EVALUATING TARGET EVENT SEQUENCE GENERATION FOR ANDROID APPLICATIONS

Authors: Casper S. Jensen  
Aarhus University, Denmark  
semadk@cs.au.dk

Mukul R. Prasad  
Fujitsu Laboratories of America, USA  
mukul@us.fujitsu.com

Anders Møller  
Aarhus University, Denmark  
amoeller@cs.au.dk

—as a part of completion for Group Paper Presentation for CMPSCI 621—
INTRODUCTION
MOTIVATING EXAMPLE: TAXCALCULATOR
UI MODEL OF A PART OF TAXCALCULATOR
1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4   deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8   taxable = 0; // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
WHAT IS THE PROBLEM?

• Automated generation of event sequence for event-driven applications for higher code coverage

Previous Works:

• Random Black-box testing [1]
• Techniques involving Symbolic Execution [2]

[1] C Hu et al Automating UI Testing for Android Applications in International workshop on Automation of Software Test

Is it possible to generate event sequences for complex targets that require long event sequence and highly constrained event parameters?
KEY IDEA & APPROACH OVERVIEW
Automated test generators for event-driven applications do not cover tests that:
- Require long event sequence
- Require specific event parameters
- Have complex targets

Automated testing with targeted event sequence generation for event-driven applications
Solution

Collider

(A Targeted Event Sequence Generation Tool)

Inspirations:

- Line reachability problem for C programs [3]
- A Model-based testing technique [4]

HIGH-LEVEL VIEW OF COLLIDER

Android Application
UI Model
Set of Targets
Collider
Test Cases for Targets
UI MODELS FOR EVENT-DRIVEN APPLICATIONS

UI Model of an event-driven application is a collection of event-handler methods that attach to GUI widgets.

Event-handler registration

= (GUI Widget Object, Event Kind, Event-handler Method)
UI MODELS FOR EVENT-DRIVEN APPLICATIONS (CONT.)

- \( M = (S, s_0, E, T) \)

- finite set of abstract states
- finite set of event-handler registrations
- transition relation
- initial state

![Diagram of UI model](image)
APPROACH OVERVIEW

1. Symbolic Summarization Phase (Symbolic analysis)
   1. **Input**: executable android application, event-handlers
   2. **Output**: event-handler summary

2. Sequence Generation Phase (Backward exploration)
   1. **Input**: event-handler summary, UI model
   2. **Output**: test case
THE PROCESS
1. SYMBOLIC SUMMARIZATION PHASE

• Goal: to produce event handler summary for each event handler
  • Event handler summary = a set of path summaries for all the execution paths within the event handler code

• The summary is computed by performing concolic testing[5]

1. SYMBOLIC SUMMARIZATION PHASE

- Goal: to produce event handler summary for each event handler
  - Event handler summary = a set of path summaries for all the execution paths within the event handler code

- The summary is computed by performing concolic testing

```c
1  void f(int x, int y) {
2      int z = 2*y;
3      if (x == 100) {
4          if (x < z) {
5              assert(0); /* error */
6          }
7      }
8  }
```

< Example code written in C >
Goal: to produce event handler summary for each event handler

- Event handler summary = a set of path summaries for all the execution paths within the event handler code

- The summary is computed by performing concolic testing

```c
1 void f(int x, int y) {
2     int z = 2*y;
3     if (x == 100) {
4         if (x < z) {
5             assert(0); /* error */
6         }
7     }
8 }
```

< Example code written in C >
I. SYMBOLIC SUMMARIZATION PHASE

• Goal: to produce **event handler summary** for each event handler
  - Event handler summary = a set of path summaries for all the execution paths within the event handler code

• The summary is computed by performing **concolic testing**

```c
1 void f(int x, int y) {
2   int z = 2*y;
3   if (x == 100) {
4     if (x < z) {
5       assert(0); /* error */
6     }
7   }
8 }
```

< Example code written in C >
I. SYMBOLIC SUMMARIZATION PHASE

• Goal: to produce event handler summary for each event handler
  • Event handler summary = a set of path summaries for all the execution paths within the event handler code

• The summary is computed by performing concolic testing

```c
1 void f(int x, int y) {
2     int z = 2*y;
3     if (x == 100) {
4         if (x < z) {
5             assert(0); /* error */
6         }
7     }
8 }
```

< Example code written in C >
2. SEQUENCE GENERATION PHASE

- Goal: to generate a test case for each given target
2. SEQUENCE GENERATION PHASE

- Goal: to generate a test case for each given target

- Partial sequence = (a path summary + an abstract state)
2. SEQUENCE GENERATION PHASE

- Goal: to generate a test case for each given target

- Partial sequence = (a path summary + an abstract state)

```
1: Initialize the worklist
2: while (worklist is not empty)
3:     identify anchors
4:     for each anchor,
5:         identify connector sequences
6:     construct a new partial sequence
7:     if (new partial sequence == test case)
8:         return new partial sequence
9:     else
10:         add new partial sequence to Worklist
11:     Reprioritize Worklist
12: return false
```
INITIALIZE THE WORKLIST

- **path summary** that triggers the **target**
- **abstract state** which has an outgoing transition to the **path summary**

Partial sequence (a path summary to target, INCOME)
IDENTIFY ANCHORS

- path summaries that affect the path condition of the partial sequence are identified as anchors.
IDENTIFY ANCHORS

- path summaries that affect the path condition of the partial sequence are identified as anchors.
IDENTIFY ANCHORS

- path summaries that affect the path condition of the partial sequence are identified as anchors.

