Barista: A Technique for Recording, Encoding, and Running Platform Independent Android Tests

Mattia Fazzini*, Eduardo Noronha de A. Freitas†, Shauvik Roy Choudhary* and Alessandro Orso*
*Georgia Institute of Technology †Instituto Federal de Goiás
{mfazzini | shauvik | orso}@cc.gatech.edu efreitas@ifg.edu.br

***Abstract*—Because mobile apps are extremely popular and often mission critical nowadays, companies invest a great deal of resources in testing the apps they provide to their customers. Testing is particularly important for Android apps, which must run on a multitude of devices and operating system versions. Unfortunately, as we confirmed in many interviews with quality assurance professionals, app testing is today a very human intensive, and therefore tedious and error prone, activity. To address this problem, and better support testing of Android apps, we propose a new technique that allows testers to easily create platform independent test scripts for an app and automatically run the generated test scripts on multiple devices and operating system versions. The technique does so without modifying the app under test or the runtime system, by (1) intercepting the interactions of the tester with the app and (2) providing the tester with an intuitive way to specify expected results that it then encode as test oracles. We implemented our technique in a tool named BARISTA and used the tool to evaluate the practical usefulness and applicability of our approach. Our results show that BARISTA (1) can faithfully encode user defined test cases as test scripts with built-in oracles that can run on multiple platforms and (2) outperforms two popular tools with similar functionality. BARISTA and our experimental infrastructure are publicly available.**

I. INTRODUCTION

Mobile platforms are becoming increasingly prevalent, and so are the mobile applications (or simply apps) that run on such platforms. Today, we use apps for many of our daily activities, such as shopping, banking, social networking, and traveling. Like all other software applications, apps must be tested to gain confidence that they behave correctly under different inputs and conditions. This is especially important nowadays, given the number of companies that make apps available to their users, as failures in an app can result in loss of reputation, and ultimately customers, for the company that provides the app. For this reason, companies are spending considerable amounts of money and resources on quality assurance (QA) activities, and in particular on testing.

In the case of Android apps, the picture is further complicated by the fragmentation of the Android ecosystem [1], which includes countless devices that come in all shapes and sizes and that can run a number of different versions of the Android system. Gaining confidence that an app works correctly across the whole range of Android devices and operating system versions is especially challenging and expensive.

To help QA testers in this difficult task, we propose a new technique for supporting testing of Android apps that has three main capabilities. *First*, it allows testers to interact with an app and (1) record the actions they perform on the app, and (2) specify the expected results of such actions using a new, intuitive mechanism. *Second*, it automatically encodes the recorded actions and specified expected results in the form of a general, platform-independent test script. *Third*, it allows for automatically running the generated test scripts on any platform (*i.e.,* device and operating system), either on a physical device or in an emulator.

In addition, there are several advantages to our approach, compared to the state of the art. One advantage is that our approach implements the record once-run everywhere principle. Testers can record their tests on one platform and ideally rerun them on any other platform. Existing approaches focused on GUI test automation through record/replay [2], [3] tend to generate tests that are brittle and break when run on platforms other than the one on which they were recorded, as confirmed by our empirical evaluation (Section V). A second advantage of our approach is that it supports the creation of oracles, and it does it in an intuitive way, whereas most existing approaches have very limited support for this aspect [2]–[6]. In general, our approach can be used with very limited training, as it does not require any special skill or knowledge. A third advantage is that, because of the way our approach encodes test cases, these tests tend to be robust in the face of (some) changes in the user interface of the app (and are unaffected by changes that do not modify the user interface). The test cases generated by our approach can therefore also be used for regression testing. From a more practical standpoint, a further advantage of our approach is that it encodes test cases in a standard format—the one used in the Espresso framework, in our current implementation. These test cases can therefore be run as standalone tests. A final, also practical and advantage of our approach is that it is minimally intrusive. Because it leverages accessibility mechanisms already present on the Android platform [7], our approach does not need to instrument the *apps under test* (*AUTs*). To use the approach, testers only have to install an app on the device on which they want to record their tests, enable the accessibility framework for it, and start recording.

Our technique offers these advantages while handling several practical challenges specific to the Android framework. First, the information required for replay is not directly available from accessibility events, and our technique needs to reconstruct it. This is particularly challenging in our context, in which BARISTA runs in a separate sandbox than the AUT.

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.21

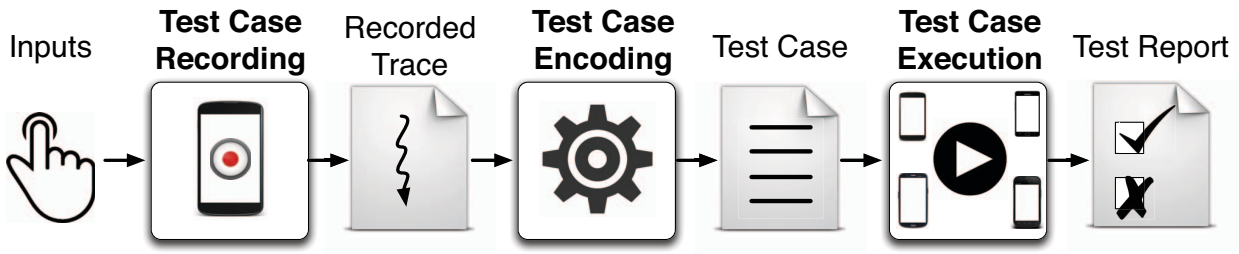

Figure 1: High-level overview of the technique.

This challenge is also a distinguishing factor with respect to related work [8] from a related domain (web app) that instead relies on a debugging interface and has direct access to the AUT. Second, our technique must process events in a timely fashion, in the face of a constantly evolving user interface. To address this challenge, our technique efficiently caches the GUI hierarchy and performs operations on its local cache.

To evaluate the practical usefulness and applicability of our technique, we implemented it in a prototype tool, called BARISTA, that encodes user recorded test cases and oracles as Espresso tests. We then performed a comparative user study in which 15 participants used BARISTA and two popular tools TESTDROID RECORDER (TR) [2] and ESPRESSO TEST RECORDER (ETR) [3], to generate tests for a set of 15 Android apps. The results of this initial study are promising. In particular, they show that BARISTA (1) can faithfully record and encode most user defined test cases, whereas the other two tools fail to do so in many cases, (2) can encode test cases that run on multiple platforms, unlike TR and ETR, and (3) provides better support for oracle generation than the two tools. In more general terms, our evaluation shows that BARISTA has the potential to improve considerably the way test cases for Android apps are generated and run, which can in turn result in an overall improvement of the Android QA process. In summary, the main contributions of this paper are:

- A technique for easily recording, encoding in a standard format, and executing in a platform-independent[1] manner test cases for Android apps.
- An implementation of the technique, BARISTA, that generates Espresso test cases and is freely available for download at http://www.cc.gatech.edu/~orso/software/barista.
- A user study, performed on a set of Android apps, that shows initial yet clear evidence of the practical usefulness and applicability of our technique, together with the improvements it provides over related work.

