Using Model Checking to Generate Test Cases for Android Applications

Ana Rosario Espada    María del Mar Gallardo
Alberto Salmerón     Pedro Merino

Universidad de Málaga, Spain

10th Workshop on Model-Based Testing
Table of Contents

1. Introduction
2. Test case generation with model checking
3. Architecture
4. Formal description of models
5. Case study
6. Conclusions and future work
Introduction

- Smartphones have become ubiquitous computing devices
- Continuously and rapidly evolving technology
- Event-driven user interface, focusing on one task at a time
- With a traditional multi-tasking operating system underneath
Introduction

- Typical errors of concurrent software may happen
- Other bugs are inherent to mobile platforms, such as
  - Incorrect implementation of lifecycle in apps or services
  - Handling of unexpected events
  - API or device compatibility problems
- Different analysis techniques have been proposed
  - Model checking ANDROID applications with JPF
  - Testing, monitoring and runtime verification
  - Automatic generation of random input events
Our proposal

- Model the possible user behaviors using state machines
  - Nested state machines representing apps, screens, etc.
  - Nondeterministic behavior within each state machine
  - Composition of state machines

- Generate test cases by exploring this model
- Monitor and analyze the execution of the test cases
- Implemented for ANDROID
We use the SPIN model checker to generate test cases.

SPIN is focused on the design and validation of computer protocols, although it has been applied to many other areas.

Given a system specification written in PROMELA, SPIN can check the occurrence of a property over all possible executions and provide counterexamples.
Test case generation with model checking

- **Model**: → PROMELA specification
- **Device**: → PROMELA process
  - Multiple devices run concurrently
  - Device state machine implemented as a loop
- Each loop branch corresponds to a transition
  - Guard declares transition trigger (e.g. button press, swipe)
  - Right hand side records transition and updates current state
- **SPIN** will explore exhaustively all possibilities (e.g. when several guards are true at the same time) to generate all possible test cases
active proctype device_A() {

}

active proctype device_B() {

}
Test case generation with model checking

```c
mtype = { state_init, state_1, state_2, ... };
typedef Device { byte transitions[MAX_TR]; short index; bool finish; } Device devices[DEVICES];
mtype state[DEVICES];

active proctype device_A() {
    state[DEVA] = state_init;
    devices[DEVA].finish = true;
}

active proctype device_B() {
    state[DEVB] = state_init;
    ... devices[DEVB].finish = true;
}
```
Test case generation with model checking

```
mtype = { state_init, state_1, state_2, ... };  
typedef Device { byte transitions[MAX_TR]; short index; bool finish; }  
Device devices[DEVICES];  
mtype state[DEVICES];  

active proctype device_A() {  
    state[DEVA] = state_init;  
do  
    ::= state[DEVA] == state_init -> transition(DEVA, BUTTON_1); state[DEVA] = state_1  
    ::= state[DEVA] == state_1 -> transition(DEVA, SWIPE); state[DEVA] = state_1  
    ::= state[DEVA] == state_1 -> transition(DEVA, BUTTON_2); state[DEVA] = state_2  
    ::= state[DEVA] == state_2 -> transition(DEVA, MESSAGE); break  
    ::= state[DEVA] == state_2 -> transition(DEVA, BACK); break  
    od;  
devices[DEVA].finish = true;  
}

active proctype device_B() {  
    state[DEVB] = state_init;  
    ...  
devices[DEVB].finish = true;  
}
```
Architecture

- **Test Generator Engine**
  - User models app user flows, associates events with UI controls (extracted with UIAUTOMATORVIEWER)
  - SPIN explores the model, generates an XML test case for each possible flow
  - Test cases are translated into JAVA classes which use the UIAUTOMATOR tool and run in the devices

- **Runtime Verification Engine**
  - Monitors the execution of the test cases
  - Implemented by the DRAGONFLY tool
Formal description of models

- Mobile applications are modeled through the composition of state machines, at different levels: view and device

- View state machines
  - A view represents a screen in an application
  - Only one view active in a device at the same time
  - User interacts with the currently active view
  - A transition may trigger another view to become active

- Device state machines
  - Composed of one or more view state machines
  - Handle transitions between view through connection states
Formal description of models

