Reuser* module aims to reproduce the patterns for the target app through generating executable and continuous target workflows. It consists of two phases, the first one is to create a mapping relation between the events of the source app and target app based on the semantic-based event fuzzy matching method (SEFM). And the second one is to generate a continuous workflow chaining the mapped events for the target app with the help of its GUI model. The details of this module will be explained in the rest of this section.

3.3.1. Semantic-based event fuzzy matching

To reuse user behavior patterns from the source app to the target app, we need to find events in the target app corresponding to the source app's behavior patterns. As mentioned above, the subtle disparity of the GUI between similar applications affects event mapping. Nevertheless, in general, events achieving the same functionality are semantically similar. Thus, this paper proposes a semantic-based event fuzzy matching (SEFM) strategy to map events across Android apps.

Next, we first analyze the constituents of events and then, based on them, give a semantic-based event similarity measurement and a fuzzy matching strategy to estimate the similarity between events, thereby achieving events mapping.

Event analysis. To match the event e_s of the source app to the event e_t of the target app, the semantic information of events is considered. In more detail, an event of Android apps can be represented as a triplet (*type*, *activity*, *attrs*), where *type* represents the event type, *activity* represents the activity on which the event triggered, and *attrs* represents attributes of the widget binding the event.

In terms of the semantic of events, they may come from attributes *attrs*, such as id, text, content description, etc. For the *id*

attribute, developers usually prefer assigning a specific meaning to the id of a widget, reflecting its functionality. Thus, it can signify the semantics of events. For the *text* and *content description*, the former is to show on the widget, and the latter aims to help users understand the functionality of the widget. And if a widget has neither *text* nor *content description*, the description information may be retrieved from its surrounding widgets within a certain kinship. In addition, the *hint* is always assigned to editable widgets for displaying the prompt message about input. For widgets with images, the image content is generally reflected in the *filename* of the image, which should also be considered.

Overall, the *id*, *text*, *content description*, description from the surrounding widgets, *hint* and the *filename* of images of the widget binding the event can imply the semantics of the event.

Semantic similarity of events' attributes. As mentioned above, one event may correspond to multiple attributes representing its semantics. Based on them, the semantic similarity between events is measured.

Concretely, for a pair of events (e.g. the source event e_s and target event e_t) to be matched, their attributes relevant to semantics can be expressed as $attrSet(e_s) = \{ea_1^s, ea_2^s, \ldots, ea_m^s\}$ and $attrSet(e_t) = \{ea_1^t, ea_2^t, \ldots, ea_n^t\}$, respectively, where both ea_i^s and ea_j^t consist of the attribute and its value, expressed by $(attr : val)$. To measure the semantic similarity between e_s and e_t, the similarity between attributes (e.g. ea_i^s and ea_j^t) from $attrSet(e_s)$ and $attrSet(e_t)$ should be measured at first.

Given two attributes ea_u^s of source event e_s and ea_v^t of target event e_t, their semantic similarity can be computed by text comparability. That is, ea_u^s and ea_v^t are regarded as natural language and preprocessed, including tokenization, lemmatization, and stopword removal. After that, two token lists for ea_u^s and ea_v^t are obtained, expressed by tl_u^s and tl_v^t, where $tl_u^s = \{tok_1^s, tok_2^s, \ldots, tok_p^s\}$, $tl_v^t = \{tok_1^t, tok_2^t, \ldots, tok_q^t\}$, and tok_i^s and tok_j^t represent tokens involved in tl_u^s and tl_v^t. To measure the semantic similarity between tl_u^s and tl_v^t, the Word2Vec model that is specific to the word vector representation for the Android domain trained by Behrang and Orso (2019), is utilized to translate tokens to word vectors and compute the cosine similarity between word vectors as the semantic similarity of tokens. In particular, the cosine similarity of all token pairs from tl_u^s and tl_v^t are computed. Then, the highest value is taken, and the pair of tokens for this highest value is "linked". Meanwhile, these tokens are removed from their token lists, and the next highest value is taken. Repeat the process until either tl_u^s or tl_v^t is empty. After all the best matched token pairs are distinguished, the average of their cosine similarity is taken as the semantic similarity of tl_u^s and tl_v^t. Namely, the semantic similarity of the attribute ea_u^s and ea_v^t.

Event fuzzy matching strategy. Based on the semantic similarity between the attributes, the semantic similarity between the source event e_s and target event e_t can be calculated. As mentioned above, widgets corresponding to similar events are similar in the semantic of attributes. But attributes representing the same semantics may come from different sources. For example, a widget *TextView* leverages the *id* to indicate its function while the matched widget *ImageView* uses the *filename* to implies its function. To identify such similar events, in this paper, we propose an event fuzzy matching strategy, which considers cross-pairing between different attributes instead of the one-to-one pairing of the same attribute.

In more detail, to compute the semantic similarity between events e_s and e_t, each attribute ea_i^s in $attrSet(e_s)$ are compared with all attributes of e_t (i.e. $attrSet(e_t)$), and the highest similarity between attributes (e.g. ea_i^s and ea_k^t) is regarded as the similarity between the attribute ea_i^s and attributes of e_t. And the average of all the highest similarity between the attributes of e_s and that of e_t is taken as the semantic similarity from e_s to e_t, expressed by $sim(e_s \rightarrow e_t)$. Likewise, each attribute ea_j^t from the $attrSet(e_t)$ is compared with all attributes of e_s (i.e. $attrSet(e_s)$), and the highest similarity between attributes (e.g. ea_j^t and ea_l^s) is regarded as the similarity between the attribute ea_j^t and attributes of e_s. And the average of all the highest similarity between the attributes of e_t and that of e_s is taken as the semantic similarity from e_t to e_s, expressed by $sim(e_t \rightarrow e_s)$. After that, the semantic similarity between e_s and e_t (i.e. $sim_{semt}(e_s, e_t)$) is the average of $sim(e_s \rightarrow e_t)$ and $sim(e_t \rightarrow e_s)$.

To better understand the SEFM strategy, take the click events on two ImageButtons at the bottom of activity a in Fig. 1 and activity a' in Fig. 2, namely add_1 and add_2, as an example. The *id* of the widget binding add_1 is "fab_new_list", the image *filename* is "ic_input_add", and the description from a surrounding widget is "CREATE A LIST". For the widget binding add_2, its *id* is "fab" and the image *filename* is "ic_note_add_white_24dp".

Assume that add_1 is the source event and add_2 is the target event. Firstly, the attributes of these two events are extracted and preprocessed. The attributes set of add_1 (i.e. $attrSet(add_1)$) is $\{ea_1^s = (id : fab\_new\_list), ea_2^s = (fn : ic\_input\_add), ea_3^s = (sd$:CREATE A LIST)$\}$, that of add_2 (i.e. $attrSet(add_2)$) is $\{ea_1^t = (id : fab), ea_2^t = (fn : ic\_note\_add\_white\_24dp)\}$. After tokenization, lemmatization and stopword removal, the token lists of each attribute in $attrSet(add_1)$ are tl_1^s = ["floating", "action", "button", "new", "list"], tl_2^s = ["add", "input"] and tl_3^s = ["create", "list"], while the token lists of each attribute in $attrSet(add_2)$ are tl_1^t = ["floating", "action", "button"] and tl_2^t = ["add", "note"], respectively.