II. TECHNIQUE

In this section, we present our technique for recording, encoding, and executing test cases for Android apps. Figure 1 provides a high-level overview of our technique, which consists of three main phases. In the *test case recording phase*, the user interacts with the AUT with the goal of testing its functionality. Our technique records user interactions together with user induced system events and offers a convenient interface to define assertion-based oracles. At the end of the recording phase, the technique enters its *test case encoding phase*, which translates recorded interactions and oracles into test cases that are (as much as possible) device independent. Finally, in the *test case execution phase*, our technique executes the encoded test cases on multiple devices and summarizes the test results in a report. In the remainder of this section, we describe these three phases in detail.

A. Test Case Recording

In this phase, the user records test cases by exercising the functionality of an app. This phase receives the package name of the AUT as input. Based on the package name provided, the technique launches the app's main activity [7] and, at the same time, creates an interactive *menu*. The menu is displayed as a floating menu above the AUT and is movable, so that it does not interfere with the user interaction with the app.

As soon as the app is launched, a second component starts operating: the *recorder*. This component is used to (1) access the UI displayed by the AUT, (2) process user interactions, (3) process system events induced by user interactions that affect recorded test cases, and (4) assist the oracle definition process. The recorder leverages the accessibility framework of the Android platform [7] to accomplish these tasks. The accessibility framework provides access to events generated by the system in response to user interface events (e.g., the click of a button). The recorder leverages the accessibility infrastructure to listen to two categories of events: events that describe a change in the UI and events that are fired as consequence of user interactions. Events in the former category are used to create a reference that uniquely identifies an element in the app's UI. We call this reference the *selector* of the element. Events in the latter category are used to identify user interactions. Recorded interactions use selectors to refer to their target UI elements. Interactions and defined oracles are logged by the recorder in the *recorded trace* in the form of *actions*. When the user stops the recorder, our technique passes the content of the recorded trace to the test case encoding phase. In the rest of this section, we discuss the content of the recorded trace, describe how the recorder defines selectors, present what type of interactions are recognized by our technique, and describe the oracle definition process.

1) Recorded Trace: Figure 2 shows an abstract syntax for the recorded trace. The beginning of the trace is defined by the $trace\text{-}def$ production rule, which indicates that a trace

[1]By platform-independent here we mean independent from the device and operating system version running on that device.

```
trace-def          ::= trace main-activity actions
main-activity      ::= string
actions            ::= action | action, actions
action             ::= interact-def | sys-interact-def | ui-assert-def
                     | af-assert-def | key-def
interact-def       ::= interact i-type selector timestamp i-props
i-type             ::= click | long click | type | select | scroll
selector           ::= resource-id | xpath | properties-based
resource-id        ::= string
xpath              ::= string
properties-based   ::= element-class element-text
element-class      := string
element-text       := string
timestamp          ::= number
i-props            ::= | exprs
sys-interact-def   ::= sys-interact sys-i-type timestamp sys-i-props
sys-i-type         ::= pause | stop | restart | start | resume | rotate | message
sys-i-props        ::= | exprs
ui-assert-def      ::= ui-assert ui-a-type selector timestamp ui-a-props
ui-a-type          ::= checked | clickable | displayed | enabled | focus
ui-a-props         ::= | selector | exprs
af-assert-def      ::= af-assert timestamp af-a-props
af-a-props         ::= | exprs
key-def            ::= key key-type timestamp
key-type           ::= action | close
exprs              ::= expr | expr, exprs
expr               ::= bool | number | string
```

Figure 2: Abstract syntax of the recorded trace.

consists of the name of the main activity followed by a list of actions. The types of actions logged into the recorded trace is indicated by production $action$. In the rest of this section, we will refer to the abstract syntax while describing the actions recorded in this phase.

2) Selectors: Our technique creates a selector for user interactions and oracles, which is used to accurately identify the UI element associated with these actions and is independent from the screen size of the device. The technique defines and uses three types of selectors: (1) the *resource ID selector* ($resource\text{-}id$ in Figure 2), (2) the *XPath selector* ($xpath$), and (3) the *property-based selector* ($property\text{-}based$). The resource ID selector corresponds to the Android resource ID that is associated to a UI element [7]; the XPath [9] selector identifies an element based on its position in the UI tree (as the UI tree can be mapped to an XML document); and the property-based selector identifies an element based on two properties: the class of the element ($element\text{-}class$) and the text displayed by the element ($element\text{-}text$), if any.

Our technique does not use the Android resource ID as its only type of selector because the Android framework does not require a developer to specify a resource ID for each UI element. Moreover, the framework cannot enforce uniqueness of IDs in the UI tree. Our technique does not use an element's screen coordinates as a selector either because the screen coordinates of a UI element can be considerably different on different devices.

The recorder aims to identify the most suitable type of selector for every interaction and oracle processed by leveraging the accessibility functionality of the Android platform. It does so by analyzing the accessibility tree for the UI displayed on the device. Each node in the tree represents an element in the UI and is characterized by two properties of interest: resource ID (if defined) and class of the UI element represented by the node. The recorder navigates the accessibility tree to track uniqueness of resource IDs and stores the IDs and the corresponding nodes in a *resource ID map*. The information stored in this map is then used every time an interaction occurs or an oracle is defined by the user. More precisely, when the recorder processes these types of actions, it considers the accessibility node associated with the action. The recorder checks whether the node has a resource ID and, if it does, checks for its uniqueness using the resource ID map. In case the resource ID is unique, the recorder creates a selector of type resource ID for that action. If the node associated to an action does not have a resource ID or the ID is not unique, the recorder generates a selector of type XPath, where the XPath selector is a path expression that identifies a specific node in the tree.

When the window containing the element affected by an interaction becomes inactive immediately after the interaction is performed (*e.g.,* when selecting an entry of a `ListPreference` dialog), the accessibility framework does not provide the reference to the node in the accessibility tree affected by the interaction. In this case, the recorder cannot define a resource ID or XPath selector and uses a property-based selector instead. The property-based selector leverages the information stored in the accessibility event representing the interaction (see Section II-A3). This type of selector identifies an element in the UI using the class of the element and the text displayed by the element (if any). We selected these two properties because they will not change across devices with different screen properties. Two UI elements that belong to the same class and display the same text would have the same selector and would thus be indistinguishable. Although this could be problematic, this type of selector is used only when the resource ID and XPath selectors cannot be used, which is not a common situation and never occurred in our evaluation.

3) Interactions: The recorder recognizes user interactions by analyzing accessibility events created by the Android platform as a result of such interactions. These events have a set of properties that describe the characteristics of the interactions. Due to space limitations, we only illustrate how the recorder processes two types of events, as other events are handled similarly.

a) Click: Our technique detects when a user clicks on a UI element by listening to accessibility events of type `TYPE_VIEW_CLICKED`. The recorder encodes an event of this type as an entry in the recorded trace ($interact\text{-}def$ in Figure 2). It labels the entry as of type ***click*** ($i\text{-}type$), identifies the interaction selector ($selector$) as discussed in Section II-A2, and saves the action timestamp ($timestamp$).

b) Type: Our technique recognizes when a user types text into an app by processing accessibility events of type `TYPE_VIEW_TEXT_CHANGED`. Naively recording events from this class, however, would result in a recorded trace that also includes spurious events in the case of programmatic (i.e., not user driven) modifications of the text. TO address this issue, our technique leverages the fact that actual typing is always followed by a `TYPE_WINDOW_CONTENT_CHANGED` event. For

each typing event, the recorder encodes the event as an entry in the recorded trace ($interact\text{-}\ def$), labels the entry as of class ***type*** ($i\text{-}type$), identifies the interaction selector ($selector$), saves the action timestamp ($timestamp$), and adds the text typed by the user to the properties of the entry ($i\text{-}props$). It is worth noting that, when a user enters text incrementally, this results in a sequence of events. This sequence of events is processed in the test case encoding phase to minimize the size of the generated test cases (see Section II-B).

