2014 Eighth International Conference on Software Security and Reliability

ADAutomation: An Activity Diagram Based Automated GUI Testing Framework for Smartphone Applications

Ang Li, Zishan Qin, Mingsong Chen* and Jing Liu
Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
Email: {ali, zsqin, mschen, jliu}@sei.ecnu.edu.cn

***Abstract*—Under the increasing complexity and time-to-market pressures, functional validation is becoming a major bottleneck of smartphone applications running on mobile platforms (e.g., Android, iOS). Due to the GUI (Graphical User Interface) intensive nature, the execution of smartphone applications heavily relies on the interactions with users. Manual GUI testing is extremely slow and unacceptably expensive in practice. However, the lack of formal models of user behaviors in the design phase hinders the automation of GUI testing (i.e., test case generation and test evaluation). While thorough test efforts are required to ensure the consistency between user behavior specifications and GUI implementations, few of existing testing approaches can automatically utilize the design phase information to test complex smartphone applications. Based on UML activity diagrams, this paper proposes an automated GUI testing framework called *ADAutomation*, which supports user behavior modeling, GUI test case generation, and post-test analysis and debugging. The experiments using two industrial smartphone applications demonstrate that our approach can not only drastically reduce overall testing time, but also improve the quality of designs.**

I. Introduction

Along with the fast growing market of smart mobile devices such as smartphones and tablet computers, the availability and popularity of smartphone applications have drastically increased. It is reported that, as of the end of October 2013, more than 1 million Apple iOS applications were active, and their overall downloads had exceeded 60 billion times [1]. Similarly, for the Android applications, more than 1 billion downloads are being conducted every month [4]. Since more and more people are relying upon smartphone applications to manage their bills, schedules, emails, shoppings, and so on, it is required that smartphone applications should be user-friendly and reliable.

GUI has become ubiquitous for interacting with today's smartphone application software. Any GUI design flaw will not only jeopardize user experience, but also cause some unimaginable catastrophes. Due to the increasing complexity of software and hardware designs, huge amounts of GUI testing efforts need to be done before the shipping of products. However, with the fierce competition in the smartphone device market, the time-to-market determines the success of smartphone application products. Undoubtedly, manual GUI testing is time-consuming and will cause the delay to market. Therefore, it is urgent to develop automated testing techniques as an alternative.

Automated GUI testing requires accurate modeling of user behaviors to simulate the interactions between users and GUIs. In classical top-down design flow, user behaviors are comprehensively defined in the design phase. They specify all the desired user behaviors which should be correctly reacted by smartphone applications. These information will then be used to instruct the GUI implementation as well as the following GUI testing. Currently, state-of-the-art GUI testing approaches and tools [24], [25], [26] are mainly based on the event modeling using finite state machines (FSMs). Although FSM-based models are promising in describing internal control flow scenarios of GUI implementations, it cannot reflect the overall user behaviors in a natural and compact manner. Therefore, it is hard for FSM-based approaches to guarantee the precision and adequacy of the automated GUI testing. The Unified Modeling Language (UML) is a de facto modeling language that can specify, visualize and construct various artifacts of software systems [5]. As a kind of behavior specifications, activity diagrams are widely used to describe both sequential or concurrent workflows of stepwise activities and actions. The flexibility in describing various control flows makes activity diagram a promising candidate for user behavior modeling to enable the automated GUI testing for smartphone applications.

Aiming at reducing overall testing efforts as well as improving test adequacy, this paper proposes an automated GUI testing framework *ADAutomation* for smartphone applications, which supports user behavior modeling and extraction, automated GUI test case generation and automated test coverage analysis and debug. This paper makes three major contributions: i) we extend the semantics of UML activity diagrams to enable user behavior modeling for smartphone applications; ii) we propose an effective approach for automated GUI test generation and test adequacy analysis from the extracted user behaviors; and iii) we implement the framework *ADAutomation* as a tool chain which performs GUI testing and debugging automatically.

The remainder of the paper is organized as follows. Section II presents related research literatures on model-driven GUI testing and automated GUI test case generation. Section III presents our automated GUI testing framework *ADAutomation* in details. Section IV shows the GUI testing results of two industrial designs using *ADAutomation*. Finally, Section V concludes the paper.

II. Related Work

User behaviors play an important role in GUI testing of smartphone applications. During the GUI testing, a test case indicates a complete user behavior which is a sequence of correlated events/actions. Figuring out dependence between events/actions of user behaviors is the key process of the GUI test case generation. As a promising heuristic approach, genetic algorithms were widely studied in user behavior exploration. Based on event-flow models, Rauf et al. utilized genetic algorithm to search for the best possible test parameter combinations according to some predefined test criterion [31]. Plan generation [33] is another approach to derive GUI test cases in the form of action sequences. For the plan generation approach, its input is the goal of a specific GUI, and its output is sequences of actions which can reach this goal. To enable the automation of GUI test case execution, Chang et al. proposed a computer vision technique [28]. It can specify which GUI components to interact with and what visual feedback to be observed. However, all the above approaches do not fully consider user behaviors for the test adequacy purpose [12].

This work was supported by NSF of China (61202103, 91118007), Innovation Program of Shanghai Municipal Education Commission (14ZZ047), Open Project of SW/HW Co-design Engineering Research Center of MoE (2013001), and Shanghai Knowledge Service Platform Project (ZF1213).

* Corresponding author.

978-1-4799-4296-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SERE.2014.20

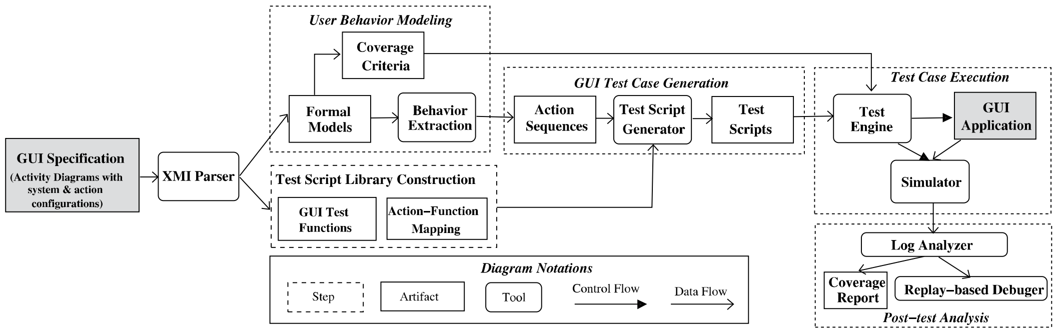

Fig. 1. The overview of our *ADAutomation* Framework

To guarantee the structural coverage of GUI applications [24], proper design models and fault models should be considered during test case generation. Model-based approaches [15] are widely investigated to enable the automation of GUI testing [20]. They can not only improve the test efficiency and sufficiency, but also can check the consistency between the specifications and implementations in a top-down design flow [10], [8]. As a promising formal model, Finite State Machine (FSM) is widely used in automated GUI test case generation. To extract FSMs from GUI designs, various approaches were investigated. For example, Shehday et al. [17] proposed a technique that transforms a given GUI into Variable Finite State Machine (VFSM) model, which can then be converted into an equivalent FSM. In [19], White et al. proposed a method that can transform a GUI into different tasks in the form of reduced FSM models. Based on FSM models, Belli [21] introduced a holistic view of fault modeling that can improve the effectiveness of generated test cases. He proposed some specific model-based coverage criteria on faulty interaction pairs and complete interaction pairs. By exploring on the obtained reduced FSM models using these coverage criteria, a set of GUI test cases can be generated. Besides FSM models, several systematic techniques based on graph models (e.g., event flow graph [30], [32], event interaction graph [27], and event dependency graph [22]) for GUI modeling have also been developed. In these graph models, nodes indicate GUI events and paths represent event sequences, which can be processed and converted into GUI test cases. Although FSM models and graph models can enable automated GUI test case generation, achieving them from complex GUI implementations could be tedious and error-prone. Furthermore, FSM models and graph models focus on the internal logic of GUI designs rather than external user behaviors, which cannot fully explore all possible input event/action sequences.