Then, the semantic similarity between any two attributes of add_1 and add_2 is computed by the cosine similarity of tokens extracted from attributes. Take attribute *filename* of add_1 and add_2 as an example, whose token lists are tl_2^s and tl_2^t, respectively. The semantic similarity between token pairs is shown in matrix $M_{ea_2^t}^{ea_2^s}$. It can be seen from this matrix, the best matched token pairs in tl_2^s and tl_2^t are ("*add*", "*add*") and ("*input*", "*note*") with cosine similarity of 1.0 and 0.1107. Thereby, the semantic similarity between *filename* of add_1 and *filename* of add_2 is $(1.0 + 0.1107)/2 = 0.5554$.

Considering the fuzzy matching strategy, the *filename* of add_1 should be also compared with the *id* of add_2, whose token lists are tl_2^s and tl_1^t. The semantic similarity between token pairs is shown in matrix $M_{ea_1^t}^{ea_2^s}$. The best matched token pairs are ("*add*", "*button*") and ("*input*", "*floating*") with cosine similarity of 0.3899 and -0.0304. So, the semantic similarity between *filename* of add_1 and *id* of add_2 is $(0.3899 + -0.0304)/2 = 0.1798$.

$$\mathbf{M}_{\mathbf{ea_2^t}}^{\mathbf{ea_2^s}} = \begin{array}{c} tl_2^s \setminus tl_2^t \\ add \\ input \end{array} \begin{array}{c} \begin{array}{cc} add & note \end{array} \\ \begin{pmatrix} \mathbf{1.0} & 0.3502 \\ -0.0447 & \mathbf{0.1107} \end{pmatrix} \end{array}$$

$$\mathbf{M}_{\mathbf{ea_1^t}}^{\mathbf{ea_2^s}} = \begin{array}{c} tl_2^s \setminus tl_1^t \\ add \\ input \end{array} \begin{array}{c} \begin{array}{ccc} floating & action & button \end{array} \\ \begin{pmatrix} 0.0776 & 0.0931 & \mathbf{0.3899} \\ \mathbf{-0.0304} & -0.0577 & 0.0745 \end{pmatrix} \end{array}$$

Based on the semantic similarity between any two attributes from $attrSet(add_1)$ and $attrSet(add_2)$, the similarity matrices between events add_1 and add_2 are as follows, where matrix $M_{e_t}^{e_s}$ is the similarity matrix from add_1 to add_2 and matrix $M_{e_s}^{e_t}$ is the similarity matrix from add_2 to add_1.

$$\mathbf{M}_{\mathbf{e_t}}^{\mathbf{e_s}} = \begin{array}{c} add_1 \setminus add_2 \\ id \\ fn \\ sd \end{array} \begin{array}{c} \begin{array}{cc} id & fn \end{array} \\ \begin{pmatrix} \mathbf{1.0} & 0.5111 \\ 0.1798 & \mathbf{0.5554} \\ \mathbf{0.8403} & 0.2901 \end{pmatrix} \end{array}$$

$$\mathbf{M}_{\mathbf{e_s}}^{\mathbf{e_t}} = \begin{matrix} add_2 \setminus add_1 \\ id \\ fn \end{matrix} \begin{matrix} id & fn & sd \end{matrix} \begin{pmatrix} \mathbf{1.0} & 0.1798 & 0.8403 \\ 0.5111 & \mathbf{0.5554} & 0.2901 \end{pmatrix}$$

From these matrices, we can see that the similarity from add_1 to add_2 (i.e. $sim(add_1 \rightarrow add_2)$) is calculated as $(1.0 + 0.5554 + 0.8403)/3 = 0.7986$, while that from add_2 to add_1 (i.e. $sim(add_2 \rightarrow add_1)$) is calculated as $(1.0 + 0.5554)/2 = 0.7777$. Thus, the semantic similarity between add_1 and add_2 (i.e. $sim_{semt}(add_1, add_2)$) is $(0.7986 + 0.7777)/2 = 0.7882$.

Besides the semantic similarity of events, we also take (1) the event *type* and (2) the name of *activity* on which the event triggered, into consideration to measure the similarity between events. This is because the event type could distinguish similar events in coarse granularity; in other words, events of a different type are intuitively not similar, which is especially reflected between *editable* events and other events. Moreover, the activity name generally reflects certain functionality of the application, and events on totally different activities are not similar to a great extent.

Thus, the similarity between two events can be computed as below. Given an event pair (e_s, e_t), the similarity between them is defined as shown in Eq. (1):

$$sim_{evt}(e_s, e_t) = \begin{cases} \alpha \cdot sim_{semt}(e_s, e_t) + \beta \cdot sim_{act}(e_s, e_t), & \text{if } type_s = type_t \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $type_s$ and $type_t$ are the event type of e_s and e_t, respectively. $sim_{semt}(e_s, e_t)$ is the attributes' semantic similarity between e_s and e_t. And $sim_{act}(e_s, e_t)$ is the activity name's semantic similarity between e_s and e_t. α and β are the weights of sim_{semt} and sim_{act}, and the sum of α and β is equal to 1.

For each source event in the user behavior pattern, SEFM computes the event similarity between it and all available events in the target app, and then takes such events whose similarity exceeds the preset threshold as the candidates. Based on the candidate events, a pattern-oriented continuous workflow generation approach is raised to link them for the target app.

3.3.2. Pattern-oriented continuous workflow generation

According to the candidate events matched with source events in behavior patterns, the *workflow generation* module tries to generate an executable workflow for the target app by chaining the discontinuous candidates with the GUI model. More specially, based on the behavior patterns, the candidate events in the GUI model are extended to the target workflow one by one by using the backward search. Moreover, since candidate events in the GUI model are discrete, there may be multiple candidate paths concatenating two ordered events together. To further improve the efficiency of target workflow generation, a candidate path selection strategy is raised to guide the search process.

In what follows, we will detail the candidate path selection strategy, and based on it, introduce the target workflow generation method.

Candidate path selection strategy. Based on the SEFM strategy, for a source event in behavior patterns, events in the target app whose similarity exceeds the preset threshold are taken as its candidate events. As mentioned above, candidate events corresponding to two ordered source events may be incontinuous in the target app, such as e'_1 and e'_2 of *ShoppingList*2 shown in Fig. 2. In order to generate the executable workflow for the target app, it is necessary to find a partial path between the two ordered candidate events to chain them in series. Particularly, there may be several partial paths, named candidate paths, between ordered candidate events. This can boil down to two reasons. One is that a source event in the behavior pattern may correspond to multiple candidate events. The other is that there may be several paths reaching a specific candidate event from the current state of the target workflow. Thus, how to select the best candidate path closest to the behavior pattern to expand the target workflow is a critical problem.

For example, Fig. 4 is a partial GUI model of a certain target app. Assume that the target workflow twf is in a state at some point, e.g., *cur* in the GUI model. And there are two candidate events matching the next source event, e.g. $cadEvt_1$ and $cadEvt_2$, which will be used to extend the twf. Obviously, $cadEvt_1$ and $cadEvt_2$ cannot be added to the twf directly, since their source states are not state *cur*. So there is a need to generate partial paths linking the *cur* to candidate events $cadEvt_1$ and $cadEvt_2$. These kinds of partial paths are candidate paths. More specially, for $cadEvt_1$, there are two candidate paths in the GUI model of target application, which are $cadPath_1 = \langle cur, e_1, s_1, e_2, s_2, e_3, s_3, e_4, s_4, cadEvt1\rangle$ and $cadPath_2 = \langle cur, e_1, s_1, e_5, s_4, cadEvt1\rangle$. For $cadEvt_2$, its corresponding candidate path $cadPath_3$ is $\langle cur, e_6, s_6, cadEvt_2\rangle$. But which candidate path should be chosen to extend the twf?

