Troyd: Integration Testing for Android

Technical Report CS-TR-5013, August 2012

Jinseong Jeon Jeffrey S. Foster

Department of Computer Science, University of Maryland, College Park

{jsjeon, jfoster}@cs.umd.edu

Abstract

We introduce Troyd, a new integration testing framework for Android apps. Troyd allows testers to write high-level scripts to drive the app under test as desired, e.g., clicking buttons on the screen, checking the contents of a text box, and so on. Troyd also provides a convenient recording mode, in which users construct Troyd scripts as the app is running; we have found that this interactivity is extremely useful in practice. Troyd is freely available as an open-source project.

1. Introduction

Android is one of the most popular mobile device platforms, with more than 600,000 apps in Google Play [3] alone. The Android SDK provides two main ways to test apps: The (Java) Instrumentation package [5], which provides APIs to monitor incoming Intents, send key events, and retrieve runtime information about instances on the screen; and monkey, which lets users script event generation. However, while very useful, both of these frameworks have important limitations: Instrumentation tests must be built and run as separate apps, which means that test scenarios are defined in advance and cannot be changed while running; and monkey specifies events using absolute coordinates, which may vary from device to device and may change as app screen layouts evolve.

In this paper, we introduce Troyd, a new integration testing framework for Android apps. Troyd, inspired by TEMA [14] and built using robotium [12], combines the scripting capabilities of monkey with the high-level APIs supplied by Instrumentation. In Troyd, test scripts consist of high-level commands, e.g., menu (press the hardware menu key), click "ok" (click the button labeled "ok"), assert_text "Andr*id" (check that there is a TextView whose contents match the given regular expression), and so on. These intuitive commands make testing scripts easy to write and understand.

Additionally, Troyd provides a recording mode in which testers type Troyd commands into a command-line interface as the app runs. When the tester is finished, Troyd then generates a test script containing the recorded commands. We have found this interactive test generation capability to be quite useful, because we can directly observe the state of the app as we are writing the test case.

Troyd is available as an open-source project at

https://github.com/plum-umd/troyd

We welcome comments, suggestions, and contributions.

2. Overview

This section uses a small example to demonstrate the overall usage model of Troyd. Suppose we want to use Troyd to test pandora [8], a personalized radio service that streams music on phones. Figure 1 shows an example Troyd test script for pandora.

In Troyd, tests are written in Ruby using the standard Ruby testing framework, which is loaded on line 1 and inherited from on line 2. Troyd test cases must also include the TroydCommands class (line 3), which provides the high-level commands used to test the app.

The testing scenario begins with the test setup (line 11), which launches the app under test by sending an Intent to its launcher Activity (the appropriate Activity name is automatically detected by Troyd's recording script). Then the test case clicks on a button to agree with the TOS (line 15); signs in (lines 16– 19); creates a station for a certain artist (line 21); signs out and in again (lines 26–30); checks if the station was correctly created (line 31); and then removes that station (line 33)

```
1  require 'test/unit'
2  class TroydTest < Test::Unit::TestCase
3    include TroydCommands
4
5    def assert_text(txt)
6      found = search txt
7      assert(found)
8    end
9
10   def setup
11     ADB.ignite "com.pandora.android.Main"
12   end
13
14   def test_station
15     click "Continue"
16     click "I have a Pandora account"
17     edit(0, "account@...")
18     edit(1, "password...")
19     click "Sign In"
20     menu
21     click "Create"
22     edit(0, "Antonio Salieri")
23     click "Search"
24     menu
25     click "Preferences"
26     click "Sign Out"
27     click "I Have a Pandora account"
28     edit(0, "account@...")
29     edit(1, "password...")
30     click "Sign In"
31     assert_text "Antonio Salieri"
32     clickLong "Antonio Salieri"
33     click "Delete"
34   end
35
36   def teardown
37     acts = getActivities
38     finish
39     puts acts
40   end
41 end
```

Figure 1. Ruby testing script generated by Troyd

to make this test script repeatable. The teardown code (lines 36–40) gets the list of visited activities, stops the app, and then prints the list of activities. (As usual, the same setup and teardown code could be used for multiple test cases, but here we have just one test case.)

