Static Window Transition Graphs for Android

Shengqian Yang*, Hailong Zhang*, Haowei Wu*, Yan Wang*, Dacong Yan†, and Atanas Rountev*
*Ohio State University, Columbus, OH, USA
Email: {yangs,zhanhail,wuhaow,wang10,rountev}@cse.ohio-state.edu
†Google Inc., Mountain View, CA, USA
Email: dacongy@google.com

***Abstract*—This work develops a static analysis to create a model of the behavior of an Android application's GUI. We propose the window transition graph (WTG), a model representing the possible GUI window sequences and their associated events and callbacks. A key component and contribution of our work is the careful modeling of the stack of currently-active windows, the changes to this stack, and the effects of callbacks related to these changes. To the best of our knowledge, this is the first detailed study of this important static analysis problem for Android. We develop novel analysis algorithms for WTG construction and traversal, based on this modeling of the window stack. We also describe an application of the WTG for GUI test generation, using path traversals. The evaluation of the proposed algorithms indicates their effectiveness and practicality.**

I. Introduction

The explosive growth in the number of deployed smartphones and tablets has significantly changed the computing landscape. The correctness, security, and performance of such devices is of paramount importance for many millions of users. For software engineering researchers, this raises high expectations for developing a comprehensive toolset of algorithms for understanding, testing, and verification of Android software.

Our focus is on a key component of such a toolset: a static analysis to create a model of an application's graphical user interface (GUI). Such a model can be used for program understanding, testing, and dynamic exploration [1]–[6]. It could also potentially be a starting point for static data-flow analyses, for example, for checking of security properties, leak defects, and other correctness properties [7]–[24].

We propose a particular form of a GUI model for Android: a *window transition graph* (WTG). Nodes in this graph represent windows and edges represent transitions between windows, triggered by callbacks executed in the UI thread. To allow the development of client data-flow analyses based on the WTG, graph edges are annotated with the sequences of callback methods invoked by the Android platform. These annotations capture *event handling callbacks* and *window lifecycle callbacks*.

Technical challenges and insights. The representation and analysis of these callback methods play a critical role in a static analysis for WTG construction, and more generally, control/data-flow analysis for Android. Transitions between windows are triggered by such methods, and during these transitions additional callbacks occur. The current state of the art in static analysis for Android is inadequate when it comes to represent such run-time behavior. For example, we have seen various cases from real applications where an event handler may force the closing of the current window and its predecessor window, while at the same time opening a new window; this leads to complicated interleavings of callbacks for the three windows. As another example, we have seen many cases where the return from a window does not come back to the predecessor, but rather to another window displayed earlier. In existing work, including our own prior work, there is no conceptual clarity on these possible run-time behaviors and how they can be analyzed in a static control-flow analysis.

We address this problem by clarifying the major elements of such behaviors, with the help of the abstraction of a *window stack*. The window stack generalizes the standard Android notion of a "back stack" [25], which stores the currently-alive activities. (Activities correspond to one category of windows.) Our generalization (1) captures additional categories of windows, and (2) models the changes to the window stack. An important observation is that a single transition in the WTG can have complex effects on the window stack: for example, it can pop and/or push windows, all as part of the same WTG edge. A major contribution of our work is the careful modeling of these stack changes and their related callbacks—both the callbacks that trigger the stack changes, and the callbacks triggered by them. To the best of our knowledge, this is the first detailed study of this important static analysis problem for Android.

The combined analysis of callbacks and the window stack also provides a solution to an important related problem: *which sequences of window transitions are feasible?* One cannot consider all WTG paths, since some such paths are provably infeasible. We can draw an analogy with the sequences of calls/returns in ordinary programs: modeling the possible states of the call stack is a key concern in static analysis of call/return sequences (which, in turn, is an important component of interprocedural data-flow analysis). However, the behavior of the window stack can be significantly more complicated. Our work provides a systematic identification of *valid* WTG paths (and, by trivial extension, valid call/return sequences), which is a critical prerequisite for future developments in interprocedural data-flow analysis for Android. As an exemplar client, we have developed a test generation tool in which valid WTG paths naturally correspond to test cases.

Contributions. The contributions of this work are: (1) definition of the window transition graph (WTG) as a GUI model for understanding, testing, dynamic exploration, and static checking of Android applications; (2) static analysis for WTG construction, employing careful modeling of the interplay between callbacks and the window stack; (3) algorithm to identify valid paths in the WTG, based on modeling of window stack changes; (4) test generation tool based on the WTG; and (5) experimental evaluation of the proposed algorithms.

II. Android Behavior and Its WTG Representation

A. Relevant Android Features

Figure 1 contains an example derived from the APV PDF viewer [26]. For simplicity, the code and its description omit a number of non-essential details. The example illustrates windows (e.g., `ChooseFileActivity`), GUI widgets (e.g., `fileListView`), and event handlers (e.g., `onItemClick`).

Windows. Subclasses of `android.app.Activity` are used to define activities, which are core application building blocks. `ChooseFileActivity`, `OpenFileActivity`, `Options`, and `About` from Figure 1 are such classes; execution starts from an instance of `ChooseFileActivity`, which shows a file list. An activity displays a window containing several GUI widgets. A widget (also referred to as a "view") is an instance of a *view class*. In Figure 1, variables that refer to widgets include `fileListView` (list of files), `l` (the same list), `item` (individual list element), `aboutItem`, `optionsItem` (both are elements of a menu, as described below), and `btn` (a button).

We also consider the two other common categories of Android windows: menus and dialogs. Instances of menu classes represent short-lived windows associated with activities ("options" menus) and widgets ("context" menus). In Figure 1 `OpenFileActitivy` has an options menu, initialized by `onCreateOptionsMenu` to contain menu items `aboutItem` and `optionsItem`. A dialog is an instance of a subclass of `android.app.Dialog`. Both menus and dialogs are used for modal events that require users to take an action before they can proceed [27].[1] We will use $\mathbf{Win}$ to denote the set of all run-time windows (activities, menus, and dialogs), and $\mathbf{View}$ for the set of all run-time widgets in these windows.

A menu/dialog takes control temporarily for a simple interaction with the user, and its lifetime is shorter than activity lifetime. The last activity that was displayed before a menu or a dialog was displayed is considered to be the *owner activity* of this menu/dialog. In the running example, the options menu is owned by `OpenFileActivity`. There are more general cases: for example, in `OpenFileActivity` there exists a button (not shown in Figure 1) for which a long-click event opens a context menu m_1, in which a menu item can be clicked to open a dialog d_1 asking for a page number in the PDF file; if an incorrect number is entered, d_1 shows another dialog d_2 with an error message. In this example `OpenFileActivity` is the owner activity of m_1, d_1, and d_2. The lifetime of a menu or a dialog is contained within the lifetime of its owner activity.