As mentioned in 3.3.1, the candidate event, whose semantic is closer to the source event's semantic, should be selected preferentially as the target event. On the other hand, since workflows achieving the same functionality are proximal in the number of steps (events) involved, the candidate path from the current state to the candidate event should be as short as possible. So, when expanding the target workflow, the candidate path selection depends on not only the semantic similarity of events but also the length of candidate paths.

Therefore, this paper proposes a candidate path priority rule considering the semantic similarity of events and the length of candidate paths. The priority score of a candidate path is defined as Eq. (2):

$$prio(cadPath) = \begin{cases} sim_{evt}(srcEvt, cadEvt), & \text{if } |cadPath| \leq 2 \\ \frac{sim_{evt}(srcEvt, cadEvt)}{1+\lg|cadPath|}, & \text{otherwise} \end{cases} \quad (2)$$

where $cadEvt$ is a candidate event matching the source event $srcEvt$ in the behavior pattern, *cadPath* is a candidate path reaching $cadEvt$ from the current state of the target workflow, and $|cadPath|$ is the length of *cadPath*. *lg* refers to logarithm. $sim_{evt}(srcEvt, cadEvt)$ is the similarity between $srcEvt$ and $cadEvt$.

From this formula, we can see that if *cadPath* is within a certain length, the event similarity $sim_{evt}(srcEvt, cadEvt)$ directly determines the priority score. Otherwise, the event similarity is penalized by the length of the candidate path, forming the priority score. This is because that for the shorter *cadPath*, the similarity between $srcEvt$ and $cadEvt$ is critical, while for the longer *cadPath*, the path length has an adverse influence on the priority of candidate paths.

After computing the priority score for all candidate paths, we sort them from highest to lowest based on their priority and then execute them in turn to verify their feasibility. The first executable path is selected to be a segment of the target workflow.

Continue the above example in Fig. 4, assume that $cadEvt_1$ has higher similarity than $cadEvt_2$. The priority score of $cadPath_1$, $cadPath_2$ and $cadPath_3$ is computed and compared as below. For $cadPath_1$ and $cadPath_2$, both of them aim to link the same event $cadEvt_1$, while the length of $cadPath_1$ exceeds 2 and that of $cadPath_2$ is within 2. The priority score of $cadPath_1$ is penalized by its length. Thus, the priority score of $cadPath_2$ is higher than that of $cadPath_1$. For $cadPath_2$ and $cadPath_3$, the length of them is within 2. So, the event similarity decides the priority score of

$cadPath_2$ and $cadPath_3$. As $cadEvt_1$ is more similar with the source event than $cadEvt_2$, the priority score of $cadPath_2$ linking $cadEvt_1$ is higher than that of $cadPath_3$ linking $cadEvt_2$. Thus, $cadPath_2$ has the highest priority. If $cadPath_2$ is executable, it will be added to the target workflow. Otherwise, we further compare the priority score of $cadPath_1$ and $cadPath_3$, and the one with higher priority will be selected to extend the workflow.

Target workflow generation by backward search. Based on behavior patterns, the target workflow is generated from the GUI model by the backward search for the target app, guided by the candidate path selection strategy.

In more detail, for an event sequence in behavior patterns from the source app, firstly, one source event is mapped to some candidate events of the target app by the SEFM strategy. Then, the candidate paths reaching candidate events from the current state of the target app are identified and evaluated by the priority rule. And the candidate path with the highest priority score is chosen to extend the target workflow. Next, this candidate path is executed. If it is executable, the next candidate event corresponding to the next source event will be handled. Otherwise, it will be abandoned, and another candidate path is selected. In extreme cases, if all the candidate paths are infeasible, the current candidate events go back to the last set of candidate events, and another candidate path reaching the last candidate event set is selected. Repeat this process until all events in the behavior pattern are traversed.

3.3.3. The algorithm of user behavior pattern reuse

The core idea of our behavior pattern reuse approach is to create an event mapping from the source app to the target app by the *SEFM* strategy and generate a continuous workflow for the target app based on the candidate path selection strategy. The pseudo-code is summarized in Algorithm 1.

Given (1) a user behavior pattern (P) of source app and (2) a GUI model (G) of target app as input, the algorithm first matches events in P with that of the target app and then generates the target workflow twf chaining the mapped events for the target app with the GUI model.

To be more specific, in *event fuzzy matching* phase, firstly, all available events (E) are extracted from the GUI model of the target app (Line 3). For each source event $srcEvt$ in P, the similarity between $srcEvt$ and all events in E is calculated based on the formula (1). And if the similarity between $srcEvt$ and event e in E exceeds the threshold value, then e is added to the candidate event set $candEvtSet$. After all events in E are compared with $srcEvt$, a candidate event set $candEvtSet$ is obtained which stores the candidate events similar to $srcEvt$ (lines 7–13). After all source events in P are traversed, all candidate event sets are identified and stored in a list *lces* (Lines 4–15).

Based on the *lces*, the target workflow is generated through linking candidate events in *lces* sequentially. In the *target workflow generation* phase, firstly, the state of *curState* is initialized as s_0 of the GUI model (Line 17), and the target workflow twf is set to an empty list (Line 18). Then, a recursive algorithm of generating target workflow (i.e., *GTWF*) is designed to generate twf for the target app, as detailed in Algorithm 2.

Concretely, the GTWF function has five parameters, which are twf, *index*, *backtrackIndex*, $invalidPathSet$ and $originalTwf$. Among them, twf is the target workflow to be generated, *index* is the index of the current candidate event set in *lces*, *backtrackIndex* is the index of the candidate event set where backtracking occurs, $invalidPathSet$ is a set of invalid paths, and $originalTwf$ is the original target workflow before backtracking occurs. The termination condition of the GTWF function is that *index* equals the size of *lces*, which means that all mapped events have been connected. At this time, the finally generated target workflow twf is returned (Lines 2–3).

The detailed generation process is as follows. Firstly, for the current candidate event set $candEvtSet$ in *lces*, all candidate paths from the *curState* to the source state of every event in $candEvtSet$ are collected (Lines 5–15). That is, for each candidate event $candEvt$ in $candEvtSet$, its source state is identified and regarded as the destination state *destState* of the current state *curState* (Lines 7–8). If *curState* and *destState* are the same state, the candidate path is a path directly linking the $candEvt$ from the *curState* (Lines 9–10). Otherwise, the GUI model G is used to generate paths linking the *curState* and *destState* by depth-first search (Line 12). And the candidate paths *candPaths* are added to the set *candPathSet* (Line 14). After that, all candidate paths except paths in $invalidPathSet$ are evaluated by the priority rule and sorted from highest to lowest based on the priority score (Lines 14–17). Then paths in *sortedcandPaths* are executed in turn, and the first executable path is obtained (Line 18). If such executable path *path* exists, it is used to extend the target workflow twf, and meanwhile, the *index* increases by 1 (Lines 19–21). Otherwise, namely, when all alternative paths cannot expand the current twf, the backtrack occurs (Lines 23–35). The termination condition of the backtrack is that *index* rolls back to 1, meaning that no feasible paths can reach the candidate event set where the backtrack starts. At this time, this candidate event set is skipped, and the twf is restored to its original state $originalTwf$ (Lines 23–25). When *index* is greater than 1, we backtrack to the last candidate event set by reviving the twf to the previous state and invalidating the last path *lastPath*. If *index* is greater than *backtrackIndex*, meaning that a new backtrack point has been encountered, then the current twf is recorded to $originalTwf$, and the new *backtrackIndex* is modified to *index* (Lines 27–33). Subsequently, the *index* decreases by 1 (Line 34). At last, the new values of parameters are passed to the next recursive call of *GTWF*.

