Analysis and Testing of Notifications in Android Wear Applications

Techincal Report OSU-CISRC-11/16-TR04, Ohio State University

Hailong Zhang and Atanas Rountev
Ohio State University, Columbus, OH, USA
Email: {zhanhail,rountev}@cse.ohio-state.edu

***Abstract*—Hundreds of millions of wearable devices are expected to be deployed in the near future. Android Wear (AW) is Google's platform for developing applications for such devices. Our goal is to make a first step toward a foundation for analysis and testing of AW apps. We focus on a core feature of such apps: *notifications* that are issued by a handheld device (e.g., a smartphone) and are displayed on a wearable device (e.g., a smartwatch). We first define a formal semantics of AW notifications in order to capture the core features and behavior of the notification mechanism. Next, we describe a constraint-based static analysis to build a model of this run-time behavior. We then use this model to develop a testing tool which executes test cases across the two devices and measures coverage for AW-specific coverage criteria. We also develop the first tool for automated test generation and GUI exploration for AW apps. These contributions advance the state of the art and enable future analyses and tools in this increasingly important area.**

I. Introduction

Wearable devices. Electronic wearable devices are designed to be worn on the body in order to enable mobility and hands-free/eyes-free activities. While smartwatches and fitness wristbands are currently the most widely used such devices, other device categories are also expected to become increasingly popular, including head-mounted displays, smart jewelry, body cameras, and smart garments.

Traditional mobile devices require direct manipulation, resulting in high cognitive and perceptual load that causes distractions for the user. Wearable devices are supposed to reduce this load, and to allow interactions that are embedded, context-aware, personalized, adaptive, and anticipatory. The long-term trend is toward devices rich with environmental and physiological sensors (e.g., GPS, accelerometer, heart rate) with a wide range of uses in healthcare, fitness, entertainment, manufacturing, construction, field work, etc. Wearable devices are expected to become one of the fastest growing markets in computing. A recent industry report forecasts that over 320 million wearable devices will be shipped in 2017 [1].

Software applications written for wearable devices present a variety of interesting challenges for software engineering researchers—for example, security/privacy, power consumption, UIs optimized for device limitations, and software evolution due to a rapidly evolving marketplace. In this context, it will be essential to develop a body of work on static and dynamic analyses for program understanding, testing, debugging, optimization, and evolution. Our work aims to make an initial contribution in this direction.

Android Wear. Android Wear (AW) is Google's software platform for developing applications for wearable devices [2]. At a high level, there are two major categories of AW apps. First, a *wearable device* may work in conjunction with a *companion handheld device* which is typically a smartphone or a tablet. The software on the wearable and the software on the handheld interact through APIs defined by the platform. A second scenario is when a stand-alone wearable device contains software running independently. Stand-alone AW apps are not well supported by the current AW version 1.5, but are expected to become more popular because of better support in the upcoming AW version 2.0 (to be released officially in the last few months of 2016). Thus, for the rest of this paper, we consider AW apps in which software runs both on a wearable and a companion handheld.

Our work focuses on a core feature of AW apps: *notifications* that are issued by the handheld and displayed on the wearable. In fact, the building and issuing of notifications is the first topic that is introduced by Google's AW developer guide [3]. When a notification is displayed, the user can perform an action that returns the flow of control back to the handheld.

Our contributions. To the best of our knowledge, this key aspect of AW app behavior has not been studied in prior work. Given the increasing importance of wearable devices and the growing popularity of Android Wear, it is highly desirable to establish foundations for analysis and testing of such applications. Our work aims to make an initial but substantial contribution toward this goal, setting the stage for future analyses and tools in this exciting new area.

The specific contributions of this work can be briefly summarized as follows. First, we define a *formal semantics of AW notifications*. Using abstracted syntax and operational semantics, we capture the core features and behavior of the AW notification mechanisms. Second, we describe a *static analysis to build a static model* of this run-time behavior. The analysis is based on static abstractions of relevant run-time entities, together with a constraint-based representation of the important relationships between these entities. Third, we use the model to develop a *testing tool* which executes test cases across the handheld and the wearable, and measures run-time coverage for several AW-specific coverage criteria. As far as we know, this is the first test coverage tool proposed for AW apps. Fourth, we develop the first tool that allows *automated test generation and GUI exploration for AW apps*, based on the output of the static analysis. Finally, we present experimental results and case studies to evaluate the proposed techniques.

```
1  class MyNotificationManager {
2    void create() {
3      Builder builder = new Builder();
4      Intent mainIntent = new Intent(MainActivity.class);
5      PendingIntent mainPI = PendingIntent.getActivity(mainIntent);
6      WearableExtender extender = new WearableExtender();
7      if (...) {
8        Notification chatPage = new Builder().build();
9        extender.addPage(chatPage);
10     }
11     Intent replyIntent = new Intent(RemoteMessagingReceiver.class);
12     PendingIntent replyPI = PendingIntent.getBroadcast(replyIntent);
13     Action replyAction = new Action.Builder(replyPI).build();
14     extender.addAction(replyAction);
15     Intent readIntent = new Intent(MarkReadReceiver.class);
16     PendingIntent readPI = PendingIntent.getBroadcast(readIntent);
17     Action readAction = new Action.Builder(readPI).build();
18     extender.addAction(readAction);
19     builder.setContentIntent(mainPI).extend(extender);
20     NotificationManager.notify(builder.build());
21   }
22 }
```

Fig. 1. Simplified code from QKSMS.

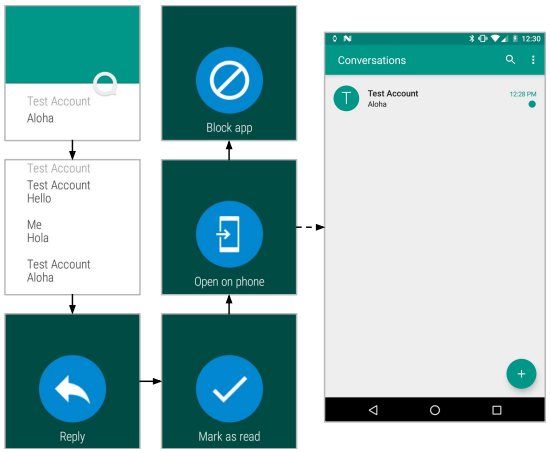

Fig. 2. Screens on a smartwatch.

II. Background and Example

Our focus are Android Wear applications which are defined for and run on a *handheld device* (e.g., a smartphone), but use a *wearable device* (e.g., a smartwatch) to display notification to the user and to receive user feedback. In essence, the wearable device becomes an extension of the GUI for the handheld device. In practice, this means that there is one application APK (running on the handheld), and API calls are issued in this APK to trigger certain behaviors on the wearable. The vast majority of current AW applications fall in this category. Two other alternatives are also possible. First, there could be an APK running on the handheld and another APK running on the wearable, with inter-device communication provided by relevant APIs. Second, there could be a standalone APK on the wearable, without the need to a companion handheld. While both of these scenarios are interesting for future work, at present they are rarely used and are not considered here.

A notification is displayed as a sequence of *screens* on the wearables. Swiping left and right allows the user to navigate between the screens. There are two categories of screens according to their functionality and content: pages and actions. A *page* displays the content of a notification, including title, text, and icon. It is a passive entity—the user observes the information but does not interact with it. An *action* is a screen containing a title and an action button; the user can click the button to execute some desirable functionality by triggering code that executes on the handheld device.

