Detecting Inconsistencies in Multi-Platform Mobile Apps

Mona Erfani Joorabchi
University of British Columbia
Vancouver, BC, Canada
merfani@ece.ubc.ca

Mohamed Ali
University of British Columbia
Vancouver, BC, Canada
mohamedha@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

***Abstract*—Due to the increasing popularity and diversity of mobile devices, developers write the same mobile app for different platforms. Since each platform requires its own unique environment in terms of programming languages and tools, the teams building these multi-platform mobile apps are usually separate. This in turn can result in inconsistencies in the apps developed. In this paper, we propose an automated technique for detecting inconsistencies in the same native app implemented for iOS and Android platforms. Our technique (1) automatically instruments and traces the app on each platform for given execution scenarios, (2) infers abstract models from each platform execution trace, (3) compares the models using a set of code-based and GUI-based criteria to expose any discrepancies, and finally (4) generates a visualization of the models, highlighting any detected inconsistencies. We have implemented our approach in a tool called CHECKCAMP. CHECKCAMP can help mobile developers in testing their apps across multiple platforms. An evaluation of our approach with a set of 14 industrial and open-source multi-platform native mobile app-pairs indicates that CHECKCAMP can correctly extract and abstract the models of mobile apps from multiple platforms, infer likely mappings between the generated models based on different comparison criteria, and detect inconsistencies at multiple levels of granularity.**

***Index Terms*—Cross-platform compatibility, mobile apps, dynamic analysis, Android, iOS**

I. INTRODUCTION

Recent industry surveys [5], [4] indicate that mobile developers are mainly interested in building *native apps*, because they offer the best performance and allow for advanced UI interactions. Native apps run directly on a device's operating system, as opposed to web-based or hybrid apps, which run inside a browser.

Currently, iOS [3] and Android [1] native mobile apps[1] dominate the app market each with over a million apps in their respective app stores. To attract more users, implementing the same mobile app across these platforms has become a common industry practice. Ideally, a given mobile app should provide the same functionality and high-level behaviour on different platforms. However, as found in our recent study [33], a major challenge faced by industrial mobile developers is to keep the app consistent across platforms. This challenge is due to the many differences across the platforms, from the devices' hardware, to operating systems (e.g., iOS/Android), and programming languages used for developing the apps (e.g., Objective-C/Java). We also found that developers currently treat the mobile app for each platform *separately* and *manually* perform screen-by-screen comparisons, often detecting many cross-platform inconsistencies [33]. This manual process is, however, tedious, time-consuming, and error-prone.

In this paper, we propose an automated technique, called CHECKCAMP (Checking Compatibility Across Mobile Platforms), which for the same mobile app implemented for iOS and Android platforms (1) instruments and generates traces of the app on each platform for a set of user scenarios, (2) infers abstract models from the captured traces that contain code-based and GUI-based information for each pair, (3) formally compares the app-pair using different comparison criteria to expose any discrepancies, and (4) produces a visualization of the models, depicting any detected inconsistencies. Our work makes the following main contributions:

- A technique to capture a set of run-time code-based and GUI related metrics used for generating abstract models from iOS and Android app-pairs;
- Algorithms along with an effective combination of mobile specific criteria to compute graph-based mappings of the generated abstract models targeting mobile app-pairs, used to detect cross-platform app inconsistencies;
- A tool implementing our approach, called CHECKCAMP, which visualizes models of app-pairs, highlighting the detected inconsistencies. CHECKCAMP is publicly available [15];
- An empirical evaluation of CHECKCAMP through a set of seven industrial and seven open-source iOS and Android mobile app-pairs.

Our results indicate that CHECKCAMP can correctly extract abstract models of the app-pairs to infer likely mappings between the generated abstract models based on the selected criteria; CHECKCAMP also detects 32 valid inconsistencies in the 14 app-pairs.

II. PERVASIVE INCONSISTENCIES

A major challenge faced by industrial mobile developers is to keep the app consistent across platforms. This challenge and the need for tool support emerged from the results of our qualitative study [33], in which we interviewed 12 senior app developers from nine different companies and conducted a

[1]In this paper, we focus on native apps; henceforth, we use the terms 'mobile app' or simply 'app' to denote 'native mobile app'.

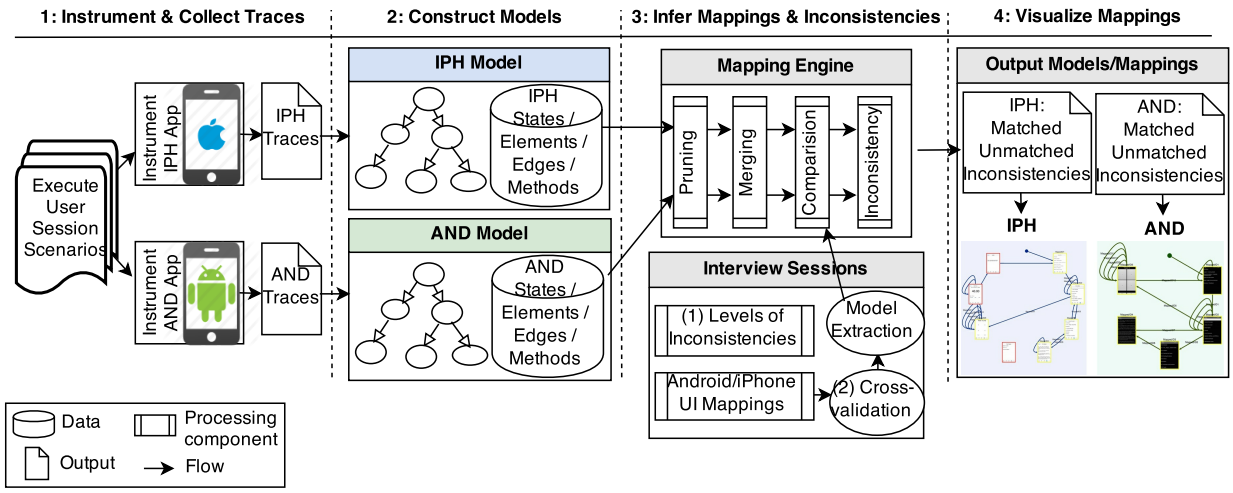

Fig. 1: The overview of our technique for behaviour checking across mobile platforms.

semi-structured survey, with 188 respondents from the mobile development community.

