Analysis and Testing of Notifications in Android Wear Applications

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

***Abstract*—Android Wear (AW) is Google's platform for developing applications for wearable 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* issued by a handheld device (e.g., a smartphone) and 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 novel testing tool for AW apps. The tool contains a testing framework together with components to support AW-specific coverage criteria and to automate the generation of GUI events on the wearable. These contributions advance the state of the art in the increasingly important area of software for wearable devices.**

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 76 million smart wearable devices will be shipped in 2020 and 22.8 million units will be Android-based [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.

Android Wear. Android Wear (AW) is Google's software platform for developing apps for wearable devices [2]. At a high level, there are two 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 platform APIs. A second scenario is when a stand-alone wearable device contains software running independently. Stand-alone apps are not well supported by AW 1.x but are expected to become more popular because of better support in AW 2.0 (released officially in February 2017). 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. 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, users 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, it is highly desirable to establish foundations for analysis and testing of wearable apps. The contributions of our work can be summarized as follows. First, we define a *formal semantics of AW notifications*. Using abstracted syntax and operational semantics, we capture the core behavior of the 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 novel *testing tool* for AW apps. The tool contains (1) a testing framework to define and execute tests across the two devices, (2) a component to measure run-time coverage for AW-specific coverage criteria, and (3) a component to automate the generation of GUI events on the wearable. Finally, we present experimental results and case studies to evaluate the proposed techniques. We plan to release publicly our implementation and experimental subjects in the near future.

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

```
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.

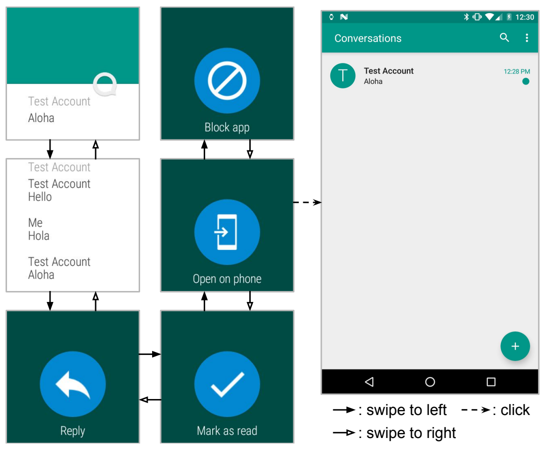

Fig. 2. Screens on a smartwatch.

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. In an exploratory study of Google Play apps we considered the top 100 apps in each app category, and identified 283 apps that contain wearable-specific code. Of those, 57% had this structure. 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, they 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 screens. There are two categories of screens. 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 from right to left 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.

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` triggers 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 their static modeling.

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. For security reasons, the intent should almost always be explicit [7]. Lines 12 and 16 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.

The next section formalizes the key abstractions for AW notifications and defines their run-time effects. This formalization serves as the foundation for the proposed static analysis.

III. Formal Semantics of AW Notifications

The formal definition of the run-time semantics of notifications in AW apps is based on semantic definitions for "plain" Java (loosely based on [8], [9]) and "plain" Android (derived from our prior work [10]–[12]), as well as a formalization newly-developed by us specifically for AW 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., [8], [9], [13]) 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

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

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; $a[b \mapsto c]$ indicates 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; we assume that the initialization of o's fields is represented by separate statements of the form $x.f = y$.

Plain Android. Our prior work on analysis of Android GUIs [10], [14] 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

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.

Analysis of 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], [11], [12]). 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. This information can potentially be used by other static analyses and their clients (e.g., testing, debugging, security analysis, and profiling).

C. Builders, Extenders, and Notifications

Several categories of API calls are related to builders, extenders, and notifications created from them. The subset of API calls relevant for our purposes is captured by the following definitions for the abstract syntax of statements s:

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

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, we generalize the heap with an artificial field `weactions` for extenders $we \in \mathsf{WearExtender}$:

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

The field stores the sequence of actions that have been added to the extender. The semantic effects are

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

where $\circ$ denotes concatenation. Operation `addaction` can also be applied to a notification builder. The modeling is similar, using an artificial field `nbactions` for builder objects.

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` stores 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. 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 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. 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 the wearable displays the actions added directly to the builder.

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:

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

Operation `buildpending` abstracts API calls that build a pending intent wrapped around a regular intent referenced by y, as illustrated at lines 5, 12, and 16 in Figure 1. The pending intent can be used when a new action object is created: in the second production, y refers to this pending intent. Operation `buildaction` represents 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 create notifications, action builders can 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. We need heap generalizations

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

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. Sometimes additional information is 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`:

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

$\mathcal{H}(\sigma(y), \texttt{pages})$ 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

This section describes a static analysis that models the propagation of notification-related 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, assignment $x = y$ is represented by an edge $y \to x$, 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 C instances. Such an analysis is classified as a flow-insensitive, context-insensitive, field-based reference analysis [15], [16]. Our analysis for AW apps generalizes this approach. Various precision extensions can be defined (e.g., [9], [16], [17]) 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 [18] and conceptually maps call statements to these abstract operations.

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,

