Sketch-Guided GUI Test Generation for Mobile Applications

Chucheng Zhang Haoliang Cheng Enyi Tang* Xin Chen Lei Bu Xuandong Li
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Software Institute of Nanjing University, Nanjing University, Nanjing, China
*Corresponding author: eytang@nju.edu.cn

***Abstract*—Mobile applications with complex GUIs are very popular today. However, generating test cases for these applications is often tedious professional work. On the one hand, manually designing and writing elaborate GUI scripts requires expertise. On the other hand, generating GUI scripts with record and playback techniques usually depends on repetitive work that testers need to interact with the application over and over again, because only one path is recorded in an execution. Automatic GUI testing focuses on exploring combinations of GUI events. As the number of combinations is huge, it is still necessary to introduce a test interface for testers to reduce its search space.**

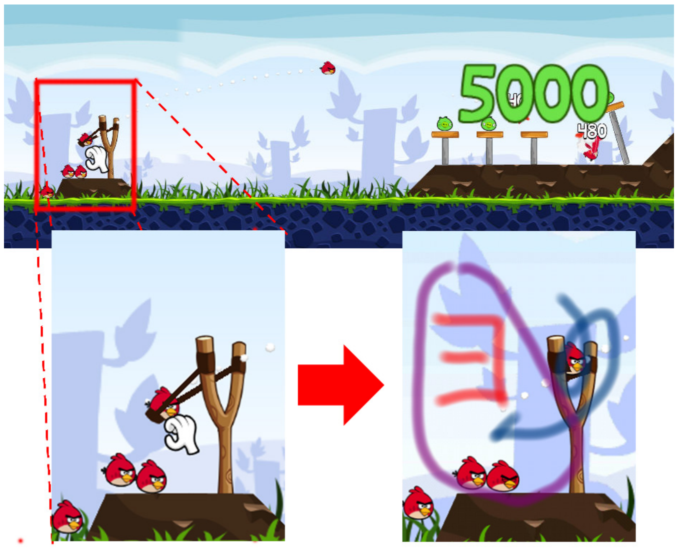
This paper presents a sketch-guided GUI test generation approach for testing mobile applications, which provides a simple but expressive interface for testers to specify their testing purposes. Testers just need to draw a few simple strokes on the screenshots. Then our approach translates the strokes to a testing model and initiates a model-based automatic GUI testing. We evaluate our sketch-guided approach on a few real-world Android applications collected from the literature. The results show that our approach can achieve higher coverage than existing automatic GUI testing techniques with just 10-minute sketching for an application.

I. Introduction

Until November 2016, the number of available mobile applications in the Google Play store is over 2.5 million [4]. Many of these applications provide rich features that help billions of users to communicate with each other. Hence, testing these applications is important. However, as most of the mobile applications interact with users through a graphical user interface (GUI), generating test cases for these GUI applications is a challenge.

Two classical approaches have been widely applied to GUI testing: manually writing GUI scripts as test cases that describe sequences of GUI actions, or recording the sequences of GUI events as test cases for playback later when testers execute the application under test. Both approaches strongly rely on manual work of professional testers. On the one hand, testers need to design and write quite a number of elaborate GUI scripts with their expertise in GUI testing. On the other hand, record and playback sequences of GUI events often depend on repetitive work that testers need to interact with the application over and over again, because only one path is recorded in an execution.

To free testers from the burden of tedious manual work, researchers have proposed a few techniques that generate GUI test cases automatically [3, 5, 6]. These techniques generate sequences of GUI events essentially by exploring the possible event combinations in applications. As the number of combinations of GUI events is often huge, how to reduce the exploring space (search space) for the automatic testing techniques is an important problem.



(a) shoot a bird by dragging it from the slingshot

(b) sketch for testing the bird-shooting

Fig. 1. An Example of Sketching to Generate High Score GUI Inputs in the Game of Angry Birds

In this paper, we propose a sketch-guided GUI test generation approach for testing mobile applications, which covers combinations of GUI events following a sketch-based specification from testers. Testers just draw a few simple strokes on the screenshots as the sketches to specify their testing purposes. Then our approach translates the sketches to a testing model and initiates a model-based automatic GUI testing.