Events. Each $w \in \mathbf{Win}$ can respond to several events. *Widget events* are of the form $e = [v,t]$ where $v \in \mathbf{View}$ is a widget and t is an event type (e.g., v could be a button and t could be "click"). We also consider five kinds of *default events*. Event $back$ corresponds to pressing the hardware BACK button, which typically (but not always) returns to the window that triggered the current window. Event $rotate$ shows that the user rotates the screen, which triggers various GUI changes. For example, if the currently-active window is a dialog, this dialog is destroyed, its underlying activity is also destroyed, and the activity (but not the dialog) is recreated and redisplayed. Event $home$ abstracts a scenario there the user switches to another application and then resumes the current application (e.g., by pressing the hardware HOME button to switch to the launcher, and then eventually returning to the application.) Event $power$ represents a scenario where the device is put in low-power state by pressing the hardware POWER button, followed by device reactivation. Event $menu$ shows the pressing of the hardware MENU button to display an options menu (or a click to display the hidden parts of an action bar). A default event will be represented as $e = [w,t] \in \mathbf{Win} \times \{back, rotate, home, power, menu\}$ where w is the currently-active window. We will use $\mathbf{Event}$ to denote the set of all widget events and default events.

```
1  class ChooseFileActivity extends Activity
2        implements onItemClickListener {
3   ArrayList<FileListEntry> fileList;
4   ListView fileListView;
5   // === Lifecycle callbacks ===
6   void onCreate() { ...
7     fileListView.setOnItemClickListener(this); }
8   // Other lifecycle callbacks: onDestroy, onStart,
9   // onRestart, onStop, onResume, onPause
10  // === Widget event handler callback ===
11  void onItemClick(ListView l, View item, int p) {
12    FileListEntry entry = fileList.get(p);
13    File file = entry.getFile();
14    if (!file.exists()) return;
15    Intent in = new Intent(OpenFileActivity.class);
16    // initialize intent based on file
17    startActivity(in); } }
18 class OpenFileActivity extends Activity {
19  MenuItem aboutItem, optionsItem;
20  // === Lifecycle callbacks ===
21  // onCreate, onDestroy, etc.
22  void onCreateOptionsMenu(Menu menu) {
23    aboutItem = menu.add("Item");
24    optionsItem = menu.add("Options"); }
25  void onOptionsMenuClosed(Menu menu) { ... }
26  // === Widget event handler callback ===
27  void onOptionsItemSelected(MenuItem item) {
28    if (item == aboutItem)
29       startActivity(new Intent(About.class));
30    if (item == optionsItem) {
31       startActivity(new Intent(Options.class));
32       this.finish(); } }
33 class Options extends Activity
34       implements OnClickListener {
35  Button btn;
36  void onCreate() { btn.setOnClickListener(this); }
37  void onClick(View v) {
38    startActivity(new Intent(About.class));
39    this.finish(); } } }
40 class About extends Activity { ... }
```

Fig. 1. Example derived from the APV PDF reader [26].

Callbacks. Each $e \in \mathbf{Event}$ triggers a sequence of callbacks that can be abstracted as $[o_1,c_1][o_2,c_2]\dots[o_k,c_k]$. Here c_i is a callback method and o_i is a run-time object on which c_i was triggered. We focus on two categories of callbacks. *Widget event handler callbacks* respond to widget events. Figure 1 shows three examples. Method `onItemClick` handles click events for items of list `fileListView`. The call at line 7 registers the activity with a listener for such events. The list, the item being clicked, and its position in the list are provided as parameters to the callback. Method `onOptionsItemSelected` handles clicks for items in the options menu, and takes the clicked item as a parameter. Method `onClick` at lines 37–39 responds to clicks on `btn`.

[1] Such windows are common: for example, in our experiments, more than half of window transitions involved menus and dialogs.

Lifecycle callbacks are used for lifetime management of windows. These methods are of significant interest to developers (e.g., in order to avoid leaks [3], [20]–[22]). There are seven kinds of lifecycle callbacks for activities, as indicated in Figure 1. For example, creation callback `onCreate` indicates the start of the activity's lifetime, and termination callback `onDestroy` indicated end of lifetime. Menus and dialogs can also have create/terminate callbacks, for example, `onCreateOptionsMenu` and `onOptionsMenuClosed` in Figure 1.[2] We will use abstract names $create$ and $destroy$ to represent these create/terminate callbacks. Similarly, $start, \ldots, pause$ will denote corresponding callbacks in activities and (if applicable) in dialogs and menus. Let **Cback** be the set of all lifecycle and widget event handler callbacks.

B. Motivation and Related Work

Section II-C describes the window transition graph (WTG), our proposed static representation of window transitions and callbacks. Each node corresponds to a window and each edge represents a window transition, labeled with a callback sequence. Figure 2 shows the WTG for the running example.

Why this static representation? A number of challenging software engineering problems for Android can be addressed with static analyses where the modeling of control flow plays a critical role. A few examples include checking of security properties (e.g., [7]–[15]), detection of energy defects (e.g., [17]–[19]), leak defects (e.g., [3], [20]–[22]), data races (e.g., [16]), and other correctness checking (e.g., [23], [24]). For example, common battery-drain defects—"no-sleep" [17] and "missing deactivation" [18], [19]—can be stated as properties of callback sequences. These sequences could potentially be derived from WTG paths. Prior work [17] defines a data-flow analysis to identify relevant API calls (e.g., GPS is turned on) and to search for no-sleep paths along which corresponding turn-off/release calls are missing. For this work, the order of callbacks is of critical importance, but their solution lacks generality and precision, and may even involve manual efforts by the user. Some dynamic analyses of energy defects [18], [19] also consider paths in which a sensor (e.g., the GPS) is not put to sleep appropriately, often because of mismanagement of lifecycle callbacks. A static approach to identify such code paths requires callback ordering information, and the WTG can provide this information. Another example is static taint analysis for Android. Representative algorithms such as [12] do not model soundly all callback interleavings and do not employ the control-flow validity constraints captured in our work. Future work could investigate whether such analyses benefit from the WTG representation. Yet another example is static detection of resource leaks. Such leaks are often the result of improper resource management under event/callback sequences [3], [20]–[22], including events such as $rotate$, $home$, and $back$. Developing static leak detectors requires callback sequences, which could be obtained from the WTG.

In addition to defect detection, the WTG is directly applicable for *GUI model construction* for program understanding, testing [1]–[4], and dynamic exploration [5], [6], [28]. In Section IV we describe a test generation tool we developed based on the WTG, using traversals of valid WTG paths.

Related work. Despite the critical importance of analyzing statically the possible GUI behaviors of an Android application, the current state of the art lacks a systematic and comprehensive solution. For example, an activity transition graph is constructed in [5] to guide run-time GUI exploration, but the underlying static analysis [7], [29] uses conservative assumptions about GUI-related control flow, and does not model the changes to the window stack. Other work that creates static GUI models (e.g., [10]) also lacks generality and representations of the window stack. Our earlier work [30] considers analysis of callbacks and determines ordering constraints between them. However, it also does not provide a comprehensive solution: (1) it considers only a limited subset of lifecycle callbacks; (2) it does not represent the interleavings of callbacks from multiple windows, as illustrated in Table I; (3) it does not model the window stack (e.g., it assumes that each $back$ event will return to the previous window); (4) it does not handle the owner-close operations described shortly; (5) it does not consider $rotate$, $home$, and $power$ events. Other work that analyzes possible callbacks in Android (e.g., [12]–[16], [31]) has similar or even more significant limitations. To the best of our knowledge, the proposed static analysis is the first comprehensive solution to the important problem of modeling the possible window/callback sequences in an Android GUI.[3]

C. Modeling of Window Transitions

Opening and closing of windows. Each callback could open a new window or close an existing one. Consider the following scenario: when an "Exit" button in an activity a is clicked, the corresponding event handler opens a new dialog d to ask the user to confirm the exit. When the dialog's "Yes" button b is selected, its handler h closes both the dialog as well as its owner activity a, and control returns back to some prior activity a'. At each event, various callbacks occur. For example, clicking b triggers $[b,h]$ $[d,destroy]$ $[a,pause]$ $[a',restart]$ $[a',start]$ $[a',resume]$ $[a,stop]$ $[a,destroy]$. Our goal is to model statically such behavior and the related changes to the window stack.[4] Note that we focus on the behavior of the main thread (i.e., UI event thread) of the application; analysis of multiple threads (e.g., as done in [16]) or of control flow across applications is not being considered. Additional limitations of the approach are discussed in Section III-D.

There are various API calls to open and close windows. For example, a call to `startActivity` opens a new activity, and a call to `finish` closes an existing one. Similarly, calls to `show` and `dismiss` can create and destroy a dialog. These will be represented with abstract operations $open(w)$ and $close(w)$, where w is the window being created/destroyed. We have never encountered an example of an execution of a callback method c that opens more than one window, and thus we assume that any path through c contains at most one $open(w)$ operation.

Operations $close(w)$ may also be triggered during an execution of c. The two common patterns are *self-close* and *owner-close*. In a self-close, c is associated with a window w and c's execution issues $close(w)$; an example is shown at line 39 of Figure 1. Another example is `onOptionsItemSelected`

[2] There is a related callback `onPrepareOptionsMenu`; for simplicity, it is not discussed here, but our implementation handles it.

[3] Since our approach is tightly coupled with Android-specific semantics, it is unlikely that it will be relevant beyond Android code.

[4] The discussion assumes Android version 4.3; some earlier versions have slight variations in certain sequences of callbacks.

TABLE I. SOME WINDOW STACK CHANGES AND CORRESPONDING CALLBACK SEQUENCES.