```
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.

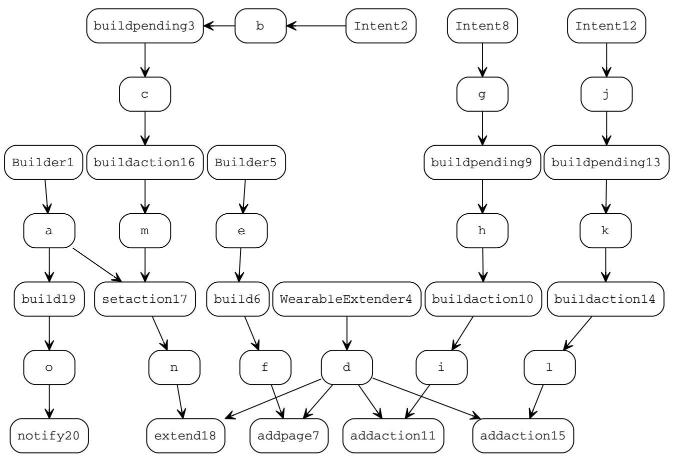

Fig. 4. Constraint graph for the running example.

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. Numeric suffixes correspond to line numbers in Figure 3.

Object creation. Node sets NB, WE, IN, BN, BP, and BA represent objects created by program statements. Let NB be the set of allocation nodes corresponding to `new` expressions for 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 inference rules. Later we describe how the relations are 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 $\mathsf{flowsto}$ 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. First, 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, $n \rightarrow x$ implies n $\mathsf{flowsto}$ x. Further, transitivity is defined as expected: for any n, n', n'', we have that n $\mathsf{flowsto}$ n' and $n' \rightarrow n''$ imply 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:

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

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 $\mathsf{extends} \subseteq \mathsf{WE} \times \mathsf{NB}$

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

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

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

Set NO 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 $\mathsf{actions}$ 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 $\mathsf{flowsto}$ 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 $x = \texttt{buildaction}(y)$, incoming edge $y \rightarrow n$ represents the flow of a pending intent. Outgoing edge $n \rightarrow x$ propagates the static abstraction of the action (i.e., node n) to variable x. The association between the action and the pending intent is represented by relation $\mathsf{pending} \subseteq \mathsf{BA} \times \mathsf{BP}$. The inference rule is as expected: pi $\mathsf{flowsto}$ n implies 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

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

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 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 built 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 $\mathsf{weactions}$, $\mathsf{nbactions}$, and $\mathsf{extends}$. 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 [11] to determine their targets.

C. Analysis Output

Four categories of information are produced by the static analysis. Part 1 of the output is 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`.

Part 2 of the output describes, for each $no \in \mathsf{NO}$, which `notify` calls it reaches. This can be used to determine the behavior of these control-flow exit points. For example, no_1 reaches `notify20`. In Part 3 of the output, for each no there is information about the screens it could trigger on the wearable. Any pair 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`.

Actions bring the control flow back to the handheld app. In Part 4 of the output, each no is associated with 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 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 Tool

The analysis could potentially be used by various clients. We illustrate one such use in the context of a testing tool developed by us. The structure of the tool is shown in Figure 5.