Recording Test Cases We could write the test case above manually, but it is often more convient to use Troyd's *test recording* capability. To start the recording phase, the tester runs Troyd's rec.rb script with the app under test:

```
.../troyd $ bin/rec.rb pandora.apk
... some messages about building Troyd, etc.
>
```

At this point, Troyd has started the emulator (if no real device is connected) and launched the app. Now testers can enter Troyd commands as they observe the running app, e.g.,

```
> click "Continue"
"click: Continue"
> click "I have a Pndora account"
"click: can't find: I have a Pndora account"
> click "I have a Pandora account"
"click: I have a Pandora account"
...
```

After conducting the test scenario of interest, testers can name that scenario via a Troyd command:

```
> sofar "station"
```

Upon receiving this command, Troyd restarts the app under test and collects the Troyd commands typed so far as a single test case. After repeating these steps, testers can stop the recording phase via a Troyd command:

```
> finish
```

Then, Troyd terminates the app under test and generates a Ruby testing script like Figure 1. Specified scenarios become distinct test cases, and all other supporting code, e.g., the setup and teardown of the target app, are automatically generated.

Additional features Once such testing scripts are generated, testers can insert assertions (e.g., line 31) to check the invariants of interest, and then repeat the same test scenarios on multiple devices or use them as regression tests as apps are updated. Troyd supports not only simple assertions like assert_text (line 5), but also complicated assertions, e.g., assert_ads, which checks if ads-related views are on the screen, or assert_died, which checks whether the app has finished when intentially driven to do so.

Ideally, Troyd would provide statement, branch, condition, etc. coverage metrics, but to our knowledge Android can only measure these when source code is available [1]. In lieu of this more detailed information, Troyd records the list of visited Activitys during the tests (line 37), which provides some limited coverage information.

3. Design

The Troyd distribution is divided into the following directories:

`apks/`	apps under test
`bin/`	main scripts
`testcases/`	generated test cases
`tools/`	external tools
`troyd/`	the controller app

The `troyd/` directory contains an app that runs on the device, acting as a controller, and the `bin/` directory includes Ruby scripts that provide several features, e.g., wrappers for Android tools, such as adb [4] (line 11 in Figure 1). The `apks/` and `testcases/` directories contain the target apps and test cases generated by Troyd. The `tools/` directory includes other open source tools, e.g., signapk from the Android source, which Troyd uses to resign the apps.

Controller App To manage the app under test, Troyd sends commands to a special controller app that runs on the mobile device and uses Instrumentation and robotium's APIs to control the app under test. (robotium provides a richer API layered on top of Instrumentation.) The controller app includes an IntentService that receives an Intent containing the target app's package name and then launches that target app. (Troyd sends this and other Intents using adb.) After starting the target app, the controller app registers a BroadcastReceiver that receives Intents containing Troyd commands, which are then carried out.

After exploring several alternatives, we settled on this design because of some restrictions in Android: only a Service or an Activity can start Instrumentation, and the am command of adb is the only way to send an Intent to the device or emulator. Unfortunately, sending an Intent using adb always creates a new process, but Instrumentation can only control an app running in the same process. As a result, we needed an intermediate component that must be neither a Service nor an Activity, but should be able to receive an Intent. Hence, our choice of a dynamically registered BroadcastReceiver.

Scripts The `bin/` directory includes the following scripts, in order of discussion:

`aapt.rb`*	wrapper for aapt
`troyd.rb`	wrapper for the controller app
`resign.rb`*	wrapper for signapk
`avd.rb`	handler for Android virtual device
`adb.rb`	wrapper for adb
`cmd.rb`	Troyd commands
`rec.rb`*	test recorder
`trun.rb`*	test runner

To run the target app on top of Instrumentation, the manifest file of the controller app must specify the package name of the target app. Script `aapt.rb` extracts that information without unpacking the target app. Using the package name, script `troyd.rb` revises the manifest file of the controller app, and then rebuilds it. Script `resign.rb` is used to resigns the controller app and the test app to have the same key, which is required to use Instrumentation. Script `avd.rb` creates and removes an Android virtual device if no real device is connected.