Algorithm 1 User behavior pattern reuse

Input: user behavior pattern P of source app, GUI model G of target app
Output: target workflow twf

```
1: //Event fuzzy matching
2: lces = List<> //store candidate events sets of P in order
3: E = getAvailableEvents(G)
4: for i = 1 to P.size() do
5:     srcEvt = P.get(i)
6:     candEvtSet = ∅
7:     for j = 1 to E.size() do
8:         e = E.get(j)
9:         simEvts = sim(srcEvt,e)// compute the event similarity
10:        if simEvts>threshold then
11:            candEvtSet.add(e)
12:        end if
13:    end for
14:    lces.add(candEvtSet)
15: end for
16: //Target workflow generation
17: curState = s0 //s0 is the initial state of G
18: twf = List<>
19: return GTWF(twf, 1, 1, ∅, twf)
```

Algorithm 2 Target workflow generation

```
1: function GTWF(twf, index, backtrackIndex, invalidPathSet, originalTwf)
2:     if index == lces.size() then
3:         return twf
4:     end if
5:     candEvtSet = lces.get(index)
6:     for each candEvt ∈ candEvtSet do //collect candidate paths
7:         curState = getCurrentState(twf)
8:         destState = getSourceState(candEvt)
9:         if isSameState(curState, destState) then
10:            candPaths = directpath
11:        else
12:            candPaths = getLeadingPaths(curState, destState, G)
13:        end if
14:        candPathSet.add(candPaths)
15:    end for
16:    candPathSet= candPathSet \ invalidPathSet //delete the invaild paths
17:    sortedCandPaths = sortByPathPriority(candPathSet)
18:    path = getFeasiblePath(sortedCandPaths)
19:    if path ! = null then
20:        twf = twf.add(path)
21:        index = index + 1
22:    else
23:        if index == 1 then // backtrack to the candidate event set
24:            index = backtrackIndex + 1 // skip the candidate event set where backtrack occurs
25:            twf = originalTwf
26:        else
27:            lastPath = getLastPath(twf)
28:            twf = twf.delete(lastPath)
29:            invalidPathSet = invalidPathSet ∪ lastPath
30:            if index > backtrackIndex then
31:                originalTwf = twf
32:                backtrackIndex = index
33:            end if
34:            index = index - 1
35:        end if
36:    end if
37:    return GTWF(twf, index, backtrackIndex, invalidPathSet, originalTwf)
38: end function
```

4. Empirical evaluation

In order to verify the validity of our user behavior pattern mining and reuse approach, we conduct experiments on five categories of similar Android apps. Moreover, four research questions are raised below.

RQ1. Do behavior patterns mined from traces by our method represent the typical functionality of similar apps?
RQ2. How effective is our SEFM strategy in event matching across similar apps? And how much influence does the fuzzy strategy have on event matching?
RQ3. Can the continuous workflow generation method create effective workflows for the target apps?
RQ4. Is our pattern reuse approach correctly transferring behavior patterns of source apps to target apps?

4.1. Evaluation metrics

To evaluate the effectiveness of our behavior pattern reuse method, we propose a series of metrics. In more detail, to measure our SEFM strategy, the matching accuracy of events is measured from different aspects as below. The straightforward metric is the correctly matched events *CM* (i.e., true positives, *TP*), which refers to the ratio of the correct counterparts in matched candidate event sets of the target app to source events in the behavior patterns. On the other side, those events that are not successfully matched can be classified into three categories. The first one is the incorrectly matched events *ICM* (i.e., false positives, *FP*), which indicates the percentage of source events in the behavior pattern that are matched to the wrong events in the target app. In fact, the source events may not have a counterpart in the target app. The second one is the incorrectly unmatched events *UM* (i.e., false negatives, *FN*), which means the percentage of source events in the behavior pattern that have a counterpart in the target app but are not matched to any event. And the third one is the correctly unmatched events *UM*! (i.e., true negatives, *TN*), representing the percentage of source events in the behavior pattern that are not matched to any event, and this is largely because that they have no counterpart in the target app.

Moreover, whether our continuous workflow generation method creates effective workflows can be analyzed from the reuse of behavior patterns and the coverage of target workflows. For the reuse of patterns, it reflects in the percentage of completely reused behavior patterns *CR* and the percentage of partially reused ones *PR*. Particularly, *CR* means the target workflows reproducing all events in behavior patterns, while *PR* indicates the target workflows failing to reproduce some events in behavior patterns that have a counterpart in the target app. In terms of the generated target workflows, whether they can cover staple states of the target app reflects our behavior pattern reuse method's effectiveness. Betweenness centrality *BC* is one of the most commonly used centrality measures, which is formally defined for the first time in Freeman (1977). In a general network consisting of vertices and edges, a vertex holding a high *BC* value has the potential to control the communication taking place between vertices. Correspondingly, in the Android app's GUI model, we think that a state *s* having higher *BC* is more likely to be triggered. In other words, state *s* is more important. Thus, this paper adopts the *BC* of a state *s* to express its centrality (importance), which is defined as the total fraction of the shortest paths between arbitrary state pairs that use or pass through *s*. To evaluate the target workflows, the *BC* of all states on the GUI model are calculated, and states are sorted by *BC* from highest to lowest. Furthermore, these states are divided into three levels according to their importance. The range of $level_1$ is the top 20% of states, the range of $level_2$ is the top 20% to 50%, and the remaining states are divided into $level_3$. If the generated target workflows cover more important states belonging to the preceding levels, then it demonstrates that our behavior pattern reuse method is effective in figuring out typical behaviors of new apps.

Besides, to verify whether our behavior pattern reuse approach can correctly transform behavior patterns of the source app into event sequences of the target app, the human experience is used to measure its effectiveness. In particular, we surveyed three postgraduates and asked them to mark each reused event sequence. Their feedback may be ✓, × and —, where ✓means that the pattern is correctly translated, × is that the pattern is incorrectly translated, and — represents they are not sure.

Table 1
Android apps used in the study.

Category	App name	Version	LOC	Source
Shopping list	S1 - Shopping list	v1.0.1	7.1K	Google Play
	S2 - Shopping list	v1.2.3	5.2K	Google Play
	S3 - Shopping list	v1.0.8	18.9K	Google Play
	S4 - OI shopping list	v1.7.0.5	32.3K	Google Play
	S5 - Fast shopping	v1.3.50	11.5K	F-Droid
Note taking	N1 - Swiftnotes	v3.1.4	4.8K	Google Play
	N2 - Note now	v2.8	6.1K	Google Play
	N3 - Notepad	v1.12	2.7K	F-Droid
	N4 - Notepad	v1.06	2.3K	F-Droid
	N5 - uNote	v1.5.1	4.3K	F-Droid
Expense tracker	E1 - Easy budget	v1.5.2	14.1K	Google Play
	E2 - Money tracker	v2.1.3	12.6K	Google Play
	E3 - Budget watch	v0.21.5	12.9K	F-Droid
	E4 - Pro expense	v1.0.0	16.8K	F-Droid
	E5 - DailyBudget	v2.0	5.8K	F-Droid
Weather	W1 - Geometric weather	v2.608	77.6K	Google Play
	W2 - Forecastie	v1.5	8.5K	Google Play
	W3 - Good weather	v4.4	9.7K	Google Play
	W4 - World weather	v1.2.5	19.7K	Google Play
	W5 - Weather	v2.4.2	25.6K	F-Droid
Todos management	T1 - Clear list	v1.5.6	15.7K	F-Droid
	T2 - Minimal	v1.2	3.4K	F-Droid
	T3 - Minitask	v1.0	38.0K	F-Droid
	T4 - Simple Todo	v1.1	3.5K	F-Droid
	T5 - Fire Todo	v1.5	3.3K	Google Play

4.2. Experimental subjects and design

Our behavior pattern reuse approach is implemented with Python, and it is precisely for Android apps. According to the suggestion of Behrang and Orso (2019), we select five categories of Android apps from Google Play or F-Droid to evaluate the effectiveness of our approach. In addition, whether the apps are commonly used and open source is still considered when selecting experimental subjects. These categories are *Shopping List*, *Note Taking*, *Expense Tracker*, *Weather*, and *Todos Management*, respectively. And each category provides a standard set of features. For each category, we select five open source applications. One of them is chosen randomly to be the source app, and the others are the target apps.