	Stack	Event	Handler	Open/Close	Stack changes	Callback sequence
1	$(\ldots, a)$	$[v,t]$	$[v,h]$	none	none	$[v,h]$
2	$(\ldots, a)$	$[v,t]$	$[v,h]$	$\mathit{open}(a')$	push a'	$[v,h][a,\mathit{pause}][a',\mathit{create}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}]$
3	$(\ldots, a', a)$	$[v,t]$	$[v,h]$	$\mathit{close}(a)$	pop a	$[v,h][a,\mathit{pause}][a',\mathit{restart}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}][a,\mathit{destroy}]$
4	$(\ldots, a)$	$[v,t]$	$[v,h]$	$\mathit{close}(a)$, $\mathit{open}(a')$	pop a, push a'	$[v,h][a,\mathit{pause}][a',\mathit{create}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}][a,\mathit{destroy}]$
5	$(\ldots, a', a)$	$[a,\mathit{back}]$	implicit	$\mathit{close}(a)$	pop a	$[a,\mathit{pause}][a',\mathit{restart}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}][a,\mathit{destroy}]$
6	$(\ldots, a)$	$[a,\mathit{rotate}]$	implicit	$\mathit{close}(a)$, $\mathit{open}(a)$	pop a, push a	$[a,\mathit{pause}][a,\mathit{stop}][a,\mathit{destroy}][a,\mathit{create}][a,\mathit{start}][a,\mathit{resume}]$
7	$(\ldots, a)$	$[a,\mathit{home}]$	implicit	none	none	$[a,\mathit{pause}][a,\mathit{stop}][a,\mathit{restart}][a,\mathit{start}][a,\mathit{resume}]$
8	$(\ldots, a)$	$[a,\mathit{power}]$	implicit	none	none	$[a,\mathit{pause}][a,\mathit{stop}][a,\mathit{restart}][a,\mathit{start}][a,\mathit{resume}]$
9	$(\ldots, a)$	$[a,\mathit{menu}]$	implicit	$\mathit{open}(m)$	push m	$[m,\mathit{create}]$
10	$(\ldots, a)$	$[v,t]$	$[v,h]$	$\mathit{open}(m)$	push m	$[v,h][m,\mathit{create}]$
11	$(\ldots, a)$	$[v,t]$	$[v,h]$	$\mathit{open}(d)$	push d	$[v,h][d,\mathit{create}]$
12	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$\mathit{close}(m)$	pop m	$[v,h][m,\mathit{destroy}]$
13	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$\mathit{close}(m)$, $\mathit{open}(a')$	pop m, push a'	$[v,h][m,\mathit{destroy}][a,\mathit{pause}][a',\mathit{create}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}]$
14	$(\ldots, a', a, m)$	$[v,t]$	$[v,h]$	$\mathit{close}(m)$, $\mathit{close}(a)$	pop m, pop a	$[v,h][m,\mathit{destroy}][a,\mathit{pause}][a',\mathit{restart}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}][a,\mathit{destroy}]$
15	$(\ldots, a, m)$	$[v,t]$	$[v,h]$	$\mathit{close}(m\&a)$, $\mathit{open}(a')$	pop $m\&a$, push a'	$[v,h][m,\mathit{destroy}][a,\mathit{pause}][a',\mathit{create}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}][a,\mathit{destroy}]$
16	$(\ldots, a, m)$	$[m,\mathit{back}]$	implicit	$\mathit{close}(m)$	pop m	$[m,\mathit{destroy}]$
17	$(\ldots, a, m)$	$[m,\mathit{rotate}]$	implicit	$\mathit{close}(m\&a)$, $\mathit{open}(a\&m)$	pop $m\&a$, push $a\&m$	$[a,\mathit{pause}][m,\mathit{destroy}][a,\mathit{stop}][a,\mathit{destroy}][a,\mathit{create}][a,\mathit{start}][a,\mathit{resume}][m,\mathit{create}]$
18	$(\ldots, a, m)$	$[m,\mathit{home}]$	implicit	$\mathit{close}(m)$	pop m	$[a,\mathit{pause}][m,\mathit{destroy}][a,\mathit{stop}][a,\mathit{restart}][a,\mathit{start}][a,\mathit{resume}]$
19	$(\ldots, a, d)$	$[v,t]$	$[v,h]$	$\mathit{open}(a')$	push a'	$[v,h][a,\mathit{pause}][a',\mathit{create}][a',\mathit{start}][a',\mathit{resume}][a,\mathit{stop}]$

associated with the options menu m: the semantics of menu-item-click event handlers includes an implicit menu self-close operation $\mathit{close}(m)$ that does not appear in the code. In owner-close operations, if c is associated with a menu m or a dialog d, it may issue $\mathit{close}(a)$ for the owner activity a. For example, the path at lines 30–32 in Figure 1 has an open operation followed by owner-close at line 32 and then an implicit self-close.

Note that the actual opening/closing of windows, as well as the related lifecycle callbacks, happen only after the callback issuing the open/close operations has completed. For example, after lines 30–32 are executed and `onOptionsItemSelected` completes, menu m and its owner a = `OpenFileActivity` are closed, activity a' = `Options` is opened, and the following callbacks are observed: $[m,\mathit{destroy}]$ $[a,\mathit{pause}]$ $[a',\mathit{create}]$ $[a',\mathit{start}]$ $[a',\mathit{resume}]$ $[a,\mathit{stop}]$ $[a,\mathit{destroy}]$. The ordering of open and close operations in a callback's execution path typically does not affect the outcome of its execution.

Behavior of the window stack. The window stack represents the set of currently-alive windows. The window that currently interacts with the user is on top of the stack. Due to space limitations, we describe the case where open and close operations appear only in widget event handler callbacks. Our algorithms and implementation also handle common cases where such operations occasionally appear in lifecycle callbacks.

The window stack starts a single element: the starting activity a. The creation of this initial state is associated with the lifecycle callback sequence to initialize a: $[\mathit{create},a]$ $[\mathit{start},a]$ $[\mathit{resume},a]$. At any moment of time, the window $w \in \mathbf{Win}$ at the top of the stack determines the possible events that could be triggered by the user. These include widget events $[v,t]$ where $v \in \mathbf{View}$ is a widget defined by w and t is the event type, as well as default events such as $[w, \mathit{back}]$, etc. When a widget event $[v,t]$ is triggered, callback $[v,h]$ is invoked. Here $h \in \mathbf{Cback}$ is the corresponding event handling method, invoked on that same widget v. If h triggers a self-close operation, w is popped from the window stack. If, in addition, h triggers an owner-close operation, the owner activity is also popped from the top of the stack. [5] Finally, if h opens a new window, this window is pushed on top of the stack.

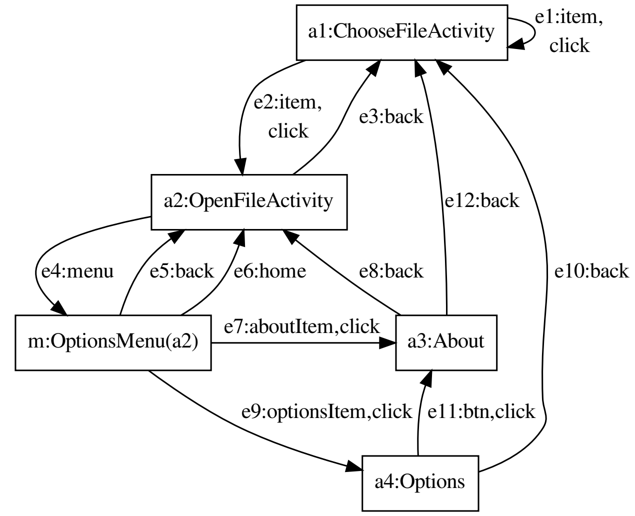
Some of these scenarios are summarized in Table I. The first column describes the stack state, with the currently-visible window on top. We use a and a' to denote activities, m to denote an options menu, and d denote a dialog. Only a representative sample of cases are described; additional details on the scenarios captured by our algorithm are presented elsewhere [32]. In several rows the event handler is listed as "implicit", because it is defined by the Android platform semantics and not by the application code. Column "Open/ Close" shows the window open/close operations triggered by the event handler. The corresponding changes to the window stack are shown in the next column. After these changes are applied, the new stack top becomes the visible window.