For example, when testers want to explore the GUI events effectively in the the game *Angry Birds*[1](shown in Figure 1), they need to generate test cases that shoot a bird at the slingshot every time by dragging it as shown in Figure 1a. With different shooting angles and power, they get different scores. Existing techniques are difficult to uncover the high score GUI test case in the game, which is hidden in a lot of GUI event sequences

[1]http://www.angrybirds.com

that dragging the bird in different ways. The manual techniques need tedious labor effort to replay the shooting over and over again with various shooting angles and power, while other automatic techniques try to traverse all the GUI components in the application other than just dragging the bird in different ways.

Our sketch-guided approach is good at working on such testing tasks. Testers just simply draw a few strokes as Figure 1b, which specifies their testing purposes. In other words, they specify a range of GUI event sequences that they want our test engine to explore deeply by a simple sketch. The sketch in Figure 1b consists of 3 stroke components with different colors: a dragging action in blue, a range specifier in purple, and an existential quantifier in red. The dragging stroke makes our test engine focus on the bird-shooting GUI inputs during the testing, and the range specifier specifies the range to which the generated test scripts should drag the bird. At last, testers draw a quantifier to specify the target of the testing, which is just to find one GUI event sequence that get the score higher than a value in the game.

II. Sketch-guided Approach

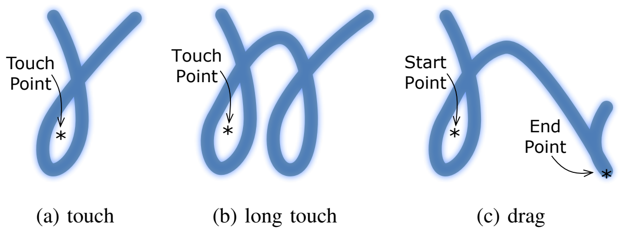
This section presents the technical details of our sketch-guided GUI test generation. Testers provide a subject program for testing, and draw a few input sketches on the screenshots taken by our testing framework. Our framework processes the sketch with 2 stages, and outputs GUI test scripts accordingly. In the first stage, the framework recognizes the input sketches to a symbolic layout with attributes ripping from the subject program. Then in the second stage, it builds a model from the symbolic layout defined by the input sketches, and generates GUI test cases by traversing paths in the model. The rest of this section describes the technical details of every stage separately.

A. Sketching Language & Sketch Recognition

The first stage of our workflow recognizes primitive shapes in the input sketches, and generates a symbolic sketch layout that holds the information of shapes and matches the GUI widgets to each corresponding shape.

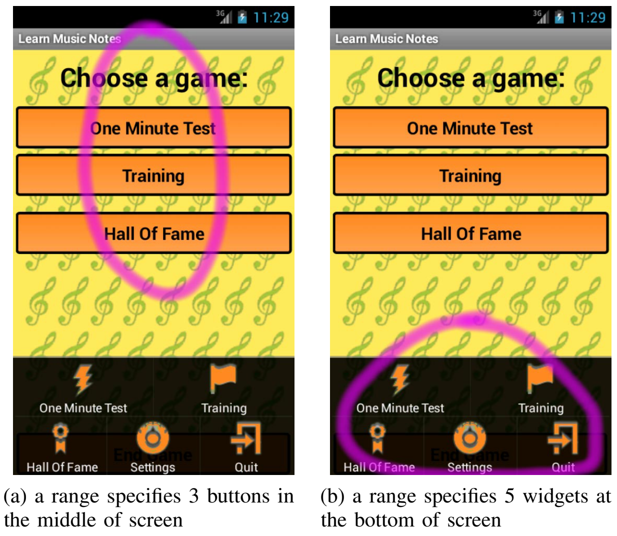
We design an expressive but simple sketching language, which defines only 4 types of primitive shapes: the *action strokes*, *range specifiers*, *quantifiers*, and *boolean connectors*. Figure 2-5 present the primitive shapes in our sketching system. Every action stroke describes a user action in the GUI testing, such as touching on a button or dragging an icon. After drawing an action stroke, the tester draws immediately a range specifier to delineate a set of GUI components that are affected by the action. Quantifiers and boolean connectors further specify the logical conditions and logical relations of the actions with different GUI components, so we also call them *logical shapes*. For example, the tester can draw an existential quantifier (∃) after a range specifier, which means the testing system just needs to find one GUI component in the specified range that makes the event sequence satisfy the test oracle.