A. Sample Android Wear App

Figure 1 presents a simplified version of code from the QKSMS open-source Android Wear app. Non-essential details have been removed or simplified for clarity. This messaging app interacts with a smartwatch to issue notifications. The call to `notify` at line 20 results in several screens being displayed on the smartwatch, as illustrated in Figure 2. The main page is displayed first. The title of this page is "Test Account" (the message sender identifier) and the page text "Aloha" is the content of the message. If the user swipes to the left, another nested page is displayed with the chat history for this message sender. Another swipe shows the "Reply" action. Through additional swiping the user can access three more actions. The last one ("Block app") is a default AW action that blocks further notification from this app.

In general, a notification has at least one page (the main page) as well as the "Block app" action. There can be additional pages following the main page. These pages are followed by a sequence of actions. When an action's button is touched by the user, the AW framework executes code on the handheld. For example, for the "Open on phone" action, a screen will be opened on the handheld to display the list of conversations. The executed code is in class `MainActivity` and is triggered using the `Intent` object at line 4 in Figure 1.

B. Main Concepts and APIs

The key concepts for the notification mechanism are: (1) a *notification builder* object is used as a factory for *notification* objects; (2) a *wearable extender* is a helper object which, when applied to a notification builder, causes the creation of wearable-specific notifications; (3) several *actions* can be included in a notification to allow the user of the wearable device to respond; (4) an *intent* inside an action determines which handheld app component is invoked in response; (5) *nested pages* can also be included in the extender/builder/notification.

Lines 3 and 8 in Figure 1 create notification builder objects. These are instances of class `NotificationCompat.Builder`, shortened to `Builder` in the example. Line 6 creates a wearable extender. The builders and the extender are ultimately used to create a notification object (call to `build` at line 20) and to display it on the wearable (call to `notify` at line 20).

In general, notifications can be displayed both on handheld devices and on wearable devices. Wearable-specific notifications are created using wearable extender objects. An extender adds more features to a builder. For example, the call to `extend` at line 19 adds the actions and nested pages of `extender` into `builder`. Earlier API calls populate the extender with these actions (lines 13 and 18, calls to `addAction`) and nested pages (line 8, call to `addPage`).

An action object describes a screen to be displayed on the wearable device. The screen contains a title (e.g., "Mark as read") and has an underlying `Intent`. When the user swipes to this screen and touches the icon, the intent is used to trigger

an app component on the handheld device. For the running example, an action object for "Reply" is created at line 13, using a helper action builder object. This action is associated with an intent to execute `RemoteMessagingReceiver` (line 11), an app component that operates on the handheld device. This component is an example of a *broadcast receiver*, a standard Android component type that operates in the background and responds to requests sent through intents. Another intent, created at line 15, is used to trigger a broadcast receiver `MarkReadReceiver` on the handheld, in response to the action created at line 17. Both actions are added to the extender, and then copied to the builder (via `extend`) and then to the notification created at line 20 via `build`.

An instance of class `Intent` contains an abstract description of an operation to be performed. This is the general Android mechanism for triggering app components. For example, if one *activity* (another standard component type in Android) in a handheld device app wants to trigger another activity in the same app, it typically invokes `startActivity` and provides as parameter an intent that describes the target activity. Similarly, a call to `sendBroadcast` is used to trigger a broadcast receiver based on a given intent. Because of the widespread use of this mechanism, prior work (e.g., [4]–[6]) has considered the semantics of intents and the static modeling of this semantics.

For an intent to be used as part of the notification mechanism analyzed in our work (which works across two devices rather than inside a single device), it has to be wrapped by a helper `PendingIntent` object. Lines 12 and 16 in the example show the creation of these helper objects. The pending intent is given to the Android notification manager as part of the action object, and when the action is actually performed (i.e., the action icon is touched by the user), the pending intent is used to access the underlying "regular" intent. At that time, the conceptual equivalent of a call such as `startActivity` or `sendBroadcast` occurs using that intent object.

The call to `setContentIntent` at line 19 is used to add a default "Open on phone" action to the builder. The action is implicitly created as part of this API call. In this example, the target of this action is `MainActivity` (via the intent created at line 4). This activity is executed on the handheld in order to display the list of conversations.

Line 8 creates a notification object and line 9 uses `addPage` to add it to the extender and thus to the notification being created by `build` at line 20. Note that both line 8 and line 20 invoke `build` on a notification builder, and produce a `Notification` instance. In this case one of the notifications (line 20) corresponds to the main notification page and the other one (line 8) to a nested page for the chat history.

In the context of this informal description, the next section formalizes the key abstractions and defines precisely their run-time effects. This formalization serves as the foundation for the proposed static analysis and its testing clients.

III. Formal Semantics of Notifications in Android Wear Applications

The formal definition of the run-time semantics of notifications in Android Wear applications is based on several sets of run-time entities and relations among them. Some of these definitions are for "plain" Java (loosely based on formalizations from [7], [8]), others for "plain" Android (derived from our prior work [9]–[11]), and some are newly-developed by us specifically for Android Wear applications.

A. Plain Java and Plain Android

Plain Java. Our discussion focuses on the semantics of individual statements inside method bodies. The modeling of the type system and the behavior due to calls and returns is well understood (e.g., [7], [8], [12]) and is elided for simplicity.

A Java program contains a set of Java classes. Each class defines a set of fields $f \in \mathsf{Field}$ and a set of methods and constructors. A method body contains declarations of local variables $x \in \mathsf{Var}$ and a control-flow graph in which nodes are statements. The syntax of these statements is defined by

$$s ::= x = \texttt{new}\ C \mid x = y \mid x = y.f \mid x.f = y$$

Generalizations to include method calls and other Java features are well known and are not discussed. The corresponding semantics is based on a set Obj of heap objects, a map Store that defines how local variables refer to these objects, and a map Heap to represent the values of object fields.

o	$\in$	Obj	heap objects
σ	$\in$	$\mathsf{Store} = \mathsf{Var} \to \mathsf{Obj}$	variable values
$\mathcal{H}$	$\in$	$\mathsf{Heap} = (\mathsf{Obj} \times \mathsf{Field}) \to \mathsf{Obj}$	field values

The semantic effects on the store and the heap are

$$\begin{array}{lcl}
\langle x = \texttt{new}\ C, \sigma, \mathcal{H}\rangle & \to & \langle \sigma[x \mapsto o], \mathcal{H}\rangle \\
\langle x = y, \sigma, \mathcal{H}\rangle & \to & \langle \sigma[x \mapsto \sigma(y)], \mathcal{H}\rangle \\
\langle x = y.f, \sigma, \mathcal{H}\rangle & \to & \langle \sigma[x \mapsto \mathcal{H}(\sigma(y), f)], \mathcal{H}\rangle \\
\langle x.f := y, \sigma, \mathcal{H}\rangle & \to & \langle \sigma, \mathcal{H}[(\sigma(x), f) \mapsto \sigma(y)]\rangle
\end{array}$$

The rules show the updated store/heap; $a[b \mapsto c]$ shows that map a is updated by (re)mapping b to c. For $x = \texttt{new}\ C$, $o \in \mathsf{Obj}$ denotes a new heap object of class C.

Plain Android. Our prior work on analysis of Android GUIs [9], [13] defined the GUI-related semantics of several important Android features (e.g., activities, menus, dialogs, widgets, layout definitions, event listeners, etc.). These definitions are not directly related to the problem considered in this paper, but the AW semantics described below can be considered as an extension of these existing definitions.

B. Notifications in Android Wear

A *notification* is a message displayed outside an application's normal GUI. For the AW applications we consider, an application running on a handheld device uses notifications to display information on a companion wearable device.