---

1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4   deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8   taxable = 0; // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
IDENTIFY CONNECTOR SEQUENCES

• For each given anchor, a set of connector sequences that lead from the anchor to the partial sequence is extracted by using the UI model

Anchor = path summaries for $e_7$
Connector = path summaries for $e_6$

```
1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4    deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8    taxable = 0;  // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
```
IDENTIFY CONNECTOR SEQUENCES

For each given anchor, a set of connector sequences that lead from the anchor to the partial sequence is extracted by using the UI model.

**Anchor** = path summaries for $e_7$

**Connector** = path summaries for $e_6$

**New Partial Sequence** =
(Anchor, Connector, original partial sequence)

```java
1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4     deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8     taxable = 0; // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
```
IDENTIFY CONNECTOR SEQUENCES

- For each given anchor, a set of connector sequences that lead from the anchor to the partial sequence is extracted by using the UI model.

```plaintext
Anchor = path summaries for e7
Connector = path summaries for e6

New Partial Sequence =
(Anchor, Connector, original partial sequence)
```

```plaintext
1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4    deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8    taxable = 0; // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;
```
RE-PRIORITIZE PARTIAL SEQUENCES

• Some of partial sequences are less important than the others

• E.g. multiple anchors that affect the same path condition, which result in multiple new partial sequences in the wordlist

• We can decrease the priority of any redundant partial sequences
EVALUATION & RESULTS
EXPERIMENTAL SETUP:

<table>
<thead>
<tr>
<th>Collider</th>
<th>symbolic execution</th>
<th>concolic execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic Java PathFinder</td>
<td>Android emulator by Android SDK</td>
<td></td>
</tr>
</tbody>
</table>

Baseline tools

- Simple Crawler
- Monkey provided by Android SDK
Q1: Was the algorithm able to reach **challenging** targets?

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Targets depending on</th>
<th>Reached</th>
<th>Potential</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequence</td>
<td>Parameters</td>
<td></td>
</tr>
<tr>
<td>TippyTipper</td>
<td>15</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>17</td>
<td>25</td>
<td>16</td>
</tr>
<tr>
<td>Munchlife</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>OpenManager</td>
<td>11</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>DieDroid</td>
<td>2</td>
<td>11</td>
<td>8</td>
</tr>
</tbody>
</table>
Q2: Does use of anchors & connectors have an effect on ability to reach targets, when compared to simple backward Breadth-First-Search technique?
Q3: Does prioritization and ignoring of less important event sequences have any effect at all?

<table>
<thead>
<tr>
<th></th>
<th>WITH PRIORITYZATION</th>
<th>WITHOUT PRIORITYZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARGETS REACHED</td>
<td>46</td>
<td>25</td>
</tr>
<tr>
<td>RUNNING TIME FOR SEQUENCE GENERATION</td>
<td>45 seconds</td>
<td>2.5 hours</td>
</tr>
</tbody>
</table>
DISCUSSION QUESTIONS
We see some of the challenging targets were reached. What happens to the missed targets? Why were they missed? What are the necessary steps?

**Possible Answer:** The algorithm supports symbolic reasoning of numeric values and booleans, resulting in imprecise treatment of, for example, strings and object. Improvisation in this field might help us reach the missed targets.
Authors use handwritten UI models for each application for symbolic generation. **Automated UI generation** models weren’t present at the time. Would human effort mean presence of errors? This also reduces the feasibility of scaling the evaluation to a larger number of benchmarks.

**Possible Answer:** One possible alternative would be using automated UI-model generation technique[6]. Would that be effective enough?

QUESTION 3

Can we use Collider for other types of event-driven applications?

Possible Answer: the authors believe that their approach might also be applicable for other types of event-driven programs such as JavaScript web applications and desktop GUI applications in a case that event-handlers are smaller in web or GUI applications
Can *Collider* be used to generate test cases for all types of targets?

**Possible Answer:** In a case that target is *simple*, collider’s execution time is greater than existing approaches.
QUESTIONS 5

Does this technique work well with all Android Applications?

Possible Answer: There was one application ("TippyTipper") which was evaluated by the authors in the paper which did not respond very well. It took 30 minutes for the event sequence generation phase to complete for this application while for other four took few seconds. This was possibly because of lot of connectors between events. Such applications might take a lot of time.
Can we apply this technique to other mobile application platforms such as IOS and windows?

Possible Answer: the authors have mentioned that their approach works well with small number of long event sequences which is the case in mobile applications. Hence, any mobile application that satisfies this condition, ex:- iOS/windows should work well with the algorithm is our assumption.