Table 1 shows the details of Android apps, including the category, app name with ID, version, size (LOC) and the source of each app. In our experiment, we choose $S1, N1, E1$ and $W1$ as the source apps of each category. Furthermore, to gather user traces for the source apps, we ask five masters to interact with the apps using Appium. These five masters are not participants in this study, and their interactions with apps are based entirely on their daily habits of using apps in the same category. In addition, the minimum relative support is set to 0.1 since the size of our trace set is relatively small, and the event similarity threshold in our experiments is set to 0.5. The parameter α and β in the event similarity formula are set to 2/3 and 1/3, respectively. In order to look for these appropriate parameter values, we conducted experiments on different parameter values. Take the minimum relative support as an example. We first set it to 0.3 and then gradually set it to 0.2 and 0.1. It is found that the user behavior patterns mined with the minimum relative support of 0.1 are better in terms of number, length, and functional representation. Then we increase and decrease the minimum relative support in the step of 0.01 around 0.1 and conduct experiments. Finally, the conclusion is that 0.1 is the most appropriate minimum relative support.

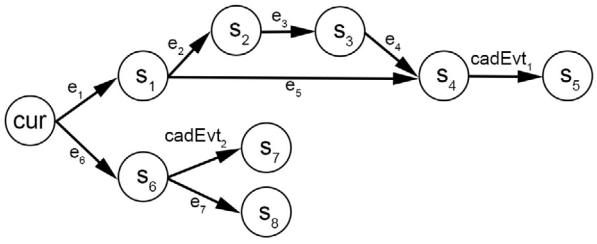

Fig. 4. Example of candidate events and corresponding candidate paths.

4.3. Experimental results and analysis

4.3.1. Results for RQ1

To evaluate the effectiveness of our behavior patterns mining method, we further analyze the functionality involved in the patterns and verify whether they can represent the typical workflows of Android apps. Table 2 shows the results of behavior patterns mined from user traces for the five source apps. This table reports the number of user traces of each app (#traces), the number of events in user traces (#evtsTraces), the number of user behavior patterns mined from user traces (#patns), the number of events in these patterns(#evtsPatns), and the typical functionality (Typical functionality) we summarized from these patterns.

As can be seen from Table 2, for five source apps $S1$, $N1$, $E1$, $W1$ and $T1$, there are 13, 5, 8, 14, and 8 behavior patterns mined from the recorded traces, respectively. Taking the source app $S1$ as an example, through analyzing 13 patterns involved in this app, we can summarize 7 typical functionalities they represent. They are adding, deleting, and editing operations to a specific list or item, respectively, as well as sorting items of a list. Obviously, these functionalities are ubiquitous in shopping list apps, thus representing the typical functionality of such a category of apps to a certain extent. The other three categories of apps' experimental results are the same, confirming that the behavior patterns do represent the typical functionality of the same category of apps.

Notably, the number of typical functionalities does not equal the number of patterns because there is a common situation that multiple patterns achieve the same functionality, even if the event sequences contained in patterns have a subtle difference. For example, there are two ways to delete an item in $S1$. One is to long-click the item and click the delete operation, while the other is that check the item, open the options menu, and delete the checked item. Both of them are common behavior patterns to achieve the functionality of deleting an item.

4.3.2. Results for RQ2

To evaluate the effectiveness of our SEFM strategy in the event mapping phase, we analyzed the event matching results with and without the fuzzy strategy, respectively. The results are listed in Table 3. Note that the benchmark of event mapping across apps is obtained through manual analysis. That is, we manually find the correct counterparts on the target app for each source event in patterns or determine that the source event has no counterpart on the target app. In Table 3, the first and second columns show the ID of the source and target apps, respectively. Column 3 and column 4 show the percentage of correctly matched events (CM) and incorrectly matched events (ICM), while column 5 and column 6 shows the percentage of unmatched events with a counterpart (UM) and with no counterpart ($UM!$) in the target app.

As Table 3 shows, for the *Shopping List*, *Note Taking*, *Expense Tracker*, *Weather* and *TodosManagement* categories, our method correctly matches (CM) 72.7%, 55.8%, 51.1%, 54.8% and

Table 2
Results of behavior patterns mined from traces.

App	#traces	#evtsTraces	#patns	#evtsPatns	Typical functionality
S1	35	274	13	65	1. add list 2. delete list 3. add item 4. delete item 5. edit the list information 6. edit the item information 7. sort items
N1	41	278	5	26	1. add note 2. delete note 3. search note and view it 4. edit and favorite note
E1	40	278	8	46	1. tutorial page 2. add expense 3. add income 4. edit expense 5. delete expense 6. delete income 7. set low balance warning threshold
W1	35	293	14	84	1. search for a new location 2. set the data refresh interval 3. set the temperature unit 4. set the distance unit 5. set the pressure unit 6. set the wind speed unit 7. view the about information 8. set the theme of the app
T1	35	259	8	36	1. choose list category 2. view task 3. add task 4. mark a task as completed and delete it 5. add list category

43.8% of source events in patterns with the fuzzy strategy, and 56.5%, 45.2%, 56.5%, 25.0% and 36.8% of that without the fuzzy strategy, respectively. As for the unmatched events *UM*, our method fails to match 2.7%, 6.7%, 7.1%, 0.3% and 2.8% of source events in patterns that have counterparts in the target app with the fuzzy strategy, while 22.7%, 10.6%, 0.6%, 8.6% and 4.9% without the fuzzy strategy. These results show that among 20 target apps, for 16 apps that belong to the *Shopping List*, *Note Taking*, *Weather* and *TodosManagement* categories, the fuzzy strategy can increase the proportion of correctly matched events and reduce the proportion of unmatched events that have counterparts. For apps in the *Expense Tracker* category, *E*2 shows that the fuzzy strategy has no effect on event matching, and the other three apps even show better results without fuzzy matching.

Besides, we further combine these metrics above into the precision and recall, where the precision is equal to $CM/(CM + ICM)$, and the recall is equal to $CM/(CM + UM)$. As Table 4 shows, with the fuzzy strategy, the average precision of the *Shopping List*, *Note Taking*, *Expense Tracker*, *Weather* and *TodosManagement* categories are 77.2%, 75.4%, 58.7%, 60.7% and 67.6%, and the recall are 96.7%, 89.0%, 88.9%, 99.5% and 94.5%, respectively. While without the fuzzy strategy, the precision are 77.2%, 69.1%, 64.2%, 29.9% and 57.2%, and the recall are 69.4%, 81.1%, 99.0%, 73.7% and 84.4%, respectively. For *Shopping List*, *Note Taking*, *Weather* and *Todos Management* categories, the precision and recall with the fuzzy strategy exceed those without the fuzzy strategy. For the *Expense Tracker* category, the results are the opposite.

Overall, the fuzzy strategy increases *CM* by 11.6% and reduces *UM* by 5.6%. And the precision and recall are increased by 8.4% and 12.2%, respectively. Thus, the effectiveness of the SEFM strategy is demonstrated.

Table 3
Results of event matching with and without the fuzzy strategy.