Instances of the relevant AW classes, and the sets of all such instances, will be denoted as follows

no	$\in$	$\mathsf{Notif} \subset \mathsf{Obj}$	notifications
nb	$\in$	$\mathsf{NotifBuilder} \subset \mathsf{Obj}$	notification builders
we	$\in$	$\mathsf{WearExtender} \subset \mathsf{Obj}$	wearable extenders
ac	$\in$	$\mathsf{Action} \subset \mathsf{Obj}$	actions
in	$\in$	$\mathsf{Intent} \subset \mathsf{Obj}$	intents
pi	$\in$	$\mathsf{PendingIntent} \subset \mathsf{Obj}$	pending intents

These objects were discussed informally in Section II.

After a notification is created in the handheld device app, it can trigger a new screen on the wearable device. This is done through a call to `notify`, as illustrated by line 20 in Figure 1. For the purposes of control-flow and data-flow analysis, `notify` causes the execution of event-processing logic on the wearable device, which then triggers event-handling code back in the handheld device, in a component such as an activity or a broadcast receiver.

The problem of analyzing inter-component control flow and data flow in Android apps is of fundamental importance and has been the target of many existing analyses (e.g., [4]–[6], [10], [11]). For AW apps, `notify` is a control-flow exit point which has to be matched with a subsequent re-entry point in the handheld app code. In essence, the notification mechanism provides a new path for inter-component control/data flow, but this time involving two devices. Our static analysis is the first approach to model this kind of inter-component interactions. The matching of control-flow exit points and re-entry points is part of the analysis output, and can be used for the purposes of other static analyses and their clients (e.g., testing, debugging, security analysis, and profiling).

C. Builders, Extenders, and Notifications

There are several categories of API calls that are related to builders, extenders, and notifications created from them. The specifics of these calls can be abstracted using several abstract operations. The subset of API calls relevant for our purposes is captured by the following definitions for the abstract syntax of statements s:

$$s \quad ::= \quad x = \texttt{addaction}(y, z) \mid x = \texttt{setaction}(y, z) \mid x = \texttt{extend}(y, z) \mid x = \texttt{build}(y) \mid \texttt{notify}(x)$$

Adding actions. Abstract operation `addaction` represents an API call that adds an action to a wearable extender, and thus to wearable-specific notifications created with the help of this extender. Parameter y refers to the extender, while z refers to the action being added. The return value of `addaction` is a reference to the updated extender (i.e., x and y are aliases).

To express the semantics of `addaction`, we generalize the heap with an artificial field `weactions` $\in$ Field for extenders $we \in$ WearExtender:

$$\mathsf{Heap} = \ldots \cup (\mathsf{WearExtender} \times \{\texttt{weactions}\} \rightarrow \mathsf{Action}^*)$$

The field stores the sequence of actions that have been added to the extender. The semantics can be expressed as

$$\langle x = \texttt{addaction}(y, z), \sigma, \mathcal{H} \rangle \rightarrow \langle \sigma[x \mapsto \sigma(y)], \mathcal{H}[(\sigma(y), \texttt{weactions}) \mapsto \mathcal{H}(\sigma(y), \texttt{weactions}) \circ \sigma(z)] \rangle$$

where $\circ$ denotes the concatenation of sequences.

The same `addaction` operation can be applied to a notification builder. The modeling is similar: a field `nbactions` is defined for builder objects, and the effects of the operation are similar to the ones for extenders.

Default action. A notification builder can have a default wearable-specific action "Open on phone", as illustrated in the running example. If `setContentIntent` is called on a builder (line 19 in Figure 1), this implicitly creates such a default action and associates it with the builder. We model these effects using an abstract operation $x = \texttt{setaction}(y, z)$ where y refers to a builder and z refers to the action. A field `default` models this association

$$\mathsf{Heap} = \ldots \cup (\mathsf{NotifBuilder} \times \{\texttt{default}\} \rightarrow \mathsf{Action})$$

The semantics of `setaction` is to map $\mathcal{H}(\sigma(y), \texttt{default})$ to $\sigma(z)$ and to copy the value of y to x.

Extending a builder. An abstract operation $x = \texttt{extend}(y, z)$ takes as input a notification builder referenced by y and a wearable extender referenced by z. The return value is a reference to the same builder object. When `extend` is executed, a snapshot of the current state of the extender is stored inside the builder. In our semantic definitions, this can be modeled by copying the action list of the extender to the builder. Thus, we introduce a field `weactions` in the builder, and set $\mathcal{H}(\sigma(y), \texttt{weactions})$ to have the value of $\mathcal{H}(\sigma(z), \texttt{weactions})$.

Building notifications. An operation $x = \texttt{build}(y)$ uses the state of the builder referenced by y to create and initialize a notification object $no \in \mathsf{Notif}$. Local variable x is assigned a reference to no. As with the builders and extenders, a key property of the object state is the list of actions, which requires the following heap extension:

$$\mathsf{Heap} = \ldots \cup (\mathsf{Notif} \times \{\texttt{actions}\} \rightarrow \mathsf{Action}^*)$$

Given $nb = \sigma(y)$, the actions for the new notification are defined as follows. If `weactions` in nb is not empty, the new notification's `actions` field is set to be $\mathcal{H}(nb, \texttt{weactions}) \circ \mathcal{H}(nb, \texttt{default})$. However, if `weactions` is empty, `actions` is set to $\mathcal{H}(nb, \texttt{nbactions}) \circ \mathcal{H}(nb, \texttt{default})$. This behavior corresponds to two scenarios. First, if nb was extended by an extender with a non-empty action list, these actions are the ones shown on the wearable (followed by nb's default action). It is also possible for an extender to provide no actions, but rather to set other options—e.g., the display style. In this case actions added directly to the builder are displayed on the wearable.

In addition, a pre-defined "Block app" action is added at the end of the action list, to allow blocking of further notifications. Figure 2 illustrates the resulting sequence of actions.

D. Actions and Intents

To model API calls related to intents, pending intents, and actions, we define the following abstract syntax:

$$s \ ::= \ x = \texttt{buildpending}(y) \mid x = \texttt{buildaction}(y)$$

Operation `buildpending` abstracts API calls that build a pending intent wrapped around a regular intent referenced by y. Lines 5, 12, and 16 in Figure 1 contain examples of such calls. The resulting pending intent can then be used when a new action object is created: in the second production above, y refers to this pending intent. Operation `buildaction` abstracts two cases: (1) a construction call in a `new Action` expression, and (2) the use of an *action builder*, as illustrated at lines 13 and 17 in Figure 1. Similarly to how notification builders are used to create notifications, action builders can be used to create actions. For simplicity we elide the relevant details, but our implementation handles both cases.

Regardless of how an action object is created, part of its internal state is a pending intent. In the semantic definitions we need heap generalizations

$$\mathsf{Heap} = \ldots \cup (\mathsf{PendingIntent} \times \{\texttt{intent}\} \rightarrow \mathsf{Intent}) \cup (\mathsf{Action} \times \{\texttt{pending}\} \rightarrow \mathsf{PendingIntent})$$

The semantics of `buildpending` and `buildaction` is as expected and is not shown in detail.

E. Nested Pages

Each notification object displays a main notification page. However, sometimes additional information may be needed to provide more details. Such information can be displayed on nested pages, accessible when the user swipes to the left. Such pages can be added by creating additional notification objects and attaching them to the main notification object. Notification `chatPage` in Figure 1 is an example of a nested page.

The abstract syntax is $s ::= x = \texttt{addpage}(y, z)$, where y refers to a wearable extender and z refers to the nested notification object. The sequence of pages added to an extender can be represented by a field `pages`:

$$\mathsf{Heap} = \ldots \cup (\mathsf{WearExtender} \times \{\texttt{pages}\} \rightarrow \mathsf{Notif}^*)$$

In the semantic definition, $\mathcal{H}(\sigma(y), \texttt{pages})$ is updated by appending $\sigma(z)$; in addition, y is copied into x. We also need to generalize builders and notifications with similar fields `pages`. The semantics of `extend` and `build` includes the copying of the value of `pages` to a builder or a notification, respectively.