The first four rows represent an event for a widget v in an activity a. If the window stack changes (rows 2–4), the callback sequences interleave lifecycle callbacks for a and the activity a' which becomes the new stack top. The implicit handlers for default events also may trigger stack changes: for example, rotating the screen destroys a and then recreates it on top of the stack (row 6). Rows 12–18 present scenarios for an options menu m, owned by an activity a. The widget events $[v,t]$ are of the form $[\mathit{menu\_item},\mathit{click}]$ with handlers h illustrated by `onOptionsItemSelected` in the running example. The implicit $\mathit{close}(m)$ operation in h is explicitly represented in the table. Row 13 corresponds to lines 28–29 in the running example, and row 15 represents the effects of lines 30–32.

Window transition graph. The WTG is defined as $G = (\mathbf{Win}, E, \epsilon, \delta, \sigma)$ with nodes $w \in \mathbf{Win}$ and edges $e \in E \subseteq \mathbf{Win} \times \mathbf{Win}$. Here we use $\mathbf{Win}$ and $\mathbf{View}$ to denote sets of static abstractions of run-time windows and widgets (while previously these sets denoted the actual run-time entities). There are various ways to define such static abstractions. We use the approach from [33], [34], which creates a separate $a \in \mathbf{Win}$ for each activity class, together with appropriate $m, d \in \mathbf{Win}$ for its menus and dialogs, and abstractions $v \in \mathbf{View}$ for their widgets (i.e., defined in layout XML files), and then propagates them similarly to interprocedural points-to analysis, but with special handling of Android API calls.

Labels $\epsilon : E \rightarrow \mathbf{Event}$ indicate that the window transition represented by an edge could be triggered due to a particular event. Labels $\delta : E \rightarrow (\{\mathit{push}, \mathit{pop}\} \times \mathbf{Win})^*$ annotate an

[5] Since the lifetime of a menu/dialog is contained within the lifetime of its owner, closing an owner implies that all owned windows have been closed.



(a) Window transition graph

e	$\delta(e)$	$\sigma(e)$	e	$\delta(e)$	$\sigma(e)$
e_1	–	1	e_7	pop m, push a_3	13
e_2	push a_2	2	e_8	pop a_3	5
e_3	pop a_2	5	e_9	pop m, pop a_2, push a_4	15
e_4	push m	9	e_{10}	pop a_4	5
e_5	pop m	16	e_{11}	pop a_4, push a_3	4
e_6	pop m	18	e_{12}	pop a_3	5

(b) Edge labels

Fig. 2. WTG for the running example.

edge with a sequence of window stack operations $push(w)$ and $pop(w)$. Finally, $\sigma : E \rightarrow ((\mathbf{Win} \cup \mathbf{View}) \times \mathbf{Cback})^*$ shows the sequence of callbacks for the transition.

The meaning of an edge $e = w_1 \rightarrow w_2$ is as follows: suppose that the currently-visible window is w_1 (i.e., it is on top of the window stack). If event $\epsilon(e)$ is issued by the GUI user, the processing of this event may trigger the stack changes described by $\delta(e)$, resulting in a new stack top element w_2. During these changes, the callback sequence $\sigma(e)$ is observed.

Example. Figure 2 shows the WTG for the running example. To simplify the figure, edges $w \rightarrow w$ for *rotate* and *home* events are not shown. Since edges for *power* are very similar to the ones for *home*, they are not shown either. The *back*-event edge from the starting activity a_1, which returns control back to the Android platform, is also not shown. Each e_i is labeled with its triggering event $\epsilon(e_i)$. Edge e_1 represents the case when the PDF file does not exist (line 14 in `onItemClick`) and the event handler returns without opening a new window. The table shows the associated stack changes as well as row numbers from Table I describing the callback sequences $\sigma(e)$.

Two acyclic paths reach a_3: $p = e_2, e_4, e_7$ and $p' = e_2, e_4, e_9, e_{11}$, where p produces a window stack (a_1, a_2, a_3) and p' produces (a_1, a_3). Edges e_8 and e_{12} correspond to possible next edges along p and p', respectively. Note that if e_8 is appended to p', the path is invalid: it represents a stack (a_1), but the end node of the path is a_2, which violates the property that the current window is on top of the window stack. Similarly, e_{12} cannot be appended to p. Our graph construction creates both e_8 and e_{12}, while our subsequent path traversal avoids the infeasible paths p', e_8 and p, e_{12}.

Algorithm 1: ConstructInitialEdges

```
foreach $w \in \mathbf{Win}$ do
    foreach widget event $[v,t]$ with callback $[v,h]$ for $w$ do
        if $MayOpenNone([v,h])$ then
            AddEdge($w, w, [v,t]$)
        foreach $open(w') \in Open([v,h])$ do
            AddEdge($w, w', [v,t]$)
    if $w$ is an activity $a$ with options menu $m$ then
        AddEdge($a, m, [a,menu]$)
    AddEdge($w, w, [w,back]$)
foreach menu and dialog $w \in \mathbf{Win}$ do
    FindOwner($w$)
if $w$ is an activity $a$ then
    AddEdge($a, a, [a,rotate]$)
    AddEdge($a, a, [a,home]$)
    AddEdge($a, a, [a,power]$)
if $w$ is an options menu $m$ with owner $a$ then
    AddEdge($m, m, [m,rotate]$)
    AddEdge($m, a, [m,home]$)
    AddEdge($m, a, [m,power]$)
if $w$ is a context menu $m$ with owner $a$ then
    ...
if $w$ is a dialog $d$ with owner $a$ then
    ...
```

III. WTG Construction Algorithm

The static analysis algorithm to construct the WTG takes as input all $w \in \mathbf{Win}$, $v \in \mathbf{View}$, and, for each w, the possible widget events $[v,t]$ and their corresponding event handler callbacks $[v,h]$. This information is computed by an existing static analysis described in [33], [34]. Given this input, the algorithm proceeds in three stages. In the first stage, initial edges e are constructed and annotated with trigger-event labels $\epsilon(e)$. This stage requires analysis of $open(w)$ operations in event handlers, as well as modeling of default events *rotate*, *home*, *power*, and *menu*. Since $close(w)$ operations are not accounted for in this stage, some of the resulting edges have incorrect target nodes. In the second stage, the initial edges are extended to include push/pop sequences $\delta(e)$ and callback sequences $\sigma(e)$. This requires analysis of self-close and owner-close operations. In the third stage, backward traversal of the graph is used to analyze the push/pop sequences along traversed paths, in order to determine the correct target nodes of edges that could not be resolved earlier.

A. Stage 1: Open-Window Operations and Default Events

In Stage 1, helper function AddEdge(w_1, w_2, ev) represents the addition to the WTG of an edge from window w_1 to window w_2. The edge is labeled with event ev: a widget event $[v,t]$, where v is an widget in w_1, or a default event $[w_1,t]$.

The first stage of the analysis applies Algorithm 1. For each window w, in addition to w's widget events $[v,t]$ and their callbacks $[v,h]$, the algorithm requires two additional properties. The first is a map *Open*, mapping each callback $[v,h]$ to the set of $open(w')$ operations that could be triggered by paths in the callback's execution. The second is a map *MayOpenNone* from $[v,h]$ to a boolean value: true if the callback's execution could complete without triggering any

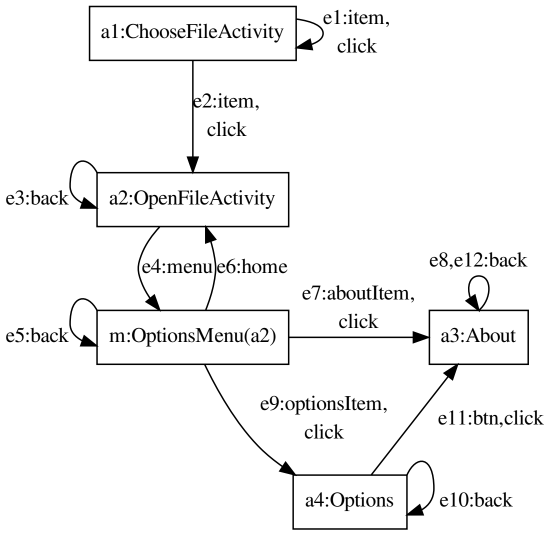


Fig. 3. WTG after Stage 1.