Based on the semantics of Petri-nets [13], UML activity diagrams are promising in describing both sequential and concurrent events/actions. Therefore, the use of UML activity diagrams for testing has been extensively studied [7], [8], [9], [10], [11]. In [16], Vieira et al. utilized activity diagrams to describe the functional GUI behaviors which need to be tested. However, for the UML activity diagram based testing, few of existing researches considered the hierarchal structures. As a high class of Petri-nets, Hierarchical Predicate Transitions Nets (HPTN) can recognize and treat both events (desirable and undesirable behaviors) and states (desirable and undesirable conditions) equally. In [18], Reza et al. proposed an HPTN based testing method to test the structural representation of GUIs. Although Petri-net base approach can perfectly describe user behaviors, none of them has been used in automated GUI testing.

The drastic increment of smartphone applications and corresponding downloads creates an impetus for developing cost-effective testing techniques to ensure the reliability [23]. In [38], [29], Hu and Neamtiu performed a study to understand the nature and possible remedies for bugs in smartphone applications, and constructed an automated testing framework for Android applications. They presented techniques for detecting GUI bugs by automatic generation of test cases, feeding the application random events, instrumenting the VM, producing log/trace files and analyzing them in post-run. In [14], Takala et al. presented how model-based test automation were implemented with Android GUI applications. However, most existing approaches focus on the testing of a single GUI view rather than the overall user behavior across multi-views.

To the best of our knowledge, our approach is the first one that systematically adopts UML activity diagrams to support user behavior modeling, automated GUI testing and GUI debugging for smartphone applications. We have developed a tool chain *ADAutomation* to support all these activities.

III. Our ADAutomation Framework

Since UML activity diagrams are good at describing the relations between actions, they naturally fit smartphone applications for GUI testing purpose. In our approach, we adopt activity diagrams to model user behaviors. By exploring all possible action sequences with a specific bound of behavior length from activity diagrams, we can generate a set of GUI test cases to validate GUI implementations.

Figure 1 shows the workflow of our automated GUI testing framework *ADAutomation*. It has five major steps: user behavior modeling, test script library construction, GUI test case generation, test case execution and post-test analysis. During the early design phase, UML activity diagrams derived from GUI specifications are used as a semi-formal model for user behavior modeling. Here, user behaviors are defined as sequences of user actions conducted on GUI widgets of smartphone applications. User behavior modeling tries to explore all possible user behaviors and proper coverage criteria for the test adequacy evaluation. To enable GUI testing, system configuration describes the target testing platform information, and for each action configuration specifies the GUI widget layout information (e.g., position, label, etc.), GUI widget layout information (e.g., position, label, etc.), and the action-function mapping. By using our XMI parser, all such information will be extracted and used to construct a test script library (i.e., a set of GUI test functions). Based on extracted action sequences from activity diagrams and derived function libraries, the test script generator can translate user behaviors into corresponding GUI test cases. During the test case execution stage, a test engine will instruct the simulator to run the test scripts on smartphone applications and log the execution and coverage information. Finally, by analyzing the simulation logs, the testing results (i.e., coverage and errors) will be reported. If there are some inconsistent behaviors between GUI specifications and implementations, the simulation logs of the error behaviors will be fed into a replay-based debugger to figure out the reasons of the inconsistencies. The following sub-sections will present the above major steps of *ADAutomation* in details.

Based on our proposed framework, we create a tool chain that integrates the UML front-end edit tool *Enterprise Architect* [36] for activity diagram modeling. In the tool chain, we use the *XCode Instruments* [37] from Apple and Android *Robotium* [3] to conduct test case execution and post-test analysis for iOS and Android application respectively. We developed the XMI parser and coverage analysis tools, which support both automated test generation and post-test analysis.

A. Behavior Modeling using Activity Diagrams

This section first presents the formal notations for UML activity diagrams. Then it proposes three coverage criteria that can measure the adequacy of generated user behaviors. Finally, it presents how to extract all the user behaviors from activity diagrams for GUI test case generation.

1) Notations: UML activity diagrams, with the Petri-net [13] like semantics, are widely used to coordinate the execution of actions and activities [5], [6]. We adopt UML 2.4 [6] as our specification for user behavior modeling. In activity diagrams, an activity consists of a set of actions connected by control flow edges to indicate the execution order. For the purpose of GUI testing, an activity can be considered as a view, and an action can be considered as external operations that can make a state change of smartphone applications. In our approach, we only adopt a subset of activity diagram elements including activity nodes, action nodes, control nodes, and control flow. We do not consider the object nodes and data flow, since GUI actions are triggered by external events/operations rather than data.

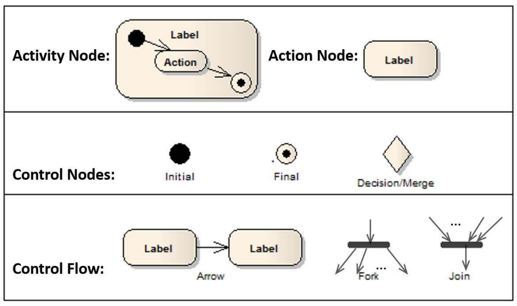

Fig. 2. Basic activity diagram constructs for GUI testing

Figure 2 shows the major constructs of UML activity diagrams used in user behavior modeling. It includes:

- *Activity node:* A smartphone application usually consists of multiple GUI activities (views), where each activity is a collection of correlated user actions. According to the definition of activity diagrams in UML 2.4 [6], concurrent actions and activities can preempt each other. However, this is not allowed in our GUI modeling. In our approach, the execution of a sequence of actions in an activity cannot be interrupted by the actions of other activities. An activity node can be refined as another activity diagram.
- *Action node:* An action indicates an individual execution step within an activity. In GUI testing, it can be considered as a sequence of correlated external events (e.g., tap, pinch or flick). Since the action is considered to be atomic, it cannot be preempted by other actions during the execution. An activity node with only one action can be considered as an action node. An action node cannot be refined further.
- *Initial node:* The initial node indicates the start of a flow, i.e., a sequence of GUI actions.
- *Final node:* When the final node is reached, the entire flow of the activity in which the final node resides will be terminated.
- *Decision/Merge node:* The diamond represents a decision or merge node, which is a kind of control nodes to determine the sequences of actions. Decision nodes choose one outgoing flow according to the constraints labeled on the outgoing edge. Merge nodes select only one incoming flow to deliver to the next activity node.
- *Arrow line:* Arrow lines indicate the execution order of actions. In the control flow, the incoming arrow line indicates the start of an action, and the outgoing arrow line represents the completion of the action.
- *Fork/Join:* Fork or join flows are used to describe concurrent behaviors of a system. They are shown by multiple arrows leaving or entering a synchronization bar respectively. In GUI-based smartphone applications, concurrent behaviors are not necessary since only one external action can be triggered at a time. We adopt fork/join flows, since they can describe the interleaving of actions.

Figure 3 presents the workflow of a GUI-based download application using an activity diagram. In this example, users must login to their accounts (the flow is described by the activity *Login*) before downloading files. If a user taps the *download* button (i.e., action c) without login, an error message will be prompted. The nested *Login* activity in the diagram has three actions (i.e., actions e, f, and g). During the processing of actions in *Login* activity, the activity cannot be disrupted by any other actions outside the activity. However, when handling actions a, b or c, users could try to login to their accounts.