Script `adb.rb` provides features to install and uninstall apps, and to communicate with the emulator or device. Script `cmd.rb` implements the high-level testing commands used inside of test scripts. Script `rec.rb`, which calls all the scripts mentioned so far, prepares the recording phase and records tester commands given through the command-line interface. Finally, script `trun.rb` is used to rerun test cases.

Scripts with asterisk symbols are executable, and the other scripts are generally only used internally. Note that although `aapt.rb` and `resign.rb` could be used by themselves, we expect testers only need to run `rec.rb` and `trun.rb` directly.

Limitations Since instances of Instrumentation can only control app components in the same process, Troyd cannot test some behavior, e.g., if the app launches the browser, Troyd loses control at that point since the browser will run in a different process. The biggest drawback of Troyd is its speed: every time a user types a new command, Troyd performs the corresponding action and waits until it takes effect in the app. This synchronization is required to guarantee test integrity, since the scripts and the controller app actually communicate asynchronously via adb. In practice, we have found the delay is a reasonable price to pay for Troyd's other benefits.

```
1  require 'test/unit'
2  class TroydTest < Test::Unit::TestCase
3    include TroydCommands
4
5    def setup
6      ADB.ignite "pkg_name_here"
7    end
8
9    def test_name
10     ...
11   end
12
13   def teardown
14     finish
15   end
```

Figure 2. Troyd test script structure

4. Troyd User Manual

To get started, install the Android SDK[1] and set environment variable $ANDROID_HOME to point to it. Be sure that paths to tools and platform-tools in the Android SDK are set to use the appropriate Android base tools (e.g., aapt, adb, android, etc.). Troyd requires Ruby, RubyGems[2] (a Ruby package manager), and Nokogiri[3] (an XML library, used to manipulate the controller's manifest file).

Testing Scripts Testers can write their own testing scripts without using Troyd's recording feature, as long as such scripts conform to the structure in Figure 2. As Troyd's test runner is built on top of Ruby unit testing framework, the testing script should be a subclass of Test::Unit::TestCase (lines 1–2), and then implement its member functions, setup (line 5) and teardown (line 13), which are run before and after every test. We recommend starting (line 6) and finishing (line 14) the app under test in setup and teardown, respectively, and performing one testing scenario per each test_* function (line 9). To use Troyd's commands, the test script should include the TroydCommands class (line 3).

Commands Troyd provides the following commands in test scripts:

- getViews : () → Array<String>

  This command returns all the View elements on the current screen.

- getActivities : () → Array<String>

  This command returns the list of Activitys visited thus far. This may be useful to check how many Activitys the current test cases cover.

- up : () → String
- down : () → String

  These commands scroll the screen up or down, and return a string containing the conducted command, "up" or "down," if successful. All the commands below whose return type is String have similar behaviors: returning the attempted command (along with parameters) as a string to indicate its success, or nil for failure.

- back : () → String
- menu : () → String

  These commands click the back and menu hardware keys.

- edit(i, "text") : (Fixnum, String) → String

  This command writes the given "text" into the i-th EditText object on the screen, where index i starts from zero.

- clear("regex") : String → String

  This command erases the text of the EditText whose text matches the given Java regular expression.

- click("regex") / clickLong("regex") : String → String

  These commands perform a short or long click on the object whose text matches the given Java regular expression. These can be applied to objects of type TextView, Button, MenuItem, RadioButton, CheckBox, ToggleButton, CheckedTextView, and CompoundButton.

- clickIdx(i) : Fixnum → String

  This command clicks the i-th TextView object in a ListView.