Two additional aspects of the semantics should be noted. First, suppose that a notification no contains a nested page no'. Even though no' may have its own actions, they do not affect the actions for no. In other words, $\mathcal{H}(no, \texttt{actions})$ is independent of $\mathcal{H}(no, \texttt{pages})$. Second, when no is actually displayed on the wearable device, repeated swiping to the left will first show the sequence of its nested pages, and then the sequence of its actions. This behavior is illustrated by Figure 2.

IV. Static Analysis

Given the abstracted language from the previous section, we develop a static analysis of the creation and propagation of notifications and related objects. Specifically, the analysis defines static abstractions of relevant objects, models the propagation of references to such objects, and determines important relationships between them.

A similar reference-propagation problem for plain Java can be solved using a *constraint graph*. A graph node corresponds to a variable $x \in \mathsf{Var}$, a field $f \in \mathsf{Field}$, or an allocation `new` C. Edges encode constraints on values. For example, an assignment $x = y$ is represented by an edge $y \rightarrow x$, showing that the set of values for y is a subset of the set of values for x. Forward reachability from `new` C determines which variables and fields refer to the corresponding C instances. Such an analysis is classified as a flow-insensitive, context-insensitive, field-based reference analysis [14], [15]. Our analysis for AW apps generalizes this approach. Various precision extensions of this standard Java analysis can be defined (e.g., [8], [15], [16]) and can be combined with our AW-specific analysis.

The conceptual input to the analysis is a program representation based on the abstracted semantics presented earlier. Figure 3 shows this representation for the running example. The analysis implementation works on the three-address Jimple representation from the Soot analysis framework [17] and conceptually maps call statements to these abstract operations.

```
1  Builder a = new Builder();
2  Intent b = new Intent(MainActivity.class);
3  PendingIntent c = buildpending(b);
4  WearableExtender d = new WearableExtender();
5  Builder e = new Builder();
6  Notification f = e.build();
7  addpage(d,f);
8  Intent g = new Intent(RemoteMessagingReceiver.class);
9  PendingIntent h = buildpending(g);
10 Action i = buildaction(h);
11 addaction(d,i);
12 Intent j = new Intent(MarkReadReceiver.class);
13 PendingIntent k = buildpending(j);
14 Action l = buildaction(k);
15 addaction(d,l);
16 Action m = buildaction(c);
17 Builder n = setaction(a,m);
18 extend(n,d);
19 Notification o = build(a);
20 notify(o);
```

Fig. 3. Abstracted program representation.

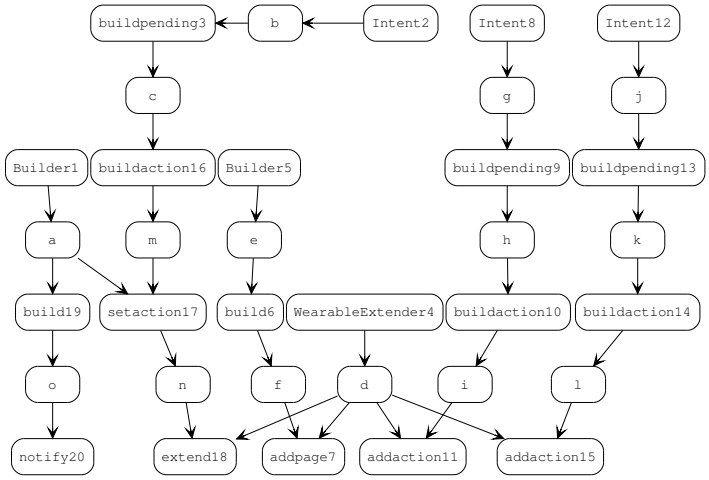

Fig. 4. Constraint graph for the running example.

A. Constraint Graph

Operation nodes. In addition to the standard constraint graph nodes listed above, we use a set OP of operation nodes. The abstract operations defined in the previous section are represented by such nodes. For $x = op(y)$, the corresponding node n has an incoming edge from the node for variable y, and an outgoing edge to the node for x. If the operation has two parameters, there is a second incoming edge. Figure 4 shows the constraint graph for the running example. Numerical suffixes correspond to line numbers in Figure 3.

Object creation. Node sets NB, WE, IN, BN, BP, and BA represent run-time objects created by program statements. Let NB be the set of allocation nodes corresponding to `new` expressions that create notification builders (e.g., nodes `Builder2` and `Builder5` in Figure 4). Similarly, let WE be the set of nodes for wearable extender `new` expressions, and IN be the similar set for intents.

In addition to `new` expressions, operation nodes may create new objects. A notification is created with $x = \texttt{build}(y)$. Each such operation corresponds to a constraint graph node $n \in \mathsf{BN} \subset \mathsf{OP}$. In the example, $\mathsf{BN} = \{\texttt{build6}, \texttt{build19}\}$.

Similarly, $x = \texttt{buildpending}(y)$ creates a pending intent and is represented by a node $n \in \mathsf{BP} \subset \mathsf{OP}$. Finally, action objects can be created either with `new` expressions, or with `build` calls on action builders. Both cases are abstracted with $x = \texttt{buildaction}(y)$, for which we have a node $n \in \mathsf{BA} \subset \mathsf{OP}$. In the example, BA contains three `buildaction` nodes.

B. Constraint-Based Analysis

We define the analysis in terms of several relations. These relations are described in declarative fashion, using several inference rules. As usual, the rules should be read "if the premises (above the line) are true, the conclusion (below the line) is also true". Later we describe how the relations are actually computed.

The flow of object references is represented by $\mathsf{flowsto} \subseteq (\mathsf{NB} \cup \mathsf{WE} \cup \mathsf{IN} \cup \mathsf{BN} \cup \mathsf{BP} \cup \mathsf{BA}) \times (\mathsf{Var} \cup \mathsf{Field} \cup \mathsf{OP})$. A pair n flowsto n' shows that an object represented by n is propagated to a variable, a field, or a parameter of an operation. The inference rules for standard propagation are straightforward. For a node $n \in \mathsf{NB} \cup \mathsf{WE} \cup \mathsf{IN} \cup \mathsf{BN} \cup \mathsf{BP} \cup \mathsf{BA}$ with a left-hand side variable x,

$$\frac{n \to x}{n \ \mathsf{flowsto} \ x}$$

Transitivity is defined as expected: for any n, n', n''

$$\frac{n \ \mathsf{flowsto} \ n' \qquad n' \to n''}{n \ \mathsf{flowsto} \ n''}$$

Builders and extenders. Additional relations are used to capture the AW-specific abstractions introduced in Section III For example, $x = \texttt{setaction}(y, z)$ takes as input a builder y and an action z. Relation $\mathsf{default} \subseteq \mathsf{NB} \times \mathsf{BA}$ represents the effects of the corresponding node n and is defined as follows:

$$\frac{nb \ \mathsf{flowsto}_1 \ n \qquad ac \ \mathsf{flowsto}_2 \ n \qquad n \to x}{nb \ \mathsf{flowsto} \ x \qquad nb \ \mathsf{default} \ ac}$$

Here the subscript indicates whether the flow is to the first or to the second parameter of the operation. The rule for `addaction` on an extender (or a builder) is similar: it adds a pair to binary relation $\mathsf{weactions} \subseteq \mathsf{WE} \times \mathsf{BA}$ (or $\mathsf{nbactions} \subseteq \mathsf{NB} \times \mathsf{BA}$).