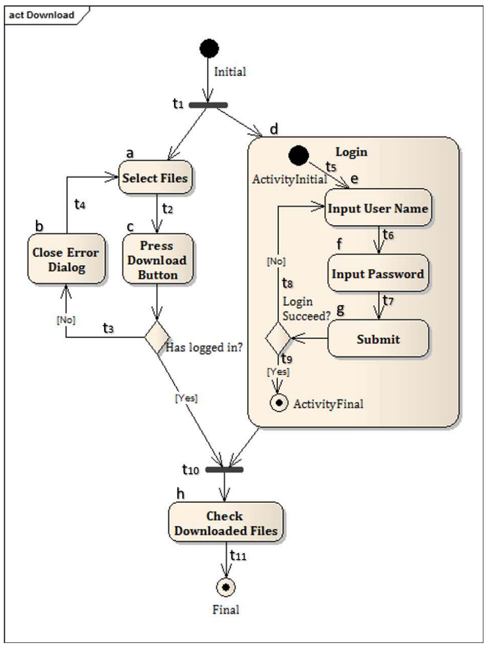

Fig. 3. The activity diagram for a file download application

As a semi-formal specification, UML activity diagram itself cannot be automatically analyzed. To accurately describe user behaviors, we extend the Petri-net as an intermediate formal model for UML activity diagrams, since Petri-net based semantics can capture both sequential and concurrent behaviors. Definition III.1 describes the relation between actions and control flow edges with a quasi-Petri-net semantics. It does not model the full features of activity diagrams, but just formally depicts the static abstracted structure of activity diagrams. Unlike the definitions in [8], definition III.1 considers nested activities based on the depth of activity hierarchy. In our approach, the actions in the current view cannot be preempted by the actions from other views. This extension enables smooth description of user events across multiple GUI views in smartphone applications.

Definition III.1. *An activity diagram is a directed graph described by a five-tuple (A, T, F, a_I, a_F) where*

- $A = \{a_1, a_2, \ldots, a_m\}$ *is a set of m action nodes.* $depth(a_i) \in \mathbb{N}$ *indicates the activity hierarchy depth of* a_i*, and* $ID(a_i) \in \mathbb{N}$ *indicates the ID of* a_i*.*
- $T = \{t_1, t_2, \ldots, t_k\}$ *is a set of k completion transitions.*
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ *is a set of flow edges between activity nodes and completion transitions.*
- $a_I \in A$ *is the initial node, and* $a_F \in A$ *is the final node. There is only one completion transition* $t \in T$ *and* $c \in C$ *such that* $(a_I, t) \in F$*, and for any* $t' \in T$ *,* $(t', a_I) \notin F$ *and* $(a_F, t') \notin F$*.* ■

In this definition, we only focus on the relations between actions. We unroll all the nested activities by removing all the activity hierarchies and making the initial/final nodes of nested activities as dummy control nodes with no delay. Note that the initial and final nodes of the topmost activities are kept. We associate each action a_i with a parameter $depth(a_i)$ to indicate the depth of activity hierarchy. The depth of actions in the topmost activity is 0, and the depth of actions in nested activities will be increased according to their hierarchy levels. As an example shown in Figure 3, the depth of a is 0, and the depth of e is 1. It is important to note that the initial/final nodes have the same depth as their enclosing activities. For example, the depth of action *ActivityInitial* in activity *Login* is 0 rather than 1.

To simplify the definition of control flows, we use the *completion transition* and *flow edge* to model system behaviors. In Definition III.1, actions are connected by flow edges associated with a completion transition. In Figure 3, there is a completion transition t_2 between the actions a and c, corresponding two flow edges (a, t_2) and (t_2, c). Because activity diagrams allow concurrent flows, the completion transition also can be used to perform synchronization. If a completion transition has multiple active incoming flow edges, it will do the join operation. If there are multiple outgoing flow edges and all the incoming flows are active, it will do the fork operation. For example, t_{10} indicates a join operation, which has two input flows (c, t_{10}) and (*ActivityFinal*, t_{10}), and one output flow (t_{10}, h). Note that in our definition the decision nodes are abstracted.

For the testing purpose, we need to enumerate all the possible user behaviors from activity diagrams. When analyzing dynamic behaviors of an activity diagram, we use the *state* to model the status of a system. The current state (denoted by *CS*) of an activity diagram indicates the set of actions which are ready for execution.

Definition III.2. *Let* $D = (A, T, F, a_I, a_F)$ *be an activity diagram. The current state CS of D is a subset of A. For any transition* $t \in T$*,*

- ${}^\bullet t$ *denotes the preset of t, i.e.,* ${}^\bullet t = \{\ a \mid (a,t) \in\ F\}$*.* $t^\bullet$ *denotes the postset of t, i.e.,* $t^\bullet = \{a \mid (t,a) \in F\}$*.*
- $enabled(CS)$ *denotes the set of completion transitions that can be triggered, i.e.,* $enabled(CS) = \{\ t \mid {}^\bullet t\ \subseteq\ CS \bigwedge \forall act \in {}^\bullet t.\ (\forall n \in CS.\ depth(act) >= depth(n))\}$*.*
- *firable(CS) denotes the set of completion transitions that can be fired from CS , then firable(CS)=*$\{\ t \mid t \in$ *enabled(CS)* $\bigwedge$ ${}^\bullet t$ *are all completed* $\bigwedge\ \exists\ n \in A.\ (CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$*. After some t is fired, the new current state* $CS' = fire(CS,t) = (CS - {}^\bullet t) \cup t^\bullet$*.* ■

Definition III.2 slightly changes the semantics of activity diagrams. Although an activity diagram can fire multiple concurrent transitions simultaneously, our approach only allows to fire one transition at a time. In our approach, only the transition associated with the deepest actions can be fired. In Figure 3, $\{a, e\}$ is the current state of the activity diagram shown in Figure 3. Although there are two outgoing transitions for the current state, only the transition from e to f is enabled, since the action e is a non initial/final action that resides in a nested activity with the largest depth. If the transition is fired, the next state will be $\{a, f\}$. It is important to note that the concurrent semantics of activity diagrams enables the interleaving between actions and activities.

Definition III.3. *A run* ρ *of an activity diagram D is a sequence of states and transitions, let*

$$\rho = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} s_n$$

where $s_0 = \{a_I\}$*,* $s_n = \{a_F\}$*, and* $s_{i+1} = fire(s_i, t_i)$ *for any i* $(0 \leq i < n)$*.* $P =< a_I,\ a_{1,1},\ a_{1,2},\ \ldots,\ a_{i,1},\ a_{i,k},\ \ldots,\ a_F >$ *is a path of* ρ *if* $s_i - s_{i-1} = \{a_{i,1},\ \ldots,\ a_{i,k}\}$ $(1 \leq i \leq n,\ k > 0)$ *where* $a_{i,k}$ *is sequentially ordered by* $ID(a_{i,k})$ *when the completion transition* t_{i-1} *is fired. Let* $P =< a_I, a_1,\ \ldots,\ a_m, a_F >$ *be a path of length m. P is a simple path if there is no action repetition, i.e.* $\forall i, j\ (0 < i < j \leq m)$*,* $a_i \neq a_j$*.* ■

The dynamic behavior of an activity diagram can be represented by a sequence of actions. We call it a *path* of the activity diagram. Due to the concurrency and loops, it is impossible to determine how many paths exist in a given activity diagram. Therefore, we adopt the *simple path*, which is a path without any loops, to reduce the overall size of the path set. When exploring a path in a UML activity diagram, we do not consider non-action nodes (i.e., control nodes) or initial/final nodes in the path length calculation. As an example shown in Figure 3, $< a_I, a, c, e, f, g, h, a_F >$ is a simple path with a length of 6.