Src	Tgt	CM		ICM		UM		UM!	
		Fuzzy	Without fuzzy	Fuzzy	Without fuzzy	Fuzzy	Without fuzzy	Fuzzy	Without fuzzy
S1	S2	83.1%	80.0%	10.7%	10.7%	3.1%	6.2%	3.1%	3.1%
	S3	86.2%	78.5%	4.6%	7.7%	4.6%	9.2%	4.6%	4.6%
	S4	64.6%	50.7%	32.3%	27.7%	3.1%	20.0%	0%	1.5%
	S5	56.9%	16.9%	40.0%	9.2%	0%	55.4%	3.1%	18.5%
Avg.		72.7%	56.5%	21.9%	13.9%	2.7%	22.7%	2.7%	6.9%
N1	N2	61.5%	42.4%	7.7%	19.2%	3.9%	11.5%	26.9%	26.9%
	N3	42.3%	38.5%	26.9%	19.2%	7.7%	7.7%	23.1%	34.6%
	N4	61.5%	46.2%	15.4%	26.9%	7.7%	11.5%	15.4%	15.4%
	N5	57.7%	53.9%	23.1%	15.4%	7.7%	11.5%	11.5%	19.2%
Avg.		55.8%	45.2%	18.3%	20.2%	6.7%	10.6%	19.2%	24.0%
E1	E2	58.7%	58.7%	37.0%	26.1%	0%	0%	4.3%	15.2%
	E3	43.5%	56.5%	32.6%	34.8%	19.6%	0%	4.3%	8.7%
	E4	56.5%	60.9%	26.1%	32.6%	8.7%	0%	8.7%	6.5%
	E5	45.7%	50.0%	50.0%	32.6%	0%	2.2%	4.3%	15.2%
Avg.		51.1%	56.5%	36.4%	31.5%	7.1%	0.6%	5.4%	11.4%
W1	W2	53.6%	19.1%	33.3%	58.3%	0%	9.5%	13.1%	13.1%
	W3	57.1%	25.0%	42.9%	75.0%	0%	0%	0%	0%
	W4	60.7%	11.9%	26.2%	54.9%	1.2%	21.3%	11.9%	11.9%
	W5	47.6%	44.0%	40.5%	40.5%	0%	3.6%	11.9%	11.9%
Avg.		54.8%	25.0%	35.7%	57.1%	0.3%	8.6%	9.2%	9.2%
T1	T2	36.1%	36.1%	30.6%	41.7%	0%	0%	33.3%	22.2%
	T3	41.7%	38.9%	25.0%	27.8%	8.3%	8.3%	25.0%	25.0%
	T4	47.2%	36.1%	22.2%	33.3%	0%	0%	30.6%	30.6%
	T5	50.0%	36.1%	8.3%	13.9%	2.8%	11.1%	38.9%	38.9%
Avg.		43.8%	36.8%	21.5%	29.2%	2.8%	4.9%	31.9%	29.2%

4.3.3. Results for RQ3

To evaluate the effectiveness of our pattern-oriented continuous workflow generation method, we assess the completeness of reused patterns and the importance of generated target workflows. Fig. 5 shows the results of behavior patterns reuse. The *CR* and *PR* are 88.3% and 9.2% on average, confirming that our pattern reuse method can effectively reuse behavior patterns of source

Table 4
Precision and recall of event matching results with and without the fuzzy strategy.

Src	Tgt	Precision		Recall	
		Fuzzy	Without fuzzy	Fuzzy	Without fuzzy
S1	S2	88.5%	88.1%	96.4%	92.9%
	S3	94.9%	91.1%	94.9%	89.5%
	S4	66.7%	64.7%	95.5%	71.7%
	S5	58.7%	64.8%	100.0%	23.4%
Avg.		77.2%	77.2%	96.7%	69.4%
N1	N2	88.9%	68.8%	94.1%	78.6%
	N3	61.1%	66.7%	84.6%	83.3%
	N4	80.0%	63.2%	88.9%	80.0%
	N5	71.4%	77.8%	88.2%	82.4%
Avg.		75.4%	69.1%	89.0%	81.1%
E1	E2	61.3%	69.2%	100.0%	100.0%
	E3	57.1%	61.9%	69.0%	100.0%
	E4	68.4%	65.1%	86.7%	100.0%
	E5	47.8%	60.5%	100.0%	95.8%
Avg.		58.7%	64.2%	88.9%	99.0%
W1	W2	61.6%	24.7%	100%	66.8%
	W3	57.1%	25.0%	100.0%	100.0%
	W4	69.9%	17.9%	98.1%	35.7%
	W5	54.2%	52.1%	100.0%	92.4%
Avg.		60.7%	29.9%	99.5%	73.7%
T1	T2	54.1%	46.4%	100.0%	100.0%
	T3	62.5%	58.3%	83.4%	61.2%
	T4	68.0%	52.0%	100.0%	100.0%
	T5	85.8%	72.2%	94.7%	76.5%
Avg.		67.6%	57.2%	94.5%	84.4%

apps. Besides, compared with the event matching results in Table 3, results in Fig. 5 show that the target workflow generation can supplement some unmatched events that have a counterpart to the target workflow, improving the effectiveness of behavior pattern reuse. Taking $E4$ as an example, 8.7% of source events in $E1$ do not find counterparts on $E4$ in event matching, but the proportion of completely reused patterns is 100%. This means that the unmatched source events discover their counterparts in the target app while generating target workflows. What is more, the sum of *CR* and *PR* of target apps belonging to the *Todos Management* category is 87.5%, while the sum of other categories is 100%. This is because, in the *Todos Management* category, there is a function represented by a user behavior pattern that does not exist in the target apps; that is, the events in the user behavior pattern have no counterparts on the target apps.

Table 5 shows the results of the state coverage on three levels of GUI model states. The average state coverage of $level_1$, $level_2$ and $level_3$ are 89.1%, 71.2% and 53.0%, respectively. The decreasing relationship of the state coverage of $level_1$, $level_2$ and $level_3$ indicates that the generated target workflows tend to cover states that are more likely to be accessed.

Overall, on average, our continuous workflow generation method completely reuses 88.3% of behavior patterns, and the generated target workflows cover 89.1% of the most important app states. Thus, according to the results, our continuous workflow generation method is effective.

4.3.4. Results for RQ4

To verify whether our approach can correctly translate the sequences of events, the human experience evaluates the behavior pattern reuse results. The feedback of the postgraduates is shown in Table 6. The results show that 61.2% of user behavior patterns are correctly transferred.

In addition, we interviewed these three postgraduates about why they think that a user behavior pattern has not been correctly reused. They consider two aspects in their judgment of

Table 5
The state coverage of target workflows on three levels.

Source app	Target app	$level_1$	$level_2$	$level_3$
S1	S2	75.0%	33.3%	44.4%
	S3	57.1%	45.5%	17.6%
	S4	100.0%	60.0%	42.9%
	S5	71.4%	66.7%	52.9%
N1	N2	100.0%	75.0%	100.0%
	N3	100.0%	100.0%	100.0%
	N4	100.0%	100.0%	71.4%
	N5	100.0%	100.0%	42.9%
E1	E2	80.0%	42.9%	30.8%
	E3	83.3%	75.0%	42.9%
	E4	80.0%	42.9%	45.5%
	E5	100.0%	100.0%	40.0%
W1	W2	100.0%	100.0%	85.7%
	W3	80.0%	42.9%	69.2%
	W4	100%	100%	60%
	W5	100.0%	100.0%	45.5%
T1	T2	100.0%	50.0%	50.0%
	T3	100.0%	75.0%	60.0%
	T4	33.3%	75.0%	28.6%
	T5	66.7%	40.0%	28.6%
Avg.		89.1%	71.2%	53.0%

Table 6
The human feedback results of the correctness of user behavior patterns reuse.