To represent the effects of $x = \texttt{extend}(y, z)$ we use a relation $\mathsf{extends} \subseteq \mathsf{WE} \times \mathsf{NB}$

$$\frac{nb \ \mathsf{flowsto}_1 \ n \qquad we \ \mathsf{flowsto}_2 \ n \qquad n \to x}{nb \ \mathsf{flowsto} \ x \qquad we \ \mathsf{extends} \ nb}$$

Notifications. Operation $x = \texttt{build}(y)$ creates a new notification based on builder y. The state of this builder, together with the state of its associated extender, determine the content of the notification. Thus, we want to record the triple of `build` call site, builder, and extender as a static abstraction of the run-time notification object. Let $\mathsf{NO} \subseteq \mathsf{BN} \times \mathsf{NB} \times \mathsf{WE}$ denote the set of all such recorded triples. For a node bn representing a `build` operation, we have

$$\frac{nb \ \mathsf{flowsto} \ bn \qquad we \ \mathsf{extends} \ nb}{(bn, nb, we) \in \mathsf{NO}}$$

This set is one of the outputs of our analysis. Further, for each triple $no \in \mathsf{NO}$, we need to determine the set of relevant actions. Relation $\mathsf{actions} \subseteq \mathsf{NO} \times \mathsf{BA}$ captures this information: no actions n shows that the actions created by node n (which is a `buildaction` site) are in the action list for no. Three rules for a `build` node bn represent this association. First, any action of the extender is copied into the notification.

$$\frac{no = (bn, nb, we) \in \mathsf{NO} \qquad we \ \mathsf{weactions} \ ac}{no \ \mathsf{actions} \ ac}$$

Second, the default action of the builder is added.

$$\frac{no = (bn, nb, we) \in \mathsf{NO} \qquad nb \ \mathsf{default} \ ac}{no \ \mathsf{actions} \ ac}$$

Finally, if there are no actions from the extender, the builder's actions are added.

$$\frac{\begin{array}{c} no = (bn, nb, we) \in \mathsf{NO} \\ nb \ \mathsf{nbactions} \ ac \qquad \nexists \ we \ \mathsf{weactions} \ ac' \end{array}}{no \ \mathsf{actions} \ ac}$$

In addition to the notifications and their actions, the analysis outputs which triples $no \in \mathsf{NO}$ flow to which calls to `notify`. For any such $no = (bn, \ldots)$, if bn flowsto n where n is a call to `notify`, the pair (no, n) is reported by the analysis.

Actions and intents. For a node $n \in \mathsf{BA}$ corresponding to $x = \texttt{buildaction}(y)$, the incoming edge $y \to n$ represents the flow of a pending intent. The outgoing edge $n \to x$ propagates the static abstraction of the created action (i.e., node n) to the left-hand-side variable x. The association between the action and the pending intent is represented by a relation $\mathsf{pending} \subseteq \mathsf{BA} \times \mathsf{BP}$. The inference rule is as expected:

$$\frac{pi \ \mathsf{flowsto} \ n}{n \ \mathsf{pending} \ pi}$$

The modeling of $x = \texttt{buildpending}(y)$ is similar: it updates a relation $\mathsf{intent} \subseteq \mathsf{BP} \times \mathsf{IN}$ which associates a pending intent with the underlying real intent.

Nested pages. The modeling of nested pages, created by $x = \texttt{addpage}(y, z)$, is similar to the modeling of actions. Relation $\mathsf{pages} \subseteq \mathsf{WE} \times \mathsf{BN}$ records which notifications are added to which extenders at `addpage` nodes n

$$\frac{we \ \mathsf{flowsto}_1 \ n \qquad bn \ \mathsf{flowsto}_2 \ n \qquad n \to x}{we \ \mathsf{flowsto} \ x \qquad we \ \mathsf{pages} \ bn}$$

Note that the actions of the notification used at `addpage` will not affect other notifications that are built with we. Thus, we abstract a nested notification using only its `build` site $bn \in \mathsf{BN}$ and do not model the specific builder/extender used at bn.

At a call to `build`, the pages list of the extender is copied to the new notification.

$$\frac{no = (bn, nb, we) \in \mathsf{NO} \qquad we \ \mathsf{pages} \ bn'}{no \ \mathsf{pages} \ bn'}$$

Here relation pages is extended to include a subset of $\mathsf{NO} \times \mathsf{BN}$.

Analysis algorithm. Computing a solution to the system of constraints is done in several stages. First, the constraints graph is build from the program representation. Next, forward reachability from $n \in \mathsf{NB} \cup \mathsf{WE} \cup \mathsf{BA}$ to `addaction`, `setaction`, and `extend` nodes is used to compute relations weactions, nbactions, and extends. Then, set NO is determined based

on reachability from notification builders to `build` nodes, and relation `actions` for $no \in \mathsf{NO}$ is computed. Finally, reachability from `build` to `notify` nodes is examined. The processing of `addpages`, `buildaction`, and `buildpending` is done in a similar manner. The `Intent` sites reaching `buildpending` nodes are analyzed with an intent analysis from our prior work [10] to determine their targets (e.g., `MainActivity`).

C. Analysis Output

Four categories of information are produced by the static analysis. First, the set NO of static abstractions represents the run-time notification objects created by `build` calls with the help of a wearable extender. Each $(bn, nb, we) \in \mathsf{NO}$ is a triple of program statements: a `build` call site bn, a `new` expression nb that creates a notification builder, and a `new` expression we for a wearable extender. In the example, NO contains $no_1 = (\texttt{build19}, \texttt{Builder1}, \texttt{WearableExtender4})$ Although here the `build` site has only one possible builder/extender, we have seen examples in real code where several builders or extenders can reach the same call to `build`.

Second, for each $no \in \mathsf{NO}$, the analysis determines which calls to `notify` it reaches. This information can be used to determine the behavior of these control-flow exit points. For example, no_1 reaches `notify20`. Third, for each no the analysis provides information about the screens it could trigger on the wearable. Any no actions ac and no pages bn corresponds to a screen. In the example we have no_1 actions `buildaction`i for $i \in \{10, 14, 16\}$ and no_1 pages `build6`. The corresponding screens are shown in Figure 2.

Actions can bring the control flow back to the handheld app. For any no, the analysis identifies the `new` sites for `Intents` that define these re-entry points. Combined with well-known techniques for intent analysis (e.g., [4], [5]), this disambiguates the control flow at `notify` calls. For any combination of no actions ac, ac pending pi, and pi intent in, a `notify` call with no can be matched with the target of intent in for the purposes of further static and dynamic analyses. In the running example we have `buildaction`i pending `buildpending`j for $(i, j) \in \{(10, 9), (14, 13), (16, 3)\}$ and `buildpending`j intent `Intent`k for $(j, k) \in \{(9, 8), (13, 12), (3, 2)\}$. Thus, for each of these three actions, the control-flow re-entry point for `notify20` can be determined by considering the corresponding intent from $\{\texttt{Intent2}, \texttt{Intent8}, \texttt{Intent12}\}$ and its target (i.e., `MainActivity`, `RemoteMessagingReceiver`, or `MarkReadReceiver`).

V. Testing

A. Coverage Criteria

Given the output of the analysis, we define several coverage criteria for testing of AW apps. The goal of these criteria is to ensure comprehensive execution of notification-related run-time behavior. This behavior is implemented by the Android platform code, across two different JVMs (one on the handheld and another on the wearable). Thus, traditional coverage such as statement or branch coverage of the handheld app code is not enough to ensure that the possible variations in run-time behavior are exercised. We propose the following coverage goals, and provide a test coverage tool to support them.