2) Test Adequacy Criteria: Testing adequacy criterion specifies the requirement of a particular testing. Unlike the traditional testing which is based on the source code coverage, during the activity diagram based GUI testing, we need to consider all the structural artifacts as well as the dynamic behaviors of activity diagrams. To measure the effectiveness of test cases derived from activity diagrams, we propose three types of coverage metrics as follows.

- *Action Coverage Criterion* requires that all the actions to be covered. The action coverage equals to the ratio between the tested actions and all the actions in an activity diagram.
- *Transition Coverage Criterion* requires that all the completion transitions to be covered. The transition coverage equals to the ratio between the checked transitions and all the transitions in an activity diagram.
- *Simple Path Coverage Criterion* requires that all the simple paths to be covered. The simple path coverage equals to the ratio between the traversed simple paths and all the simple paths in an activity diagram.

The action coverage criterion checks the reachability of the specified actions in activity diagrams. The transition coverage criterion tests all the branches along the control flow. Due to the existence of loops, it is hard to determine how many paths in a given activity diagram. We adopt the simple path coverage since all the simple paths should be all feasible during the execution. Note that transition coverage is stronger than the action coverage. However, since simple paths may not consider some actions/transitions in the loop, the simple path coverage criterion may not cover all transitions or actions.

In practice, achieving 100% coverage of the above categories still cannot guarantee the test adequacy. The above coverage criteria are only used to describe basic scenarios that should be investigated. According to Definition III.3, the simple path does not consider loops, and the order of actions associated to fork transitions is fixed. Therefore, a simple path may correspond to a large set of possible event sequences in real executions. In this case, just checking one representative from this set is not enough, though 100% simple path coverage can be achieved. In other words, only when a sufficiently large set of possible user behaviors (random or directed) is tested, the above coverage criteria can really reflect the test adequacy.

3) User Behavior Extraction: As described in Section III-A, UML activity diagrams can be used to model user behaviors. For the adequacy of GUI testing, it is necessary to figure out as many as possible user behaviors. However, due to the loops in activity diagrams, it is impossible to enumerate all such information. Therefore, during the user behavior extraction, we restrict the length of paths within a limited bound. Since the fork operation triggers outgoing actions simultaneously by semantics, during the behavior extraction, we fix the execution order of the adjacent actions of fork transitions. For the other non-adjacent actions in the parallel flows, different interleavings will be considered as different behaviors.

Algorithm 1: Our Path Exploration Algorithm

```
Input: i) $G$ is the activity diagram for path exploration;
       ii) $L$ is the max length of user behaviors set by tester;
       iii) $CS$ indicates the current state;
       iv) $path$ indicates the incomplete run till $CS$;
       v) $Paths$ stores the complete user behaviors found so far;
Output: A set of user behaviors $Paths$ starting from $CS$
1  pathExploration($G, L, CS, path, Paths$) begin
2      if $path.length > L$ then
3          return $\varnothing$;
4      end
5      $firableTrans = firable(CS)$;
6      if $|firableTrans| == 0$ then
7          $Paths.add(path)$;
8          return $Paths$;
9      end
10     foreach $tran \in firableTrans$ do
11         $preSet = {}^{\bullet}tran$;
12         $postSet = tran^{\bullet}$;
13         $CS' = (CS - preSet) \cup postSet$;
14         $segment = ordered(postSet - preSet)$;
15         $path' = path.append(segment)$;
16         $pathExploration(G, L, CS', path', Paths)$;
17     end
18     return $Paths$;
19 end
```

Algorithm 1 presents the procedure to obtain all the user behaviors whose lengths are no larger than a given bound. Unlike the algorithm proposed in [10], [8], our approach considers both the hierarchy information presented in Section III-A and the bound information. The algorithm searches the user behaviors in a depth-first way. In this algorithm, lines $2-4$ handle the case when the searched path is longer than the specified length limit. Line 5 calculates the firable transitions of the current state CS according to the Definition III.2. If none of the transition can be fired, that means a new complete path has been found. In this case, lines $6-9$ record the new path and abort the following recursive search. Since GUI testing can only trigger one action at a time, when there are multiple firable transitions, we need to fire each of them individually. Lines 11 and 12 figure out the preset and post of a firable transition. Lines 13-15 calculate the new state and new incomplete path when the fire of the transition is triggered. Line 16 continues to explore the remaining part of the paths recursively. Finally, line 18 reports all the explored paths.

B. Test Script Library Construction

When all the action sequences are extracted from a given UML activity diagram, we need to translate them into corresponding GUI test cases. In our approach, each action in a given activity diagram has one associated GUI test function. A GUI test case is described using a sequence of test functions. Therefore, it is necessary to set up a test script library consisting of test functions of all user actions, where each function in the library can describe a set of strongly correlated GUI events. As well, a one-to-one mapping from the actions of activity diagrams to the functions in the test script library should also be provided. Consequently, the GUI test case generation is a process to transform a sequence of actions to a test script that consists of a sequence of test functions. Such a script can be fed into some test engine to control the GUI test process.

At present, there are only a few test engines that support behavior-level GUI test automation for smartphone applications. The most popular ones are: i) *Apple XCode Instruments* which uses *UIAutomation JavaScript library* to conduct black-box GUI testing; and ii) *Android Instrumentation*, which uses *Robotium* to do black-box GUI testing on APK-packaged applications. These tools enable users to write test scripts for GUI testing. Due to the space limitation, this subsection will only introduce the automated GUI test case generation with examples using *XCode Instruments* and its *UIAutomation* library for iOS applications. The same process can be easily applied to the test case generation for Android applications.

1) Activity Diagram Annotation: Since UML activity diagrams only focus on the GUI action flow modeling, they do not contain any details of target operating systems, GUI layout or user actions. To ensure that the derived GUI test cases can be applied on different smartphone platforms, when modeling activity diagrams, testers need to specify the target testing platform (e.g., iOS or Android) and provide the layout information of both manipulated GUI widgets and corresponding GUI operations for each activity diagram action. Such configuration information will be saved in XML format and integrated into the XMI (XML Metadata Interchange) files of UML activity diagrams. During test case generation, these information will be extracted to construct test script libraries for different smartphone platforms.

```
<Platform>
   <Name>iOS</Name>
   <Version>6.0</Version>
   <TestEngine>
      <Name>Xcode Instruments</Name>
      <Version>4.6</Version>
      <Delay>
         <Unit>second</Unit>
         <Value>0.2</Value>
      </Delay>
   </TestEngine>
</Platform>
```

Listing 1. An example of system configuration for *PicFlick*

Our approach uses the *system configuration* as a top level description for the whole activity diagram, which specifies the target test platform of the smartphone applications as well as the parameters (e.g., name, version, delays between GUI events, etc) for the corresponding test engine. Listing 1 shows a system configuration for the activity diagram of the application *PicFlick* presented in Section IV.

To enable automated test function derivation, the relation between an action and corresponding GUI operations should be clearly specified. We created the *action configuration* to specify such relation. For each action, the configuration contains following information.

1) **Widget features** specify the attributes of corresponding GUI widgets, including ID, position, size, and etc.
2) **GUI operations** describe consecutive user operations conducted on the associated GUI widgets.
3) **Test logs** instrument proper log information based on the results of GUI operations.