A. Testing Framework

Since notifications are handled by the Android platform on two devices, frameworks such as Robotium [19] and Espresso [20] cannot be used to write AW test cases. We developed and made public AW UIAutomator Server [21], a testing framework for AW apps. The implementation adds AW-specific functionality to the existing UIAutomator Server developed by others [22]. Our AW UIAutomator Server creates a JSON-RPC server on each device listening to incoming events. Developers can write simple Python scripts to send events such as swiping and clicking, entering text into input fields, and simulating the sending of an SMS. The approach works for both emulated and real wearable devices.

The framework contains a library with a crawler of GUI widget hierarchies for Android emulators. Given a notification that has been displayed in the emulator, the library communicates with the GUI widget server to record the current widgets on the wearable screen (including string titles) and then parses this information into abstract objects. For real devices, since the GUI widget server is disabled by default, we cannot use this crawler. The library includes an alternative crawler based on the pytesseract OCR tool [23]. The crawler takes screenshots and recognizes characters via OCR in order to build a representation of the wearable's screen. As described below, we use these capabilities to identify the static IDs of notifications, pages, and actions, in order to check coverage.

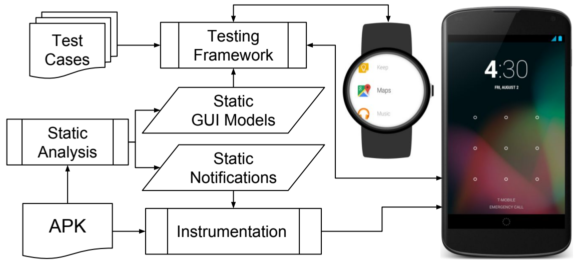

Fig. 5. Overview of testing tool.

B. Coverage Criteria

Consider a set of test cases written and executed using our framework. One interesting question is whether this set comprehensively exercises notification-related run-time behavior. This behavior is implemented by the Android platform code, across two different JVMs. 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. For example, in `QKSMS`, both creating and updating a conversation would trigger a notification. The notification builders for these two operations may both reach function `multipleSenders` which includes several branches and transitively calls `extend`, `build` and `notify`. Statement/branch coverage cannot ensure that all possible behaviors are covered at run time. We propose the following AW-specific coverage goals, and provide coverage measurements for them.

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 any 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`.

C. Measuring Test Coverage

Static analysis. The analysis produces a list of static notifications, each one 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. This information is available in Part 2 of the analysis output (Section IV-C). 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.

Part 3 of the analysis output defines a GUI model for each static notification. This model is simply the set of static abstractions for the notification's actions and nested pages, as defined by relations actions and pages.

Instrumentation. The instrumentation component 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 example, the title of the chat page will be changed from "Test Account" to "1859080457 Test Account". We record coverage for the nested page if we can observe this ID in the string title of a page on the wearable's screen during test execution.

To identify an action, the instrumentation inserts its 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 the `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` or `onStartCommand`. We record action coverage if we can 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.)

D. Automated Generation of Wearable GUI Events

Using the testing framework, a tool user can write a test case containing a mix of GUI events on the handheld and the wearable—e.g., trigger a `notify` site in the handheld app, issue swipe events on the wearable to get to a particular action, and then click the action button. If the goal is to write many such test cases to achieve high coverage of the GUI structure on the wearable, part of this process can be automated. Suppose a tool user writes a test case $T_{initial}$ (for the handheld app) that issues a notification object at a `notify` site. The elements of the GUI model for this notification—that is, its nested pages and actions—can be automatically triggered. For each nested page, we can append to $T_{initial}$ a sequence of swipe events that stops when it reaches that nested page. Similarly, for any action in the GUI model, we can append to $T_{initial}$ swipe events to reach the action, followed by a click event to trigger it. The GUI model does not represent ordering of actions/pages and the number of swipe events needed to reach a particular action/page is not known statically. Thus, after each swipe event, a run-time check (similar to the one used for coverage tracking) determines whether the target action/page is reached. This automation allows several augmented test cases to be automatically generated and executed, starting from a single $T_{initial}$ written by the tool user.