In this work, to identify the most pervasive cross-platform inconsistencies between iOS and Android mobile app-pairs, we conducted an exploratory study by interviewing three industrial mobile developers, who actively develop apps for both platforms. The following categories and examples are extracted from the interviews as well as a document shared with us by the interviewees, containing 100 real-world cross-platform mobile app inconsistencies. Ranked in the order of impact on app behaviour, the most pervasive inconsistency categories are as follows:

Functionality: The highest level of inconsistencies is missing functionality; e.g., "*Notes cannot be deleted on Android whereas iOS has the option to delete notes.*" Or "*After hitting send, you are prompted to confirm to upload – this prompt is missing on iOS.*"

Data: When the presentation of any type of data is different in terms of order, phrasing/wording, imaging, or text/time format; e.g., "*Button on Android says 'Find Events' while it should say 'Find' similar to iOS.*"

Layout: When a user interface element is different in terms of its layout such as size, order, or position; e.g., "*Android has the 'Call' button on the left and 'Website' on the right - iPhone has them the other way around.*"

Style: The lowest level of inconsistency pertains to the user interface style; i.e., colour, text style, or design differences, e.g., "*iOS has Gallery with a blue background while Android has Gallery with a white background*".

We propose an approach that is able to automatically detect such inconsistencies. Our main focus is on the first two since these can impact the behaviour of the apps.

III. Approach

Figure 1 depicts an overview of our technique called CheckCamp. We describe the main steps of our approach in the following subsections.

A. Inferring Abstract Models

We build separate dynamic analyzers for iOS and Android, to instrument the app-pair. For each app-pair, we execute the same set of user scenarios to exercise similar actions that would achieve the same functionality (e.g., reserving a hotel or creating a Calendar event). As soon as the app is started, each analyzer starts by capturing a collection of traces about the runtime behaviour, UI structures, and method invocations. Since the apps are expected to provide the same functionality, our intuition is that their traces should be mappable at an abstract level. The collected traces from each app are used to construct a *model*:

Definition 1 (Model). *A Model μ for a mobile app M is a directed graph, denoted by a 4-tuple $< \alpha, \eta, V, E >$ where:*

1) *α is the initial edge representing the action initiating the app (e.g., a tap on the app icon).*
2) *η is the node representing the initial state after M has been fully loaded.*
3) *V is a set of vertices representing the states of M. Each $v \in V$ represents a unique screen of M annotated with a unique ID.*
4) *E is a set of directed edges (i.e., transitions) between vertices. Each $(v_1, v_2) \in E$ represents a clickable c connecting two states if and only if state v_2 is reached by executing c in state v_1.*
5) *μ can have multi-edges and be cyclic.*

Definition 2 (State). *A state $s \in V$ represents the user interface structure of a single mobile app screen. This structure is denoted by a 6-tuple, $< \gamma, \theta, \tau, \lambda, \Omega, \delta >$, where γ is a unique state ID, θ is a classname (e.g., name of a `View Controller` in iOS or an `Activity` in Android), τ is the title of the screen, λ is a screenshot of the current screen, Ω is a set of user interface elements with their properties such as type, action, label/data, and δ is a set of auxiliary properties (e.g., tag, distance) used for mapping states.*

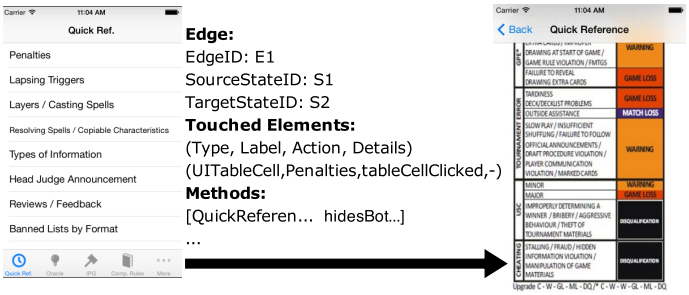

Fig. 2: An *edge* object of MTG iPhone app with its touched element and methods.

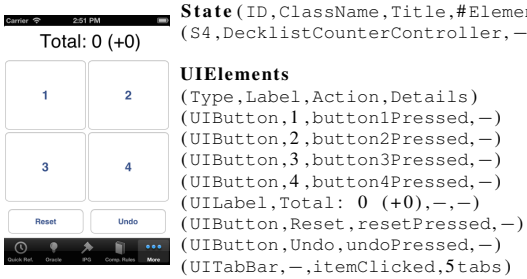

Fig. 3: A snapshot of a *state* in MTG iPhone app with its captured UI *element* objects.

Definition 3 (Edge). *An edge $e \in E$ is a transition between two states representing user actions. It is denoted by a 6-tuple, $< \gamma, \theta, \tau, \lambda, \Omega, \delta >$, where γ is a unique edge ID, θ is a source state ID, τ is a target state ID, λ is a list of methods invoked when the action is triggered, Ω is a set of properties of a touched element*[2] *(i.e. type, action, label/data) and δ is a set of auxiliary properties (e.g., tag, distance) used for mapping purposes.*

1) iOS App Model Inference: In iOS, events can be of different types, such as touch, motion, or multimedia events. We focus on touch events since the majority of actions are of this type. A touch event object may contain one or more finger gestures on the screen. It also includes methods for accessing the UI view in which the touch occurs. We track the properties of the UI element that the touch event is exercised on. To capture this information, we employ the *Category and Extension* [9] feature of Objective-C, which allows adding methods to an existing class without subclassing it or knowing the original classes. We also use a technique called *Method Swizzling* [31], which allows the method implementation of a class to be swapped with another method. To that end, we define a category extension to the `UIApplication` class and a new method in this category. We then swap a built-in method, responsible for sending an event, with the new method. The `swap` method call modifies the method implementations such that calls to the original method at runtime result in calls to our method defined in the category. Additionally, we capture the invoked method calls after an event is fired. We use aspects to dynamically hook into methods and log method invocations. Once an *event* is fired at runtime, all the invoked methods and their classes are traced and stored in a global dataset.

For each event fired, we add an edge to the model. Figure 2 shows an *edge* object of an iPhone app (called MTG, used in our evaluation in Section V) including its captured touched element and invoked methods.

To construct the model, we need to capture the resulting state after an event is triggered. In iPhone apps, a UI state includes the current visible view controller, its properties, accompanied by its set of UI elements. We use a delay to wait for the UI to update properly after an event, before triggering another event on a UI element. Based on our empirical analyses, a two second waiting time is enough for most iOS apps. An event could potentially move the UI forward, backward, or have no effect at all. If the action results in staying in the current view controller, different state mutations could still occur. For instance, UI element(s) could dynamically be changed/added/removed, or the main view of the view controller be swapped and replaced by another main view with a set of different UI element(s). At a low level, moving the UI forward or backward loads a view controller in iPhone apps. Similar to capturing properties of each edge, our approach for capturing UI-structure of each state, combines reflection with code injection to observe *loading view controller* methods.

Once we obtain a reference to the view controller, our approach takes a snapshot of the state and captures all the UI *element* objects in an array associated to the view controller, such as tables with cells, tab bars with tab items, tool bar items, navigation items (left, right, or back buttons), and it loops through all the subviews (e.g., labels, buttons) of the view controller. For each of them, we create an *element* object with its `ID`, `type`, `action`[3], `label`, and `details`.