Note that not everything in our framework is represented by the sketch. Specifically, testers specify the text-input actions



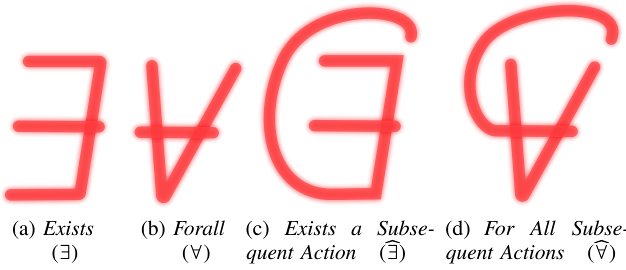
(a) touch (b) long touch (c) drag

Fig. 2. Typical Types of *Action Strokes* in Our System



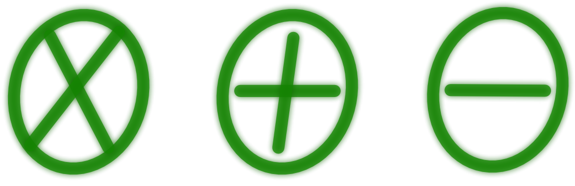
(a) a range specifies 3 buttons in the middle of screen (b) a range specifies 5 widgets at the bottom of screen

Fig. 3. Examples of *Range Specifiers* on Screenshots



(a) *Exists* (∃) (b) *Forall* (∀) (c) *Exists a Subsequent Action* ($\bar{\exists}$) (d) *For All Subsequent Actions* ($\bar{\forall}$)

Fig. 4. Types of *Quantifiers* in Our System



(a) boolean *AND* (⊗) (b) boolean *OR* (⊕) (c) boolean *NOT* (⊖)

Fig. 5. Types of *Boolean Connectors* in Our System

and the test oracle directly through a textbox in our system and links them to the symbolic layout later after other sketches are recognized. The test oracle is some constraints that should be satisfied when the generated test cases are performed, such as `Color(x,y)==RGB(200,0,0)` where `x` and `y` are coordinates of a user-specified point. If the application crashes during testing, the test oracle is always unsatisfied.

Figure 2 shows several typical types of action strokes, which specify the user actions in the application's GUI systems. For a touchscreen-based mobile application, the typical actions such as touching, long touching a GUI widget, or dragging a widget to another place are expressed separately as strokes in Figure 2a, Figure 2b, and Figure 2c. Testers can further extend the system with more GUI actions when necessary. Our sketching system not only recognizes the action strokes, but also records the coordinates of key points and its corresponding GUI widgets specified by the strokes. For example, when a tester draws a drag action on the screenshot, our system records the coordinates of its start point and end point along with the information of GUI widgets at these points.

Our system rips the information of GUI widgets from the application under test when we take the screenshots, and binds every screenshot with the bound of widgets on it. When the tester draws an action on a screenshot, our system directly obtains the information of widgets from the binding. The range specifiers shown in Figure 3 also use such a technique to collect the widget information in the range. Usually, the range specifiers denote a set of GUI widgets in the application as Figure 3a and Figure 3b show. In some Games, it also denotes a set of coordinates on the screen when our system does not get any widgets in the range like Figure 1b.