Listing 2 shows an action configuration example for the action *"Choose Photos in the Album"* in *PicFlick*.

```
<UserActions>
   <FunctionName>selectPhotos</FunctionName>
   <Log>"Select␣Photos"</Log>
   <Operation>
       <OPIndex>0</OPIndex>
       <Object>
          <Type>TableView</Type>
          <OBIndex>0</OBIndex>
          <Log>
             <Cond>NotEqualNULL</Cond>
             <Pass>"TableView␣0␣exits"</Pass>
             <Fail>"TableView␣0␣does␣not␣exit"</Fail>
          </Log>
       </Object>
       <action>Tap</action>
       <!--AccessMode can be ALL, RAND, and SELECT-->
       <AccessMode>ALL</AccessMode>
       <ScreenShot>Enabled</ScreenShot>
   </Operation>
   <Operation> ...... </Operation>
</UserActions>
```

Listing 2. An action configuration example of *PicFlick*

To achieve widget features, testers should figure out the GUI layout and design information for smartphone applications. Such information can be obtained from GUI designers if the smartphone application has not been developed yet. If the smartphone GUI design is ready and the code is available, many existing tools can be used to figure out the GUI layout. For example, based on the Apple *UIAutomation* library, each GUI component of an iOS application can be accessed as a *UIAElement* object using Javascript. By calling *logElementTree()* function provided in *UIAutomation* library, we can obtain the corresponding hierarchy tree of all the *UIAElements* of the current view. For Android platform, the similar hierarchy tree of GUI layout could be obtained by using *HierarchyViewer* contained in *Android Developer Tools* (ADT).

Based on widget features, test functions can get references to GUI widgets to perform various operations. Generally, there are three major ways to achieve references to GUI widgets: by name, by layout and widget index, and by the coordinate of screens. Although our action configuration supports all these three approaches, we do not suggest to access widgets directly using the coordinate of screens. This is because the location calculation of widgets can be very complex in many scenarios (e.g., rotation of screens, different zoom-levels of screens). Listing 2 shows an example which access a *TableView* widget using the layout and index information. The associated first operation (with *OPIndex* 0) will tap all the elements of the first table (with *OBIndex* 0) in the current view.

In addition to describing user operations on GUI widgets, our action configuration allows to specify the instrumentation of log information as well as *ScreenShot* operations for GUI test design. Such configuration items enable the automated post-test analysis and debug. During the test case execution, our derived test cases support the log of both successful and failed testing scenarios. The log operations are triggered by specific conditions or events to record the success points and failure points during the simulation. As an example in Listing 2, no matter whether the *TableView* exists or not, a corresponding log information will be reported. Since various log functions have been widely used in GUI test automation on both iOS and Android systems, during the test script library construction, the log specifications in the action configuration can be easily transformed and instrumented into test script functions.

2) Script Library Construction: In *ADAutomation*, we use the *Enterprise Architect* as our front end activity diagram edit tool. By using this tool, we can attach the activity diagram with the system configuration as well as action configurations to enable the automated function library generation. By using the XMI parser in our framework, the test script library can be generated automatically.

```
1   var target  = UIATarget.localTarget();
2   var app     = target.frontMostApp();
3   var win     = app.mainWindow();
4
5   function selectPhotos()
6   {
7       UIALogger.logMessage("Select␣Photos");
8       target.delay(0.2);
9       var tView = win.tableViews()[0];
10      if(tView==null){
11          UIALogger.logFail("TableView␣0␣does␣not␣exist");
12      } else {
13          UIALogger.logPass("TableView␣0␣exists");
14      }
15      var widgetID = 0;
16      while(widgetID<tView.visibleCells(),length){
17        tView.visibleCells()[photoID].tap();
18        widgetID = widgetID + 1;
19      }
20      screenShot();
21  }
22  function choosePhotoAlbum()
23  {
24      UIALogger.logMessage("Choose␣a␣photo␣album");
25      ...
26      screenShot();
27  }
28  function photoViews() {...}
29  function dragPhoto() {...}
30  ...
```

Listing 3. An overview of the test script library for *PicFlick* (iOS)

Listing 3 presents the iOS version of the test script library derived using our XMI parser for the *PicFlick* design presented in Section IV. The library consists of two parts: global variables and test function definitions. Global variables are used to declare some system variables shared by all test functions, and test function definitions describe the user operation details for each action. Since we adopt the *UIAutomation* tool for GUI testing, the whole function library is constructed in the form of Javascript. There are totally 30 functions in this library. Due to the space limit, we only present 4 of them. To enable the following debug and error analysis, each function is instrumented with log information. As specified in Listing 2, the line 7 of Listing 3 indicates the log action start; lines 11 and 13 are used to record null object reference exceptions during the execution; and line 20 shoots the GUI screen to record the action result.

C. Automated Test Script Generation

Generally, a GUI test case contains three major parts:

1) **Initialization** includes the functional library and initializes the global data structure.
2) **Body** dispatches each user action in the same order as the sequence of user actions.
3) **Finalization** ends current test and print log information.

For the same smartphone application, the initialization and the finalization parts of the test cases are same. Therefore, we can use the same template for these two parts in all derived test cases. Since the relation between actions and functions is a one-to-one mapping (see the annotation <*FunctionName*> in Listing 2), the translation from a user behavior to the corresponding GUI test case is straightforward. For example, let a_I →*Go to Photos View*→*Choose Album*→*Choose Photos in the Album*→a_F be a complete sequence of user actions. Based on the extracted action-function mapping from the UML activity diagram, we can generate the iOS test script in the form of Javascript as follows.

```
#import "picflick.js"
UIALogger.logStart("Testing␣starts");
photoViews();
choosePhotoAlbum();
selectPhotos();
UIALogger.logPass("Testing␣ends!");
```

Listing 4. A translated GUI test script for *PicFlick* (iOS)

D. GUI Testing and Error Diagnosis

Test engines (e.g., *XCode Instruments*) play an important role during the testing. It can not only start the simulator to set up the testing environment, but also can trigger external events to interact with the GUI implementations using generated test scripts. During the simulation, we need to validate each test case against the GUI implementation. If a test case can be simulated successfully, it will be used to calculate the accumulative coverage information. Otherwise, the failed test cases as well as the failed scenarios will be recorded for the following error diagnosis.

Algorithm 2: Calculation of Simple Path Coverage

```
Input: i) $SP$ is the set of simple paths of an activity diagram;
       ii) $P$ is the set of the paths associate with the passed test cases;
Output: Simple path coverage for $P$ over $SP$
1  SPCoverage($SP$, $P$) begin
2      $SP\_Size = |SP|$;
3      while $P \neq \emptyset$ & $SP \neq \emptyset$ do
4          $p = P.element(0)$ in the form $< x_1, x_2, \ldots, x_m >$;
5          $P = P - \{p\}$;
6          foreach $sp \in SP$ do
7              /*$sp$ is in the form $< y_1, y_2, \ldots, y_n >$*/;
8              if $\{y_1, y_2, \ldots, y_n\} \subseteq \{x_1, x_2, \ldots, x_m\}$ & $p$ can be simulated
               on the activity diagram then
9                  $SP = SP - \{sp\}$;
10             end
11         end
12     end
13     return $(SP\_Size - |SP|)/SP\_Size$;
14 end
```

Calculating the GUI coverage is a major task during the test case simulation. In our approach, we only deal with three kinds of coverage criteria introduced in Section III-A2. Generally, the action coverage and the transition coverage are easier to be obtained than the simple path coverage, since it only needs to record which action/transition is covered. For the simple path coverage, we need to figure out the matching between a simple path and the traversed paths by the passed test cases. Algorithm 2 describes our approach for calculating the simple path coverage. This algorithm firstly extracts one path from the set *P* (line 4). Then the algorithm iteratively compares this path with each simple path in the set *SP* (lines 6-11). If such two paths match (line 8), then the compared simple path will be removed from the set *SP*. When either *P* or *SP* becomes empty, the whole matching process will be terminated. Finally, line 13 reports the simple path coverage information.