$open(w')$ (i.e., there is an execution path without window-open operations), and false otherwise. Both of these maps can be computed using an approach from [30], in which interprocedural control-flow traversal of h (and its transitive callees) is performed to find calls such as `startActivity`.

Algorithm 1 considers each event and its callback. If $[v,h]$ could be executed without opening a new window, an edge $w \to w$ is created. Edge e_1 in Figure 2 illustrates this case; the edge is created because there is a path in `onItemClick` (through line 14 in Figure 1) for which no windows are created. We will refer to such edges as *no-open* edges. Next, each possibly-opened window w' is considered. At line 6, an edge from w to w' is created for event $[v,t]$. Line 8 handles default event $menu$ for activities. The edges created at lines 6 and 8 push a new window on top of the window stack, and will be referred to as *window-open* edges.

At line 9, initial edges for $back$-button events are created. The targets of these edges (as well as their callback sequences) will not be known until Stage 3. Next, for each menu and dialog w, its owner activity is determined by traversing backward the newly-created window-open edges, using helper function FINDOWNER.[6] Finally, default events $rotate$, $home$, and $power$ are handled. This handling is consistent with the description in Table I. The cases for context menus and dialogs are not shown, but they are similar to those for options menus.

Example. Figure 3 shows the WTG for the running example after Stage 1 has completed. The edge numbering is the same as in the final WTG from Figure 2. Similarly to that earlier figure, certain $rotate$, $home$, and $power$ edges are not shown for simplicity. Edge e_1 is created because $MayOpenNone$ is true for the corresponding event handler, while e_2 shows that this handler could open a_2. The owner of m is a_2, and the $home$ edge for m reflects that. The $back$-event edges have incorrect targets that will be fixed later. The $back$-event edge for a_3 is labeled as e_8, e_{12} since eventually it will lead to the creation of two separate edges e_8 and e_{12}.

B. Stage 2: Close-Window Operations

In this stage the analysis first considers each edge e for a widget event $[v,t]$ and handler $[v,h]$. Using the interprocedural control-flow reachability analysis from [30], h under calling context v is analyzed for self-close operations (e.g., calls to `finish`) and e is classified in one of three disjoint categories: must-not-self-close, may-self-close, and must-self-close. If h under context v does not contain a path reaching a self-close operation, e is in the first category. If some but not all paths reach a self-close, the second category applies. If every path reaches a self-close, the edge is must-self-close.

In a similar manner, classification is performed for owner-close operations. The analysis considers each menu and dialog w and w's owner activity a. For an edge $e = w \to \ldots$ for a widget event $[v,t]$, we can classify e as must-not-close-owner, may-close-owner, and must-close-owner.

Example. In Figure 3, e_7 and e_9 are must-self-close due to the implicit $close(m)$ in `onOptionsItemSelected`. Edge e_{11} is also must-self-close due to the call to `finish` at line 39 in the running example. (If, hypothetically, this call were guarded by a conditional, the classification would have been may-self-close.) The other two widget event edges e_1 and e_2 are must-not-self-close. For owner-close operations, e_7 is must-not-close-owner, while e_9 is must-close-owner, since under widget context `optionsItem` the handler definitely closes the owner activity a_2 (line 32 in the running example).

This classification is used to create push/pop labels $\delta(e)$ for the analyzed edges. For example, e_9 opens a_4 while definitely closing m and its owner a_2; thus, $\delta(e_9) = pop\ m,\ pop\ a_2,\ push\ a_4$. Algorithm 2 provides some details on this process. One important observations is that a single edge created by Stage 1 may be expanded into several edges, with different $\delta(e)$ labels. For example, if (hypothetically) e_{11} were may-self-close, it would expand to two edges from a_4 to a_3, one labeled with $push\ a_3$ (line 7 in the algorithm) and the other with $pop\ a_4,\ push\ a_3$ (line 5 in the algorithm). Helper function EXPANDEDGE(e, d) takes an edge e created by Stage 1 and constructs an "expanded" version of it with $\delta(e) = d$. After Stage 2, the edges from Stage 1 are discarded.

Some details of the processing are elided due to space limitations. For example, the handling of dialogs is similar to that of menus, but with the additional possibility that a self-close operation is not executed. The handling of $rotate$, $home$, and $power$ events is consistent with the push/pop sequences listed in Table I, and is not shown in Algorithm 2. After the algorithm completes, all edges have labels $\delta(e)$. The labels created for the running example are shown in Figure 2b. At this point, there is still a single $back$-event edge for a_3 (labeled with $pop\ a_3$); Stage 3 creates two separate edges from it.

Certain edges have incorrect targets and have to be processed by Stage 3. These edges do not open new windows, but close existing ones: namely, (1) edges for $back$ events, and (2) no-open edges that contain close operations. In both cases, the top of the stack after executing the edge is some (yet) unknown previously-opened window. The rest of the edges have correct target nodes and their callback sequences $\sigma(e)$ can be determined at this time, using the Android semantic specification illustrated by Table I. For edges $e_1, e_2, e_4, e_6, e_7, e_9, e_{11}$ from Figure 3, the callback sequences computed by Stage 2 are

[6] In general, w could have multiple owners, e.g., due to subclassing of activities; the necessary algorithmic generalizations are straightforward.

Algorithm 2: ExpandEdgesWithLabels

```
foreach $w \in \mathbf{Win}$ do
    if $w$ is an activity $a$ then
        foreach window-open edge $e = a \to w'$ do
            if $e$ is may/must-self-close then
                ExpandEdge($e, [pop\ a, push\ w']$)
            if $e$ is not must-self-close then
                ExpandEdge($e, [push\ w']$)
        foreach no-open edge $e = a \to a$ do
            if $e$ is may/must-self-close then
                ExpandEdge($e, [pop\ a]$)
            if $e$ is not must-self-close then
                ExpandEdge($e, [\ ]$)
        if exists $e = a \to m$ for default event $[w, menu]$ then
            ExpandEdge($e, [push\ m]$)
    if $w$ is a menu $m$ with owner $a$ then
        foreach window-open edge $e = m \to w'$ do
            if $e$ is may/must-owner-close then
                ExpandEdge($e, [pop\ m, pop\ a, push\ w']$)
            if $e$ is not must-owner-close then
                ExpandEdge($e, [pop\ m, push\ w']$)
        foreach no-open edge $e = m \to m$ do
            if $e$ is may/must-owner-close then
                ExpandEdge($e, [pop\ m, pop\ a]$)
            if $e$ is not must-owner-close then
                ExpandEdge($e, [pop\ m]$)
    if $w$ is a dialog $d$ with owner $a$ then
        ...
foreach edge $w \to w$ for default event $[w, back]$ do
    ExpandEdge($e, [pop\ w]$)
```

listed in Figure 2b. The rest of the edges in Figure 3 have incorrect target nodes, and since $\sigma(e)$ depends on the target of e, their callback sequences cannot yet be determined.

C. Stage 3: Backward Analysis of the Window Stack

Edges with incorrect targets require further processing. They are of the form $e = w \to w$, with labels $\delta(e)$ containing no *push* but at least one *pop*. To identify the correct target of e, Stage 3 performs a backward traversal from w, using correct edges finalized in Stage 2, to examine all paths ending at w. This traversal is parameterized by a value k, which defines the largest number of edges along any path being considered.[7] For each such path $e_1, e_2, \ldots, e_n$, where $n \leq k$ and the target node of e_n is w, we need to consider the sequence of push/pop operations $\delta(e_1), \delta(e_2), \ldots, \delta(e_n), \delta(e)$ and to decide (1) whether this sequence represents valid run-time behavior, and (2) what could be the top of the window stack after the sequence is executed.

