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, China

June 30, 2014
Outline

- Introduction
- Behavior Modeling Using Activity Diagrams
  - Graph–based Notations
  - User Behavior Modeling & Extraction
- Our Automated GUI Testing Framework
  - Test Script Library Generation
  - GUI Testing & Error Diagnosis
- Experiments
- Conclusion
Outline

- Introduction
- Behavior Modeling Using Activity Diagrams
  - Graph–based Notations
  - User Behavior Modeling & Extraction
- Our Automated GUI Testing Framework
  - Test Script Library Generation
  - GUI Testing & Error Diagnosis
- Experiments
- Conclusion
More and more people rely on smartphone apps to manage their daily life (bills, shopping, surfing, emails, etc.).

GUI testing is becoming a major bottleneck in overall testing (up to 40% of time and resources).

Time-to-market pressure requires automated GUI testing techniques.

Total number of both iOS and Android APP Downloads: >100 billion
Summary of Related Work

- **Capture & Replay / Script Driven Testing**
  - Flexible for debug purpose (tools: Robotium, UIAutomation, etc.)
  - Require expert-knowledge to detect errors, need human intervention

- **Random Testing**
  - Can be fully automated (tools: Monkey, UI-Auto-Monkey, etc.)
  - Coverage convergence cannot be guaranteed. The coverage may take quite long time to go from 90% to 99%.

- **Model-Based Approaches (e.g., FSM, Event-Flow)**
  - Can be automated with less test cases.
  - Focus on the internal logic rather than user behaviors

- **Few of existing testing approaches make use of user behavior information.**
  - How to model user behaviors accurately?
  - How to check the inconsistency between GUI specifications and GUI implementations?
Overview of Our ADAutomation Framework

User Behavior Modeling

Formal Models → Behavior Extraction

Coverage Criteria

GUI Test Functions → Action-Function Mapping

Action Sequences

Test Script Generation

GUI Test Case Generation

Test Script Library Construction

Test Case Execution

Log Analyzer

Coverage Report → Replay-based Debugger

GUI Application

Test Engine/Engineer

Post-test Analysis
Outline

- Introduction
- Behavior Modeling Using Activity Diagrams
  - Graph–based Notations
  - User Behavior Modeling & Extraction
- Our Automated GUI Testing Framework
  - Test Script Library Generation
  - GUI Testing & Error Diagnosis
- Experiments
- Conclusion
User Behavior Modeling

- XML Parser
- GUI Specification (Activity Diagrams with system & action configurations)
- User Behavior Modeling
  - Formal Models
  - Behavior Extraction
  - Coverage Criteria

Test Script Library Construction
- GUI Test Functions
- Action-Function Mapping
- Action Sequences
- Test Script Generation
- Test Scripts
- GUI Test Case Generation

Post-test Analysis
- Log Analyzer
  - Coverage Report
  - Replay-based Debugger

Test Case Execution
- Simulator
  - Application
  - Test Engine
Graph-Based Notations

- **UML Activity Diagram Notations**
  - *Actions* (i.e., external operations) and *activities* (i.e., GUI views, which indicate a collection of correlated actions)
  - Control nodes and flows (indicating the execution order of actions)
    - Control nodes
      - Initial
      - Final
      - Decision/Merge
    - Control flow
      - Arrow
      - Fork
      - Join
User Behavior Modeling

- As a semi-formal specification, UML activity diagrams cannot be automatically analyzed.

- We extend the relation between actions with a quasi-Petri-net semantics for GUI testing purpose.
  - Concurrent action execution $\rightarrow$ Interleaving of actions
  - Activities $\rightarrow$ GUI views
    - Actions in the current view cannot be preempted by actions from other views.
  - Activity depth $\rightarrow$ Action execution priority
    - Nested activities are indexed by their depth
    - At any time, only actions in the deepest activity can be fired.

- The behavior of an activity diagram can be represented by a sequence of actions (e.g., $<$Initial, a, b, c, …, Final$>$).
The action order of each thread should be kept. Independent actions can be interleaved.

The execution of action1, action2, and action3 cannot be interrupted by the action4 and action5.
To enable sufficient testing, we need to enumerate all possible user behaviors from activity diagrams, which can be explored in a depth-first way.