Source app	User	✓	×	–
S1	User1	65.4%	25.0%	9.6%
	User2	69.2%	25.0%	5.8%
	User3	69.2%	25.0%	5.8%
N1	User1	70.0%	20.0%	10.0%
	User2	80.0%	20.0%	0%
	User3	75.0%	20.0%	5.0%
E1	User1	53.1%	46.9%	0%
	User2	52.1%	40.6%	6.3%
	User3	50.0%	40.6%	9.4%
W1	User1	57.1%	37.5%	5.4%
	User2	57.1%	28.6%	14.3%
	User3	55.4%	32.1%	12.5%
T1	User1	43.8%	53.1%	3.1%
	User2	56.3%	40.6%	3.1%
	User3	46.9%	50.0%	3.1%
Avg.		61.2%	33.6%	6.2%

incorrect reuse. On one hand, the function represented by the user behavior pattern does not exist on the target app, and on the other hand, the generated target workflow in the target app is not equivalent to the user behavior pattern in the source app. For uncertain reuse results, a part of events in the user behavior patterns are accurately reused, but the remaining events may not exist on the target apps, or they may not affect the overall behavior logics.

4.4. Threats to validity

The primary external threat to the validity of our results is the generalization to other apps and categories. To mitigate this threat, we randomly select real-world apps from five different categories. The first internal threat is that our approach currently only supports three-screen events, namely clickable, long-clickable and editable events, and does not support user gestures, which feature may be considered in future work. Moreover, the second internal threat is that there may be mistakes when manually inspecting the pattern reuse results. In order to reduce the error caused by this threat, we repeatedly check the matching relation between the source event and the expected target event to improve the credibility of the results.

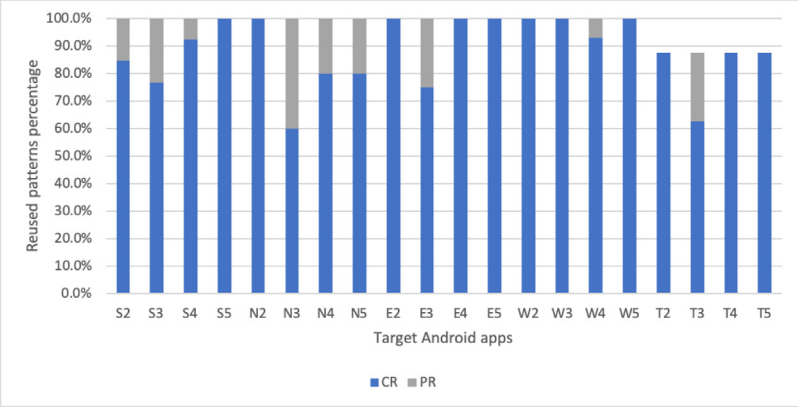

Fig. 5. The percentage of completely reused patterns and partially reused patterns.

5. Related work

5.1. Test case reuse

Test case reuse is an important means to reduce test costs in the testing field. In general, test case reuse can be divided into two types, one is test case migration, and the other is generic test case reuse.

In test case migration, there are some studies in the Web and Android field aiming to migrate test cases between applications with similar functionality (Rau et al., 2018a,b; Behrang and Orso, 2018b,a, 2019; Lin et al., 2019). For example, Rau et al. (2018b) first raise a test transferring method across Web applications, which creates and leverages event mappings across different applications, and thus to transfer tests from one to another. Lin et al. (2019) propose a framework called CRAFTDROID that leverages information retrieval, along with static and dynamic analysis techniques, to extract the human knowledge from an existing test suite for one app and transfer the test cases and oracles to be used for testing other apps with the similar functionality. Moreover, Qin et al. (2019) proposes a tool called TestMig to migrate test cases from the iOS version to the Android version. TestMig executes the GUI tests of the iOS version and records their GUI event sequences. Guided by the iOS GUI events, TestMig explores the Android version of the application to generate the corresponding Android event sequences. All of these work share a common intuition that applications in the same category are implement similar functionality and user flows.

Unlike the existing research, our approach focuses on the user behavior patterns mined from real-world user traces, representing the typical functionality of similar apps, that is, the workflows that users are most likely to execute on such apps. But the migrated test cases are manually written by researchers according to specific usage scenarios. On the other hand, events involved in user behavior patterns are not necessarily continuous since the pattern only means the sequence of triggering events, while the test case must be a continuous event sequence.

In generic test case reuse, some researchers are dedicated to the reuse of generic test cases applicable to certain types of activities or functions. Moreira et al. (2013) defines several base UI test patterns and proposes a domain-specific language PARADIGM for combining test patterns with building newer ones and promoting reuse. The above works are to reuse pre-defined generic test cases, while our work is to reuse the behavior patterns of a specific app to similar apps. Rosenfeld et al. (2018) aims at relieving some of the manual functional testing burdens by automatically classifying each of the application's screens to a set of common screen behaviors for which generic test scripts can be instantiated and reused. Hu et al. (2018) presents a system for synthesizing highly robust, highly reusable UI tests, called AppFlow. AppFlow utilizes machine learning to classify each of the app's activities to a certain type and summarizes a set of common behaviors that can be instantiated and reused for each type of apps. It enables developers to write a library of modular tests for the main functionality of an app category, thereby tests a new app in the same category by synthesizing full tests from the modular ones in the library. Mariani et al. (2018) proposes Augusto, a test case generation technique that encodes the commonly expected semantics of application-independent functionalities (AIF) into models using Alloy (Jackson, 2002). Moreover, with the semantic models, Augusto can precisely identify AIFs and then generate complex test cases for applications under test.

5.2. Usage of user trace

In recent years, user trace is used in many test-related studies in the web field. For example, Ernst et al. (2002) focuses on data values and implements a technique named Daikon to extract models from execution traces in the form of Boolean expressions. Based on the work of Ernst et al. (2002) and Lorenzoli et al. (2008) generates an EFSM model considering both data values and component interactions which are two different aspects of application execution. Brooks and Memon (2007) leverage event sequences that users execute on the application to develop a probabilistic usage model of the application. Wang et al. (2018, 2019) propose a trace-based model construction approach for web applications, which represents dynamic behaviors of web applications as traces and optimize minimal traces set by the greedy algorithm. Milani Fard et al. (2014) implements a tool named $T_{ESTILIZER}$, which mines the human knowledge present in the form of input values, event sequences, and assertions existing in manually-written test cases. Moreover, $T_{ESTILIZER}$ leverages the human knowledge to infer a model and expand the model by exploring uncovered paths and states, finally generating new test cases. Ermuth and Pradel (2016) adopts data mining techniques, in particular frequent subsequence mining and inference of finite state machines, to infer macro events from user traces, that is, logical steps that users commonly perform. Furthermore, this work combines random test generation and macro events to generate test cases from the tested web application model.

There have been many studies on Android automated testing (Choi et al., 2013; Azim and Neamtiu, 2013; Amalfitano et al., 2014; Choudhary et al., 2015; Mahmood et al., 2014; Linares-Vásquez et al., 2017; Su et al., 2017; Yang et al., 2018; Salihu et al., 2019). But these studies do not consider the context between events, thereby are not possible to generate test case with user

logic. Thus, in the Android field, research has gradually begun to introduce user trace. For example, Linares-Vásquez et al. (2015) proposes a novel approach $M_{ONKEYLAB}$ that automatically generates test cases by mining user traces and deriving language models. Inspired by crowdsourcing's popular use in software engineering research (Ponzanelli et al., 2013; Chen and Kim, 2015; Mao et al., 2015; Wang et al., 2016; Mao et al., 2017a), Mao et al. (2017b) introduces a tool P_{OLARIZ} which generates replicable test scripts from crowd-based testing by extracting cross-app motif events from traces that are recorded by Android *getevent* command and replayed by RERAN (Gomez et al., 2013).