Example. Suppose that $k = 2$ and we consider $e_5 = m \to m$ in Figure 3, labeled with $pop\ m$. Two paths ending at m need to be examined: e_2, e_4 and e_6, e_4. The edge labels for the first path (including e_5's label) are $push\ a_2$, $push\ m$, $pop\ m$. This is a feasible sequence whose execution is guaranteed to leave a_2 as the top of the stack. Thus, e_5 should have a_2 as a target, and the analysis creates this corrected edge. For the second path, the edge labels (including e_5) are $pop\ m$, $push\ m$, $pop\ m$. Although this is a feasible sequence, it does not provide enough information to decide what would be the top of the stack after executing these operations, and the analysis does not create any edges due to this path.

As another example, consider edge $e_{10} = a_4 \to a_4$. For $k = 4$, the relevant path is e_0, e_2, e_4, e_9. Here e_0 is an implicit edge entering a_1, labeled with $push\ a_1$; this edge represents the triggering of the start activity a_1 by the Android platform. The sequence for $e_0, e_2, e_4, e_9, e_{10}$ is $push\ a_1$, $push\ a_2$, $push\ m$, $pop\ m$, $pop\ a_2$, $push\ a_4$, $pop\ a_4$. This sequence leaves a_1 as the top of the stack. Thus, e_{10} should be redirected to a_1 (as shown in the graph in Figure 2).

As a final example, consider *back*-event edge $a_3 \to a_3$. Path e_2, e_4, e_7, with this edge appended, has the sequence $push\ a_2$, $push\ m$, $pop\ m$, $push\ a_3$, $pop\ a_3$. Thus, this *back*-event edge should have a_2 as target. In the final graph from Figure 2, e_8 is this redirected edge. Another relevant path is $e_0, e_2, e_4, e_9, e_{11}$; the sequence along the path, appended with the *back*-event edge, is $push\ a_1$, $push\ a_2$, $push\ m$, $pop\ m$, $pop\ a_2$, $push\ a_4$, $pop\ a_4$, $push\ a_3$, $pop\ a_3$, which leaves a_1 as the top of the stack. In this case an edge from a_3 to a_1 needs to be introduced (e_{12} from Figure 2).

Stage 3 analyzes an edge $e = w \to w$ as follows. A stack containing *push* and *pop* operations is maintained. The stack is initialized with the reverse of $\delta(e)$; for all examples from above, this is an operation $pop\ w$. Backward traversal from w is performed, limiting path length to at most k edges. When an edge e_i is encountered during the traversal, the reverse of its $\delta(e_i)$ sequence is used to update the stack. If $pop\ w'$ is seen, it is just added on top of the stack. If $push\ w'$ is encountered and the stack is not empty, the top of the stack must be $pop\ w'$ (otherwise the path is infeasible and is ignored) and $pop\ w'$ is removed from the stack. If $push\ w'$ is observed when the stack is empty, the traversal stops and w' is identifies as a possible target, leading to a new edge $w \to w'$.

Example. Consider edge $e_{10} = a_4 \to a_4$. Starting from a stack containing $pop\ a_4$, edges e_9, e_4, e_2, e_0 are visited to produce the following sequence: $push\ a_4$, $pop\ a_2$, $pop\ m$, $push\ m$, $push\ a_2$, $push\ a_1$. Operations $push\ a_4$ and $push\ a_2$ empty the stack. Since $push\ a_1$ occurs for an empty stack, edge e_{10} becomes $a_4 \to a_1$.

D. Limitations

The algorithm and its implementation have several limitations. As discussed earlier, control flow due to multiple threads or across multiple applications is not modeled. The modeling of GUI widgets and event handlers [33] captures many commonly-used Android widgets, but is not fully comprehensive. Furthermore, custom window/widget systems cannot be handled. Asynchronous transitions (e.g., due to timers and sensor events) are not represented in the WTG. The interprocedural intent analysis used to resolve $open(w)$ calls [30] considers only explicit intents, as they are designed for use inside the same application [35]. More general intent analyses (e.g., [11], [29], [36]) could be used instead. Our analysis also does not model the different launch modes for activities [25]. Due to these limitations, some window transitions are missing:

[7] An alternative would be to traverse all acyclic paths, without a length limit.

for example, for the 20 apps used in our evaluation, on average 13% of the WTG nodes have no incoming edges. While most of these limitations are orthogonal to the contributions of this paper, they emphasize the need to advance the state of the art in static analysis for Android, and in particular the comprehensive modeling of Android-specific control flow and data flow.

E. Path Validity

The analysis outlined in the previous sections does not ensure that each path represents a feasible run-time execution. Consider again the final WTG (after Stage 3) shown in Figure 2. Paths $p = e_0, e_2, e_4, e_7$ and $p' = e_0, e_2, e_4, e_9, e_{11}$ both reach node a_3. However, p cannot be extended with edge e_{12} because the corresponding edge labels would be *push* a_1, *push* a_2, *push* m, *pop* m, *push* a_3, *pop* a_3. This leaves a_2 as the top of the window stack, while the target node of e_{12} is a_1. Similarly, if p' were extended with e_8, the top of the stack would be a_1 while the target of e_8 is a_2.

The WTG can be augmented with a path validity check, which "simulates" the window stack along a given path of interest, and decides whether the path is valid. This is similar in spirit to classical interprocedural analyses, where the sequence of calls and returns along a path is used to simulate the call stack, in order to decide path validity [37]. A WTG edge may correspond to several push/pop operations, but the validity of these operations is still based on the same style of push/pop matching as in traditional analyses. As discussed in the next section, one use of this validity check is during test generation, to avoid the creation of unexecutable test cases. Path validity checks may also be needed for static checking of correctness properties, in order to avoid analyzing infeasible paths that lead to false positives.

IV. Test Generation

One possible application of the WTG is for model-based test generation (e.g., [1]–[4], [19]). To illustrate this use of the WTG, we developed a prototype test generation tool. The tool traverses certain WTG paths and for each path creates a test case implemented with the Robotium testing framework [38]. For a path $p = e_1, e_2, \ldots$, the event label $\epsilon(e_i)$ is translated to corresponding Robotium API calls to trigger the event. Some events may require additional input from the tester—e.g., to decide which item in a list to click. Since the static analysis solution is conservative, it is possible that event $\epsilon(e_i)$ may not be feasible at run time, or even if it is feasible, the target window of e_i after the run-time event is not as expected. Each test case includes run-time checks to detect such scenarios and report the test case as infeasible.

One can consider various test generation schemes (e.g., leak testing in [3] considers neutral-effect cycles in a manually-constructed model). In our proof-of-concept tool, we use a simple path-based approach. Starting from the implicit edge e_0 showing the invocation of the start activity, we append m distinct edges to create a path $p = e_0, e_1, \ldots, e_m$. A naive approach is to simply explore all such paths. A more precise approach is to apply the validity check from Section III-E each time the path is extended with a new edge. The next section shows that this validity check, which is based on our proposed tracking the push/pop sequences, can reduce substantially the number of test cases being generated.

V. Experimental Evaluation

The WTG was constructed for the 20 open-source applications used in our prior work [30], [33]. The first goal of the evaluation is to characterize the effects of different stages of the algorithm, as well as its overall cost. The second goal is to evaluate precision, relative to a manually-constructed model. The third goal is to evaluate precision for the test generation from Section IV. The implementation is available as part of our public analysis toolkit [39].

A. Algorithm for Building the WTG

Table II provides measurements of the number of WTG nodes and edges. Column "Stage 1" shows the number of edges before considering any close-window operations (Algorithm 1). After Stage 2, the edges are expanded with push/pop sequences, based on analysis of close-window effects. Column $\Delta_{1,2}$ shows the increase due to this expansion. One can observe that an edge from Stage 1 can often have several possible push/pop sequences. This indicates that an event handler may exhibit a variety of behaviors. Our analysis discovers such variations and represents them with separate edges (Algorithm 2). We are not aware of any existing work that performs such detailed analysis of Android event handlers.

The large number of edges for `FBReader` and `XBMC` is caused by a known limitation of our prior analyses [30], [33]: both analyses use a context-insensitive call graph based on class hierarchy analysis. For example, in `FBReader`, two utility methods are responsible for over 96% of the WTG edges. Both methods take parameters of an interface type which is implemented by 130 classes. Class hierarchy resolution for calls on these parameters is highly imprecise. More precise call graph construction is likely to solve this problem.