After typing text, a user can click the input method action key to trigger developer defined actions. Because the Android system does not generate accessibility events for this type of interactions, our technique provides an on-screen keyboard that can be used by the tester as a regular keyboard and records this type of interactions as well. In response to this event, the recorder adds an entry ($key\text{-}def$) to its recorded trace (***action***). Our technique handles in a similar fashion the key that, when clicked, hides the on-screen keyboard (***close***).

4) User-induced System Events: User interactions can lead to system events that affect the AUT and consequently the behavior of recorded test cases. We classify these events under three categories: events that trigger callbacks of the activity lifecycle; runtime changes in the configuration of the device; and messaging objects (intents) that originate from other apps and trigger the execution of components in the AUT. We illustrate how our technique accounts for these events.

a) Activity Lifecycle Callbacks: These are triggered by the Android system as result of certain user interactions and can be divided into (1) callbacks generated as the user navigates through the AUT and (2) callbacks triggered when the AUT is not running in the foreground. Our technique does not take any action on callbacks of the former type because they are automatically triggered in the test scripts generated by our technique. Conversely, our technique detects and suitably processes the latter type of callbacks. The recorder detects when an activity of the AUT stops running in the foreground by analyzing accessibility events of type `TYPE_WINDOW_STATE_CHANGED`. In this case, the recorder checks if the activity is in its PAUSED or STOPPED state by accessing the activity manager running in the Android system. When the AUT starts running in the foreground again (detected by the recorder using the accessibility event mentioned above), the recorder creates entries ($sys\text{-}interact\text{-}def$) of type ***pause*** and ***resume*** ($sys\text{-}i\text{-}type$) if the activity was in the paused state. Otherwise, if the activity was in the stopped state, it adds ***pause***, ***stop***, ***restart***, ***start***, and ***resume*** entries.

b) Device Configurations: Configurations can be changed at runtime by the Android system as result of certain user actions. Among those, screen orientation is particularly important for test case recording because an activity of the AUT can display different UI elements based on the orientation of the device. Our technique records such changes so that the test execution phase can properly execute recorded interactions. The recorder listens for configuration change events generated by the Android system and when it detects a screen orientation change it stores the change as an entry ($sys\text{-}interact\text{-}def$) of type ***rotate*** ($sys\text{-}i\text{-}type$) having the current orientation value as its property ($sys\text{-}i\text{-}props$).

Table I: Assertable properties for UI-based oracles.

Property	*Description*
CHECKED	The element is checked
CLICKABLE	The element can be clicked
DISPLAYED	The element is entirely visible to the user
ENABLED	The element is enabled
FOCUS	The element has focus
FOCUSABLE	The element can receive focus
TEXT	The element contains a specific text
CHILD	Child-parent relationship between two elements in the UI
PARENT	Parent-child relationship between two elements in the UI
SIBLING	Sibling relationship between two elements in the UI

c) Intents: Intents are the messaging objects used by the Android system to enable communication between different apps. An app can let the system know about what messages is interested in receiving by using intent filters [7]. Our technique allows users to define and send intents to the AUT so that they can test the behavior of the AUT upon receiving these messages. Users can also define the properties of an intent through the menu provided by our technique. When an intent is defined, the recorder saves it and its properties as an entry ($sys\text{-}interact\text{-}def$) of type ***message*** into the recorded trace and then sends the intent to the AUT.

5) Oracles: Oracles are an essential part of a test case. Our technique uses assertion based oracles that can be of two types: UI-based and activity-flow-related oracles. The former check for properties of UI elements, whereas the latter check for properties of intents used to transfer control between AUT components.

a) UI-based Oracles: These oracles can either check the state of a UI element at a specific point of the execution or check the relationship between two UI elements. Table I reports the properties that can be asserted using UI-based oracles and provides a brief description of them. Variations of the properties listed in Table I can also be asserted. For instance, our technique can be used to assert that the percentage of visible area of an element is above a user defined threshold. Moreover, the technique can also define assertions that check that a property of an element does not have a certain value. The menu and the recorder contribute together to the creation of assertions. Figures 3, 4, and 5 show part of the assertion creation process. The user starts the process by clicking the *assert button* in the menu (the button with the tick symbol in Figure 3). This creates the *assertion pane*, a see-through pane that overlays the device screen entirely (Figure 4). This pane intercepts all user interactions and is configured so that the Android system does not generate accessibility events for interactions intercepted on the pane, so that no spurious events are recorded. At this point, the user can define assertions either automatically or manually. With the automatic process, the user selects an element in the UI, and our technique automatically adds assertions for each property of the element. With the manual process, assertions are defined directly by the user. For the sake of space, we describe in detail only the manual process. The automatic process follows similar principles.

Figure 3:
Menu overlay.

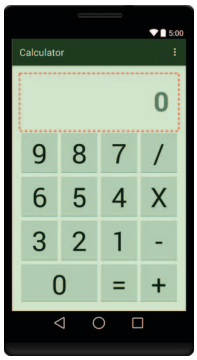

Figure 4:
Assertion pane.

Figure 5:
Oracle selection.

As shown in Table I, the user can assert properties that affect either a single element or a pair of elements. We illustrate how the technique works when asserting properties that affect one element. (Assertions that affect a pair of elements are defined similarly.) The user selects an element in the UI by long clicking (tap-hold-release) on it. In response to the long click, BARISTA sends the x and y coordinates of the location being pressed to the recorder, which explores the accessibility tree to find the node identified by the location, computes the screen location of the node's vertexes, and sends these coordinates back to BARISTA. BARISTA uses the coordinates to highlights the element, as shown in Figure 4.

The user can then either change the currently selected element through dragging or accept it. At this point, the recorder identifies the node on the accessibility tree as usual (in case the user changed it), checks the node class, and based on this information builds a list of assertable properties. The top of the list is populated with properties that are specific to the node. As shown in Figure 5, these properties are displayed in the proximity of the selected element. The user can then choose a property and the value to be considered in the assertion, and BARISTA sends the property and the value to the recorder. The recorder creates an entry in the recorded trace ($ui\text{-}assert\text{-}def$), suitably labels the entry based on the selected assertion property ($ui\text{-}a\text{-}type$), identifies the selector for the assertion ($selector$), and adds the user defined value for the assertion to the properties of the entry ($ui\text{-}a\text{-}props$). After the recorder successfully adds the assertion to its recorded trace, it signals the end of the assertion definition process to BARISTA, which removes the assertion pane from the screen, so that the user can continue to interact with the app.

b) Activity-Flow-Based Oracles: Apps use intents to transfer control flow between app components. Our technique allows users to check the properties of intents in their recorded test cases. The user can enable this type of assertions by setting a flag at the beginning of the recording process. The recorder recognizes intents being used within the AUT by reading a log of system messages generated by the Android system. When the recorder detects that the AUT used an intent to transfer control between two app components (by processing system message entries in the log), it adds an assertion ($af\text{-}assert\text{-}def$) into the recorded trace that checks for the values describing the properties of the intent (action, data, type, category).