However, due to the loops, it is impossible to enumerate all the user behaviors.

- **Restrict the length of path with a limited bound**

To mimic the user behaviors, the depth-first user behavior exploration needs to consider the hierarchy information of activities.

- Consider the depth information of enclosed activity for each action
- Only the actions within the deepest activity can be explored further
Forward Path Exploration

Select Files

Press Download Button

Input User Name

Input Password

Login Succeed?

Check Downloaded Files

Final

Has Logged in?

Check Downloaded Files

Forward Path Exploration

 CS=<a,c> path=<a>

 CS=<a,d> path=<a,d>

 CS=<a,e> path=<a,d,e>

 CS=<b,c> path=<a,d,e,f,b>

 CS=<g> path=<a,d,e,f,b,g>
Backward Path Exploration

Select Files

Press Download Button

Input User Name

ActivityInitial

Login Succeed?

Submit

Check Downloaded Files

Initial

ActivityFinal

CS=<Final>
path=<a,d,e,f,b,g>

CS=<g>
path=<a,d,e,f,b,g>

CS=<b,c>
path=<a,d,e,f,b>

CS=<a,c>
path=<a,d,e,f,b,a>

CS=<b,c>
path=<a,d,e,f,b,a,b>

Final
To measure the effectiveness of user behaviors extracted from activity diagrams, we propose three types of coverage metrics.

- **Action coverage** check the reachability of each action.
- **Transition coverage** checks all the conditional branch along the control flows.
- **Simple path** checks possible combination of actions and transitions.
  - A simple path is a path (user behavior) without action repetition.
  - When a user behavior remove all its repeated actions, it should match some simple path.
An Example of Simple Path Coverage

- A user behavior matches a simple path if it can simulate on the activity diagram and it contains all the actions of the simple paths.
  - E.g., the user behavior described in blue lines matches the simple path in green lines.
- Simple path coverage requires that all the simple paths to be covered.
Outline

● Introduction
● Behavior Modeling Using Activity Diagrams
  ◆ Graph–based Notations
  ◆ User Behavior Modeling & Extraction
● Our Automated GUI Testing Framework
  ◆ Test Script Library Generation
  ◆ GUI Testing and Error Diagnosis
● Experiments
● Conclusion
Test Script Library Construction

- XML Parser
- User Behavior Modeling
  - Coverage Criteria
  - Formal Models
  - Behavior Extraction
  - GUI Test Functions
  - Action-Function Mapping
- Test Script Library Construction
- GUI Test Case Generation
  - Action Sequences
  - Test Script Generation
  - Test Scripts
- Test Case Execution
  - Simulator
    - Application
    - Test Engine
- Post-test Analysis
  - Log Analyzer
    - Coverage Report
  - Replay-based Debugger
- GUI Specification
  (Activity Diagrams with system & action configurations)
<Platform>
  <Name>iOS</Name>
  <Version>6.0</Version>
</Platform>

<TestEngine>
  <Name>…</Name>
  <Version>…</Version>
</TestEngine>

<Delay>
  <Unit>Second</Unit>
  <Value>0.2</Value>
</Delay>
Activity Diagram Annotation

- **Action configuration**
  - **Widget features** specify the attributes of corresponding GUI widgets such as ID, name, position, size, etc.
  - **GUI operations** describe user operations conducted on associated GUI widgets.
  - **Test logs** instrument proper log information based on the result of GUI operations.
For each action, our tool will generate one corresponding function. Therefore, there will be a mapping between the action and its associate test function.

For the name of derived test function, we use the following convention:

- **Action node information**
  - Name="Select Files"
  - ID=1

- **Function name**
  - _1_SelectFiles
A Test script library is a set of system settings and generated test functions

- System configuration → System settings
- Action configuration → Test Functions

A skeleton of a test script library *library.js* in JavaScript

```javascript
var delay = 0.2;  // System setting

function screenshot() { ……} // System Functions
function _1_SelectFiles() {……}
function _2_PressDownloadButton() {……}
function _3_InputUserName() {
  UIALogger.logMessage("Action:InputUserName");
  win.textFields()[0].setValue("…");
  screenshot();
}
function _4_InputPasswd() {……}
function _5_Submit() {……}
function _6_CheckDownloadFiles() {……}
……
```

// Test functions
The process of test case generation is a one-one mapping from a user behavior to a test case.