VI. Experimental Evaluation

A. Study Subjects

We evaluated the proposed static analysis on eight open-source AW applications from F-Droid [24]. 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 (in)feasibility of the static solution. For 4 apps we were able to obtain sufficient understanding 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. Jimple is Soot's intermediate representation; the table shows the number of statements in this IR. For open-source apps, we manually identified and filtered out all third-party libraries, and

TABLE I
CHARACTERISTICS OF STUDY SUBJECTS.

Application	Classes	Methods	Jimple Stmts	Time (sec)	`notify` calls (AW/All)	`build` calls (AW/All)	`extend` calls	$\lvert\mathsf{NO}\rvert$	(no, n)	(no, n, bn)	(no, n, ac, t)
`QuickLyric`	1139 (100)	7584 (471)	121772 (8876)	12.24	2/5	2/5	2	2	2	0	2
`WhatsappBetaUpdater`	387 (36)	1891 (146)	29832 (1930)	2.41	1/1	1/1	1	1	1	0	1
`QKSMS`	1592 (672)	9193 (4360)	140234 (73896)	11.67	1/1	3/5	3	7	6	6	18
`Loop`	555 (201)	5070 (1373)	72796 (22808)	6.38	1/1	1/1	1	1	1	0	3
`Silence`	4898 (948)	33860 (5782)	523060 (74000)	68.74	2/6	2/9	3	3	3	0	7
`Tasks`	1357 (956)	6249 (4409)	83602 (55223)	7.67	1/1	1/4	1	1	1	0	3
`Telegram`	4363 (3778)	23405 (19227)	510145 (447549)	28.42	1/4	1/6	2	1	1	0	2
`org.toulibre.cdl`	172 (146)	896 (753)	10582 (8509)	1.04	1/1	1/1	1	1	1	0	2
`ArcusWeather`	6361	36805	485714	92.19	1/5	2/8	1	2	1	3	1
`GroupMe`	4699	26937	362634	40.96	1/3	2/8	2	2	1	1	5
`Slack`	5697	34867	418256	152.06	3/12	5/14	3	5	3	2	3
`Signal`	5987	41008	588386	68.91	2/9	2/12	3	3	3	0	7

then counted the number of application classes, methods, and Jimple statements. These numbers are shown in parentheses in the corresponding columns. Our analysis does not distinguish application classes from third-party library classes.

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 all `notify` and `build` calls and those with at least one wearable extender for the notification and the builder. In total there are 49 `notify` calls and 74 `build` calls, of which 17 and 23 are wearable specific respectively. All these calls are analyzed to determine which subset is for wearables and which is for handhelds. Column 8 shows the number of `extend` calls.

Column "$\lvert\mathsf{NO}\rvert$" 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-B. 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. The small number of nested pages implies the simplicity of the GUI structure of AW notifications. This is because of the design principles based on *micro-interactions* that suggest to "keep the number of detail cards as low as possible" [25].

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.

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

QKSMS. Whenever a message arrives, this app issues a notification on the wearable. We could not trigger one of the six static notifications. This 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. Soot failed to generate a valid instrumented APK file for it. Thus, we manually instrumented the code. The application requires two handheld devices for testing: one for sending messages and one for receiving messages and bridging notifications to a wearable. We utilized 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 techniques to handle such imprecision (e.g., object sensitivity [9], [26]) and they can be easily integrated with our approach.

Slack. This business app is used for team communication, file sharing, archiving, 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 interprocedural constant propagation analysis.

C. Automated Generation of GUI Events

As described in Section V-D, if a tool user writes a test case that issues a particular notification, our approach can generate several augmented test cases by appending swipe events and click events on the wearable. One question is how this approach compares with random testing. To get insights about this comparison, we performed a case study with `ArcusWeather`. A notification in this app has a relatively complex GUI structure, with three nested pages and one action. Using the static GUI model to guide the exploration, our approach generates four augmented test cases, one for each page and action. The test case to cover the first nested page requires two events: a swipe up to activate the notification, and a swipe to the left to reach the page. To cover the second page, the test case requires three events. Four events are required to cover the last page. The test case to cover the action requires six events, with the last one being a click.