For the failed test cases, we need to figure out the reasons for errors. In our framework, we log various information during the execution of each test case. *MDAutomation* can record GUI actions and their timestamps, user-defined events and suspected screens. Based on the capture-playback function provided by existing tools (e.g., *XCode Instruments*), we can replay the fail scenarios for the debug purpose.

IV. Case Study

By using our proposed framework *ADAutomation*, we did experiments on two smartphone applications: *PicFlick* [34] and *Newsyc* [35]. All the experimental results were obtained on a MacBook Pro machine with Intel Core i5 2.4GHz processor and 4 GB RAM.

A. Experiment 1: PicFlick

PicFlick is a free Wi-Fi based remote picture print management application developed by Eastman Kodak Company. It allows registered users to access, edit photos and send them over a wireless connection to any supported KODAK All-In-One Printer. In this experiment, we did the GUI testing on the beta versions of its both iOS and Android applications. Since the layout and design of both iOS and Android versions of *PicFlick* are almost same, they can be tested using the same UML activity diagram.

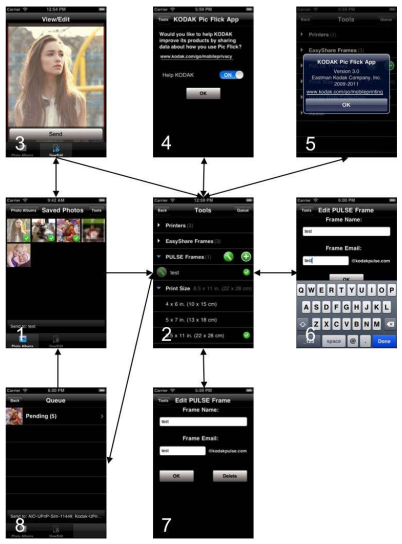

Fig. 4. Eight views of *PicFlick* and their relations in the iOS version

Figure 4 shows all the eight GUI views of *PicFlick* on iOS and the corresponding view switches indicated by the arrow lines. The view 1 presents the entrance of the application, where users can access and select pictures for the following processing. View 2 enters the view *Tools* in which we can set the configurations of pictures (e.g., size, quality and etc.) and remote devices (e.g., printers or digital frames). View 3 shows one of the selected pictures. View 4 is used to collect the user feedback (i.e., usage data) of the product. View 5 displays

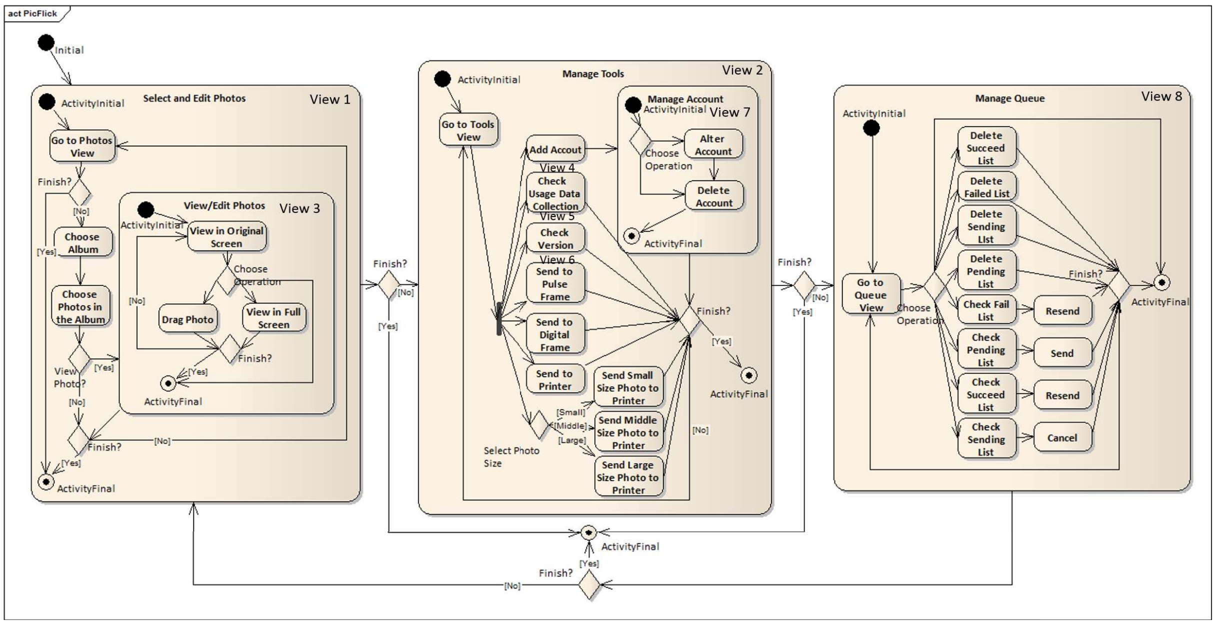

Fig. 5. The Activity Diagram for PicFlick

the product version information. View 6 creates a new picture frame, while view 7 edits an existing frame. By entering the view 8, we can check the status of various lists of processing/processed pictures, including pending list, failed list, and etc.

Before GUI test case generation, we need to model the user behaviors based on GUI specifications (e.g., views provided in Figure 4, and corresponding action information). Since this case study was conducted on an existing industrial design, we can easily achieve its GUI specification from its user manual. To evaluate the effectiveness of our approach, we reinterpret the GUI specification using an activity diagram. Some people may argue that the drawing of UML activities is time-consuming. However, these kind of formal or semi-formal behavior modeling work should be done in the early design phase anyway. Although we spent around three hours to construct the activity diagram, such time cannot be accounted into the overall testing time.

Figure 5 shows the workflow of the *PicFlick* using a UML activity diagram, which models the GUI-oriented user behaviors for the PicFlick application. For each external user input, we model it using an action. For correlated actions inside a GUI view, we group them within in an activity. Since *PicFlick* has 8 individual views, there are totally 8 activities in the activity diagram. Our user behavior modeling approach allows nested activities, which can merge correlated activities together. As shown in Figure 4, the views 4, 5, 6, and 7 can only switch from and return to the view 2. Therefore, we can make them as nested activities inside the activity labeled view 2 in Figure 5. The modeling of view 3 is subtle, since view 3 can switch to view 2 at any time. Therefore, in the activity of view 3, each action can terminated and quit the activity of view 1 immediately.

By using our XMI parser, we achieved two script libraries for both iOS and Android GUI testing respectively. Both libraries contain around 500 lines of code. In this experiment, we set the time interval between two consecutive input actions to 0.2 seconds. We extracted all the possible user behaviors from Figure 5 with a limited number of actions. Based on the derived test script library for *PicFlick*, our tool can automatically translate the user behaviors into concrete GUI test cases. Table I shows the result of GUI testing for *iOS*-based *PicFlick*. In this table, column 1 presents the limited number of actions for each test case. As an example, the last row indicates the test result of all the derived test cases whose length is no larger than 18. Columns 2 and 3 indicate the number of generated test cases and the failed test cases during the testing. Columns 4 presents the overall testing time (test case generation time + execution time). The last 3 columns present the proposed coverage metrics using the generated test cases.