Figure 3 shows a snapshot of a state in the MTG iPhone app including its UI *element* objects. For instance, the top left button in Figure 3 has 'UIButton' as `type`, '1' as `label`, 'button1Pressed' as `action` (the event handler). We set `details` for extra information such as the number of cells in a list. Using this information, we create a state node in the model.

2) Android App Model Inference: At a high level, our Android dynamic analyzer intercepts method calls executed while interacting with an app and captures UI information (state) upon the return of these methods. Similar to iOS, Android has different types of events. In our approach, we focus on user-invoked events since they contribute to the greatest changes in the UI and allow the app to progress through different states. These types of events get executed when a user directly interacts with the UI of an app, for instance by clicking a button or swiping on the screen. When a user interacts with a UI element, the associated event listener method is invoked, and the element is passed as one of its arguments. To create a new edge in our model, we inspect

[2] A touched element is the UI element which has been exercised when executing a scenario (e.g., a cell in a table, a button, a tab in a tab bar).

[3] `action` pertains to the event handler, representing the method that will handle the event.

Algorithm 1: Pruning a Given Model

```
   input : State Graph (G) of a Given Model (M)
   output: Pruned State Graph (P)
1  begin
2      S ← GETVERTICES(G)
3      E ← GETEDGES(G)
4      foreach i = 0, i <COUNT(S), i++ do
5          s1 ← S[i]
6          foreach j = i + 1, j <COUNT(S), j++ do
7              s2 ← S[j]
8              if s1(class) ≡ s2(class) &
               s1(#elements) ≡ s2(#elements) then
9                  elFlag ← TRUE
10                 foreach e1 ∈ s1.Elements do
11                     e2 ← GETELEMENTATINDEX(s1, e1)
12                     if e1.type ≠ e2.type ||
                       e1.action ≠ e2.action then
13                         elFlag ← FALSE
14                         break
15                     end
16                 end
17                 if elFlag then
18                     REMOVEDUPLICATESTATE(S,s2)
19                     UPDATEEDGES(E,s1,s2)
20                 end
21             end
22         end
23     end
24     return P(S,E)
25 end
```

these arguments and extract information about the UI element that was interacted with by the user. This inspection also allows us to separate user-invoked events from other types, by checking whether the method argument was a UI element such as a button or table that the user can interact with. We compare the argument against the *andoird.widget* package [11], which contains visual UI elements to be used in apps.

In our android analyzer, a UI state includes the current visible screen, its properties, accompanied by its set of UI elements. When an executed method returns, we use the activity that called the method to retrieve information about the state of the UI. To access the UI layout of the current view, we use a method provided by the Android library called *getRootView* [11]. This method returns a `ViewGroup` object, which is a tree-like structure of all the UI elements present in the current screen of the app. We traverse this tree recursively to retrieve all the UI elements. Additionally, we capture some unique properties of the UI elements such as labels for `TextViews` and `Buttons`, and number of items for `ListViews`. These properties are used during the mapping phase to compare iOS and Android states at a lower level.

B. Mapping Inferred Models

Next, we analyze each model-pair to infer likely mappings implied by the states and edges through a series of phases. Prior to the *Mapping* phase, two preprocessing steps are required namely *Pruning* and *Merging*.

1) Pruning: The first step in our analysis, is to prune the graph obtained for each platform, in order merge duplicate states. This step is required as our dynamic analyzers capture any state we encounter after an event is fired without checking if it is a unique state or a duplicate state. This check can be carried out either separately in each analyzer tool or once in the mapping phase. Having it in the mapping phase ensures that the pruning procedure is consistent across platforms. Identifying a new state of a mobile app while executing and changing its UI is challenging. In order to distinguish a new state from previously detected states, we compare the state nodes along with their properties, as shown in Algorithm 1.

As input, Algorithm 1 takes all *States* and *Edges*, obtained from the graph (G), and outputs a pruned graph (P). We loop through all the states captured (line 4), and compare each state with the rest of state space (line 6) based on their classes and number of UI elements (line 8). Next, we proceed by checking their UI elements (line 10) for equivalency of `types` and `actions` (line 12). Thus, data changes do not reflect a unique state in our algorithm. In other words, two states are considered the same if they have the same class and set of UI elements along with their respective properties. Detected duplicate states are removed (line 18) and the source and target state IDs for the edges are adjusted accordingly (line 19).

2) Merging: Platform-specific differences that manifest in our models are abstracted away in this phase. This step is required since such irrelevant differences can occur frequently across platforms. For instance, the iPhone app may offer *More* as an option in its tab controller which is different from the Android app. If the iPhone app has more than five items, the tab bar controller automatically inserts a special view controller (called the *More view controller*) to handle the display of additional items. The More view controller lists the additional view controllers in a table, which appears automatically when it is needed and is separate from custom content. Thus, our approach merges the *More* state with the next state (view controller) to abstract away iPhone differences that are platform-specific and as such irrelevant for our analysis. Similarly, the Android app may offer an option *Menu* panel to provide a set of actions. The contents of the options menu appear at the bottom of the screen when the user presses the *Menu* button. When a state is captured on Android and then the option *Menu* is clicked, our approach merges the two states together to abstract away Android differences. Other differences such as Android's hardware back button vs. iPhone's soft back button are taken into account in our graph representations.

3) Mapping: The collected code-based (e.g., `classname`) and GUI-based (e.g., screen `title`) data for states and edges are used in this phase to map the two models, as shown in Algorithm 2. As input, Algorithm 2 takes iPhone (IG) and Android (AG) graphs, produced after the pruning and merging phases, and outputs those models with a set of computed auxiliary mapping properties for their states and edges (MIG and MAG). The algorithm operates on the basis of the following assumptions (1) the model of an app starts with an initial edge that leads to an initial state and (2) conceptually, both models start with the same initial states. An array, *edgePairs*, holds the initial iPhone and Android