For every range specifier, we designate a quantifier in our sketching language. The quantifier specifies the quantity of events in the range that satisfy the oracle, which affects the search condition in our test generation. For example, the universal quantifier ($\forall$) in Figure 4b requires all events in the range to eventually satisfy the condition in the test oracle, whereas the existential quantifier ($\exists$) in Figure 4a drives the test engine to search for just one event that satisfies the condition, which is often used in generating a GUI event that triggers the bug. We introduce 2 recursive quantifiers in our sketching language shown in Figure 4c and Figure 4d. We call them the recursive existential quantifier ($\widehat{\exists}$) and the recursive universal quantifier ($\widehat{\forall}$) separately, which recursively specify the quantity of all subsequent actions of the current event in the range.

We also introduce the boolean connectors in our sketching language to make it expressive and powerful. Figure 5 presents the boolean connectors in our system, which represents boolean *and* ($\otimes$) in Figure 5a, *or* ($\oplus$) in Figure 5b, and *not* ($\ominus$) in Figure 5c.

The following context-free grammar (also called BNF, Backus-Naur form) defines the syntax of our sketching language:

$$\begin{aligned}
sketch &\rightarrow tracelist \\
tracelist &\rightarrow eventtrace \mid tracelist, eventtrace \\
eventtrace &\rightarrow eventstep \mid eventtrace; eventstep \\
&\quad \mid eventtrace : tracelist \\
eventstep &\rightarrow actionset \mid eventstep\ \textbf{boolcon}\ actionset \\
actionset &\rightarrow \textbf{act} \mid \textbf{act} + \textbf{range} + \textbf{quant}
\end{aligned} \tag{1}$$

where the `act`, `range`, `quant`, and `boolcon` in the grammar means an action stroke, range specifier, quantifier, and boolean connector separately, and the punctuation marks in the grammar such as +, :, ; and , are gestures or buttons in our sketching system that specify the relations of shapes in the sketch. The grammar defines the drawing order for testers in producing their sketches. Testers start their sketching by drawing an action stroke to specify an event (such as touch) on a GUI widget. Then they optionally add a range specifier to make the event affect on a set of widgets, and a quantifier to present the quantity of events in the range that satisfy the oracle. At this point, testers have specified an $actionset$ with actions on different widgets, and they draw boolean connectors to connect multiple $actionset$s and build an $eventstep$ in their testing. An $eventtrace$ is either a sequence of $eventstep$s connected with semicolons in the grammar, or a fork from an $eventstep$ in the trace to a list of $eventtrace$s with a colon at the fork point. A sketch is finally a $tracelist$ that consists of multiple $eventtrace$s.

B. Event-flow Modeling & Script Generation

The second stage of our workflow builds a GUI model for test generation from the symbolic sketch layout generated by the previous stage. The symbolic sketch layout L is defined as a 3-tuple $\langle l, W, O_e \rangle$, where l is a symbolic $tracelist$ organized as the grammar in Equation 1, W is a mapping that maps the coordinates in l to the widgets in the application under test, and O_e is another mapping that maps every $eventtrace$ in l to a test oracle specified by the tester.

The coordinate-widget map $L.W$ in the sketch layout L stores the bound of every widget in the application under test. When our framework queries the coordinates of a point p_t on a screenshot, the map $L.W(p_t)$ analyzes the bound of every widget and returns the widget that the sketch locates the point in. We translate all coordinates to widgets in our GUI model. So the output test scripts can directly perform actions on the GUI widgets, which generalizes the testing on devices with different screen resolutions.

Our framework models the sketch layout as a partial abstract event-flow graph (P-aEFG), which is a variation of the event-flow graph (EFG) defined by Memon et al. for GUI testing [13]. Not as the EFG models representing all possible event sequences on a GUI, our P-aEFG just partially extracts the abstract event-flow that represents possible interactions defined by the sketch. Hence, the P-aEFG is often smaller. In our P-aEFG, a vertex represents an abstract event that summarizes the events may be performed at the same application state. And an edge in a P-aEFG from the vertex v_1 to v_2 means that an event e_2 in the abstract event v_2 may be performed immediately after every event e_1 in the abstract event of v_1, and every sequence of e_1, e_2 should be tested when it is specified by the sketch. We formally define a P-aEFG (G) as a 4-tuple $\langle V, E, V_0, O \rangle$, where:

1) V is a vertex set and each vertex $v \in V$ is an abstract event specified in the sketch. An abstract event represents a set of events that may be performed at the same application state.