- An extracted user behavior example:
  

- The corresponding generated test case:

  ```
  #import “Library.js”
  UIALogger.logStart(“Testing starts”);
  _1_SelectFiles( );
  _3_InputUserName();
  _4_InputPasswd();
  _5_Submit( );
  _2_PressDownloadButton ( );
  _6_CheckDownloadFiles( );
  UIALogger.logPass(“Testing ends”);
  ```
Overview of ADAutomation Framework

- **XML Parser**
- **User Behavior Modeling**
  - Formal Models
  - Behavior Extraction
- **Test Script Library Construction**
  - GUI Test Functions
  - Action-Function Mapping
  - Action Sequences
  - Test Script Generation
  - Test Scripts
- **GUI Test Case Generation**
- **Post-test Analysis**
  - Log Analyzer
  - Coverage Report
  - Replay-based Debugger
- **Test Case Execution**
  - Simulator
  - Application
  - Test Engine
Error Diagnosis

- Error Diagnosis
  - Log information

- Playback
Outline

- Introduction
- Behavior Modeling Using Activity Diagrams
  - Graph–based Notations
  - User Behavior Modeling & Extraction
- Our Automated GUI Testing Framework
  - Test Script Library Generation
  - GUI Testing & Error Diagnosis
- Experiments
- Conclusion
All the experimental results were obtained on a MacBook Pro machine with Intel Core i5 2.4GHz processor and 4 GB RAM.
Case Study 1: *PicFlic*

- *PicFlic* is a free Wi-Fi based remote picture print management application developed by Eastman Kodak company.

- The left picture shows the eight views of *PicFlick* on iOS and the corresponding view switches indicated by arrow lines.

- We did GUI testing on both iOS and Android versions of *PicFlick*.
The Activity Diagram of *PicFlick*

- Designing this activity diagram from user specification needs around 6 hours.
- 30 actions, 86 transitions, and 955 simple paths are derived from this activity diagram.
- It can generate 12776 test cases with a bound limit 18.
# Test Results of *PicFlick*

- Testing Time (test case generation time + simulation time):
  - 70217 seconds (less than **20 hours**) for PicFlick (iOS).
  - Compared to **2-3 man month** manual testing in industrial.

<table>
<thead>
<tr>
<th>Bound Size</th>
<th>Test Case #</th>
<th>Failed Case #</th>
<th>Test Time (s)</th>
<th>Action Coverage</th>
<th>Transition Coverage</th>
<th>Sim. Path Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>16.06</td>
<td>13.33%</td>
<td>13.95%</td>
<td>0.21%</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
<td>0</td>
<td>103.37</td>
<td>43.33%</td>
<td>46.51%</td>
<td>1.05%</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
<td>0</td>
<td>438.57</td>
<td>66.67%</td>
<td>67.44%</td>
<td>2.62%</td>
</tr>
<tr>
<td>8</td>
<td>176</td>
<td>5</td>
<td>1207.15</td>
<td>83.33%</td>
<td>82.56%</td>
<td>5.03%</td>
</tr>
<tr>
<td>10</td>
<td>432</td>
<td>14</td>
<td>2773.44</td>
<td><strong>100%</strong></td>
<td><strong>100%</strong></td>
<td>8.07%</td>
</tr>
<tr>
<td>12</td>
<td>881</td>
<td>22</td>
<td>5231.80</td>
<td><strong>100%</strong></td>
<td><strong>100%</strong></td>
<td>19.71%</td>
</tr>
<tr>
<td>14</td>
<td>2224</td>
<td>83</td>
<td>13235.72</td>
<td><strong>100%</strong></td>
<td><strong>100%</strong></td>
<td>45.60%</td>
</tr>
<tr>
<td>16</td>
<td>5784</td>
<td>331</td>
<td>33481.70</td>
<td><strong>100%</strong></td>
<td><strong>100%</strong></td>
<td>81.66%</td>
</tr>
<tr>
<td>18</td>
<td>12776</td>
<td>993</td>
<td>70217.14</td>
<td><strong>100%</strong></td>
<td><strong>100%</strong></td>
<td><strong>98.95%</strong></td>
</tr>
</tbody>
</table>

- Android version shows the similar testing results.
### Bugs Found in *PicFlick*

- **993** of 12776 test cases resulted in application crashes on iOS, and **5** suspected bugs were found.