TABLE I
TESTING RESULT FOR *PicFlick (iOS)* USING *ADAutomation*

Bound Size	Test #	Failed #	Testing Time(s)	Action Coverage	Trans. Coverage	S. P. Coverage
1	1	0	8.02	6.67%	6.98%	0.10%
2	2	0	16.06	13.33%	13.95%	0.21%
3	4	0	32.15	26.66%	27.91%	0.42%
4	13	0	103.37	43.33%	46.51%	1.05%
5	30	0	231.01	66.67%	66.28%	1.89%
6	59	0	438.57	66.67%	67.44%	2.62%
7	111	2	797.56	66.67%	67.44%	3.67%
8	176	5	1207.15	83.33%	82.56%	5.03%
9	277	11	1832.30	93.33%	97.67%	6.39%
10	432	14	2773.44	100%	100%	8.07%
11	624	17	3848.24	100%	100%	12.16%
12	881	22	5231.80	100%	100%	19.71%
13	1352	39	3.33.48	100%	100%	31.24%
14	2224	83	13235.72	100%	100%	45.60%
15	3653	127	21627.05	100%	100%	63.21%
16	5784	331	33481.70	100%	100%	81.66%
17	8786	411	49429.83	100%	100%	95.07%
18	12776	993	70217.14	100%	100%	98.95%

To measure the test adequacy, we explored all the coverage elements on the UML activity diagram. We obtained 30 actions, 86 transitions, and 955 simple paths in total to evaluate the test coverage. From this table, we can find that when using all the test cases with a bounded length of 10, we can get 100% activity coverage and transition coverage. The simple path coverage increases more slowly than the other two coverages. Although increasing the bound of the user behaviors can further improve the simple path coverage, the number of generated test cases will be increased exponentially. Since the maximum length of all simple paths of the activity diagram in Figure 5 is less than 19, we do not consider user behaviors whose lengths are larger than 18. This is because that they will not improve the simple path coverage any more.

For the iOS version, when the bound of test case length equals to 18, the test case generation will cost 1896.14 seconds, and the test simulation will cost 68321 seconds. The overall time is 70217.14 seconds. In practice, such testing time can be further reduced by running GUI testing on multiple computers in parallel. However, if all these test cases are generated and run manually, for such a complex smartphone application, it will cost around 2-3 man months. Additionally, due to the lack of user behavior models, it is hard for testers to enumerate all the possible user behaviors. In this case, the test adequacy is mainly determined by the expertise and experiences of testers, which is not objective. During the execution of the 12776 generated test scripts, 11783 of them passed, but 993 of them resulted in application crashes. By checking the simple path coverage and scanning the logs generated during the test case execution, we found 5 suspected bugs in the application.

TABLE II
DETAILS OF BUGS FOUND IN *PicFlick* (*iOS*) TESTING

Index	Failure Scenarios	Failed Test #	Reasons of the failures
1	If the picture is too large, then the drag of the picture may crash.	121	Due to the limited resource for the smartphone application, the drag of big pictures will use up all the allocated CPU and memory resources.
2	If users send pictures to digital frames and printers at the same time, the application will crash.	806	The implementation of the task scheduling between sending list and pending list is wrong.
3	Fail to delete tasks from pending list.	40	The implementation of the delete operation of the pending list is wrong.
4	Fail to tap the sending list button in the *Queue* view.	12	After selecting devices to send photos, the sending list button is disabled by mistake.
5	Fail to find printers which appear in the *Tools* view.	14	The implementation of the connection between PicFlick and the drivers of printers is wrong.

Table II lists the details of the bugs found in the iOS version. In the table, column 1 indicates the index of the errors. Column 2 presents the failure descriptions. Column 3 presents how many test cases failed during the simulation because of the same fault. We had reported all these failures to the developers of the application, and all of them are confirmed as bugs which need to be fixed. Based on the developers' confirmation, the last column presents the reasons for all these failures.

Similar to the approach presented in [29], we also conducted the random testing using the Monkey event generator *UI AutoMonkey* [2] for random GUI test case generation. Unlike [29], we performed the random black-box GUI testing without instrumenting any code to observe event activations. By using *XCode Instruments*, all the random execution results were recorded for coverage analysis and debugging. Table III shows the random testing results using *UI AutoMonkey* on the iOS version of *PicFlick*. For each case with different numbers of actions, we conducted the random testing for one day individually. Here, unlimited means that a test case will be generated as long as possible on-the-fly until it fails. Due to the randomness, we can find that the case with the bound of 20 achieves the worst coverage. This is because that the random event sequences are so short that can hardly hit GUI widgets in different views. Therefore, the simple path coverage is lower than 1%. When adopting larger bound test cases, we can find that all actions and transitions can be covered. Although the random approach use more time (24 hours) than our approach (19.5 hours with the bound size 18), our approach can achieve better simple path coverage (98.95%) than the random approach with unlimited bound (93.89%). From this table, we can find that the random approach only found three bugs which have been reported in Table II. No extra new errors were detected by using the random approach.

TABLE III
RANDOM TESTING RESULTS FOR *PicFlick* (*iOS*)

Bound Size	Action Coverage	Transition Coverage	Simple Path Coverage	Bugs Found
20	47.22%	59.34%	0.93%	0
50	97.22%	97.80%	65.60%	1
100	97.22%	97.80%	85.80%	2
200	97.22%	97.80%	85.80%	2
400	100%	100%	93.89%	3
unlimited	100%	100%	93.89%	3

Since we adopted the same UML activity diagram, when we tested the Android version of *PicFlick* and set the bound of test cases to 18, we get the same test set with 12776 test cases. Running all these test cases using ADT together with Robotium needs 55226.31 seconds (test case generation time + test case execution time). During the test case execution, 925 test cases failed. We checked the crash reports, and found that the Android version reported fewer failure types than the iOS version. All the 925 failures are caused only by the faults 1 and 2 listed in Table II. The failure scenarios 3, 4 and 5 did not appear in the Android implementation. We also did the random testing for the Android version of *PicFlick* using the same random test case set. The test case execution time and coverage results are similar to to the results shown in Table III. The random testing did not find any new faults.

B. Experiment 2: Newsyc

Newsyc is an open-source *Hacker News* client for iOS devices. Besides reading hacker news, users can post or share news to other people. The GUI implementation *Newsyc* has four GUI views (i.e., news list view, news browsing view, comment view, and system setting view). Therefore, the corresponding design activity diagram has 4 activities and 10 actions in total. Table IV shows the testing results for *Newsyc* using our *ADAutomation* framework.

TABLE IV
TESTING RESULT FOR *Newsyc* USING OUR APPROACH

Bound Size	Test Case #	Test Time (s)	Action Coverage	Transition Coverage	Simple Path Coverage
2	1	8.48	25.00%	29.41%	6.67%
3	3	26.11	33.33%	35.29%	20.00%
4	7	62.45	33.33%	47.06%	20.00%
5	11	99.66	33.33%	47.06%	20.00%
⋮	⋮	⋮	⋮	⋮	⋮
15	4995	47463.04	33.33%	47.06%	20.00%

From Table IV, we can find that when the bound size of test cases exceeds 4, all the investigated coverage results won't change any more. We checked the log information collected by *ADAutomation*, and found a fault in the GUI implementation. When users enter the news browsing view for the first time, they cannot bookmark the news, share the news, or modify the font size. This is because the buttons in this view were all mistakenly disabled during the first entrance of the view. However, all these buttons will be enabled when the users enter the view again later. This certainly is an unacceptable bug, since this scenario is not user-friendly. Consequently, in this case, most of the automatically derived test cases will fail due to the incomplete execution of the derived test cases (i.e., action sequences). In our approach, if one action of a test case fails, the execution of the test case will be aborted.