Recall that some of the Stage 2 edges have incorrect target nodes. Column "Stage 3" shows the number of edges after the correct targets have been determined. This is achieved with backward path analysis, based on a parameter k for path length; the column contains measurements for $k = 4$. Column $\Delta_{2,3}$ shows the size of the difference (number of edges removed and added) between the edge sets from Stage 2 and Stage 3. The backward path traversal, combined with tracking of feasible push/pop sequences along the path (Section III-C), results in significant changes to the graph. The next four columns show the effects of increasing the path length limit k. In general, newly-created edges require backward traversals of non-trivial length. Thus, one cannot consider just the edges entering a node w to determine the targets of Stage 3 edges $w \rightarrow \ldots$; rather, paths of length k reaching w must be examined. To the best of our knowledge, ours is the first approach to perform such static modeling of possible transitions in Android GUIs. For most programs, the graph stabilizes at $k = 4$; for the rest, slightly larger values of k (not shown here) are needed.

The last column shows the running time of the analysis in seconds. This measurement includes the time for the event handler analysis from [30], which is invoked on-demand inside our analysis. Overall, the running times are suitable for practical use, even though we have not made any significant effort to optimize the implementation. However, as indicated by the results for `FBReader`, scalability limitations could be encountered for large WTGs.

TABLE II. WTG CONSTRUCTION ALGORITHM: NUMBER OF NODES/EDGES AND ANALYSIS COST.

Application	SLOC	Nodes	Edges					k				Time
			Stage 1	Stage 2	$\Delta_{1,2}$	Stage 3	$\Delta_{2,3}$	$k=1$	$k=2$	$k=3$	$k=4$	(sec)
APV	3832	14	77	101	24	105	58	75	95	104	105	5
Astrid	24487	93	594	740	146	838	236	675	836	838	838	18
BarcodeScanner	6549	20	90	121	31	128	65	98	118	128	128	6
Beem	12962	24	99	125	26	132	65	100	118	132	132	6
ConnectBot	32638	37	185	233	48	237	112	182	211	234	237	8
FBReader	45510	45	286	17774	17488	41942	26326	29473	39820	41941	41942	2086
K9	52240	55	258	411	153	516	221	433	486	509	516	25
KeePassDroid	27457	41	272	468	196	643	389	399	598	640	643	9
Mileage	9881	75	409	562	153	676	268	485	636	676	676	7
MyTracks	23389	61	212	314	102	391	197	294	363	391	391	7
NotePad	4986	22	122	191	69	213	110	162	195	213	213	6
NPR	12118	32	344	502	158	590	106	525	590	590	590	6
OpenManager	2562	18	95	116	21	116	64	84	113	116	116	5
OpenSudoku	6079	35	173	232	59	237	125	180	208	237	237	6
SipDroid	24533	31	176	305	129	406	331	226	364	396	406	12
SuperGenPass	2119	9	49	63	14	64	39	45	58	64	64	5
TippyTipper	1739	10	54	63	9	65	16	56	61	65	65	5
VLC	10670	26	117	130	13	131	45	112	131	131	131	6
VuDroid	2380	7	30	44	14	47	23	34	41	47	47	4
XBMC	23295	67	1080	3819	2739	4279	722	3690	4241	4278	4279	16

TABLE III. FEASIBILITY OF WTG EDGES.

Application	WTG	Manual	Infeasible
APV	105	105	0
BarcodeScanner	128	106	22
OpenManager	116	109	7
SuperGenPass	64	64	0
TippyTipper	65	65	0
VuDroid	47	45	2

B. Manual Examination of WTGs

For in-depth evaluation of analysis precision, we examined the WTG ($k = 4$) for APV, BarcodeScanner, OpenManager, SuperGenPass, TippyTipper, and VuDroid. These applications had the smallest numbers of WTG nodes, and thus could be examined manually with reasonable effort.

Column "WTG" in Table III replicates the Stage 3 measurements from Table II. Column "Manual" shows the number of WTG edges that were manually confirmed to be feasible using run-time test cases. The last column contains the number of infeasible WTG edges. The infeasibility was asserted by examining the source code. In general, the number of infeasible edges is small (around 6% across the six applications). We determined the root causes of all infeasible edges. In all cases, the infeasibility was due to deficiencies in the earlier work on window/widget modeling [33], [34] and event handler analysis [30]. If these existing static analyses were to be improved, the WTG would achieve perfect precision. These results highlights the need for continued advances in static analysis of GUI structure and behavior for Android applications. Still, the small number of infeasible edges is a positive indicator that highly-precise static GUI models can be constructed automatically.

C. Test Generation

Recall that our prototype test generator considers paths $p = e_0, e_1, \ldots, e_m$ (all e_i are distinct) and generates test cases from them. Here e_0 represents the invocation of the start activity by the Android platform. The numbers of paths for $m = 2$ and $m = 3$ are shown in Table IV. Columns "All" contain the number of all paths, while columns Δ show the reduction (in percent) when the path validity check from Section III-E is applied. For FBReader the number of paths with $m = 3$ was too large enumerate in reasonable time.

For several applications the path validity check reduces the number of test cases. For example, for $m = 3$ (i.e., test cases containing three GUI events), 10 applications show reductions of 26% or more. Such reductions indicate that statically we can eliminate significant numbers of infeasible test cases.

Of course, even if a path satisfies the static validity condition, it could still result in an infeasible test case. As indicated earlier, due to deficiencies in prior static analyses, some WTG edges (and thus paths) may be infeasible. To understand better this infeasibility, for the six applications studied in Section V-B we generated test cases from the statically-feasible paths for $m = 2$. Although the sequences of events (implemented through Robotium [38] API calls) are generated automatically, some test cases still require manual effort: for example, for BarcodeScanner, we need to manually set up a variety of actual barcode images to drive the different test cases. Due to this manual effort, we did not consider larger values of m.

The number of test cases (with path validity) is shown in column "Static" in Table V. We set up and executed all 1581 test cases indicated in this column. The next column "Feasible" shows the number of these test cases that were feasible at run time—that is, they could match the event sequence and target windows of the static path. In BarcodeScanner, the event handler analysis from [30] leads to infeasible edges that make most of the test cases infeasible. As described in [30], the application processes eleven types of barcodes, and the GUI behavior (subset of visible widgets and subset of handler effects) differs based on the barcode type. This variability cannot easily be modeled statically. In OpenManager, the 6.5% of infeasible test cases are due to inter-application interactions. When the main activity is invoked by another application (rather than by the user), that activity computes information about a file, returns it to the invoking application, and closes itself. Our analysis does not model the interactions between multiple applications and does not recognize that the activity-close operation happens under these conditions. Overall, with the exception of one application, the vast majority of statically-generated test cases are feasible at run time.

TABLE IV. NUMBER OF PATHS FOR TEST GENERATION.

Application	$m = 2$		$m = 3$	
	All	Δ (%)	All	Δ (%)
APV	116	24.1	1416	37.7
Astrid	232	62.1	1822	75.3
BarcodeScanner	526	1.9	7675	4.6
Beem	138	26.1	929	38.6
ConnectBot	287	26.1	3384	40.3
FBReader	33404638	84.9	N/A	N/A
K9	12393	19.8	443647	27.0
KeePassDroid	20	0.0	48	0.0
Mileage	16	0.0	45	0.0
MyTracks	1331	9.4	35212	20.5
NotePad	217	17.5	2625	26.0
NPR	4171	21.0	251251	30.5
OpenManager	392	0.8	5803	1.5
OpenSudoku	111	23.4	980	33.3
SipDroid	905	32.9	13604	51.6
SuperGenPass	195	0.0	2110	0.0
TippyTipper	341	0.0	5405	0.0
VLC	42	0.0	131	0.0
VuDroid	52	0.0	276	0.0
XBMC	5728	62.3	1330605	71.0

TABLE V. RUN-TIME FEASIBILITY OF GENERATED TEST CASES.

Application	Static	Feasible
APV	88	88
BarcodeScanner	516	88
OpenManager	389	364
SuperGenPass	195	195
TippyTipper	341	341
VuDroid	52	52

Summary. Columns $\Delta_{1,2}$ and $\Delta_{2,3}$ of Table II indicate that event handlers can have complex behaviors and their transitions depend on non-trivial sequences of preceding events. Our analysis is the first to model these features, leading to improved static GUI models and test case generation. For six applications, manual comparison with run-time behavior indicates that the analysis achieves good precision.