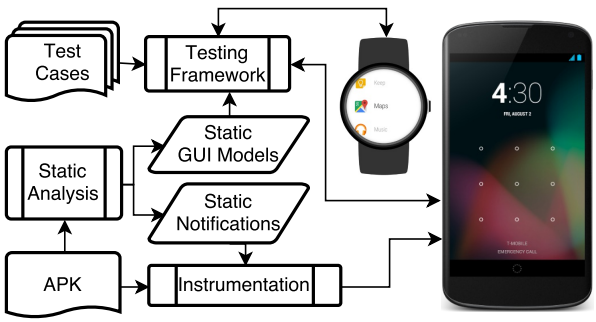

Fig. 5. Overview of testing tool.

Notification sites. Recall that the analysis computes a set of triples $no = (bn, nb, we)$ to represent notification objects. For each no, the analysis determines which calls to `notify` are reached by no. Let n denote such a call site. We define *notification site coverage* as follows: for each n and no that reaches it, execute at least one test case that invokes n with a notification built from nb, extended by we, and built at bn.

This criterion covers all static abstractions of notifications along with every possible `notify` call site where they are issued. We have seen applications in which multiple builders and extenders flow to a single `build` site, and a single builder flows to multiple `build` sites. All such scenarios are captured by this definition. For the example in Figure 3, a test case should cover $(\texttt{build19}, \texttt{Builder1}, \texttt{WearableExtender4}, \texttt{notify20})$.

Nested pages. Nested pages are optionally used by a notification to display supplementary information. The running example illustrates this scenario: a chat page is added only when the condition is true at line 7 in Figure 1. To exercise the run-time behavior related to such pages, we define the following *nested page coverage* criterion: for every no pages bn, such that no reaches a `notify` site n, execute at least one test case in which no is issued by n and a page created by `build` site bn is displayed on the wearable device as part of the run-time notification. Figure 2 shows such an execution: when the notification created by `build19` reaches `notify20`, and then a "swipe left" event occurs on the wearable, the nested page created at `build6` is displayed on the wearable.

Actions. We also consider *action coverage*: for each no actions ac and each `notify` site n reached by no, at least one test case triggers n to issue no and within the run-time notification an action represented by ac is displayed on the wearable. In addition, every possible static target of ac should be re-entered in the handheld device by clicking the action button on the wearable. For the running example, three test cases are needed, one for each action ("Block app" is not of interest). They should trigger the corresponding intent targets on the handheld, i.e., perform the "Reply" action and enter `RemoteMessagingReceiver`, perform the "Mark as read" action and enter `MarkReadReceiver`, and perform the default "Open on phone" action and enter `MainActivity`.

B. Testing Tool

Figure 5 illustrates the structure of our testing tool. In addition to the static analysis described in the previous section, the tool uses (1) code instrumentation to track notifications and

related objects, and (2) a test execution framework based on Google's UI Automator [18].

Static analysis. The static analysis takes an APK file as input and applies the constraint analysis from Section IV based on the Soot framework [17]. The output of the analysis is a set of static notifications stored in an XML file. Each notification is defined by the site of the `build` call, the site of the `notify` call, and the sites of the `new` expressions for the notification builder and wearable extender. We assign an integer ID for each site using a hash code based on the corresponding Soot statement, the signature of the surrounding method, and the statement line number. The IDs of the four sites are used to compute an ID for the notification. This ID is then used by our testing framework to identify the GUI widgets on the screen of the wearable device and to compute coverage, as described shortly. We also assign an integer ID to every action based on its `buildaction` site.

The analysis also computes a static GUI model for each static notification. The structure of AW GUIs is described elsewhere [19]. We model two key GUI components: action card and page card. An *action card* corresponds to an `ActionPage` object at run time and is represented by a `buildaction` site in the static analysis. It contains only one action button with some text as a title. A *page card* corresponds to an instance of `CardFrame` at run time; in the static analysis, it is represented by $no \in \mathsf{NO}$ for the main notification page and $bn \in \mathsf{BN}$ for a nested page. The card displays information from a notification. Figure 2 illustrates these components. For each static notification no, relations `actions` and `pages` computed by the static analysis are used to define the GUI model for no. This GUI model is used later for coverage tracking, automated test generation, and GUI exploration.

Instrumentation. The instrumentation tool takes as input an APK file and the output of the static analysis. For every `new` expression for notification builders and wearable extenders, the instrumentation records the integer ID of the site and associates it with the run-time object. We also record the ID of a call to `build` and associate it with the notification created by it. During testing, before each call to `notify`, the instrumentation checks the IDs of the three sites for the run-time notification plus the ID of the `notify` site. If they match the sites of a static notification, we record that the test covers this part of the notification-sites criterion. We also check the nested pages of the notification. If the page's sites match the ones of the pages from the static notification, we prepend the static ID to the page's title. For the running example, the title of the chat page will be changed from "Test Account" to "1859080457 Test Account". We record coverage for the nested-pages criterion if we can observe this ID in the string title of a page from the screen on the wearable device, during run-time test execution.

Similarly, in order to identify an action, the instrumentation inserts the action's static ID as a prefix of its title. We also add the action's ID as an extra string inside its target `Intent`, and instrument the entry points of the corresponding static targets on the handheld. If a target is an activity, we instrument its `onCreate` method. If a target is a broadcast receiver, the entry point is its `onReceive` method. If a target is an intent service, the entry point is `onHandleIntent`; for a normal service, the entry is `onBind`. One detail to notice is that, since pending intents are registered with Android's `ActivityManagerNative`, we cannot simply change their internal `Intents`. Our tool creates a shadow pending intent at every `new` expression for a pending intent. That shadow object acts similarly to the original one, except that its internal `Intent` contains an extra string with a static action ID. We replace the action's target pending intent with its corresponding shadow object so that we can fetch and check the ID of the action that was the sender of the underlying `Intent`. This check is done when control flow returns back to the entry point of an activity, broadcast receiver, or service in the handheld app. We record action coverage if we are able to retrieve the action ID from the title on the wearable's screen when the notification is issued, and it matches the ID we get from the `Intent` on the handheld after performing the action (the `Intent` is available in `onCreate`, etc. via standard APIs.)

```
1  def test():
2      # test for 'Open on phone' action
3      setup()
4      h.send_sms('+12345', 'Aloha')
5      w().swipe.left() # chat history page
6      w().swipe.left() # 'Reply' action
7      w().swipe.left() # 'Mark as read' action
8      w().swipe.left() # 'Open on phone' action
9      w.click()
10     grep_hit_target_from_logcat()
11     teardown()
```

Fig. 6. Sample test case to trigger the "Open on phone" action.

Testing framework. Since notifications are handled by the Android platform on two independent devices and JVMs, unit testing frameworks such as Robotium [20] and Espresso [21] cannot be applied. Google's UI Automator [18] allows tests running on multiple devices across different processes. However, it requires developers to write extra Java code and configurations. To ease the burden of writing tests, we extended UI Automator Server [22] to support AW devices. Figure 6 shows a sample test case to trigger the "Open on phone" action in the running example; details are elided for brevity. Test case execution communicates with a remote JSON-RPC [23] server running on the wearable device. This server is part of our testing framework. The framework also includes a library containing a socket-based crawler of GUI widget hierarchies. The library builds a socket connection to and communicates with Android's GUI widget server to record the current GUI widgets on the wearable screen (including string titles). It then parses the widget hierarchy information into abstract objects. We use this functionality to identify the static IDs of run-time notifications, pages, and actions, in order to check if the execution behavior during testing meets our coverage criteria.

C. Automated Test Generation and GUI Exploration

The testing tool described above can be used by developers to write test cases and measure run-time coverage. In addition, the tool can be extended to *automatically generate test cases* that aim to achieve high coverage of actions and pages. Given a test case (written by a developer) that issues a notification object at a `notify` site, we can perform automated exploration of the GUI model of this notification. Specifically, for any nested page, we can automatically generate a test case containing a sequence of "swipe left" actions that stops when it reaches the nested page. Similarly, for any action in the GUI model, a test case can be generated automatically to cover it. This test case uses "swipe left" to reach the action,

TABLE I. CHARACTERISTICS OF STUDY SUBJECTS.