B. Test Case Encoding

The test case encoding phase receives as input the recorded trace and a user-provided flag (*retain-time flag*) that indicates whether the timing of recorded interactions should be preserved. For instance, if a user sets a 30-seconds timer in an alarm clock app and wants to check with an assertion the message displayed when the timer goes off, he or she would set the retain-time flag to true to ensure that the assertion is checked 30 seconds after the timer is started. The test case encoding phase produces as output a test case that faithfully reproduces the actions in the recorded trace. In the current version of our technique, the generated test case is an Android UI test case based on the Espresso framework [10]. In the rest of this section, we illustrate how the technique translates the recorded trace into a test case.

The test case encoding phase starts by translating the $main\text{-}activity$ entry of the recorded trace into a statement that loads the starting activity of the recorded interactions. It then translates actions into statements grouping generated statements into a single test procedure.

Statements that reproduce user interactions and UI-based oracles are divided into three parts. The first part is used by the test case execution engine to retrieve the UI element affected by the action. Our technique places the selector ($selector$) of the action in this part of the statement. The second part of the statement consists of the action that the test case execution engine performs on the UI element identified by the first part of the statement. The technique encodes this part of the statement with the Espresso API call corresponding to the action being processed ($i\text{-}type$ or $ui\text{-}a\text{-}type$). The third part of the statement accounts for parameters involved in the action and it is action specific. To generate this part of the statement, our technique processes the properties of the action ($i\text{-}props$ or $ui\text{-}a\text{-}props$). Statements representing user-induced system events and activity-flow-based oracles do not follow this structure. Actions representing user-induced system events are translated into statements that call procedures of the currently executing activity. Actions representing activity-flow-based oracles translates into statements that check the properties of intents generated by the execution of the AUT.

The content of the generated test case is affected by the retain-time flag as follows. If the retain-time flag is set, our technique places an additional statement between the statements representing two subsequent actions. This statement pauses the execution of the test cases (but not the execution of the app being tested) for a duration that is equal to the difference of the timestamps associated with the two actions.

Overall, our technique translates interactions and oracles into a single line statement. The one-to-one mapping between actions and statements favors readability and understanding of generated test cases, thus addressing a well-known problem with automatically generated tests.

C. Test Case Execution

The test case execution phase takes as input the test case produced by the second phase of the technique, together with a user-provided list of devices on which to run the test case, and performs three main tasks: (1) prepare a device environment for the test case execution, (2) execute the test case, and (3) produce the test report.

The first step installs the AUT and the generated test case on all devices in the user-provided list. Once the execution environment is set up, the technique executes the test case on each device in the user-provided list in parallel. The execution of a test case is supported through our extension of the Espresso framework and works as follows. The test case execution engine begins with loading the starting activity of the test. From this point, the engine synchronizes the execution of the test case's steps with the updates in the UI of the AUT.

The engine processes user interaction and UI-based oracle statements as follows. It first navigates the UI displayed by the device to find the UI element referenced by the action. If the element is not present, the execution of the test case terminates with an error. If the element is present, the execution engine behaves differently according to whether it is processing an interaction or an oracle statement. In the former case, the execution engine injects a motion event into the app or performs an API call on the UI element being targeted by the interaction. In the case of an oracle statement, the execution engine retrieves all elements in the UI that hold the property expressed by the oracle's assertions and checks whether the element targeted by the oracle is one of these elements. If the element is not present, the test case terminates with a failure. Otherwise, the execution continues.

The engine processes user-induced system event statements by mediating the execution of the system event with the Android system. Activity-flow-based oracle statements are processed as follows. During the execution of the test case, the test engine stores intents being sent by the AUT into a buffer. When the engine reaches an oracle statement it checks that the buffer contains an intent with the same properties as the one expressed by the statement. After the execution of such statement, the engine clears the buffer to make sure that following oracle statements will match new intents.

At the end of the execution, the technique produces a test execution report that contains: (1) the outcome of the test case on each device, (2) the test case execution time, and (3) debug information if an error or failure occurred during execution.

III. Implementation

We implemented our technique in a framework called BARISTA. There are three main modules in the framework: (1) the *recording module*, which implements the aspects of the test case recording phase (Section II-A); (2) the *encoding module*, which encodes test cases as presented in the test case encoding phase (Section II-B); and (3) the *execution module*, which executes test cases as described in the test case execution phase (Section II-C). The recording module is implemented as an Android app and runs on devices that use the platform API level 16 and above. The app does not require root access to the device to operate and does not require the device to be connected to an external computational unit during recording, as the test case recording happens directly and entirely on the device. The encoding and execution modules are part of a web service implemented in Java. We describe these three components in more detail.

There are three fundamental components in the BARISTA app: (1) the *menu component*, (2) the *recording component*, and (3) the *input method component*. The three components correspond, respectively, to the menu, recorder, and keyboard presented in Section II-A. The three components run in distinct processes, which in turn are different from the process in which the AUT is running. This design allows BARISTA to perform its test case recording phase on all apps installed on the device without the need to instrument these apps. When the user ends the recording phase, the app attaches the trace to an HTTP requests and sends it to the BARISTA web service.

The encoding module of the BARISTA web service uses the JavaWriter 2.5 library [11] to create the source code of the generated test cases. BARISTA encodes test cases based on the Espresso 2.2 framework [10]. More precisely, BARISTA extends Espresso to provide a larger API that implements the concepts introduced by the technique. The extended API includes the notion of XPath selector (added to the `ViewMatcher` class), a select action for multiple view elements (implemented by extending the `ViewAction` class), and an extended support for the scroll functionality. The BARISTA web service uses the adb server to prepare device environments and execute test cases. Test reports are produces using Spoon 1.7 [12].

IV. Limitations

As we stated in Section II-A, our technique leverages the accessibility functionality of the Android platform to detect user interactions. In this way, the technique does not need to run on a "rooted" device, does not need customization of the underlying platform, and does not need to instrument the AUT. However, the accessibility infrastructure does not currently offer support for complex multi-touch gestures (*e.g.,* pinch in and out). We are currently investigating ways to address these limitations. Fortunately, these actions are mostly used in games and are not predominant in other types of app.

Our technique binds interactions and oracles with UI elements. Certain Android apps, however, rely on bitmapped (rather than UI) elements. Hence, the technique cannot currently handle such apps. Luckily, as in the previous case, the vast majority of these apps are games, whereas other types of app tend to rely exclusively on standard UI elements.