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

- **925** of 12776 test cases resulted in application crashes on Android, but only bug 1 and 2 were reported.
Random Testing for *PicFlick*

- We conduct the random testing using UI AutoMonkey on iOS

<table>
<thead>
<tr>
<th>Bound Size</th>
<th>Action Coverage</th>
<th>Transition Coverage</th>
<th>Simple Path Coverage</th>
<th>Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>47.22%</td>
<td>59.34%</td>
<td>0.93%</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>97.22%</td>
<td>97.80%</td>
<td>65.60%</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>97.22%</td>
<td>97.80%</td>
<td>85.80%</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>97.22%</td>
<td>97.80%</td>
<td>85.80%</td>
<td>2</td>
</tr>
<tr>
<td>400</td>
<td>100%</td>
<td>100%</td>
<td>93.89%</td>
<td>3</td>
</tr>
<tr>
<td>unlimited*</td>
<td>100%</td>
<td>100%</td>
<td>93.89%</td>
<td>3</td>
</tr>
</tbody>
</table>

- Random testing took **24 hours**, but only 3 out of 5 known bugs were found. No new errors were detected using the random approach.

- Simple path coverage is **93.89%** using the random approach compared to **98.95%** using ADAutomation.

- Random testing on Android shows the similar results.
Case Study 2: Newsyc

- Newsyc is an open source *Hacker News* client.
- The GUI implementation has four views (i.e., news list view, news browsing view, comments view, and system setting view).
- Its activity diagram has 12 actions, 35 transitions, and 5 simple paths.
- 4995 test cases were generated form this diagram with a bound limit of 15.
### Testing Result for *Newsync* (iOS)

<table>
<thead>
<tr>
<th>Bound Size</th>
<th>Test Case #</th>
<th>Test Time (s)</th>
<th>Action Coverage</th>
<th>Transition Coverage</th>
<th>Sim. Path Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>8.48</td>
<td>25.00%</td>
<td>29.41%</td>
<td>6.67%</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>26.11</td>
<td>33.33%</td>
<td>35.29%</td>
<td>20.00%</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>62.45</td>
<td>33.33%</td>
<td>47.06%</td>
<td>20.00%</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>99.66</td>
<td>33.33%</td>
<td>47.06%</td>
<td>20.00%</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4995</td>
<td>47145.0</td>
<td>33.33%</td>
<td>47.06%</td>
<td>20.00%</td>
</tr>
</tbody>
</table>

- **One Bug Found** - When users enter the news browsing view for the first time, they cannot bookmark the news, share the news, or modify the font size.

- **Test time** (test case generation time + simulation time) is 47145 seconds (about 13 hours).
Random Testing for *Newsyc*

<table>
<thead>
<tr>
<th>Bound Size</th>
<th>Action Coverage</th>
<th>Transition Coverage</th>
<th>Simple Path Coverage</th>
<th>Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100.00%</td>
<td>97.06%</td>
<td>53.33%</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>100.00%</td>
<td>100.00%</td>
<td>80.00%</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>0</td>
</tr>
<tr>
<td>unlimited*</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00 %</td>
<td>0</td>
</tr>
</tbody>
</table>

- Random testing takes 12 hours.
- Achieved 100% coverage in all categories.
- No bug was found.
GUI testing is becoming a major bottleneck in smartphone application development.

This paper proposes an efficient automated GUI testing framework for smartphone applications

- User behavior modeling using activity diagram
- Three test adequacy criteria
- Automated GUI test library construction
- Tool chain for automated GUI testing

Successfully applied on various smartphone applications

- Significant reduction in overall GUI testing time
- More bugs found than random methods
Thank you !