Algorithm 1 Abstract Event-flow Modeling

```
Input: $L\langle l, W, O_e\rangle$
Output: $G\langle V, E, V_0, O\rangle$
 1: for all $es \in$ eventStep($L.l$) do
 2:     $t_v \leftarrow$ syntaxTree($es$)
 3:     for all $p_t \in$ pointCoodinate($t_v$) do
 4:         $t_v \leftarrow t_v[p_t \hookrightarrow L.W(p_t)]$ //substitute coordinates to widget
 5:     end for
 6:     V($es$) $\leftarrow t_v$                    //hold the vertex for every $eventstep$
 7:     $G.V \leftarrow G.V \cup \{t_v\}$
 8: end for
 9: for all "$es_1; es_2$" $\in$ subString($L.l$) do
10:     $G.E \leftarrow G.E \cup \{(\mathrm{V}(es_1), \mathrm{V}(es_2))\}$
11: end for
12: for all "$et : tlist$" $\in$ subString($L.l$) $\wedge$ $\forall et' \in tlist$ do
13:     $es_1 \leftarrow$ lastEventStep($et$)
14:     $es_2 \leftarrow$ firstEventStep($et'$)
15:     $G.E \leftarrow G.E \cup \{(\mathrm{V}(es_1), \mathrm{V}(es_2))\}$
16: end for
17: for all $et \in$ eventTrace($L.l$) do
18:     $es \leftarrow$ firstEventStep($et$)
19:     $G.V_0 \leftarrow G.V_0 \cup \{\mathrm{V}(es)\}$
20: end for
21: $G.O \leftarrow$ oracleParse($L$,V)
```

Algorithm 2 `oracleParse`: Marks Every Test Oracle on the Path of the Abstract Event-flow Graph

```
Input: $L\langle l, W, O_e\rangle$, V  //the map holds the vertex for each $eventstep$
Output: $O$                         //the map binds every oracle to the path
 1: for all $et \in$ eventTrace($L.l$) do
 2:     $p \leftarrow$ V(firstEventStep($et$))
 3:     $P \leftarrow P \cup \{p\}$                  //put the path $p$ to the set $P$
 4:     $\forall p \in P$ do
 5:         if "$es_1; es_2$"$\in$ subString($et$)
                $\wedge$ V($es_1$) == lastVertex($p$) then
 6:             $p \leftarrow$addVertex($p$,V($es_2$))  //catenate a vertex at the end
 7:         else if "$et' : tlist$" $\in$ subString($et$)
                $\wedge$ V(lastEventStep($et'$)) == lastVertex($p$) then
 8:             $P \leftarrow P - \{p\}$
 9:             for all $et'' \in tlist$ do
10:                 $p' \leftarrow$addVertex($p$,V(firstEventStep($et''$)))
11:                 $P \leftarrow P \cup \{p'\}$
12:             end for
13:         end if
14:     end for
15:     $\forall p \in P,\ O(p) \leftarrow L.O_e(et)$
16:     $P \leftarrow \varnothing$
17: end for
```

2) $E \subseteq V \times V$ is a directed edge set between vertices. An edge (v_1,v_2) $\in E$ iff $\forall e_1 \in v_1$, $\forall e_2 \in v_2$, e_1 may be performed immediately after e_2, and the combination of e_1, e_2 should be tested according to the specification.
3) V_0 is a set of start vertices representing the events that are available for the testers when the application starts.
4) O is a map that binds every test oracle to the path in the graph. A path p in the P-aEFG is a sequence of vertices $(v_1, v_2, ..., v_n) \in V \times V \times ... \times V$, while a test oracle o is a constraint that the application should satisfy when the corresponding sequence of events have been performed.