Algorithm 2: Mapping two (iOS & Android) Models

```
   input : iPhone State Graph (IG)
   input : Android State Graph (AG)
   output: IG with Mapping Properties (MIG)
   output: AG with Mapping Properties (MAG)
1  begin
2      IS ← GETVERTICES(IG)
3      AS ← GETVERTICES(AG)
4      IE ← GETEDGES(IG)
5      AE ← GETEDGES(AG)
6      edgePairs[0] ← INSERTEDGEPAIR(IE[0], AE[0])
7      foreach i = 0, i <COUNT(edgePairs), i++ do
8          pair ← edgePairs[i]
9          if NOTMAPPED(pair) then
10             s1 ← GETSTATE(IS,pair[iphTrgtId])
11             s2 ← GETSTATE(AS,pair[andTrgtId])
12             iphEdges ← GETOUTGOINGEDGES(s1,IE)
13             andEdges ← GETOUTGOINGEDGES(s2,AE)
14             /*Find closest edge-pairs*/
15             nextPairs ← FINDEDGEPAIRS(iphEdges,andEdges)
16             SETSTATEMAPPINGPROPERTIES(s1,s2)
17             SETEDGEMAPPINGPROPERTIES(nextPairs)
18         end
19         foreach j = 0, j <COUNT(nextPairs),j++ do
20             edgePairs[i+j+1] ← INSERTEDGEPAIR(nextPairs[j])
21         end
22     end
23     return (MIG,MAG)
24 end
```

edges (line 6) and other edge-pairs are inserted through the main loop (line 20). To find the edge-pairs, we first obtain the initial iPhone and Android states (line 10 and 11) based on the target state IDs in the initial edge-pair. We then obtain all the outgoing iPhone edges (`iphEdges` in line 12) and Android edges (`andEdges` in line 13) from the already mapped state-pair. To identify closest iPhone and Android edge-pairs (line 15), we loop through the outgoing edges and calculate σ_{Ed}, based on a set of comparison criteria as defined in Formula 1:

$$\sigma_{Ed} = \min_{\substack{\forall Ed_{iph} \in iphEdges \\ \forall Ed_{and} \in andEdges}} \left(\frac{f(Ed_{iph}, Ed_{and})}{\sum_{i=1}^{N_{flags}} F_i}\right) * 100 \quad (1)$$

where

$$\begin{aligned} f(Ed_{iph}, Ed_{and}) = {} & F_{action} * LD(Iph_{action}, And_{action}) \\ & + F_{label} * LD(Iph_{label}, And_{label}) \\ & + F_{type} * Corresponds(Iph_{type}, And_{type}) \\ & + F_{class} * LD(Iph_{class}, And_{class}) \\ & + F_{title} * LD(Iph_{title}, And_{title}) \\ & + F_{elms} * \sum_{i=1}^{N_{ElPairs}} Similarity(Iph_{elms}, And_{elms}) \\ & + F_{methods} * LD(Iph_{methods}, And_{methods}) \end{aligned}$$

with the `action`, `label`, and `type` of the touched element, `classname`, `title` and attributes of UI elements in the target state, and the `method` calls invoked by the event.

The edge-pair with the lowest computed σ_{Ed} value is selected as the closest Android-iPhone edge-pair and their mapping properties are appended to the model accordingly (line 17). To instantiate different combinations of this metric, we use a set of binary flags, denoted as F_{action}, F_{label}, F_{type}, F_{class}, F_{title}, F_{elms} and $F_{methods}$. The value of each flag is 1 or 0 to activate or ignore a criterion. We propose six different instantiations, listed in Table I, and compare them in our evaluation to assess their effectiveness (discussed in Section V).

TABLE I: Six combinations for mapping.

ID	Combinations of Comparison Criteria
Comb1	ClassName
Comb2	TouchedElement (action, label, type)
Comb3	TouchedElement+ClassName
Comb4	TouchedElement+ClassName+Title
Comb5	TouchedElement+ClassName+Title+UIElements
Comb6	TouchedElement+ClassName+Title+UIElements+Methods

LD in Formula 1 is a relative *Levenshtein Distance* [37] between two strings, calculated as the absolute distance divided by the maximum length of the given strings (See Formula 2). Some string patterns that are known to be equivalent are chopped from the strings before calculating their distance. For instance, the words "Activity" in Android `classname` and "ViewController"/"Controller" in iPhone `classname` are omitted.

$$LD(str, str') = \frac{distance(str, str')}{maxLength(str, str')} \quad (2)$$

Corresponds in Formula 1 is used for comparing the element's type based on the corresponding Android-iPhone UI element equivalent mappings. Since iOS and Android have different UI elements, a mapping is needed to find equivalent widgets. We analyzed GUI elements that exist for both native Android [2] and iPhone [23] platforms and identified the differences and similarities on the two platforms. We used and extended upon existing mappings that are available online [17]. During the interview sessions (See Section II), we cross-validated over 30 control, navigation, and UI element mappings (such as button, label, picker and slider) that function equivalently on the two platforms, so that the generated models can be used in this phase. We have made these UI equivalent mappings publicly available [15]. *Corresponds* returns 1 if two elements are seen as equivalent and thus can be mapped, and 0 otherwise.

Further, *Similarity* in Formula 1 is a relative number ([0,1]) between two sets of elements in the two (target) states calculated as follows:

$$Similarity(elAry, elAry') = \frac{elPairCount(elAry, elAry')}{maxCount(elAry, elAry')} \quad (3)$$

where the number of elements that can be mapped is divided by the maximum size of the given arrays. Similar to the touched element, `action`, `label`, and `type` properties of UI elements are used to compute mapping between them.

Finally, going back to our algorithm, mapped edge-pairs are inserted to the main array (line 20), and the next set of states and edges are considered for mapping recursively until no other outgoing edges are left.

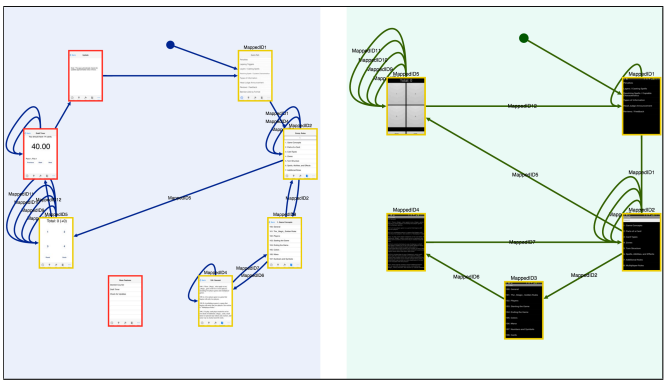

Fig. 4: Visualization of mapping inferences for MTG iPhone (left) and Android (right) app-pairs. The result indicates 3 *unmatched states* shown with red border (including 2 *functionality* inconsistencies where iPhone has more states than Android and 1 platform specific inconsistency with *MoreViewsController* on iPhone). Other 5 *matched* states have *data* inconsistencies shown with yellow border.

4) Detecting Inconsistencies: Any *unmatched* state left without mapping properties from the previous phase is considered as a *functionality* inconsistency. For a *matched* state-pair, since their incoming edges are mapped, we assume that these target states should be equivalent conceptually. *Data* inconsistencies pertain to text properties of the screen such as titles, labels, buttons, and also the number of cells in a table and tabs. Image related and style related properties are out of scope. We calculate data inconsistencies, σ_{State}, in a pair of mapped states by computing LD between two titles as well as text properties of the elements-pairs.

$$\sigma_{State} = \lceil LD(Iph_{title}, And_{title}) \rceil + \sum_{i=1}^{N_{ElPairs}} \lceil LD(Iph_{txt}, And_{txt}) \rceil \tag{4}$$

To compute the correspondence between the elements, we loop through the two arrays of elements. First, we compare the elements' types based on the corresponding Android-iPhone UI element equivalent mappings [14]. For any two elements with the same type and a textual label, we compute LD. We ignore image element types e.g., a button with an image. Where we have multiple elements of the same type, the lowest computed LD is selected as the closest elements-pairs. The σ_{State} is added as *mapping distance* to the models with the same *mapping tag* for the two states (line 16). Additionally, the detected inconsistencies are added to *mapping result* which are later manifested through our visualization.