Although our technique generates by design single-app test cases (i.e., records only the actions performed within the AUT), the accessibility framework allows for observing system-wide interactions. The technique could therefore be extended to handle such system-wide interactions as well.

Finally, our technique does not have sandboxing capabilities at the moment, so it is not able to record inputs that require

Table II: Description of our benchmark apps.

ID	*Name*	*Category*	*Installations (#K)*	*LOC (#K)*
A1	DAILY MONEY	Finance	500 - 1000	10.7
A2	ALARM KLOCK	Tools	500 - 1000	6.1
A3	QUICKDIC	Books	1000 - 5000	289.7
A4	SIMPLE C25K	Health	50 - 100	1.5
A5	COMICS READER	Comics	100 - 500	8.4
A6	CONNECTBOT	Communication	1000 - 5000	24.3
A7	WEATHER NOTIFICATION	Weather	100 - 500	13.2
A8	BARCODE SCANNER	Shopping	100000 - 500000	47.9
A9	MICDROID	Media	1000 - 5000	5.6
A10	EP MOBILE	Medical	50 - 100	31.4
A11	BEECOUNT	Productivity	10 - 50	16.2
A12	BODHI TIMER	Lifestyle	10 - 50	10.5
A13	ANDFHEM	Personalization	10 - 50	60.3
A14	XMP MOD PLAYER	Music & Audio	10 - 50	58.7
A15	WORLD CLOCK	Travel & Local	50 - 100	31.4

sandboxing (e.g., sensors data, networking data, camera data). Adding these capabilities would require our technique to instrument the AUT or the Android framework, which would change the nature of the approach and is something that we plan to explore in future work.

V. EMPIRICAL EVALUATION

To assess the expressiveness, efficiency, and ultimately usefulness of our approach, we used BARISTA to perform a user study involving 15 human subjects and 15 real-world Android apps. Because defining oracles is a fundamental part of generating test cases and of our approach, to perform an apple-to-apple comparison we used as a baseline for our evaluation: TESTDROID RECORDER (TR) [2] and ESPRESSO TEST RECORDER (ETR) [3]. The former tool records test cases in the Robotium [13] format while the latter records test cases in the Espresso [10] format. We therefore did not consider pure record/replay tools with no oracle definition capabilities, such as RERAN [14], VALERA [15], and MOSAIC [16]. We considered including ACRT [4] in our study, as it can record tests in Robotium format. Unfortunately, however, ACRT does not work with recent Android versions, so using it would have required us to backport our benchmark applications to an earlier Android version.

In our empirical evaluation, we investigated the following research questions:

RQ1: Can BARISTA record user defined test cases? If so, how does it compare to TR and ETR?

RQ2: Is the test case recording process with BARISTA more efficient than the one with TR and ETR?

RQ3: Does BARISTA's encoding preserve the functionality of the test cases? How does BARISTA compare to TR and ETR in this respect?

RQ4: Can test cases generated by BARISTA run on different devices? How platform independent are they with respect to test cases generated by TR and ETR?

In the remainder of this section, we first describe the benchmarks used in the evaluation. We then present the user study, discuss evaluation results, and conclude illustrating anecdotal evidence of BARISTA's usefulness using feedback from developers that used it.

A. Experimental Benchmarks

For our empirical evaluation, we used a set of real-world Android apps. Specifically, we selected 15 free and open-source apps from the F-Droid catalog [17] (ETR requires an app source code). Our choice of apps is based on three parameters: (1) popularity, (2) diversity, and (3) self-containment. As a popularity measure, we used the number of installations for an app according to the Google Play store [18]. We selected apps from different categories to have a diverse corpus of benchmarks and prioritized apps for which we did not have to build extensive stubs (*e.g.,* apps that do not rely on a hard to-replicate backend database). Table II shows the lists of apps we used. For each app, the table shows its ID (*ID*), name (*Name*), category (*Category*), the range of its installations (*Installations*), and the number of lines of code (*LOC*). It is worth noting that none of the apps in Table II had a reference test suite.

B. User Study

For our experimentation, we recruited 15 graduate students from three institutions. We asked the participants to perform four tasks: (#1) write natural language test cases (NLTCs), (#2) record NLTCs using TR, (#3) record NLTCs using ETR, and (#4) record NLTCs using BARISTA. Before performing the user study, we conducted a three-hour tools demonstration session to familiarize the participants with the three tools. We did not inform the subjects of which tool was ours and which ones were the baseline (but they obviously could have discovered this by searching the name of the tools).

All participants started from the task #1. In this task we provided the participants with three benchmark apps, so that each app was assigned to three different users. We asked the participants to explore the apps' functionality and then define five NLTCs for each app assigned to them. NLTCs were written purely in natural language, without the use of any framework and without even following any particular structure. After they all completed task #1, we manually analyzed the NLTCs for possible duplicates and checked with the participants in case of ambiguities. Table III shows the properties of the NLTCs we collected. For each app, the table shows the number of distinct NLTCs (*NLTC(#)*), average number of interactions per test case (*I(#)*), and average number of assertions per test case (*A(#)*). The total number of distinct NLTCs is 215. All NLTCs have at least one assertion. A1 is the app having the NLTC with the highest number of interactions (27), while A11 is the app with the NLTC having the highest number of assertions (10). All NLTCs are expected to pass.

In tasks #2, #3 ,and #4, we asked participants to record NLTCs using TR, ERT, and BARISTA, respectively. For each task, each participant was provided with a set of NLTCs written for three apps. The set of NLTCs provided to a specific participant was different between the three tasks. However, the set of all test cases across the three tasks was the same. We also decided not to give participants NLTCs they wrote, so as to mimic a scenario in which the test specifications are provided by a requirements engineer and the testing is performed by a

Table III: Information on the NLTCs considered: *NLTC(#)* = number of NLTCs for the app; *I(#)* = average number of interactions across NLTCs; *A(#)* = average number of assertions across NLTCs.

ID	*NLTC(#)*	*I(#)*	*A(#)*
A1	15	9.33	3.40
A2	15	7.07	1.40
A3	14	6.21	1.36
A4	14	4.36	3.14
A5	14	3.50	1.93
A6	13	8.92	1.23
A7	14	3.29	2.79
A8	14	2.93	1.86
A9	12	4.08	1.25
A10	15	6.47	3.00
A11	15	6.73	2.20
A12	15	3.67	1.73
A13	15	3.93	3.13
A14	15	4.87	2.47
A15	15	4.47	3.27
Total	215	5.33	2.30

Table IV: Results of the test case recording process, for each app considered: *C(#)* = number of test cases that could be recorded; *NC(#)* = number of test cases that could not be recorded; *AS(#)* = number of assertions skipped; *AA(#)* = number of assertions altered; and *T(s)* = average recording time.