Like the above work, our approach also leverages user traces of existing apps and mines frequent and reusable user behavior patterns from them to help understand other similar new apps.

6. Conclusion

To help understand a new app's behavior quickly, in this paper, we propose a user behavior pattern mining and reuse approach across similar Android apps. In the pattern mining phase, we adopt the sequential pattern mining technique *CloFAST* to mine frequent patterns from real-world user traces. In the pattern reuse phase, we leverage the semantic-based event fuzzy matching strategy to match candidate events on the target app for each source event in patterns. Moreover, we further use the GUI model of the target app to supplement new events among the discontinuous ordered candidate events, thereby chaining them in series to generate the target workflow. We implement the approach and evaluate its effectiveness on 25 open source apps from five categories. The results show that the behavior patterns of source apps are correctly transferred to the target apps. And 88.3% of patterns are completely reused, and the coverage rate of the generated target workflow to the states of $level_1$ reaches 89.1%.

In future work, we plan to introduce user gestures into our research to expand more user interactions. And our existing research results can be combined with existing Android automated testing methods to generate test cases that are more likely to be executed by users.

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.

Acknowledgment

The work describes in this paper is supported by the National Natural Science Foundation of China under Grant No. 61702029, 61872026 and 62077003.

References

Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.M., 2014. MobiGUITAR: Automated model-based testing of mobile apps. IEEE Softw. 32 (5), 53–59.

Azim, T., Neamtiu, I., 2013. Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 641–660.

Behrang, F., Orso, A., 2018a. Automated test migration for mobile apps. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings. pp. 384–385.

Behrang, F., Orso, A., 2018b. Test migration for efficient large-scale assessment of mobile app coding assignments. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 164–175.

Behrang, F., Orso, A., 2019. Test migration between mobile apps with similar functionality. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).

Brooks, P.A., Memon, A.M., 2007. Automated GUI testing guided by usage profiles. In: Proceedings of the Twentysecond IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM, pp. 333–342.

Chen, F., Kim, S., 2015. Crowd debugging. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 320–332.

Choi, W., Necula, G., Sen, K., 2013. Guided gui testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages Applications, Vol. 48. ACM, New York, NY, USA, pp. 623–640.

Choudhary, S.R., Gorla, A., Orso, A., 2015. Automated test input generation for android: Are we there yet? (E). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 429–440. http://dx.doi.org/10.1109/ASE.2015.89.

Ermuth, M., Pradel, M., 2016. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 82–93.

Ernst, M.D., Cockrell, J., Griswold, W.G., Member, IEEE, 2002. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.

Freeman, L.C., 1977. A set of measures of centrality based on betweenness. Sociometry 40 (1), 35–41.

Fumarola, F., Lanotte, P.F., Ceci, M., Malerba, D., 2016. CloFAST: closed sequential pattern mining using sparse and vertical id-lists. Knowl. Inf. Syst. 48 (2), 429–463.

Gomez, L., Neamtiu, I., Azim, T., Millstein, T., 2013. Reran: Timing-and touch-sensitive record and replay for android. In: 2013 35th International Conference on Software Engineering (ICSE). IEEE, pp. 72–81.

Hu, G., Zhu, L., Yang, J., 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 269–282.

Jackson, D., 2002. Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. (TOSEM) 11 (2), 256–290.

Kudo, T., Bulc Xe, R.F.F., o Neto, Vincenzi, A.M.R., 2019. A conceptual metamodel to bridging requirement patterns to test patterns. In: XXXIII Brazilian Symposium on Software Engineering (SBES 2019).

Lin, J.W., Jabbarvand, R., Malek, S., 2019. Test transfer across mobile apps through semantic mapping. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).

Linares-Vásquez, M., Moran, K., Poshyvanyk, D., 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 399–410.

Linares-Vásquez, M., White, M., Bernal-Cárdenas, C., Moran, K., Poshyvanyk, D., 2015. Mining android app usages for generating actionable gui-based execution scenarios. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 111–122.

Lorenzoli, D., Mariani, L., Pezzè, M., 2008. Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering. pp. 501–510.

Mahmood, R., Mirzaei, N., Malek, S., 2014. Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 599–609.

Mao, K., Capra, L., Harman, M., Jia, Y., 2017a. A survey of the use of crowdsourcing in software engineering. J. Syst. Softw. 57–84.

Mao, K., Harman, M., Jia, Y., 2017b. Crowd intelligence enhances automated mobile testing. In: IEEE/ACM International Conference on Automated Software Engineering.

Mao, K., Yang, Y., Wang, Q., Jia, Y., Harman, M., 2015. Developer recommendation for crowdsourced software development tasks. In: 2015 IEEE Symposium on Service-Oriented System Engineering. IEEE, pp. 347–356.

Mariani, L., Pezzè, M., Zuddas, D., 2018. Augusto: Exploiting popular functionalities for the generation of semantic gui tests with oracles. In: Proceedings of the 40th International Conference on Software Engineering. pp. 280–290.

Milani Fard, A., Mirzaaghaei, M., Mesbah, A., 2014. Leveraging existing tests in automated test generation for web applications. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 67–78.

Moreira, R.M., Paiva, A.C., Memon, A., 2013. A pattern-based approach for GUI modeling and testing. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 288–297.

Ponzanelli, L., Bacchelli, A., Lanza, M., 2013. Leveraging crowd knowledge for software comprehension and development. In: 2013 17th European Conference on Software Maintenance and Reengineering. IEEE, pp. 57–66.

Qin, X., Zhong, H., Wang, X., 2019. Testmig: Migrating gui test cases from ios to android. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 284–295.

Rau, A., Hotzkow, J., Zeller, A., 2018a. Poster: Efficient GUI test generation by learning from tests of other apps. In: IEEE/ACM International Conference on Software Engineering: Companion.

Rau, A., Hotzkow, J., Zeller, A., 2018b. Transferring Tests Across Web Applications. Springer, Cham.

Rosenfeld, A., Kardashov, O., Zang, O., 2018. Automation of android applications functional testing using machine learning activities classification. pp. 122–132.

Salihu, I.-A., Ibrahim, R., Ahmed, B.S., Zamli, K.Z., Usman, A., 2019. AMOGA: A static-dynamic model generation strategy for mobile apps testing. IEEE Access 7, 17158–17173.

Singh, S., Gadgil, R., Chudgor, A., 2014. Automated testing of mobile applications using scripting technique: A study on appium. Int. J. Curr. Eng. Technol. (IJCET) 4 (5), 3627–3630.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z., 2017. Guided, stochastic model-based GUI testing of Android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 245–256.

Wang, W., Guo, J., Li, Z., Zhao, R., 2018. Efsm-oriented minimal traces set generation approach for web applications. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), Vol. 1. IEEE, pp. 12–21.

Wang, W., Guo, X., Li, Z., Zhao, R., 2019. Test case generation based on client-server of web applications by memetic algorithm. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp. 206–216.

Wang, J., Wang, S., Cui, Q., Wang, Q., 2016. Local-based active classification of test report to assist crowdsourced testing. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 190–201.

Yang, S., Wu, H., Zhang, H., Wang, Y., Swaminathan, C., Yan, D., Rountev, A., 2018. Static window transition graphs for Android. Autom. Softw. Eng. 25 (4), 833–873.