VI. RELATED WORK

Control-flow analysis for Android. The control/data flow of Android applications is driven by the GUI and static analysis of this flow is an important problem. One of the first related efforts is the SCanDroid analysis tool [7], [29] which employs control-flow analysis and intent analysis in the context of a security analysis. Later work on related security problems [8], [9], [11] also uses intent analysis and control-flow analysis. These techniques do not attempt detailed analysis of GUI-related control/data flow due to widgets and event handlers.

An activity transition graph, used for run-time GUI exploration [5] and based on [7], [29], has some similarities to our WTG. This representation does not capture menus/dialogs, does not consider the general GUI effects of event handlers (e.g., window-close) and the triggered callbacks, and does not model the window stack and its state changes. A similar model of activity transitions is used in an analysis of security-sensitive behaviors [10]. This approach is much less comprehensive than ours, in terms of both the model and the analysis algorithm.

The taint analysis in FlowDroid [12] models the effects of callbacks by creating an artificial main method. The flow of control in this method encodes possible sequences of callbacks, but does not account for the general GUI effects of event handlers, and does not represent control flow that spans multiple activities. This approach cannot capture the callback sequences described in Table I. Control flow involving dialogs, menus, and window termination is also not captured. A more comprehensive solution is available from our prior work [30], where ordering constraints between callbacks are represented by a callback control-flow graph. This work provides event handler analysis for our WTG analysis (to detect window open/close effects of callbacks). However, this earlier approach does not represent the window stack, the push/pop sequences at transitions, or the path feasibility based on these sequences. It also ignores several categories of lifecycle callbacks, cannot represent correctly the callback interleavings from Table I, does not handle operations that close the owner, and does not represent the effects of $rotate$, $home$, and $power$.

An existing operational semantics for activities [40] captures aspects of Android control flow, including callbacks and the activity stack, but does not define GUI static models or analysis algorithms. Various other static analyses aim to model the sequences of callbacks in Android, in the context of security analysis (e.g., [13]–[15], [31]), GUI model construction (e.g., [28]), race detection (e.g., [16]), leak analysis (e.g., [17], [22]), and static checking (e.g., [23], [24]). None of these techniques provide comprehensive behavior definition/analysis for the key aspects of GUI behavior: widgets, event handlers, callback sequences, and window stack changes. We develop a more general approach for static analysis and representation of Android GUI behavior, which provides a promising starting point for generalizing existing (and future) static analyses.

GUI models for understanding and testing. Reverse engineering of GUI models is well developed in prior work (e.g., [41]–[43]) and has more recently been used for Android (e.g., [5], [6], [28], [44], [45]). The models are usually constructed through dynamic exploration. As results from [30] indicate, a static approach could produce more comprehensive models—of course, at the expense of potential infeasibility. For the purposes of program understanding, hybrid static/dynamic techniques are the most promising. Existing examples of such techniques [5], [28] may benefit from our static GUI models, including the path validity check which could be beneficial for dynamic GUI crawling. Finite state machines and similar models for GUI testing have been used widely (e.g., [1]–[4], [42], [46]–[49]), and various test generation schemes can be employed (e.g., [47]). Techniques have been proposed to improve the generated test cases (e.g., [50]–[52]) and it may be possible to integrate them with WTG-based test generation.

VII. CONCLUSIONS

A representation of window/callback sequences is a foundation for static analyses for Android. One can draw an analogy with the interprocedural control-flow graph [37], a key representation for traditional static analysis. We propose the WTG as a similarly-important static model for Android, and develop algorithms for its construction and traversal. In the future, it is important to generalize this work to handle more comprehensive control flow in Android applications.

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

REFERENCES

[1] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an Android application," in *ICST*, 2011, pp. 377–386.

[2] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *ISSTA*, 2013, pp. 67–77.

[3] D. Yan, S. Yang, and A. Rountev, "Systematic testing for resource leaks in Android applications," in *ISSRE*, 2013, pp. 411–420.

[4] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in Android applications," in *MOBS*, 2013, pp. 1–6.

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

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

[7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated security certification of Android applications," University of Maryland, College Park, Tech. Rep. CS-TR-4991, 2009.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *MobiSys*, 2011, pp. 239–252.

[9] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *NDSS*, 2012.

[10] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An automatic system for revealing UI-based trigger conditions in Android applications," in *SPSM*, 2012, pp. 93–104.

[11] 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.

[12] 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.

[13] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *CCS*, 2012, pp. 229–240.

[14] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in *ICSE*, 2014, pp. 1036–1046.

[15] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of Android malware through static analysis," in *FSE*, 2014, pp. 576–587.

[16] Y. Lin, C. Radoi, and D. Dig, "Retrofitting concurrency for Android applications through refactoring," in *FSE*, 2014, pp. 341–352.

[17] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?" in *MobiSys*, 2012, pp. 267–280.

[18] Y. Liu, C. Xu, S. C. Cheung, and J. Lu, "GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications," *TSE*, vol. 40, pp. 911–940, Sep. 2014.

[19] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, "Detecting energy bugs and hotspots in mobile apps," in *FSE*, 2014, pp. 588–598.

[20] "Stopping and restarting an activity," `developer.android.com/training/basics/activity-lifecycle/stopping.html`.

[21] P. Dubroy, "Memory management for Android applications," in *Google I/O Developers Conference*, 2011.

[22] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in Android applications," in *ASE*, 2013, pp. 389–398.

[23] S. Zhang, H. Lü, and M. D. Ernst, "Finding errors in multithreaded GUI applications," in *ISSTA*, 2012, pp. 243–253.

[24] E. Payet and F. Spoto, "Static analysis of Android programs," *IST*, vol. 54, no. 11, pp. 1192–1201, 2012.

[25] "Tasks and back stack," `http://developer.android.com/guide/components/tasks-and-back-stack.html`.

[26] "APV PDF viewer," `code.google.com/p/apv`.

[27] "Android dialogs," `http://developer.android.com/guide/topics/ui/dialogs.html`.

[28] 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.

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

[30] 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.

[31] S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. Van Horn, "Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation," in *SPSM*, 2013, pp. 21–32.

[32] S. Yang, "Static analyses of GUI behavior in Android applications," Ph.D. dissertation, Ohio State University, 2015.

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

[34] 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.

[35] "Intents and intent filters," `developer.android.com/guide/components/intents-filters.html`.

[36] 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.

[37] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," in *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice Hall, 1981, pp. 189–234.

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

[39] "GATOR: Program Analysis Toolkit For Android," `web.cse.ohio-state.edu/presto/software/gator`.

[40] E. Payet and F. Spoto, "An operational semantics for Android activities," in *PEPM*, 2014, pp. 121–132.

[41] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *WCRE*, 2003, pp. 260–269.

[42] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software," *TSE*, vol. 31, no. 10, pp. 884–896, 2005.

[43] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *ISSTA*, 2012, pp. 67–77.

[44] P. Tramontana, "Android GUI Ripper," `wpage.unina.it/ptramont/GUIRipperWiki.htm`.

[45] P. Wang, B. Liang, W. You, J. Li, and W. Shi, "Automatic Android GUI traversal with high coverage," in *CSNT*, 2014, pp. 1161 – 1166.

[46] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *ISSRE*, 2000, pp. 110–121.

[47] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *FSE*, 2001, pp. 256–267.

[48] A. M. Memon, "An event-flow model of GUI-based applications for testing," *STVR*, vol. 17, no. 3, pp. 137–157, 2007.

[49] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *TOSEM*, vol. 18, no. 2, pp. 7:1–7:35, 2008.

[50] X. Yuan and A. M. Memon, "Generating event sequence-based test cases using GUI run-time state feedback," *TSE*, vol. 36, no. 1, pp. 81–95, 2010.

[51] X. Yuan, M. B. Cohen, and A. M. Memon, "GUI interaction testing: Incorporating event context," *TSE*, vol. 37, no. 4, pp. 559–574, 2011.

[52] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon, "Lightweight static analysis for GUI testing," in *ISSRE*, 2012, pp. 301–310.