- clickImg(i) : Fixnum → String

  This command clicks the i-th ImageView or ImageButton object on the screen.

- clickItem(i, j) : (Fixnum, Fixnum) → String

  This command clicks the j-th item of the i-th Spinner.

- clickOn("x.y") : String → String

  This command clicks the given coordinate on the screen.

[1] http://developer.android.com/sdk/index.html
[2] http://rubygems.org
[3] http://nokogiri.org

- drag("x.y", "x.y") : (String, String) → String

  This command touches the given location and drags to the new position on the screen.

- assert_text("regex") : String → nil
- assert_not_text("regex") : String → nil

  These commands check that the screen has a TextView object that matches (does not match for _not_) the given Java regular expression. As usual, these raise an assertion failure exception if not satisfied.

- assert_checked("regex") : String → nil

  This command checks that the check box whose text matches the given Java regular expression is marked. It raises an assertion failure otherwise.

- assert_died : () → nil

  This command checks whether or not the app is finished. For example, some apps pop up *terms of service* that users must agree to, and assert_died is useful to check that refusing the terms indeed stops the app. It raises an assertion failure otherwise.

- assert_ads : () → nil

  This command checks if there is an ads-related View object on the current screen. This is implemented on top of other commands: it finds AdView instances after retrieving all the View objects on the screen via the getViews command. It raises an assertion failure if no ads-related View is found.

- sofar("name") : String → nil

  This command restarts the app, and the commands recorded so far will be emitted as a test case named test_name.

- finish : () → nil

  This command stops recording; emits recorded test cases; and kills the emulator (if the recording phase used the emulator).

Recording Script `rec.rb` starts the recording phase as follows:

```
.../troyd $ bin/rec.rb target.apk [options]
```

This script generates and starts the emulator (if no real device is connected); rebuilds the controller app to specify the target app; resigns the target app to have the same key as the controller app; installs the controller and target apps on the emulator (or device); launches the target app via the controller app; and then waits for tester commands. While controlling the app by the commands described above, one can name each meaningful test case via the sofar command and finish the recording via the finish command, as mentioned earlier.

Script `rec.rb` provides the following options:

`--avd` ⟨*avd-name*⟩

Reuse the given virtual device name, instead of creating a fresh one.

`--opt` ⟨*avd options*⟩

Options that should be passed directly to avd.

`--dev` ⟨*serial number*⟩

If multiple devices are connected, Indicate which device to use.

`--no-rec`

Disable test recording mode.

For example, suppose we are generating a test case for the scenario that clicking a "Refuse" button on the TOS page of a certain app stops the app. After launching the app via Troyd and seeing the prompt (>), we would type the following commands:

```
.../troyd $ bin/rec.rb target.apk
... some messages about building Troyd, etc.
> click "Refuse"
> assert_died
> sofar "Refuse"
> finish
```

The above commands indeed stop the app by clicking "Refuse" button on the screen, and Troyd places the generated Ruby testing script, which includes the test case below, into the `testcase/` directory.

```
def test_Refuse
  click "Refuse"
  assert_died
end
```

Replaying Once test cases are recorded, script `rec.rb` copies the target apk file to the `apks` directory, and places the testing script in `testcases` directory, with the same package name. For each apk file, script `trun.rb` finds the script with the same package name and runs that testing script. One can replay all or part of the test cases as follows:

```
.../troyd $ bin/trun.rb [options]
.../troyd $ bin/trun.rb --only pkg1,pkg2
```

Script `trun.rb` provides the same options as script `rec.rb`, except it does not include `--no-rec`. Note that, by default, `trun.rb` sets the `-no-window` option for avd, which hides the emulator screen. Similar to the

recording phase, the test runner is able to use the real device if connected. Otherwise, the script will automatically generate a new virtual device and clean it up after finishing regression tests.