Eventually, at the end of this phase, each state is marked as either *unmatched*, *matched with inconsistencies* or *completely matched* in the two models, ready to be visualized in the next phase. Thus, we automatically detect mismatched screens by using one platform's model as an oracle to check another platform's model and vice versa.

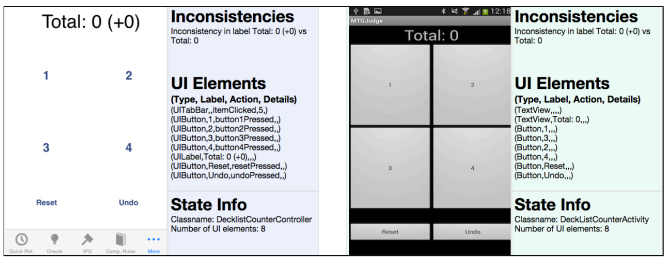

Fig. 5: Zooming into a selected *State* (or *Edge*) represents detected inconsistencies and UI-structure (or touched element and methods) information of iPhone (left) and Android (right) app-pairs.

C. Visualizing the Models

After calculating the likely mappings and detecting potential inconsistencies, we visualize the iOS and Android models, side-by-side, colour coding the mapping results. Red, yellow and dark green border colours around states show *unmatched*, *matched with inconsistencies* and *completely matched* states, respectively. Matched states and edges share the same *mapping tag*>. Figure 4 depicts an example of the output of the visualization phase (it is minimized because of space restrictions). The models can be zoomed in and list detected inconsistencies as well as UI-structure information on selected state(-pair) or touched element and methods information on selected edge(-pair) (See Figure 5).

IV. Tool Implementation

Our approach is implemented in a tool called CHECKCAMP [15].

Its iPhone analyzer is implemented in Objective-C. We use and extend a number of external libraries. ASPECTS [6] uses Objective-C message forwarding and hooks into messages to enable functionality similar to Aspect Oriented Programming for Objective-C. DCINTROSPECT [10] is a library for debugging iOS user interfaces. We extend DCINTROSPECT to extract a UI element's action message, target object, it's properties and values.

The Android analyzer is implemented in Java (using Android 4.3). To intercept method calls, we rely mainly on ASPECTJ.

Our Mapping and visualization engine is written in Objective-C and implements the states recognition and the states/edges mapping steps of the technique. The output of the mapping engine is an interactive visualization of the iOS and Android models, which highlights the inconsistencies between the app-pairs. The visualization is implemented as a web application and uses the CYTOSCAPE.JS library [14], which is a graph theory library to create models.

V. Evaluation

To evaluate the efficacy of our approach we conducted an empirical evaluation, which addresses the following research questions:

TABLE II: Characteristics of the experimental objects, together with total number of edges, unique states, elements and manual unique states counts (MC) across all the scenarios.

ID	App [URL] (#Scenarios)	#LOC		#Edges		#Unique States		#Elements		#MC States	
		AND	IPH	AND	IPH	AND	IPH	AND	IPH	AND	IPH
1	MTG-Judge [16] (2)	3,139	1,822	23	38	11	14	118	125	11	14
2	Roadkill-Reporter [20], [22] (1)	1,799	474	3	17	1	5	48	103	4	5
3	NotifyYDP [26] (1)	1,673	1,960	5	18	2	5	101	96	2	5
4	Family [13] (1)	∼12K	∼14K	10	24	3	4	93	372	3	4
5	Chirpradio [19], [21] (1)	1,705	881	3	4	1	1	9	24	1	1
6	Whistle [24] (1)	702	111	3	4	1	1	6	4	1	1
7	Redmine [18] (1)	1,602	48	6	8	5	4	68	26	5	4
8	Industry App A (2)	8,376	4,015	37	46	13	14	1,041	1,286	13	13
9	Industry App B (4)	∼70k	∼28K	49	53	22	22	715	796	22	22
10	Industry App C (6)	∼68K	∼30K	76	87	37	36	1,142	1,028	37	36
11	Industry App D (4)	∼69K	∼28K	66	71	29	31	940	1,803	29	29
12	Industry App E (2)	∼68K	∼26K	23	28	11	12	353	265	11	12
13	Industry App F (3)	∼68K	∼28K	53	57	28	28	635	2,182	28	28
14	Industry App G (4)	∼69K	∼29K	53	56	27	27	813	1,128	27	27

RQ1. How accurate are the models inferred by CHECKCAMP?

RQ2. How accurate are the mapping methods? Which set of comparison criteria provides the best results?

RQ3. Is CHECKCAMP capable of detecting valid inconsistencies in cross-platform apps?

A. Experimental Objects

We include a set of seven large-scale industrial and seven open-source iPhone and Android app-pairs (14 app-pairs in total). The industrial app-pairs are collected from two local mobile companies in Vancouver. The open-source app-pairs are collected from Github. We require the open-source app-pairs to be under the same GitHub repository to ensure that their functionally is meant to be similar across iPhone and Android. Table II shows the app-pairs included in our evaluation. Each objects's ID, name, resource, and their characteristics in terms of their size and complexity is also presented. XCODE STATISTICIAN [25] and ECLIPSEMETRICS [12] are used to measure lines of code (LOC) in the iOS and Android apps, respectively.

B. Experimental Procedure

We used iOS 7.1 simulator and a Samsung Galaxy S3, to run the iPhone and Android apps, respectively. To collect traces, two graduate students were recruited. First, they installed a fresh version of each pair of the apps, which were then instrumented by CHECKCAMP. Next, to collect consistent traces, we wrote a set of scenarios for our collected app-pairs and gave each student one scenario for each app to access all use-cases of the Android or iPhone versions of the apps according to the given scenarios. Note that the same user scenario is used for both the iOS and Android versions of an app. The scenarios used in our evaluation are available online [15].

Once traces were collected, CHECKCAMP was executed to obtain the models and mappings. To asses the accuracy of the models generated (RQ1), we compare the number of generated unique states to the actual number of unique states for each app-pair. To form a comparison baseline, we manually examine and navigate the user scenarios for each app-pair and document the number of unique states.