Algorithm 1 parses the sketch layout L and builds the P-aEFG model G for test case generation. It first generates the vertices of the graph from the sketch layout (line 1-8). The function `eventStep` at line 1 returns a set of all top-level *eventstep*s that is syntactically not a substring of any other *eventstep* in the sketch layout. As every *eventstep* denotes the events may be performed at the same application state, we generate the vertex of our P-aEFG by substituting the coordinates of points to the corresponding widgets in the syntax tree of an *eventstep*. After the substitution, every range specifier in the symbolic sketch layout becomes a set of corresponding GUI widgets in the abstract event. Line 9-16 of Algorithm 1 links vertices with edges depending on the gestures (a semicolon or a colon) in the sketch layout. Then it stores all the start vertices in $G.V_0$ (line 17-20). Finally, the algorithm marks every test oracle on the path of the P-aEFG by calling `oracleParse` at line 21, and stores it in the map $G.O$ of the P-aEFG.

Algorithm 2 depicts details of the function `oracleParse`, which extracts the oracle constraints from the symbolic layout $L.O_e$ and marks them on the path p in the P-aEFG. Since testers have specified the oracle constraints for every *eventtrace* during sketching, Algorithm 2 seeks the set of paths P that corresponds to the symbolic event trace et (line 4-14), and marks the oracle constraints in the returned map (line 15).

After building the P-aEFG model G, our framework generates test cases directly by traversing the P-aEFG model. Generally, every GUI test case corresponds to a path in G, which is defined by a sequence of GUI events and an oracle constraint in our testing system. For every path p in the model G, our framework picks the first event from the start set $G.V_0$, develops a sequence of GUI events by picking the events one by one along the path, gets the oracle constraint from $G.O(p)$, and outputs the test case to a GUI test script.

III. Evaluation

We have implemented a prototype of our sketch-guided testing system[2], and performed our evaluation on 10 Android applications collected by Choi et al. [9], which are listed in Table I. All the applications are open source projects from *F-Droid* app market. The smallest project is `music note` with 1345 instructions in its bytecode, whereas the largest one is `anymemo` with 72145 bytecode instructions. Further details of each project can be referred to [9, 10].

We recruited 12 volunteers who were 3rd-year college students in Computer Science from our university to evaluate the system. All of them are familiar with Android devices but only 6 of them have taken the software testing class in our university. We pair every one of them with a student who has

[2]http://software.nju.edu.cn/eytang/artifacts/sketch/sketchgeneration.tar.gz.

TABLE I
Code Coverage Comparison of the Sketch-guided Testing and a few Recent Automatic GUI Testing Techniques

App	*% Line Coverage*				*% Branch Coverage*	
	Monkey	*AndroidRipper*	*MobiGUITAR*	*Sketch-guided*	*SwiftHand*	*Sketch-guided*
music note	46.85	4.01	29.83	84.12	62.30	73.63
whohas	59.35	16.31	32.89	78.35	54.40	61.39
explorer	69.36	2.52	16.49	69.40	62.94	53.42
weight	45.07	14.97	17.32	70.06	51.45	62.02
tippy	78.62	7.68	13.35	88.97	61.50	65.82
myexpense	42.57	12.82	9.48	62.04	34.40	45.42
mininote	29.15	4.25	6.39	41.89	24.20	28.14
mileage	30.60	9.52	14.68	59.79	24.50	39.64
anymemo	25.51	2.80	3.99	51.97	36.20	41.67
sanity	22.17	3.99	9.36	31.33	16.69	18.69
Average	44.93	7.89	15.38	63.79	42.86	48.98

TABLE II
Branch Coverage (%) of the Sketch-guided Testing with Different Sketching Time

App	*2 min*	*4 min*	*6 min*	*7 min*	*8 min*	*after 2 minute Extra Feedback*
music note	52.75	**72.73**	73.31	73.61	73.63	73.63
whohas	25.32	**60.83**	61.28	61.36	61.39	61.39
explorer	**45.39**	47.30	47.31	47.36	47.37	53.42
weight	**42.40**	42.45	42.48	42.48	42.48	62.02
tippy	16.84	**41.44**	44.30	44.36	44.39	65.82
myexpense	6.04	9.83	18.08	**36.45**	38.64	45.42
mininote	**25.49**	25.68	25.77	25.77	25.80	28.14
mileage	16.83	20.07	25.68	32.37	**37.05**	39.64
anymemo	4.95	17.18	23.57	29.79	**33.31**	41.67
sanity	2.17	5.39	9.96	**11.45**	11.68	18.69

not taken the software testing class, so every test group consists of two volunteers. With a 5-minute training, we ensure that all volunteers work with our testing system correctly.