ID	TR					ETR					BARISTA				
	C(#)	*NC(#)*	*AS(#)*	*AA(#)*	*T(s)*	*C(#)*	*NC(#)*	*AS(#)*	*AA(#)*	*T(s)*	*C(#)*	*NC(#)*	*AS(#)*	*AA(#)*	*T(s)*
A1	15	0	9	20	176	15	0	0	36	97	15	0	2	0	119
A2	4	11	0	2	108	15	0	14	3	27	15	0	0	0	42
A3	9	5	5	1	11	13	1	3	9	40	14	0	2	0	18
A4	9	5	8	7	27	14	0	26	13	30	14	0	3	0	29
A5	12	2	2	2	38	14	0	1	20	19	14	0	0	0	9
A6	6	7	0	1	18	13	0	0	13	29	13	0	0	0	11
A7	11	3	13	5	14	13	1	5	21	15	14	0	0	0	8
A8	11	3	5	0	25	14	0	0	17	21	14	0	0	0	5
A9	11	1	3	3	23	12	0	0	12	28	12	0	0	0	11
A10	13	2	10	2	61	14	1	17	8	38	15	0	0	0	56
A11	12	3	10	0	66	15	0	1	13	56	15	0	0	0	57
A12	15	0	5	0	25	15	0	4	14	28	15	0	0	0	22
A13	13	2	14	3	123	15	0	0	39	51	15	0	0	0	46
A14	15	0	7	2	97	11	4	1	24	43	15	0	2	0	49
A15	15	0	17	0	83	14	1	1	35	124	15	0	2	0	57
Total	171	44	108	48	60	208	7	74	277	43	215	0	11	0	36

QA tester. For each of the three tasks, we asked the users to reproduce the steps of the NLTCs as faithfully as possible, unless the tool prevented them to do so (*e.g.,* they could skip assertions that the tool was unable to encode). Finally, we grouped participants so that some performed the task #2 before the other two tasks, others started from task #3, and still others started from task #4.

The experimental setup to perform task #2 was structured as follows. We asked users to record NLTCs on a device running Android API level 19. The device was connected to a MacBook Pro (2.3 GHz i7 processor and 8GB memory) running Eclipse 4.4, with TR installed as a plugin. To define an assertion using TR, users might need to specify the Android resource IDs of the element involved in the assertion. We thus made the UIAUTOMATORVIEWER tool [19] available to users, so that they could easily explore an app's UI. In task #3, we asked users to record NLTCs on a device running Android API level 19. The device was connected to a MacBook Pro (2.3 GHz i7 processor and 8GB memory) running Android Studio 2.2 with ETR installed. To perform task #4, we asked users to record NLTCs using a device running API level 19 with BARISTA installed. We did not impose any timeout to perform the three tasks.

C. Results

RQ1: To answer the part of RQ1 about BARISTA's expressiveness, we checked the test cases recorded by users using BARISTA against the corresponding NLTCs. The third part of Table IV (columns below BARISTA header) shows the results of this check. For each app, we report the number of test cases that could be recorded (*C*), the number of test cases that could not be recorded (*NC*), the number of assertions skipped (*AS*), and the number of assertion altered (*AA*). We considered an NLTC as recorded if the generated test case contained all interactions defined in it, and not recorded otherwise. We considered an assertion as skipped if the user did not define it, whereas we considered an assertion as altered if the user defined an assertion with a different meaning from the one in the NLTC. When using BARISTA, participants could record all test cases, skipped 11 assertions (2.2% of the total number of assertions), and did not alter any assertion. The 11 assertions that users could not express with BARISTA do not directly check for properties of UI elements (*e.g.,* an NLTC for A4 states "assert that the alarm rings").

The first (TR) and second (ETR) sections of Table IV help us answer the second part of RQ1, which compares BARISTA to the baseline. 44 test cases could not be recorded using TR. 36 of those could not be recorded because TR altered the functionality of 10 apps, preventing users from performing certain interactions. In the remaining cases, users stopped recording the test case after making a mistake. Even without considering the last eight test cases, which mostly depend on user errors, BARISTA could record 20.1% more test cases than TR. 7 test cases were not recorded using ETR because users stopped recording the test case after making a mistake.

As the table also shows, users skipped 108 assertions while using TR and 74 while using ETR (the assertions skipped while using BARISTA are included in both sets). The reason behind these two high numbers is that TR and ETR offer a limited range of assertable properties. For instance, TR does not allow for checking whether a UI element is clickable or whether an element is checked, while ETR offers only three assertable properties: *text is*, *exisit*, and *does not esist*. In the test cases generated by TR and ETR, we can also note that 48 and 277 assertions (sum of column *AA*) were different from the ones defined in the corresponding NLTCs. An example of such assertion mismatch is an NLTC from A1, for which the user recorded "assert button is enabled" instead of "assert button is clickable". We asked the participants involved why they modified these assertions, and they said that it was because they could not find a way to record the original assertion with the tool. Among the test cases recorded by all tools, BARISTA could faithfully express 65.2% more assertions than TR and 3.8X more assertions than ETR.

These results provide initial evidence that BARISTA can record test cases and is more expressive than TR and ETR.

RQ2: To answer RQ2, we compare the amount of time taken by the participants to record test cases using TR, ETR, and

BARISTA. For each app, Table IV reports the average time in seconds (*T(s)* columns) taken to record test cases. The average time is computed considering the test cases that were recorded by all three tools and in which no assertion was skipped. The amount of time associated with each test case is calculated from the moment in which the user recorded the first action to the time in which the user terminated the recording process. Recording test cases with BARISTA was faster than TR for 13 apps and faster then *ETR* for 10 apps. BARISTA has the lowest average recording time considering all apps and it is 32.3% faster than TR and 19.9% faster than ETR.

We can thus conclude that, on average, BARISTA is more efficient in recording test cases than TR and ETR.

RQ3: To answer the part of RQ3 about BARISTA's correctness, we executed the 215 test cases generated using BARISTA on the device on which they were recorded. We report the execution results in the third part of Table V (columns below BARISTA header). For each app, we report the number of test cases executed (*T*), the number of test cases that worked correctly (*W*), the number of test cases that terminated with an error or failure due to a problem in the tool encoding or execution phase (*NW*), and the number of test cases that terminated with an error or failure due to a user mistake in the recording process (*M*). We consider a test case as working correctly if it faithfully reproduces the steps in its corresponding NLTC. Across all benchmark apps, 97.2% of the recorded test cases worked correctly, and 12 apps had all test cases working properly. The test case from A5, which is marked as not working, terminated with an error because the file system of the device changed between the time the test case was recorded and the time the test case was executed. The five test cases marked as user mistakes terminated with an assertion failure. In two of these cases, the user asserted the right property but forgot to negate it. In the remaining three test cases, the user asserted the right property but on the wrong UI element. We presented the errors to users and they confirmed their mistakes.

The first and second part of Table V (columns below TR and ETR headers) lets us answer the second part of RQ3, which compares the correctness of the test cases generated by BARISTA with respect to that of the baseline. Across all benchmark apps, only 64.9% of the recorded test cases with TR worked correctly. This number corresponds to 51.6% of the NLTCs. The 49 test cases classified as not working could not replicate at least one of the interactions from their corresponding NLTCs. Users made 11 mistakes using TR. In the majority of these cases (6), the user entered the wrong resource ID when recording an assertion. In the case of ETR, only 38.9% of the recorded tests worked correctly. 121 test cases did not work because of the following reasons: (1) the UI reference generated by the tool could not identify the corresponding UI element (75 test cases), (3) the tool generated additional actions that changed the behavior of the test case (30 test cases), and (3) the tool did not generate test case actions for certain user interactions (16 test cases). Users made six mistakes using ETR. In all cases, users altered an assertion making the test case fail.