To evaluate the accuracy of the mappings (RQ2), we measure precision, recall, and F-measure for each combination, listed in Table I, and app-pair as follows:

Precision is the rate of mapped states reported by CHECKCAMP that are correct: $\frac{TP}{TP+FP}$

Recall is the rate of correct mapped states that CHECKCAMP finds: $\frac{TP}{TP+FN}$

F-measure is the harmonic mean of precision and recall: $\frac{2\times Precision\times Recall}{Precision+Recall}$

where TP (true positives), FP (false positives), and FN (false negatives), respectively, represent the number of states that are correctly mapped (both fully matched or matched with inconsistencies), falsely mapped, and missed. To document TP, FP, and FN, associated with each app for our combinations of comparison criteria, we manually examine the apps and compare the formed baseline against the reported output.

To validate detected inconsistencies (RQ3), for the best combination calculated in RQ2, we manually examine the reported inconsistencies in each app-pair. The results from our analysis are presented in the next section.

Note that, to the best of our knowledge, there are currently no similar tools to compare the results of CHECKCAMP against. That is why our baselines are created manually.

C. Results and Findings

RQ1: Inferred models. We ran multiple *Scenarios* to cover all the screens/states in each app. For each scenario, the initial model is constructed over its traces and analyzed by CHECKCAMP. Table II presents the total number of *Edges*, *Unique States*, and *UI Elements* for all the scenarios running on each Android and iPhone app, produced by CHECKCAMP. The last column of the table also shows the number of *Unique States* counted *manually*. As far as RQ1 is concerned, our results show that CHECKCAMP is able to identify unique states of a given iPhone and Android app-pair and generate

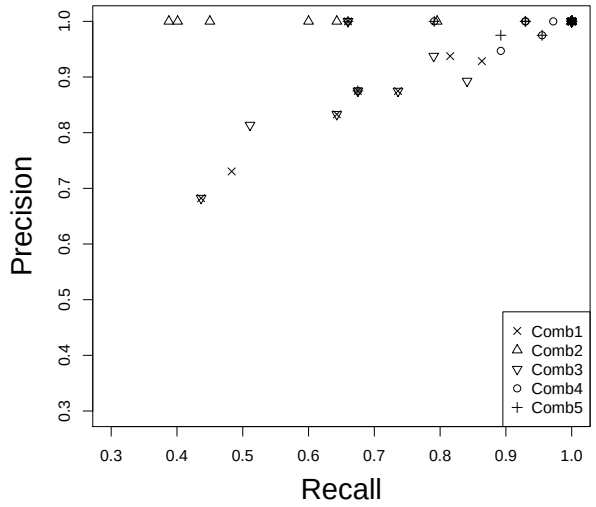

Fig. 6: Plot of precision and recall for the five mapping combinations of each app-pair.

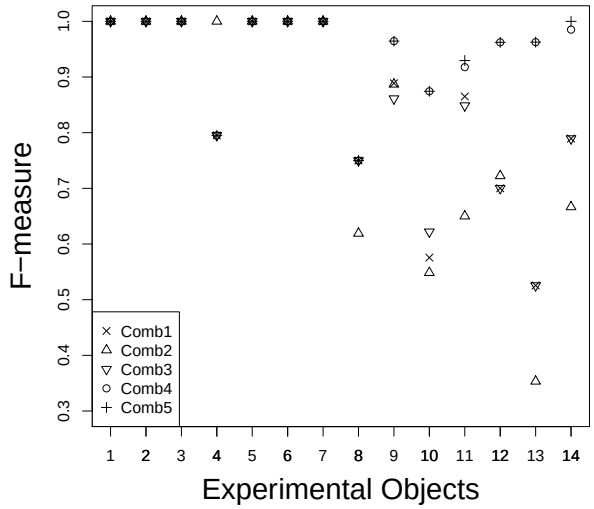

Fig. 7: F-measure obtained for the five mapping combinations on each app-pair.

their state models correctly for each scenario. However, there is a few cases in our industry iPhone apps (IDs 8 and 11) and Android app (ID 2) where the number of manual unique states does not exactly match the number of unique states collected by the dynamic analyzer. This is mainly because our approach currently takes into account the type of the class (either *Activity* in Android or *View Controller* in iOS) in defining a state and thus separate states are captured for different *View Controllers* (discussed in Section VI under Limitations).

RQ2: Different mapping combinations. The precision and recall rates, measured for the first five combinations, listed in Table I, for our 14 app-pairs, are presented in Figure 6. The F-measure is shown in Figure 7. We do not include Combination 6 in these figures since apart from the touched element's event-handler (i.e., `action`), comparing the rest of the method calls did not improve the mapping (discussed in Section VI under Conclusive Comparison Criteria). As far as RQ2 is concerned, our results show that CHECKCAMP is highly accurate in mapping state-pairs. As expected, the results are higher in the open-source apps due to the relative simplicity compared to the industry apps. The comparisons in Figure 6 and Figure 7 reveal that Combination 5 followed by Combination 4 provide the best mapping results in recall, precision, and F-measure for the industry apps. While the results of the combinations have less variation in the open-source apps, Combination 2 shows the best results for them. For the best combinations:

- The recall is 1 for the open-source apps, and for the industry apps it oscillates between 0.68–1 (average 0.88) meaning that our approach can successfully map most of the state-pairs present in an app-pair.
- The precision is 1 for the open-source apps, and for the industry apps it oscillates between 0.88–1 (average 0.97), which is caused by a low rate of false positives (discussed in Section VI under Limitations).
- The F-measure is 1 for open-source apps, and varies between 0.75–1 (average 0.92), for industry apps.

RQ3: Valid inconsistencies. As far as RQ3 is concerned, for the best combinations calculated in RQ2, Table III depicts the number of reported inconsistencies by CHECKCAMP along with some examples. We manually examined and validated (inconsistency categories) in each app-pair across the scenarios. We also computed the average rank and percentage of severity of the valid detected inconsistencies. The used severity ranks are presented in Table IV, which are adopted from Bugzilla [7] and slightly adapted to fit inconsistency issues in mobile apps. We computed the percentage of the valid inconsistencies' severity as the ratio of the average severity rank to the maximum severity rank (which is 5).

We found a number of valid *functionality* inconsistencies in the open-source apps, and interestingly, two in the industrial apps (IDs 10 and 12). However, in some app-pairs, functions such as email clients or opening browsers behaved differently on the two platforms. For instance, in the case of app-pair with ID 3, opening browsers and email clients take the Android app user to outside of the application while that is not the case in the iPhone app. As such, the two models have mismatched states in Table II as CHECKCAMP is not capturing states outside of the app.

Among the data inconsistencies in Table III, are inconsistencies in the number of cells and text of titles, labels, and buttons. Most of the false positives in the reported data inconsistencies (in particular in app-pair with ID 8) are due to the UI structure of a state being implemented differently on the two platforms (discussed in Section VI under Limitations).

TABLE III: Number of reported inconsistencies by CHECKCAMP, validated, average and percentage of their severity with examples in each app-pair.