5. Related Work

robotium [12] is an open-source project that provides a richer set of APIs than Instrumentation. For instance, Instrumentation provides an API to generate a pointer event for a certain position, which is similar to monkey in that both of them use absolute coordinates. On the other hand, robotium can, given a button's label, find that label on the screen and click it by calculating its position. However, robotium has the same drawback as Instrumentation: the test scenarios are defined in advance and cannot be changed while running. As mentioned earlier, Troyd's controller app is built on top of robotium, and thus offers similarly high-level commands.

robolectric [11] is very similar to robotium in that both of them are unit testing frameworks. The only difference is that robolectric runs on the Java VM, rather than the Dalvik VM. By running on the JVM, robolectric removes part of the time-consuming testing loop, namely rebuilding the app and reinstalling it into the emulator or device. However, since robolectric runs on the JVM, it does not have the full Android system available. Instead, it includes reimplementations of some important system libraries and returns `null` values for calls to unimplemented APIs. Thus, while useful, robolectric cannot fully test apps.

nativedriver [7] is an Android version of selenium [13], a browser automation framework. The nativedriver server package is attached to the app under test at link time. Then the augmented app is run on the emulator, and the server opens TCP sockets so that users can control the app. This is similar to Troyd's controller app, but nativedriver's approach requires the app to have INTERNET permission for the TCP connections, whereas Troyd does not need any extra permissions.

TEMA [14] is a model-based Android GUI testing system. It leverages monkey [6] to simulate user interactions and has keywords similar to Troyd commands, e.g., press or tap. TEMA requires testers to manually create a *model* for the app under test, and generated results are sequences of TEMA keywords. In contrast, Troyd allows testers to perform testing scenarios while running an app, and then generates standalone testing scripts using recorded commands.

There are several commercial tools, such as Testdroid [2], Ranonex [9], FAST [10], and eggPlant [15], that support automated testing of Android apps. Such commercial tools provide more powerful recorders that can literally record tester actions, whereas Troyd requires testers to type the commands accordingly.

6. Conclusion

We introduced Troyd, a new integration testing framework for Android apps. Troyd provides a rich set of commands to simulate user interactions and to retrieve properties of interest at runtime. Troyd's recording capability allows developers and users to write their own test scenarios while using the app under test. The resulting generated tests can be used as regression tests or compatibility tests.

Acknowledgments

This research was supported in part by NSF CNS-1064997 and by a research award from Google.

References

[1] EMMA: a free Java code coverage tool. http://emma.sourceforge.net/.

[2] Bitbar. Automated Testing Tool for Android - Testdroid. http://testdroid.com/.

[3] J. Constine. Google Play: 600K Apps, 1.5B Installs Per Month, 20B Total, Now With Byte-Sized Smart App Updates. TechCrunch, June 27 2012. http://techcrunch.com/2012/06/27/google-play/.

[4] Google. Android Debug Bridge, . http://developer.android.com/guide/developing/tools/adb.html.

[5] Google. Testing Fundamentals, . http://developer.android.com/guide/topics/testing/testing_android.html.

[6] Google. UI/Application Exerciser Monkey, . http://developer.android.com/guide/developing/tools/monkey.html.

[7] native driver. Native application GUI automation with extended WebDriver API. http://code.google.com/p/nativedriver/.

[8] Pandora. Pandora internet radio. https://play.google.com/store/apps/details?id=com.pandora.android.

[9] Ranonex. Android Test Automation - Automate your App Testing. http://www.ranorex.com/mobile-automation-testing/android-test-automation.html.

[10] W. River. Wind River Framework for Automated Software Testing. http://www.windriver.com/announces/fast/.

[11] Robolectric. Test-Drive Your Android Code. http://pivotal.github.com/robolectric/.

[12] Robotium. User scenario testing for Android. http://code.google.com/p/robotium/.

[13] selenium. Browser automation framework. http://code.google.com/p/selenium/.

[14] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2011)*, pages 377–386, Mar. 2011.

[15] TestPlant. eggPlant for mobile testing. http://www.testplant.com/products/eggplant/mobile/.