One could attempt to achieve the same coverage with random testing. To explore this possibility, we used Google's Monkey tool [27] for random testing. We configured Monkey to explore package `com.google.android.wearable.app` and category `android.intent.category.HOME` so that it would avoid opening irrelevant apps and triggering useless events. Since the targeted events were swipe and click, we also configured the tool to only perform *motion* and *touch* to simulate these two operations. The most common scenario of triggering random events is to open the app list or the navigation bar and start exploring them. In this case, the actual notification is lost from the screen. It is very unlikely that additional events will lead back to the notification and reach the desired page or action. We consider this to be a "stuck" state. In our study we check for a stuck state every 30 events and restart Monkey once such a state is reached. This process is restarted several times, always beginning with the original user-defined test case to trigger the notification. The execution is stopped after a successful run—that is, when the expected page/action is reached/triggered. We record the total number of events in all unsuccessful runs and in the final successful run. Because of the randomness, we perform 10 separate executions of this process and measure the mean numbers of events.

To cover the first nested page, Monkey requires 156 events. To reach the second and third page, Monkey needs 204 and 280 events, respectively. To trigger the action, Monkey requires 291 events. There are two reasons for the large numbers of events. First, random testing is very likely to open the app list and invoke other default apps instead of exploring GUI elements of the notification. Second, randomly generated events can have a bouncing effect producing many useless swipes, e.g., landing on page 2 from page 1 and then going back to page 1. This highlights the benefits of generating GUI events based on a static analysis model rather than randomly.

VII. Related Work

There is a significant body of work on static analysis and testing for Android, (e.g., [4], [5], [10]–[12], [14], [28]–[36]), but very little work exists for Android Wear.

Android Wear. 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 OS. Liu and Lin [39] examine CPU usage, idle episodes, and thread-level parallelism 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 [31] uses guided random GUI exploration. GUIRipper [32] generates a dynamically built GUI model. MobiGUITAR [33] utilizes an enhanced version of GUIRipper and applies test adequacy criteria to it in order to generate test cases. A^{3}E [30] uses GUI exploration based on a control-flow model from static analysis. PUMA [34] is a framework that separates the logic for exploring app execution and the logic for analyzing app properties. ACTEve [35] 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 directly 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. This growth presents interesting challenges for software engineering researchers. Our work focuses on Android Wear and one of its core 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. Our evaluation indicates that the analysis has practical cost and high precision.

There are many open problems in this area, both for AW 1.x and for the more sophisticated AW 2.0. 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 a wearable and a handheld, custom UIs on the wearable, techniques to reduce battery consumption, security analysis, and support for AW evolution.

Acknowledgments. We thank the ICSE reviewers for their valuable feedback. This material is based upon work supported by the U.S. National Science Foundation under awards 1319695/1526459 and by a Google Faculty Research Award.

References

[1] IDC Research Inc., "Worldwide smartwatch market will see modest growth in 2016 before swelling to 50 million units in 2020," http://www.idc.com/getdoc.jsp?containerId=prUS41736916, Sep. 2016.

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

[3] "Building apps for wearables," https://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," http://spruce.cs.ucr.edu/SCanDroid.

[7] "PendingIntent," https://developer.android.com/reference/android/app/PendingIntent.html.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

[19] "Robotium: User scenario testing for Android," https://github.com/robotiumtech/robotium.

[20] "Testing UI for a single app," https://developer.android.com/training/testing/ui-testing/espresso-testing.html.

[21] "Android Wear (AW) UIAutomator server," https://github.com/presto-osu/aw-uiautomator-server.

[22] "Android UIAutomator server," https://github.com/xiaocong/android-uiautomator-server.

[23] "Python-tesseract: A Python wrapper for Google's Tesseract-OCR," https://pypi.python.org/pypi/pytesseract.

[24] "F-Droid application market," https://f-droid.org.

[25] "UI patterns for Android Wear," https://developer.android.com/design/wear/patterns.html.

[26] 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.

[27] "UI/Application exerciser Monkey," https://developer.android.com/tools/help/monkey.html.

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

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

[32] 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.

[33] 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.

[34] 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.

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

[36] H. Zhang, H. Wu, and A. Rountev, "Automated test generation for detection of leaks in Android applications," in *AST*, 2016, pp. 64–70.

[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 Int. Symp. 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 *Int. Conf. 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.