We also did the random testing on the newsyc design using the tool *UI AutoMonkey*. Table V shows the results. For different bound size, we applied the random tests on newsyc for 12 hours respectively. Surprisingly, although we can achieve 100% coverage in all categories when the bound size is larger than 50, the aforementioned bug cannot be detected. This is because in the random testing, we applied the method proposed in Algorithm 2 for the simple path coverage calculation. For the ease of comparison, we only consider valid GUI actions in the execution of a random test case. The invalid GUI events (e.g., click of disabled buttons) are not considered during the comparison. Therefore, although all simple paths are covered, the matching method in Algorithm 2 cannot accurately reflect the actual action executions. It is important to note that for random testing, the low coverage ratio can indicate the test adequacy and sometimes can be used to infer bugs. However, the high coverage ratio cannot fully guarantee test adequacy as well as lack of bugs.

TABLE V
RANDOM TESTING RESULTS FOR *Newsyc*

Bound Size	**Action Coverage**	**Transition Coverage**	**Simple Path Coverage**	Bugs found
10	100.00%	97.06%	53.33%	0
20	100.00%	100.00%	80.00%	0
50	100.00%	100.00%	100.00%	0
100	100.00%	100.00%	100.00%	0
200	100.00%	100.00%	100.00%	0

V. Conclusions

The GUI-intensive nature makes the reliability and maneuverability of GUI be two most important issues during the design of smartphone applications. Due to the increasing complexity, to achieve these two goals, a large quantity of testing efforts are required. To alleviate time-to-market pressure coupled with the stringent validation requirement, this paper proposes the framework *ADAutomation* that can enable automated GUI testing for smartphone applications from user behavior models established in the design phase. Based on UML activity diagrams, *ADAutomation* supports user behavior modeling, automated GUI test case generation, test case simulation and error diagnosis. A tool chain based on this framework has been developed. To demonstrate the efficacy of our approach, we did experiments on two industrial designs. The results show that our approach can not only reduce the overall test time, but also can effectively detect fatal faults in complex GUI implementations.

References

[1] Wikipedia, App Store (iOS). *http://en.wikipedia.org/wiki/IOS_App_Store*.

[2] UI AutoMonkey. *https://github.com/jonathanpenn/ui-auto-monkey*.

[3] Android Robotium. *http://code.google.com/p/robotium/*.

[4] Xyologic App Download Statistics. *http://xyo.net/app-downloads-reports/*.

[5] J. E. Rumbaugh, I. Jacoboson and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 2001.

[6] OMG, UML Superstructure V2.4. *http://www.omg.org/spec/UML/2.4/ Superstructure/Beta2/PDF/*.

[7] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li and G. Zheng. *Generating test cases from UML activity diagram based on gray-box method*. In Prof. of Asia-Pacific Software Engineering Conference (APSEC), pages 284–291, 2004.

[8] M. Chen, X. Qiu and X. Li. *Automatic test case generation for UML activity diagrams*. In Proc. of International Workshop on Automation of Software Test (AST), pages 2–8, 2006.

[9] M. Chen, P. Mishra and D. Kalita. *Coverage-driven automatic test generation for UML activity diagrams*. In Proc. of ACM Great Lakes Symposium on VLSI (GLSVLSI), pages 139–142, 2006.

[10] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao and X. Li. *UML Activity Diagram-Based Automatic Test Case Generation For Java Programs*. The Computer Journal, 52(5):545–556, 2009.

[11] M. Chen, X. Qin, H. Koo and P. Mishra. *System-Level Validation: High-Level Modeling and Directed Test Generation Techniques*. Springer, 2012.

[12] A. M. Memon, M. L. Soffa and M. E. Pollack. *Coverage criteria for GUI testing*. In Prof. of International Symposium on Foundations of Software Engineering (FSE), pages 256–267, 2001.

[13] J. L. Peterson. *Petri Nets Theory and the Modeling of Systems*. Prentice-Hall, NJ, 1981.

[14] T. Takala, M. Katara, and J. Harty. *Experiences of system-level model-based GUI testing of an Android application*. In Proc. of International Conference on Software Testing, Verification and Validation, pages 377–386, 2011.

[15] A. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton and B. M. Horowitz. *Model-based testing in practice*. In Proc. of International Conference on Software Engineering (ICSE), pages 285–294, 1999.

[16] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan and J. Kazmeie. *Automation of GUI testing using a model-driven approach*. In Proc. of International Workshop on Automation of Software Test (AST), pages 9–14, 2006.

[17] R. Shehady and D. Siewiorek. *A method to automate user interface testing using variable finite state machines*. In Proc. of International Symposium on Fault Tolerant Computing (FTCS), pages 80–88, 1997.

[18] H. Reza, S. Endapally and E. Grant. *A model-based approach for testing GUI using hierarchical predicate transition nets*. In Proc. of International Conference on Information Technology: New Generations, pages 366–370, 2007.

[19] L. White and H. Almezen. *Generating test cases for GUI responsibilities using complete interaction sequences*. In Proc. of International Conference on Software Maintenance (ICSM), pages 473–482, 2005.

[20] S. Arlt, C. Bertolini, S. Pahl, M. Schäf. *Trends in Model-based GUI Testing*. Advances in Computers, 86:183–222, 2012.

[21] F. Belli. *Finite state testing and analysis of graphical user interfaces*. In Proc. of International Symposium on Software Reliability Engineering (ISSRE), pages 34–43, 2001.

[22] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee and A. Memon. *Lightweight static analysis for GUI testing*. In Proc. of International Symposium on Software Reliability Engineering (ISSRE), pages 301–310, 2012.

[23] Q. Xie. *Developing cost-effective model-based techniques for GUI testing*. In Proc. of International Conference on Software Engineering (ICSE), pages 997–1000, 2006.

[24] F. Gross, G. Fraser and A. Zeller. *EXSYST: search-based GUI testing*. In Proc. of International Conference on Software Engineering (ICSE), pages 1423–1426, 2012.

[25] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine and A. M. Memon. *Using GUI ripping for automated testing of Android applications*. In Proc. of International Conference on Automated Software Engineering (ASE), pages 258–261, 2012.

[26] D. R. Hackner and A. M. Memon. *Test case generator for GUITAR*. In Proc. of International Conference on Software Engineering (ICSE), pages 959–960, 2008.

[27] X. Yuan and A. M. Memon. *Using GUI run-time state as feedback to generate test cases*. In Proc. of International Conference on Software Engineering (ICSE), pages 396–405, 2007.

[28] T. H. Chang, T. Yeh and R. C. Miller. *GUI testing using computer vision*. In Proc. of ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), pages 1535–1544, 2010.

[29] C. Hu, and I. Neamtiu. *Automating GUI testing for Android applications*. In Proc. of International Workshop on Automation of Software Test (AST), pages 77–83, 2011.

[30] P. A. Brooks, and A. M. Memon. *Introducing a test suite similarity metric for event sequence-based test cases*. In Proc. of International Conference on Software Maintenance (ICSM), pages 243–252, 2009.

[31] A. Rauf, S. Anwar, M. A. Jaffer, and A. A. Shahid. *Automated GUI test coverage analysis using GA*. In Proc. of International Conference on Information Technology : New Generations, pages 1057–1062, 2010.

[32] A. M. Memon. *An Event-Flow Model of GUI-Based Applications for Testing*. Software Testing, Verification and Reliability, 17(3):137–157, 2007.

[33] A. M. Memon. *Hierarchical GUI Test Case Generation using Automated Planning*. IEEE Transactions on Software Engineering, 27(2):144–155, 2001.

[34] KODAK PicFlick. *http://www.kodak.com/go/mobileprinting*.

[35] Newsyc. *https://github.com/Xuzz/newsyc*.

[36] Enterprise Architect. *http://www.sparxsystems.com*.

[37] Xcode 4, Apple Developer. *https://developer.apple.com/xcode/*.

[38] C. Hu and I. Neamtiu. *A GUI bug finding framework for Android applications*. In Proc. of the ACM Symposium on Applied Computing (SAC), pages 1490–1491, 2011.