- $\rightarrow / \rightarrow_i$: transition relation of the view state machines $M/M_i$
- $\rightarrow_c$: transition relation that connects view state machines
- $\rightarrow_d$: transition relation that connects device state machines
  - Constructed from relations $\rightarrow / \rightarrow_i$ and $\rightarrow_c$
- Transitions are labeled with the event required to fire them
  - E.g. $s \xrightarrow{e} s'$: event $e$ must be fired to transit from $s$ to $s'$
- Test case: sequence of events
View state machines

**View state machine**

\[ M = \langle \Sigma, I, \rightarrow, E, C, F \rangle \]

- \( \Sigma \): finite set of states
- \( I \subseteq \Sigma \): set of initial states
- \( C \subseteq \Sigma \): connection states (to a different state machine)
- \( F \subseteq \Sigma \): set of final states
- \( E \): set of user events
- \( \rightarrow \subseteq \Sigma \times E \times \Sigma \): labeled transition relation
- \( I, C \) and \( F \) are mutually disjoint

- \( E \) can be divided into two disjointed sets:
  - \( E^+ \): user events (e.g. button press, swipe)
  - \( E^- \): system events (e.g. message reception)
View state machines

Flow

Given a view state machine \( M = \langle \Sigma, I, \rightarrow, E, C, F \rangle \), we define the set \( \text{Flow}(M) = \{ s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} s_n | s_0 \in I, s_n \in F \cup C \} \) of all sequences of transitions, allowed by \( M \), starting at an initial state of \( M \), and ending at a final or connection state of \( M \).

- Given a flow \( \phi = s_0 \xrightarrow{e_1} \cdots \xrightarrow{e_n} s_n \in \text{Flow}(M) \), the sequence of events (i.e. the test case) determined by \( \phi \) is \( \text{test}(\phi) = e_1 \cdots e_n \).
- Given a state machine \( M \), the set of test cases allowed by \( M \) is \( \text{TC}(M) = \{ \text{test}(\phi) | \phi \in \text{Flow}(M) \} \).
Composition of view state machines

- Given a set of state machines $M_i = \langle \Sigma_i, I_i, \rightarrow_i, E_i, C_i, F_i \rangle$
  - $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$
  - $I = \bigcup_{i=1}^{n} I_i$
  - $E = \bigcup_{i=1}^{n} E_i$
  - $C = \bigcup_{i=1}^{n} C_i$
  - $F = \bigcup_{i=1}^{n} F_i$

- $E \subseteq E$ the set of call events that provoke the switch between active view state machines

Connection relation

The connection of view state machines $M_1, \ldots, M_n$ is given by a binary relation $\mathcal{R}(M_1, \cdots, M_n) \subseteq C \times E \times I$, that connects connection states with initial states

- We denote 3-tuples $(s_i, e, s_j)$ of $\mathcal{R}(M_1, \cdots, M_n)$ as $s_i \xrightarrow{e} c s_j$
Composition of view state machines

Device state machine

Given a finite set of view state machines, 
\( M_i = \langle \Sigma_i, l_i, \longrightarrow_i, E_i, C_i, F_i \rangle \), and a connection relation of 
\( M_1, \ldots, M_n \), the device state machine

\[ D = M_1 \| \cdots \| M_n \| R(M_1, \cdots, M_n) \]

is defined as the state machine \( \langle \Sigma \times \Sigma^* \times \mathcal{E}^*, l, \longrightarrow_d, E, F \rangle \) where

- \( \Sigma^* \) is the set of finite sequences of states of \( \Sigma \), and \( \mathcal{E}^* \) is the set of finite sequences of call events
- transition relation \( \longrightarrow_d \) is defined by the following rules
Composition of view state machines

- The states of a device state machine are called *configurations*.
- A configuration is a 3-tuple \( \langle s, h, eh \rangle \)
  - \( s \): the current state of the active view state machine
  - \( h = s_1 \cdot s_2 \cdots s_n \): the stack of states that constitutes the history of created view state machines, where \( s_i \in C \)
  - \( eh = e_1 \cdot e_2 \cdots e_n \): the history of events that provoked the creation of new view state machines, where \( e_i \in \mathcal{E} \)
Composition of view state machines

Transition within a view state machine:

\[
R1. \quad \frac{s \xrightarrow{e_i} s'}{\langle s, h, eh \rangle \xrightarrow{e} \langle s', h, eh \rangle}
\]
Composition of view state machines

- Transition to a new state machine, without reusing:

\[ s \in C_i, s \xrightarrow{e} s', \neg \text{reuse}(e) \]
\[ \langle s, h, eh \rangle \xrightarrow{e} \langle s', h \cdot \text{return}(s), eh \cdot e \rangle \]