ID	#Reported (Categories)	#Validated (Categories)	Severity (Avg,%)		Examples of Reported Inconsistencies
1	13 (2 func, 11 data)	13 (2 func, 11 data)	2.6	52%	Android missing 'Draft Time'/'Update' functionality (Figure 4) # table cells: iPhone(12628) vs. Android(6336)
2	4 (3 func, 1 data)	1 (1 func)	5	100%	Android missing 'Help' functionality
3	2 (2 data)	2 (2 data)	2	40%	Title: iPhone 'Notify YDP' vs. Android ''
4	3 (1 func, 2 data)	1 (1 func)	5	100%	Android missing 'Change Password' functionality
5	1 (1 data)	1 (1 data)	2	40%	Button: iPhone '' vs. Android 'Play'
6	0	0	–	–	–
7	5 (1 func, 4 data)	5 (1 func, 4 data)	2.6	52%	iPhone missing a functionality
8	14 (14 data)	2 (2 data)	2	40%	Button: iPhone 'Reset' vs. Android 'RESET'
9	2 (2 data)	0	–	–	–
10	5 (1 func, 4 data)	3 (1 func, 2 data)	3	60%	iPhone missing 'Map' functionality
11	2 (2 data)	1 (1 data)	2	40%	Title: iPhone 'May 14' vs. Android 'Schedule'
12	2 (1 func, 1 data)	2 (1 func, 1 data)	3.5	70%	Android missing 'Participants' functionality
13	1 (1 data)	1 (1 data)	2	40%	Title: iPhone 'Details' vs. Android 'Hotels'
14	0	0	–	–	–
All	54 (9 func, 45 data)	32 (7 func, 25 data)	3	60%	–

TABLE IV: Bug severity description.

Severity	Description	Rank
Critical	Functionality loss, no work-around	5
Major	Functionality loss, with possible work-around	4
Normal	Makes a function difficult to use	3
Minor	Not affecting functionality, behaviour is not natural	2
Trivial	Not affecting functionality, cosmetic issue	1

Thus, CHECKCAMP could not map the elements correctly and reported incorrect inconsistencies.

VI. DISCUSSION

In this section, we discuss our general findings, limitations of CHECKCAMP, and some of the threats to validity of our results.

A. Comparison Criteria

Among the code-based and GUI-based comparison criteria, our evaluation shows that the most effective in the mapping phase pertains to information about the text, action, and type of UI elements that events are fired on, as well as the classname and title of the states. In addition, while we extract a set of method calls after an event fires, our investigation shows that only the action of the touched UI element is effective. We found that even after omitting OS built-in methods, such as delegate methods provided by the native SDK, or library API calls, the method names are quite different in the two platforms and thus provided no extra value in the mapping phase.

B. Limitations

There are some limitations to our current implementation. First, deciding what constitutes a UI state is not always straight forward. For instance, consider two screens with a list of different items. In the Android version of an app the same *Activity* is used to implement the two screens while on the iPhone version separate *View Controllers* exist and currently as shown in Algorithm 1, the type of the class (either *Activity* in Android or *View Controllers* in iOS) is checked (line 8) for identifying a state and thus (mistakenly) separate states are captured in iPhone.

Next, the low rate of false positives in RQ2 include examples where even considering our selected properties all together, CHECKCAMP still lacks enough information to conclude correct mappings. For instance, if an `ImageButton` which contains an image as a background is exercised, there would be no text/label to be compared. Another limitation is with respect to the string edit distance used in our algorithm; for instance, the two classnames *DetailedTipsViewController* and *TipsDetailActivity* are falsely reported as being different based on their distance. This means, if the outgoing edges can not be mapped correctly in Algorithm 2, CHECKCAMP halts and cannot go any further. Backtracking based approaches can be considered to recover if it performs incorrect matches.

Another limitation is related to the high false-positive rate in the reported data inconsistencies in RQ3. In states with multiple elements of the same type, e.g., buttons with images or text properties, our programatic approach in CHECKCAMP cannot map them correctly. Another reason, occurred in some cases, is the UI structure of a state-pair is implemented differently. For instance, in an Android state, buttons exist with text properties whereas in the corresponding iPhone state, those texts are implemented through labels along with buttons. However, this limitation could be addressed through image-processing techniques [28], [42] on the iPhone and Android screenshots collected by the dynamic analyzers. This could enable the detection of other types of inconsistencies between app-pairs including image-related data, layout, or style.

C. Applications

There are various applications for our technique. First of all, our technique supports mobile developers in comprehending, analyzing, and testing their native mobile apps that have implementations in both iOS and Android. Many developers interact with GUI to comprehend the software by creating a mental model of the application [41]. On average, 48% of

a desktop applications's code is devoted to GUI [40]. We believe the amount of GUI-related code is higher in mobile applications due to their highly interactive nature. Thus, using the models to provide a visualization of the apps accompanied with the UI-structure and method calls in the visualization output, would support mobile developers and testers in their program comprehension and analysis tasks and to obtain a better understanding of their mobile apps. The models inferred by CHECKCAMP can also be used for generating test cases. In terms of scalability, the results in Table II show that our approach is scalable to large industrial mobile apps consisting of tens of thousands of LOC and many states.

D. Threats to Validity

The fact that we form the comparison baselines manually could be a threat to internal validity. We did look for other similar tools to compare our results against, without success. Manually going through the different applications to create baselines is labour intensive and potentially subject to errors and author's bias. We tried to mitigate this threat by asking the first two authors to create the comparison baselines together before conducting the experiment. Additinally, we had a small number of scenarios in particular for the open source apps. We tried to mitigate this threat by assuring that these scenarios covered the app screens/states fully. A threat to the external validity of our experiment is with regards to the generalization of the results to other mobile apps. To mitigate this threat, we selected our experimental objects from industrial and open-source domains with variations in functionality, structure and size. With respect to reproducibility of our results, CHECKCAMP, the open-source experimental objects, their scenarios and results are publicly available [15].

VII. RELATED WORK

Dealing with multiple platforms is not specific to the mobile domain. The problem also exists for cross-browser compatibility testing. However, in the mobile domain, each mobile platform is different with regard to the OS, programming languages, API/SDKs, and supported tools, making it much more challenging to detect inconsistencies automatically.

Mesbah and Prasad [38] propose a functional consistency check of web application behaviour across different browsers. Their approach automatically analyzes the given web application, captures the behaviour as a finite-state machine and formally compares the generated models for equivalence to expose discrepancies. Their model generation [39] and mapping technique is based on DOM states of a web application while CHECKCAMP deals with native iOS and Android states and mappable code-based and GUI related metrics of the two mobile platforms. Choudhary *et al.* [30] propose a technique to analyze the client-server communication and network traces of different versions of a web application to match features across platforms.