Every test group is assigned only 8 minutes to complete their sketching in the evaluation. Then our testing system generates test cases and performs the testing with a 15-minute time limit on an Apple MacBook Pro with Intel Core i5 CPU 2.6GHz, and 8GB Physical Memory. After that, testers have an extra 2 minutes to provide a feedback and fix the sketches, which can further be processed with our testing system for 5 minutes. Hence, the total time of our sketch-guided testing is up to 30 minutes.

We compare the code coverage of our sketch-guided testing with 30-minute running of a few recent automatic GUI testing tools, Monkey, AndroidRipper [1], MobiGUITAR [2], and SwiftHand [9]. Monkey is an automated fuzz testing tool provided by the Android development kit, which creates random inputs without considering application's state. AndroidRipper [1] and MobiGUITAR [2] are two state-of-the-art model-based GUI testing tools for mobile applications. SwiftHand [9] is a recent tool that optimizes the exploration strategy of test generation with a machine learning algorithm. We perform every tool on the same platform and environment as our sketch-guided testing, and collect the line coverage with `Emma`[3]. Because the internal instrumentation of SwiftHand, which is a critical part of SwiftHand's functionality, conflicts with `Emma` (as Choudhary et al. mentioned [10]), we just compare the branch coverage of our sketch-guided approach with SwiftHand.

Table I depicts the results of our comparison. To reduce the randomness and non-deterministic nature introduced by different test groups, we present the average results in the table. From the results, our sketch-guided approach obtains higher code coverage than the automatic GUI testing techniques, which means the guidance of the sketches is helpful in GUI testing. The line coverage of our sketch-guided approach is 41.98% higher than Monkey, 708.49% higher than AndroidRipper, and 314.76% higher than MobiGUITAR in average. And the average branch coverage of our sketch-guided approach is 14.76% higher than SwiftHand. Our comparison results of Monkey, AndroidRipper, and SwiftHand are in accordence with the experiments in [10].

[3]http://emma.sourceforge.net/

An exception in Table I is that the branch coverage of SwiftHand on `explorer` is a little higher than the sketch-guided testing. The reason is that `explorer` only provides simple features of browsing files in the current device, and testers cannot specify meaningful guidance other than randomly surfing in the file system. So the line coverage of Monkey, which creates random inputs to test the application, is similar to our sketch-guided testing on `explorer`. However, SwiftHand applies a machine learning strategy to the control flow, which focuses on the internal structure of the application and achieves higher coverage on `explorer`. Nonetheless, our sketch-guided testing achieves higher branch coverage than SwiftHand on most applications, which indicates that testers' guidance is still better than machine learning algorithms with current technology.

From the results in Table I, all the 4 GUI testing techniques achieve low coverage on a few applications such as `sanity`, because a large number of modules and features in the application are not activated by GUI actions, but the environments, such as sensors, GPS, etc. Our sketch-guided approach does not support the environment testing currently. Such limitation can be overcome by extending our sketching system in future.

We collect the sketches at different scheduled time from testers in our testing, and compute the code coverage from the sketches of different sketching time to estimate the necessary manual effort in our sketch-guided testing. Table II presents the branch coverage that our sketch-guided testing achieves with different sketching time. From Table II, simple mobile applications only need a very short sketching time (just less than 2-4 minutes) to achieve a stable code coverage, and all the applications in our evaluation can achieve good coverage in about 8-minute sketching time. Hence, the manual effort in our sketch-guided testing is acceptable.