- Reusing, but no previous view state machine to reuse:

\[ s \in C_i, s' \in I_j, s \xrightarrow{e} s', \text{reuse}(e), \text{top}(s_1 \cdots s_n, j) = \perp \]
\[ \langle s, h, eh \rangle \xrightarrow{e} \langle s', h \cdot \text{return}(s), eh \cdot e \rangle \]

- Reusing:

\[ s \in C_i, s' \in I_j, s \xrightarrow{e} s', \text{reuse}(e), \text{top}(s_1 \cdots s_n, j) = s_k \]
\[ \langle s, s_1 \cdots s_n, e_1 \cdots e_n \rangle \xrightarrow{e} \langle s_k, s_1 \cdots s_{k-1}, e_1 \cdots e_{k-1} \rangle \]
Flow continues with the previous view state machines, after the current one finishes:

\[ R5. \quad s \in F_i, auto\_return(e) \implies \langle s, h \cdot s', eh \cdot e \rangle \xrightarrow{d} \langle s', h, eh \rangle \]

If \( auto\_return(e) \) is false, the current configuration cannot evolve
Composition of view state machines

Given a device state machine $D$:

1. **The trace-based semantics** determined by $D$ ($\mathcal{O}(D)$) is given by the set of finite/infinite sequences of configurations (flows) produced by the transition relation $\rightarrow_d$ from an initial state, that is, $\mathcal{O}(D) = \{ \langle s_0, \varepsilon, \varepsilon \rangle \xrightarrow{e_0}_d \langle s_1, h_1, eh_1 \rangle \cdots | s_0 \in I \}$.

2. Given a flow $\phi = c_0 \xrightarrow{e_1}_d c_1 \xrightarrow{e_2}_d c_2 \cdots \in \mathcal{O}(D)$, the test case determined by $\phi$ is the sequence of events $test(\phi) = e_1 \cdot e_2 \cdots$.

3. The set of test cases determined by a set of flows $T$ is $TC(T) = \{ test(t) | t \in T \}$.

Thus, a flow $\phi \in \mathcal{O}(D)$ consists of a (finite or infinite) sequence of view state machine flows connected through connection states.
Composition of device state machines

- Composition of several devices is carried out by interleaving.
- Communication between devices is modeled with user events in the sender (e.g. \(e^+\)) and system events in the receiver (e.g. \(e^-\)).
- \(dh\): set of system events produced but not yet consumed.
- Sender transition:

\[
\begin{align*}
R6. \quad & c_0 \xrightarrow{e^+} c_1 \\
\langle c_0, c_0', dh \rangle \xrightarrow{e^+} d \parallel d' \langle c_1, c_0', dh + \{ e^+ \} \rangle
\end{align*}
\]

- Receiver transition (cannot proceed until \(e^+ \in dh\))

\[
\begin{align*}
R7. \quad & c_0' \xrightarrow{e^-} c_1', \; e^+ \in dh \\
\langle c_0, c_0', dh \rangle \xrightarrow{e^-} d \parallel d' \langle c_0, c_1', dh - \{ e^+ \} \rangle
\end{align*}
\]
Case study

- A single **Android** device with two applications: Facebook and YouTube
  - A user comments on Facebook posts, and visits links that play on the YouTube application

- Modeling
  - Can be done during application development or afterwards
  - State machines could be modeled with UML, then translated into final the XML model
  - We allow several levels of nesting: device → application → view → state machine
Case study
Case study

<Application name="Facebook" package="com.facebook.android">
  <Views>
    <View name="HomeView" controlsFile="Home.xml"/>
  </Views>
  <StateMachine name="HomeUpdate">
    <States>
      <State name="S0"/>
      <State name="S1"/>
    </States>
    <Transitions>
      <Transition ID="1" event="Swipe" prev="" next="S0" type="Simple"/>
      <Transition ID="2" event="Comment" prev="S0" next="S0" through="CommentView" type="View"/>
      <Transition ID="3" event="Swipe" prev="S0" next="S1" type="Simple"/>
      <Transition ID="4" event="ClickYouTubeLink" prev="S0" next="S0" through="ViewingMovieStateMachine" type="StateMachine"/>
      <Transition ID="5" event="Swipe" prev="S1" next="S1" type="Simple"/>
      <Transition ID="6" event="Comment" prev="S1" next="S0" through="CommentView" type="View"/>
      <Transition ID="7" event="Swipe" prev="S1" next="" type="Simple"/>
    </Transitions>
  </StateMachine>
</Application>
State machine transition events must be associated with UI controls.