Table V: Results of test case execution: *T(#)* = number of executed test cases; *W(#)* = number of working test cases; *NW(#)* = number of test cases that did not work due to a problem with the tool; and *M(#)* = number of test cases that did not work due to a user mistake.

ID	TR				ETR				BARISTA			
	T(#)	*W(#)*	*NW(#)*	*M(#)*	*T(#)*	*W(#)*	*NW(#)*	*M(#)*	*T(#)*	*W(#)*	*NW(#)*	*M(#)*
A1	15	8	6	1	15	6	9	0	15	15	0	0
A2	4	3	1	0	15	7	8	0	15	15	0	0
A3	9	5	4	0	13	6	7	0	14	14	0	0
A4	9	3	5	1	14	8	6	0	14	12	0	2
A5	12	10	0	2	14	9	5	0	14	13	1	0
A6	6	4	2	0	13	5	8	0	13	13	0	0
A7	11	9	2	0	13	3	6	4	14	11	0	3
A8	11	8	2	1	14	8	6	0	14	14	0	0
A9	11	11	0	0	12	3	8	1	12	12	0	0
A10	13	9	4	0	14	6	9	0	15	15	0	0
A11	12	8	4	0	15	0	15	0	15	15	0	0
A12	15	12	3	0	15	7	7	1	15	15	0	0
A13	13	1	9	3	15	4	11	0	15	15	0	0
A14	15	9	5	1	11	4	7	0	15	15	0	0
A15	15	11	2	2	14	5	9	0	15	15	0	0
Total	171	111	49	11	208	81	121	6	215	209	1	5

Overall, BARISTA nearly doubles the percentages of working test cases compared to TR and ETR. Based on these results, we can answer RQ3 as follows: there is evidence that test cases generated by BARISTA work correctly, and that BARISTA can outperform TR and ETR in this respect.

RQ4: To answer the part of RQ4 on BARISTA's cross-device compatibility, we executed the test cases recorded using BARISTA on seven (physical) devices: LG G FLEX (D1), MOTOROLA MOTO X (D2), HTC ONE M8 (D3), SONY XPERIA Z3 (D4), SAMSUNG GALAXY S5 (D5), NEXUS 5 (D6), and LG G3 (D7). (We acquired these devices in early 2015 with the goal of getting a representative set in terms of hardware and vendors.) We executed all the test cases that did not contain a user mistake, and among those, 206 test cases worked on all devices. Overall, the average compatibility rate across all apps and devices was 99.2%. Two test cases (from A13) did not work on D7 because that device adds additional space at the bottom of a `TableLayout` element. The additional space moves the target element of an action out of the screen, preventing BARISTA from successfully interacting with that element. (The two test cases work on D7 by adding a scroll action to the test cases.) Also, one test case (from A13) did not work on D1, D5, and D7 because these devices display an additional element in a `ListView` component. For this reason, an interaction in the test case selects the previous to last element instead of the last element.

To answer the second part of RQ4, which involves comparing cross-device compatibility of test cases generated using BARISTA with respect to the baseline, we executed on the seven devices considered the test cases (that did not contain a user mistake) recorded using TR and ETR. For TR, 108 tests worked on all devices, and the average compatibility rate across all apps and devices was 68.3%. Many of the failing tests also failed on the device on which they were recorded. In addition, TR generated three test cases that did not work on D5: one test (from A9) failed to identify the target element

of an interaction based on the x and y coordinates stored in the test case; two tests (from A15) used an index to select the target element of an interaction that was not valid on the device. For ETR, 62 tests worked on all devices, and the average compatibility rate across all apps and devices was 37.3%. Also in this case, many tests failed on the device on which they were recorded as well. In addition ETR generated 2 tests that did not work on D1, 1 test that did not work on D2, 19 tests that did not work on D4, 3 tests that did not work on D5, and 14 tests that did not work on D7. 37 of these failures were caused by the UI reference generated by the tool. The remaining two failures were caused by an unsatisfiable constraint in the test. Finally, it is worth noting that, whereas for the three BARISTA-generated tests that are not cross-device compatible, the corresponding TR- and ETR-generated tests are also not cross-device compatible, the opposite is not true; that is, for the TR- ETR-and generated tests that are not cross-device compatible, the corresponding BARISTA-generated tests are cross-device compatible.

Based on these results, we can conclude that tests generated using BARISTA can run on different devices in a majority of cases, and that BARISTA generated a greater number of cross-device-compatible tests than TR and ETR.

D. Developers Feedback

We recently publicly released BARISTA and also directly contacted several developers in various companies to introduce our tool and ask them to give us feedback in case they used it. Although this is admittedly anecdotal evidence, we want to report a few excerpts from the feedback we received, which echo some of our claims about BARISTA's usefulness. Some feedback indicates the need for a technique such as BARISTA: *"I have been looking for something like* BARISTA *to help me get into automation for a while"*. Other feedback supports the results of our empirical evaluation on the efficiency of BARISTA: *"Overall, a very interesting tool! For large-scale production apps, this could save us quite some time by generating some of the tests for us"*. Finally, some other feedback points to aspects of the technique that should be improved and that we plan to address in future work: *"There are a few more assertions I'd like to see. For example, testing the number of items in a* `ListView`*"*. We are collecting further feedback and will make it available on the BARISTA's website.

E. Threats To Validity

There are both internal and external threats to validity associated with the results of our empirical evaluation.

In terms of internal validity, the participant of the user study were not familiar with the apps they generated test cases, which may not be the case in real-world situations. However, it is not uncommon for testers to test someone else's software.

In terms of external validity, our results might not generalize to other apps. To mitigate this threat, we used randomly selected real-world apps. Our results might also not generalize to other devices. To mitigate this threat, we selected a representative set of devices in terms of hardware and vendors.

VI. RELATED WORK

In the domain of desktop apps, there is a large body of techniques that focus on GUI test automation using a record/replay approach [20]–[24]. BARISTA can be related to MARATHONITE [24] and ABBOT [20] in that they all work at a higher level of abstraction recording semantic actions and they identify elements using their delineating properties rather than using their coordinates. However, BARISTA differs from the two techniques in that they use dynamic code instrumentation while our technique uses the accessibility infrastructure offered by the Android framework.

There is also a rich set of techniques for GUI test automation through record/replay in the web app domain [8], [25]–[30]. BARISTA can be related to SELENIUM IDE in that they both record semantic actions and offer the opportunity to express oracles in the recording process. However, SELENIUM IDE has direct access the AUT while BARISTA can access the AUT only upon receiving certain accessibility events, this difference makes the recording task of our technique more challenging. Furthermore, SELENIUM IDE runs with heightened privileges [31].