In the mobile domain, Rosetta [34] infers likely mappings between the JavaME and Android graphics APIs. They execute application pairs with similar inputs to exercise similar functionality and logged traces of API calls invoked by the applications to generate a database of functionally equivalent trace pairs. Its output is a ranked list of target API methods that likely map to each source API method. Cloud Twin [36] natively executes the functionality of a mobile app written for another platform. It emulates the behaviour of Android apps on a Windows Phone where it transmits the UI actions performed on the Windows Phone to the cloud server, which then mimics the received actions on the Android emulator. To our best knowledge, none of the related work addresses inconsistency detection across iOS and Android mobile platforms.

VIII. CONCLUSION AND FUTURE WORK

This work is motivated by the fact that implementation of mobile apps for multiple platforms – iOS and Android – has become an increasingly common industry practice. As a result, a challenge for mobile developers and testers is to keep the app consistent, and ensure that the behaviour is the same across multiple platforms. In this paper, we proposed CHECKCAMP, a technique to automatically detect and visualize inconsistencies between iOS and Android versions of the same mobile app. Our empirical evaluation on 14 app-pairs shows that the GUI model-based approach can provide an effective solution; CHECKCAMP can correctly infer models, and map them with a high precision and recall rate. Further, CHECKCAMP was able to detect 32 valid functional and data inconsistencies between app versions.

While we are encouraged by the evaluation results of CHECKCAMP, there are several opportunities in which our approach can be enhanced and extended for future research. The immediate step would be to conduct an in-depth case study, carried out in an industrial setting with a number of developers using CHECKCAMP. This would help validate the efficiency of the mapping and the visualizations. Additionally, the execution of consistent scenarios can be enhanced by the use of mobile apps that have test suites such as CALABASH [8] scripts. The traces generated by test suites can be leveraged in the mapping engine to enhance the approach.

Systematically crawling to recover models is also an alternative to using scenarios. While there are limitations of automated model recovery, it could complement human-provided scenarios, to ensure better coverage. We have taken the first required steps for automatically generating state models of iPhone applications [32] through a reverse engineering technique. There have been similar techniques for Android applications [27], [29], [35], [43].

Another direction is to improve the current dynamic analyzers to capture information regarding each device's network communication (client-server communication of platform-specific versions of a mobile application), as well as the API calls made to utilize the device's native functionality such as GPS, SMS, Calendar, Camera, and Gallery.

ACKNOWLEDGMENTS

This work was supported in part by an NSERC Strategic Project Grant and a UBC four year fellowship (4YF).

REFERENCES

[1] Android Market Stats. http://www.appbrain.com/stats/.

[2] android.widget Package. http://developer.android.com/reference/android/widget/package-summary.html.

[3] App Store Metrics. http://148apps.biz/app-store-metrics/.

[4] Appcelerator / IDC Q3 2014 Mobile Trends Report. http://www.appcelerator.com/enterprise/resource-center/research/appcelerator-2014-q3-mobile-report/.

[5] Appcelerator / IDC Q4 2013 Mobile Trends Report. http://www.appcelerator.com.s3.amazonaws.com/pdf/q4-2013-devsurvey.pdf.

[6] Aspects. https://github.com/steipete/Aspects.

[7] Bugzilla Severity Definitions. https://wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity.

[8] Calabash. http://calaba.sh/.

[9] Categories and Extensions. http://developer.apple.com/library/ios/#documentation/cocoa/conceptual/objectivec/chapters/occategories.html.

[10] DCIntrospect. https://github.com/domesticcatsoftware/DCIntrospect.

[11] . The developer's guide - Android developers. https://developer.android.com/guide/index.html.

[12] Eclipse Metrics plugin. http://metrics2.sourceforge.net/.

[13] Family App for iPhone and Android. https://github.com/FamilyLab/Family.

[14] Graph theory (a.k.a. network) library for analysis and visualisation. https://http://js.cytoscape.org/.

[15] iOS and Android Dynamic Analyzers, Mapping and Visualization Engine together with Open-source Experimental Scenarios and Results. https://github.com/saltlab/camp.

[16] MTGJudge App for iPhone and Android. https://github.com/numegil/MTG-Judge.

[17] PortKit: UX Metaphor Equivalents for iOS & Android. http://kintek.com.au/blog/portkit-ux-metaphor-equivalents-for-ios-and-android/.

[18] Redmine App for iPhone and Android. https://github.com/webguild/RedmineMobile.

[19] The Android version of Chirpradio. https://github.com/chirpradio/chirpradio-android.

[20] The Android version of Roadkill Reporter. https://github.com/calebgomer/Roadkill_Reporter_Android.

[21] The iOS version of Chirpradio. https://github.com/chirpradio/chirpradio-ios.

[22] The iOS version of Roadkill Reporter. https://github.com/calebgomer/Roadkill_Reporter_iOS.

[23] UIKit Framework Reference. https://developer.apple.com/library/ios/documentation/uikit/reference/uikit_framework/Introduction/Introduction.html.

[24] Whistle App for iPhone and Android. https://github.com/yasulab/whistle.

[25] Xcode Statistician. http://xcode-statistician.mac.informer.com/.

[26] YDP App for iPhone and Android. https://github.com/alakinfotech/YDP.

[27] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013.

[28] T.-H. Chang, T. Yeh, and R. C. Miller. GUI testing using computer vision. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 1535–1544. ACM, 2010.

[29] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, 2013.

[30] S. R. Choudhary, M. Prasad, and A. Orso. Cross-platform feature matching for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014. ACM, 2014.

[31] Cocoa developer community. Method Swizzling. http://cocoadev.com/wiki/MethodSwizzling.

[32] M. Erfani Joorabchi and A. Mesbah. Reverse engineering iOS mobile applications. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 177–186. IEEE Computer Society, 2012.

[33] M. Erfani Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile app development. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'13, pages 15–24. ACM, 2013.

[34] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring likely mappings between APIs. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 82–91. IEEE Press, 2013.

[35] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81. IEEE Press, 2013.

[36] E. Holder, E. Shah, M. Davoodi, and E. Tilevich. Cloud twin: Native execution of Android applications on the Windows Phone. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 598–603. IEEE, 2013.

[37] M. Y. Kao. *Encyclopedia of Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[38] A. Mesbah and M. R. Prasad. Automated cross-browser compatibility testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*, pages 561–570. ACM, 2011.

[39] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. In *ACM Transactions on the Web (TWEB)*, volume 6, pages 3:1–3:30. ACM, 2012.

[40] B. A. Myers and M. B. Rosson. Survey On User Interface Programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI'92, pages 195–202. ACM, 1992.

[41] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE Computer Society, 2012.

[42] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for Dynamic Analysis of iOS Applications. In *Proceedings of the Workshop on Open Research Problems in Network Security (iNetSec)*, pages 65–77, 2011.

[43] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pages 250–265. Springer-Verlag, 2013.