Application	Classes	Methods	Jimple Stmts	Time (sec)	`notify` calls	`build` calls	`extend` calls	$\|\mathsf{NO}\|$	(no, n)	(no, n, bn)	(no, n, ac, t)
`QuickLyric`	1139	7913	121772	12.24	2	2	2	2	2	0	2
`WhatsappBetaUpdater`	387	1993	29832	2.41	1	1	1	1	1	0	1
`QKSMS`	1592	9573	140234	11.67	1	3	3	7	6	6	18
`Loop`	555	5225	72796	6.38	1	1	1	1	1	0	3
`Silence`	4898	35618	523060	68.74	2	2	3	3	3	0	7
`Tasks`	1357	6532	83602	7.67	1	1	1	1	1	0	3
`Telegram`	4363	24389	510145	28.42	1	1	2	1	1	0	2
`org.toulibre.cdl`	172	928	10582	1.04	1	1	1	1	1	0	2
`ArcusWeather`	6361	36805	485714	92.19	1	2	1	2	1	3	1
`GroupMe`	4699	26937	362634	40.96	1	2	2	2	1	1	5
`Slack`	5697	34867	418256	152.06	3	5	3	5	3	2	3
`Signal`	5987	41008	588386	68.91	2	2	3	3	3	0	7

and "click" to trigger it. This machinery can also be used to perform *automated exploration* of the GUI structure on the wearable device. Such automated exploration for plain Android has been developed in prior work (e.g., [24]) for the purposes of program understanding and systematic testing. As far as we know, we have developed *the first tool that allows automated test generation and GUI exploration for AW apps*. The proposed approach can be easily modified to support other testing frameworks such as Appium [25].

VI. EXPERIMENTAL EVALUATION

A. Study Subjects

We evaluated the proposed static analysis on eight open-source AW applications from F-Droid [26]. They were selected because they were the only F-Droid apps having the string "WearableExtender" in their decompiled code and allowing installation on an actual AW smartwatch. We wrote test cases to achieve high coverage for the criteria introduced in the previous section. We then compared the resulting run-time notifications against the static ones reported by our analysis.

In addition to these open-source apps, we wanted to demonstrate applicability to closed-source apps. Only APKs are available for such apps. In the absence of source code, it is very challenging to trigger the necessary run-time conditions to achieve high coverage, and to reason about the (in)feasibility of the static solution. In order to provide some initial insights, we studied several closed-source Google Play AW apps. For 4 of them, we were able to obtain sufficient understanding of the app to be able to write meaningful test cases and to make high-confidence judgments on solution feasibility.

Characteristics of the study subjects are shown in Table I; the closed-source apps are listed at the bottom of the table. The number of classes is shown in column "Classes". This includes all classes in an APK except `android.support` libraries. Since Android SDK packs all necessary classes of third-party libraries into APKs, this number also includes those classes. During the experiment, we did not observe any notification-related operations in library classes. Jimple is Soot's intermediate representation; the table shows the number of statements in this IR. Column "Time" shows the running time of the static analysis. On average, the cost of the analysis is around 1.5 seconds per 10K Jimple statements, on a PC with 3.40GHz CPU and 16GB memory. Columns 6 and 7 show the number of `notify` and `build` calls with at least one wearable extender for the notification and the builder, respectively. Column 8 shows the number of `extend` calls.

TABLE II. ACHIEVED RUN-TIME COVERAGE.

Application	notification site coverage	nested page coverage	action coverage
`QuickLyric`	2/2	0	2/2
`WhatsappBetaUpdater`	1/1	0	1/1
`QKSMS`	5/6	5/6	15/18
`Loop`	1/1	0	3/3
`Silence`	3/3	0	7/7
`Tasks`	1/1	0	3/3
`Telegram`	1/1	0	2/2
`org.toulibre.cdl`	1/1	0	2/2
`ArcusWeather`	1/1	3/3	1/1
`GroupMe`	1/1	1/1	3/5
`Slack`	1/3	0/2	1/3
`Signal`	3/3	0	7/7

Column "$|\mathsf{NO}|$" shows the size of set NO, which contains the static abstractions of notifications. The last three columns correspond to the coverage criteria defined in Section V-A. Column "(no, n)" corresponds to the notification-site criterion. Here n is a call to `notify` reached by $no \in \mathsf{NO}$. In some cases (e.g., `QKSMS`) the number of such pairs is smaller than the size of NO because some of the notifications are used as nested pages and not as parameters of `notify`. Column "(no, n, bn)" corresponds to the nested-page criterion. Site bn is a `build` call that creates a nested page added to no through some wearable extender. Column "(no, n, ac, t)" corresponds to action coverage. Here ac is an action of no and t is a handheld app re-entry point triggered by this action.

B. Case Studies

For each application, we wrote test cases to try to achieve complete coverage with respect to the criteria defined earlier. The source code of the app, when available, was examined to ensure that we have indeed achieved the greatest possible coverage. The creation of these test cases was done both to (1) validate the working of our testing tool, and (2) to evaluate the precision of the static analysis, since any coverage goal that cannot be achieved indicates analysis imprecision.

The results from these case studies are shown in Table II. In general, very high coverage was achieved, indicating that the static analysis solution is typically feasible at run time. For 9 of the 12 apps, perfect analysis precision was observed. Additional observations from these studies are presented below.

QKSMS. This application is an alternative to Android's default messaging application. Whenever a message arrives, a notification is issued on the wearable to inform the user. We could not find a way to trigger one of the six static notifications. The missing case occurs when the user receives several messages

from multiple senders. The processing logic for this case is complicated and, to the best of our understanding, the code that issues the notification is dead code.

Telegram. This is a popular chatting application, with close to two million downloads in the Google Play store. It provides secure point-to-point communication. However, Soot failed to generate a valid instrumented APK file for it. Thus we manually instrumented the code and then proceeded to write test cases as with the other apps. This application requires two handheld devices for testing: one for sending messages and one for receiving messages and bridging notifications to a wearable. We utilized UI Automator and our testing framework to manage three devices at the same time and achieved complete coverage.

GroupMe. This is an app for group chats and sharing. It also needs two handheld devices for testing. There are 5 tuples (no, n, ac, t) reported by the static analysis, but only three of them are feasible. The reason for the infeasibility is that a superclass `BaseNotification` contains code for building and issuing notifications (both on the handheld and on the wearable), and only one of its subclasses is related to wearable-only notifications. The spurious targets t come from other subclasses of `BaseNotification`. There are standard static analysis techniques to handle such sources of imprecision (e.g., object sensitivity [8], [27]) and they can be easily integrated with our approach.

Slack. This business app is used for team communication, file sharing, archiving, search, cloud integration, etc. Out of the three static pairs (no, n), only one is feasible. The other two coverage criteria are also affected by this imprecision. We determined that the two infeasible notifications are issued by code that could never be executed at run time. This dead code could be discovered by an interprocedural constant propagation analysis. However, without such a pre-analysis, our analysis reports the effects of the dead code as part of the static solution.

C. Summary

These studies provide initial evidence that the static analysis is precise and can be executed with practical cost. In addition, the testing experiments validate the design and implementation of the proposed testing tool. Both the analysis and the tool advance the state of the art in analysis/testing of AW apps, and can provide a starting point for future work on such apps.

VII. Related Work

There is a significant body of work on static analysis and testing for Android, (e.g., [4], [5], [9]–[11], [13], [24], [28]–[34]), but very little work exists for Android Wear.