In the domain of mobile apps, there are techniques that focus on GUI test automation through record/replay [2]–[6] and techniques that focus mainly on the record/replay task [14]–[16], [32]. TESTDROID RECORDER [2] is a tool, implemented as an Eclipse plugin, that records interactions from a connected device running the AUT. BARISTA is similar to TR in that they both record interactions at the application layer, however the approach used by TR presents some limitations. First, TR uses identifiers that do not reliably identify elements in the GUI. Second, generated tests rely on `sleep` commands, which make tests slow and unreliable. Third, the tool does not suitably instrument the UI of the AUT to process user inputs leading to missed interactions in recorded test cases.

ESPRESSO TEST RECORDER [2] is part of Android Studio and generates test cases by recording user interactions from a connected device running the AUT. Similarly to BARISTA, the tool generates Espresso test cases. However, the tool support for defining oracles is limited and generated test cases use references to UI elements that tend to be inaccurate.

ACRT [4] is a research tool that, similarly to TR, generates ROBOTIUM tests from user interactions. ACRT is based on an app instrumentation approach that modifies the layout of the AUT to record user interactions and adds a custom gesture to certain element of the GUI to allow the user to definition oracles. The support for interactions and oracles is limited and the technique does not consider how to uniquely identify elements in the GUI.

SPAG [5] uses SIKULI [33] and ANDROID SCREENCAST [34] to create a system in which the screen of the AUT is redirected to a PC and the user interacts with AUT using the PC. SPAG−C [6] extends SPAG using image comparison methods to validate recorded oracles. The approach for oracle definition presented in SPAG and SPAG−C is minimally invasive, as it does not modify the AUT. However, expressing

oracles for a specific element in the GUI is a practical challenge and the image comparison approach can miss small but significant differences. MOBIPLAY [32] is a record/replay technique based on remote execution. The technique is similar to BARISTA in that inputs to the AUT are collected at the application layer. However, MOBIPLAY input collection approach requires modifications in the Android software stack. In addition, MOBIPLAY records inputs based on their screen coordinates, while BARISTA collects them so that they are platform-independent. Finally, MOBIPLAY does not support the definition of oracles.

RERAN [14] records low level system events by leveraging the Android GETEVENTS utility and generates a replay script for the same device. The low level approach presented by RERAN is effective in recording and replaying complex multi-touch gestures. However, generated scripts are not suitable for replay on different devices because recorded interactions are based on screen coordinates. VALERA [15] redesigns and extends RERAN with a stream-oriented record-and-replay approach. MOSAIC [16] extends RERAN to overcome the device fragmentation problem. The technique abstracts low-level events into an intermediate representation before translating them to a target system. RERAN, VALERA, and MOSAIC are powerful techniques for record and replay. However, they do not support oracle definition, which constitute a fundamental aspect of GUI testing.

VII. CONCLUSION

We presented a new technique for helping testers create (through recording), encode (using a standard format), and run (on multiple platforms) test cases for Android apps. One distinctive feature of our technique is that it allows for adding oracles to the tests in a visual and intuitive way. We implemented our technique in a freely available tool called BARISTA. Our empirical evaluation of BARISTA shows that it can be effective in practice and improve the state of the art.

There are a number of possible directions for future work. We will extend our current evaluation by (1) performing an additional user study with a large number of experienced Android developers and (2) running the generated test cases also on different versions of the Android OS. We will study ways to factor out repetitive action sequences, such as app initialization, so that testers do not have to repeat them for every test. We will investigate how to add sandboxing capabilities to BARISTA, so that it can generate tests that are resilient to changes in the environment. Based on feedback from developers, we will extend the set of assertable properties that testers can use when defining oracles. We will investigate the use of fuzzing for generating extra tests by augmenting those recorded, possibly driven by specific coverage goals. We will study ways to help developers fix broken test cases during evolution (*e.g.,* by performing differential analysis of the app's UI). Finally, we will investigate the use of our technique to help failure diagnosis; a customized version of BARISTA could be provided to users to let them generate bug reports that allow developers to reproduce an observed failure.

ACKNOWLEDGMENTS

We thank the students who kindly participated in our user study. This work was partially supported by the National Science Foundation under grants CCF-1453474, CCF-1564162, CCF-1320783, and CCF-1161821, and by funding from Google, IBM Research, and Microsoft Research.

REFERENCES

[1] OpenSignal, "Android Fragmentation Visualized," https://opensignal.com/reports/2014/android-fragmentation/.
[2] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: automated remote UI testing on Android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, 2012.
[3] Google, "Espresso Test Recorder," https://developer.android.com/studio/test/espresso-test-recorder.html.
[4] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-Replay Testing for Android Applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, 2014.
[5] Y.-D. Lin, E.-H. Chu, S.-C. Yu, and Y.-C. Lai, "Improving the accuracy of automated GUI testing for embedded systems," *Software, IEEE*, 2014.
[6] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai, "On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices," *Software Engineering, IEEE Transactions on*, 2014.
[7] J. Annuzzi Jr, L. Darcey, and S. Conder, *Advanced Android Application Development*. Pearson Education, 2014.
[8] SeleniumHQ, "Selenium IDE," http://docs.seleniumhq.org/projects/ide/.
[9] W3C, "XML Path Language," https://www.w3.org/TR/xpath-30/.
[10] Google, "Espresso," https://google.github.io/android-testing-support-library/.
[11] Square, "JavaPoet," https://github.com/square/javapoet.
[12] ——, "Spoon," http://square.github.io/spoon.
[13] H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.
[14] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "RERAN: Timing- and Touch-sensitive Record and Replay for Android," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.
[15] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet Lightweight Record-and-replay for Android," in *Proceedings of the 2015 International Conference on Object Oriented Programming Systems Languages & Applications*, 2015.
[16] M. H. Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: Cross-Platform User-Interaction Record and Replay for the Fragmented Android Ecosystem," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015.
[17] F-Droid, "F-Droid," https://f-droid.org.
[18] Google, "Google Play," https://play.google.com/store.
[19] ——, "Automating User Interface Tests," http://developer.android.com/tools/testing-support-library/index.html.
[20] Abbot, "Abbot Java GUI Test Framework," http://abbot.sourceforge.net/doc/overview.shtml.
[21] Jacareto, "Jacareto," http://sourceforge.net/projects/jacareto.
[22] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay Tool for Observation-Based Testing," in *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, 2000.
[23] Pounder, "Pounder," http://pounder.sourceforge.net.
[24] MarathonITE, "marathonITE Powerful Tools for Creating Resilient Test Suites," http://marathontesting.com.
[25] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic Capture and Replay for JavaScript Applications," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.
[26] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.
[27] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive Record/Replay for Web Application Debugging," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 2013.

[28] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing," in *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, 2010.

[29] S. Andrica and G. Candea, "WaRR: A Tool for High-Fidelity Web Application Record and Replay," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2011.

[30] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI Testing Tools Using Accessibility Technologies," in *Proceedings of the 2009 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[31] SeleniumHQ, "Selenium Heightened Privileges Browsers," http://www.seleniumhq.org/docs/05_selenium_rc.jsp.

[32] Z. Qin, Y. Tang, E. Novak, and Q. Li, "MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications," in *Proceedings of the 2016 International Conference on Software Engineering*, 2016.

[33] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," in *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, 2009.

[34] Android Screencast, "Android Screencast," https://code.google.com/p/androidscreencast/.