IV. Related Work

As smartphones and tablets become very popular today, various techniques focus on testing GUIs of mobile applications, such as guiding random testing with different strategies [11], applying static analysis to improve the testing quality [7, 19], combining GUI ripping with other testing techniques [15], testing mobile applications with symbolic execution [20], and even the computer vision technique [8]. Some recent research on the record-and-replay techniques proposes lightweight approaches to reproduce high-bandwidth stream of inputs and concurrency events [23]. Choi et al. present the tool SwiftHand [9], which generates test scripts for mobile applications with a machine learning algorithm. A few empirical studies compare different testing tools publicly available [10]. All these techniques do not provide the sketch-guided interface as ours, which conveniently helps testers to specify their testing purposes.

Memon et al. provide a survey on model-based testing techniques for GUI-based applications [16], and define a GUI model called Event-flow Graph (EFG) [13, 17]. The EFG model has later been improved by the observe-model-exercise paradigm [21] and a few regression testing techniques [12, 14, 18]. Nguyen et al. integrate these techniques in GUITAR [22], a tool for testing GUIs of PC applications. And two branches of the tool, AndroidRipper [1] and MobiGUITAR [2], are further developed for testing mobile applications with different model traversing strategy. Different from these model-based approaches, our sketch-guided GUI testing provides a natural interface for testers to specify their testing purposes. The sketches both guide the model generation and the model traversing stages, and effectively improve the test coverage.

V. Conclusion and Future Work

We present a sketch-guided GUI test generation approach for testing mobile applications. Testers just need to draw a few simple strokes on the screenshots following the syntax of the sketching language we defined in our system. Then our approach translates the strokes to the Partial Abstract Event-Flow Graph (P-aEFG), and initiates a model-based automatic GUI testing. We evaluate our sketch-guided approach on a few real-world Android applications collected from the literature. The results show that our approach can achieve higher coverage than existing automatic GUI testing techniques with just 10-minute sketching for an application. In the future, we will extend our approach to support in testing the environment of mobile applications, which guides the application in different device environments and uncovers more bugs consequentially.

Acknowledgements

This research was supported in part by National Natural Science Foundation of China (No. 61772260, 61402222, 61632015 and 61690204) and National Key Research and Development Program of China (No. 2016YFB1000802).

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using GUI ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, 2012.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, Sept. 2015.

[3] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE 2012, pages 59:1–59:11, 2012.

[4] AppBrain. Android operating system statistics. http://www.appbrain.com/stats/stats-index, last accessed in November 2016.

[5] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pages 641–660, 2013.

[6] Y. M. Baek and D. H. Bae. Automated model-based android GUI testing using multi-level GUI comparison criteria. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 238–249, Sept. 2016.

[7] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *30th International Conference on Automated Software Engineering (ASE)*, pages 669–679, 2015.

[8] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544. ACM, 2010.

[9] W. Choi, G. Necula, and K. Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In *Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pages 623–640, 2013.

[10] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *30th International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[11] L. Clapp, O. Bastani, S. Anand, and A. Aiken. Minimizing GUI event traces. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 422–434, New York, NY, USA, 2016. ACM.

[12] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon. SITAR: GUI test script repair. *IEEE Transactions on Software Engineering*, 42(2):170–186, Feb. 2015.

[13] A. M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, Sept. 2007.

[14] A. M. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):4:1–4:36, Nov. 2008.

[15] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.*, pages 260–269, Nov. 2003.

[16] A. M. Memon and B. N. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 80 of *Advances in Computers*, pages 121–162. Elsevier, 2010.

[17] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *8th European Software Engineering Conference Held Jointly with 9th International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 256–267, 2001.

[18] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, Oct. 2005.

[19] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 559–570, New York, NY, USA, 2016. ACM.

[20] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[21] B. Nguyen and A. Memon. An observe-model-exercise paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 40(3):216–234, March 2014.

[22] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1):65–105, 2014.

[23] Z. Qin, Y. Tang, E. Novak, and Q. Li. MobiPlay: A remote execution based record-and-replay tool for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 571–582, 2016.