Android Wear. Ahola [35] highlights open issues and missing features in the AW platform. Lyons [36] provides suggestions for the design of apps for smartwatches, using feedback from a user study. Min et al. [37] present an exploratory investigation of the battery usage of smartwatches and emphasize that "checking smartphone notifications" is the most common usage for smartwatches. Chauhan et al. [38] characterize various properties (e.g., domain categories, external tracking, information leakage) of apps for AW and other wearable OSes. Liu and Lin [39] examine CPU usage, idle episodes, thread-level parallelism, and microarchitectural behaviors of AW devices. They provide evidence of execution inefficiencies and design flaws in the AW platform. Other researchers have considered the use of AW devices in areas such as healthcare [40], text recognition [41], and mobile biometrics [42]. There is no existing work on modeling the AW notification mechanism and using this modeling in a testing tool, which is the target of our work.

Testing and GUI exploration for plain Android. Choudhary et al. [43] summarize many existing testing and GUI exploration approaches for Android apps. Dynodroid [30] uses guided random GUI exploration. GUIRipper [31] generates a dynamically built GUI model. MobiGUITAR [32] utilizes an enhanced version of GUIRipper and applies test adequacy criteria to it in order to generate test cases. A^{3}E [24] uses GUI exploration based on a control-flow model from static analysis. PUMA [33] is a framework that separates the logic for exploring app execution and the logic for analyzing app properties. ACTEve [34] is a concolic testing tool which symbolically tracks events from their generation to their handling. None of these tools are designed for AW apps, and they cannot be used for analysis and test coverage of AW notifications.

VIII. Conclusions and Future Work

The popularity of wearable devices are expected to increase dramatically over the next decade. Wearables hold the promise of context-aware interactions based on a rich variety of sensor information. Applications in diverse areas such as industry, healthcare, well-being, retail, and entertainment will be deployed in the future. This growing area presents an interesting challenge for the software engineering research community.

Our work focuses on the popular Android Wear platform, and on one of the core AW interaction mechanisms: control flow due to notifications. We abstract the essential concepts of the mechanism and define an analysis to model them statically. The resulting information provides a starting point for further client analyses. We present one such client: a testing tool which measures run-time coverage for notification-related entities, and also allows for automated test generation and GUI exploration. To the best of our knowledge, this is the first work to develop such testing features for AW apps. Our evaluation indicates that the analysis has practical cost and high precision.

There are many open problems in this area, both for the current AW version 1.5 and for the more sophisticated version 2.0 which will be released officially very soon. Both apps with two APKs (one on the handheld and one on the wearable) and apps with wearable-only APKs are expected to become increasingly popular. Examples of interesting problems include data synchronization between wearable and a handheld, custom UIs on the wearable, techniques to reduce battery consumption (e.g., using ambient mode [44]), security analysis, and support for the future evolution of the AW platform. We expect to see significant research advances in this area. The work presented here is a building block for such advances.

References

[1] Gartner, Inc., "Worldwide wearable devices sales," Jan. 2016, `gartner.com/newsroom/id/3198018`.

[2] "Android Wear," developer.android.com/wear.

[3] "Building apps for wearables," developer.android.com/training/building-wearables.html.

[4] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. le Traon, "Effective inter-component communication mapping in Android with Epicc," in *USENIX Security*, 2013.

[5] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to Android inter-component communication analysis," in *ICSE*, 2015, pp. 77–88.

[6] "SCanDroid: Security Certifier for anDroid," spruce.cs.ucr.edu/SCanDroid/tutorial.html.

[7] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 2005.

[8] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *POPL*, 2011, pp. 17–30.

[9] A. Rountev and D. Yan, "Static reference analysis for GUI objects in Android software," in *CGO*, 2014, pp. 143–153.

[10] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *ICSE*, 2015, pp. 89–99.

[11] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for Android," in *ASE*, 2015, pp. 658–668.

[12] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *TOPLAS*, vol. 23, no. 3, pp. 396–450, May 2001.

[13] D. Yan, "Program analyses for understanding the behavior and performance of traditional and mobile object-oriented software," Ph.D. dissertation, Ohio State University, Jul. 2014.

[14] B. G. Ryder, "Dimensions of precision in reference analysis of object-oriented programming languages," in *CC*, 2003, pp. 126–137.

[15] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *CC*, 2003, pp. 153–169.

[16] M. Sridharan and R. Bodik, "Refinement-based context-sensitive points-to analysis for Java," in *PLDI*, 2006, pp. 387–400.

[17] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *CC*, 2000, pp. 18–34.

[18] "UI Automator testing framework," goo.gl/3VedFe.

[19] "UI patterns for Android Wear," goo.gl/KGFykx.

[20] "Robotium testing framework for Android," code.google.com/p/robotium.

[21] "Espresso testing framework," goo.gl/aVoiFf.

[22] "UI Automator server," github.com/presto-osu/android-uiautomator-server.

[23] "JSON-RPC," json-rpc.org.

[24] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *OOPSLA*, 2013, pp. 641–660.

[25] "Appium testing framework," appium.io.

[26] "F-Droid application market," f-droid.org.

[27] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *TOSEM*, vol. 14, no. 1, pp. 1–41, 2005.

[28] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *PLDI*, 2014, pp. 259–269.

[29] W. Yang, M. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *FASE*, 2013, pp. 250–265.

[30] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *FSE*, 2013, pp. 224–234.

[31] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. Memon, "Using GUI ripping for automated testing of Android applications," in *ASE*, 2012, pp. 258–261.

[32] D. Amalfitano, A. Fasolino, P. Tramontana, B. Ta, and A. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, pp. 53–59, 2015.

[33] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *MobiSys*, 2014, pp. 204–217.

[34] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *FSE*, 2012, pp. 1–11.

[35] J. Ahola, "Challenges in Android Wear application development," in *International Conference on Web Engineering*, 2015, pp. 601–604.

[36] K. Lyons, "What can a dumb watch teach a smartwatch?: Informing the design of smartwatches," in *ACM International Symposium on Wearable Computers*, 2015, pp. 3–10.

[37] C. Min, S. Kang, C. Yoo, J. Cha, S. Choi, Y. Oh, and J. Song, "Exploring current practices for battery use and management of smartwatches," in *ACM International Symposium on Wearable Computers*, 2015, pp. 11–18.

[38] J. Chauhan, S. Seneviratne, M. A. Kaafar, A. Mahanti, and A. Seneviratne, "Characterization of early smartwatch apps," in *Workshop on Sensing Systems and Applications Using Wrist Worn Smart Devices*, 2016, pp. 1–6.

[39] R. Liu and F. X. Lin, "Understanding the characteristics of Android Wear OS," in *MobiSys*, 2016, pp. 151–164.

[40] H. Dubey, J. C. Goldberg, M. Abtahi, L. Mahler, and K. Mankodiya, "EchoWear: Smartwatch technology for voice and speech treatments of patients with Parkinson's disease," in *Proceedings of the Conference on Wireless Health*, 2015, p. 15.

[41] L. Arduser, P. Bissig, P. Brandes, and R. Wattenhofer, "Recognizing text using motion data from a smartwatch," in *Workshop on Sensing Systems and Applications Using Wrist Worn Smart Devices*, 2016, pp. 1–6.

[42] A. H. Johnston and G. M. Weiss, "Smartwatch-based biometric gait recognition," in *International Conference on Biometrics Theory, Applications and Systems*, 2015, pp. 1–6.

[43] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for Android: Are we there yet?" in *ASE*, 2015, pp. 429–440.

[44] "Creating wearable apps," developer.android.com/training/wearables/apps/index.html.