UIAUTOMATORVIEWER can extract control information from live ANDROID applications.

Controls include which action they support, e.g. click, long click or scroll.

Some controls can be enriched with parameters, e.g. for test input generation.
Case study

You like this.

Like

Users: Uma

clickYouTubeLink

Comment

HomeView

/Swipe

/Swipe

/Swipe

/Swipe

/ClickYou TubeLink

/HomeUpdate

/HomeView

S0

S1

/Comment

/Comment

<node index="0" text="" testGroup="" ....
<node index="0" ....
<node testGroup="clicLike" IsFixedValue="" PatternOrValue="" index="0"
text="Like" resource-id="id/feed_feedback_like_container" clickable="true"
long-clickable="false" password="false" ... />
Case study

- Test case generation with model checking: same principle as before, with more layers
- XML model translated into PROMELA specification
  - Device $\rightarrow$ process
  - Application/view/inner state machines $\rightarrow$ inlines ("functions")
- Nested state machines $\rightarrow$ nested inline calls
  - Device processes contain the topmost state machines
  - A state machine may call another one by calling their inline
- Limited exploration depth
  - State must be stored in a stack ("backstack") when transitioning to a new state machine
  - Backstack/transition history limit number of state machine transitions/transitions in a single test case
  - History part of global SPIN state: more test cases
- Test cases generated as XML
typedef Backstack { mtype states[MAX_BK]; short index; }
Backstack backstacks[DEVICES];
#define currentBackstack devices[device].backstack
#define currentState currentBackstack.states[currentBackstack.index]

active proctype device_219dcac41() {
    if
        true -> app_219dcac41_Facebook(D_219dcac41);
        true -> app_219dcac41_YouTube(D_219dcac41);
    fi;
    devices[D_219dcac41].finished = true
}

inline statemachine_Facebook_HomeView_HomeUpdate(device) {
    currentState = State_Facebook_HomeView_HomeUpdate_init;
    pushToBackstack(device, State_Facebook_HomeView_HomeUpdate_init);
    do
        currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
            transition(device, VIEW_HomeView, 2);
            view_Facebook_CommentView(device);
            currentState = State_Facebook_HomeView_HomeUpdate_S0
        ::
        currentState == State_Facebook_HomeView_HomeUpdate_S0 ->
            transition(device, VIEW_HomeView, 4);
            statemachine_Youtube_MovieView_ViewingMovieStateMachine(device);
            currentState = State_Facebook_HomeView_HomeUpdate_S0
        ;
    od;
    popFromBackstack(device)
}
Case study

- Each XML test case is transformed into a JAVA class
  - Subclass of UiAutomatorTestCase
  - Compiled, installed and executed on the device

```java
public class TestDevice1 extends UiAutomatorTestCase {
    // Transition 2: previous S0 next S0 on view HomeView
    public void TestFacebookComment2() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector()
            .className("android.widget.TextView").index(1).textContains("Comment"));
        control.click();
    }

    // Transition 4: previous S0 next S0 on view HomeView
    public void TestFacebookClicYouTubeLink27() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector()
            .className("android.view.View").index(3));
        control.click();
    }

    // Transition 1: previous next Y0 on view MovieView
    public void TestYouTubeplaypause28() throws UiObjectNotFoundException {
        UiObject control = new UiObject(new UiSelector()
            .className("android.view.View").index(4));
        control.click();
    }
}
```
Case study

- Test generation results
  - Backstack fixed to 4; change devices and max. transitions
  - Device A has been assigned only the Facebook application (although YouTube is reachable)
  - Both devices are independent

<table>
<thead>
<tr>
<th>Devices</th>
<th>Config.</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>Transitions</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>✓</td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>20</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>26</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>10</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>12</td>
</tr>
</tbody>
</table>
Conclusions

- Model-based testing approach for generating test cases for Android applications
- Models capture user behavior and interaction between applications; realistic behaviors vs. random input events
- Flexible models built by composing state machines
- SPIN generates are possible test cases
- Adaptable to other mobile platforms
Future work

- Connect with our runtime verification monitor DRAGONFLY
- Include additional runtime information in the traces
- Analyze other properties, e.g. energy consumption
Thanks for your attention
Questions?