PLDI : U: Property-Based Randomized Test Generation for Android Applications

Peilun Zhang
University of Colorado at Boulder

Abstract

Mobile apps are stateful and interacting with both complicated framework and also environments like sensor system and network system. As a test developer, it is difficult for her to use either automated testing tools, which are fully-automated but offer low testing relevance, or scripted testing tools, which are labor-intense but address certain specific test problems, alone. In this paper, we illustrate, from the developers' perspective, how we enable a fusion of scripted and automated testing together with our programming abstractions. We enable the developer to implement user interaction sequence generators that generate sets of user interaction sequences to avoid implementing test cases one by one. We present insight-parametric generators that use developer's knowledge as the input of defining generators to efficiently conduct testing. Driven by real issues reported online, we illustrate that ChimpCheck can enable the developer to derive test suites with manageable manual efforts and still efficiently test common scenarios encountered in Android testing.

1 Problem and Motivation

Testing interactive applications (e.g, Android applications) is notoriously difficult and labor intensive due to their event-driven nature and multiple sources of inputs (e.g. network transactions, sensor updates). Sufficiently exercising the applications (apps) requires the developers to run a large suite of UI event sequences to test their apps' ability to handle the multitude of asynchronous events dictated by the users (e.g. button clicks but also screen rotations). [1]

Consider the problem of testing a chat service app as shown in Figure 1. A user story for this app is: (1) Type in some random strings to see if the authentication system handles invalid input (2) Type in a test account and password into the username and password text boxes, respectively; (3) Click on button `LOG IN`; and (4) Type in some text and send it to one of her friends.

This description captures the main features that a test developer want to validate, but an effective test suite should need more than corresponding test case to see if the app is robust in handling this flow. For example, the developer needs to test both valid and invalid scenarios (e.g. a correct pair of test account and password, and another one of random test account and password). In addition, as Android apps are interacting with both the complicated event-driven Android framework and an environment ranging from user actions to cloud servers, a different ordering of sequential events can lead to a totally different, sometimes erroneous state. Even in a controlled testing environment, developers still need to exercise their apps with sufficiently large suites of user-interactions events to feel confident in handling the asynchronous events that might get triggered by real users. However, developing such test cases one-at-a-time in popular scripted testing tools (like Espresso [3]) is too time-consuming.

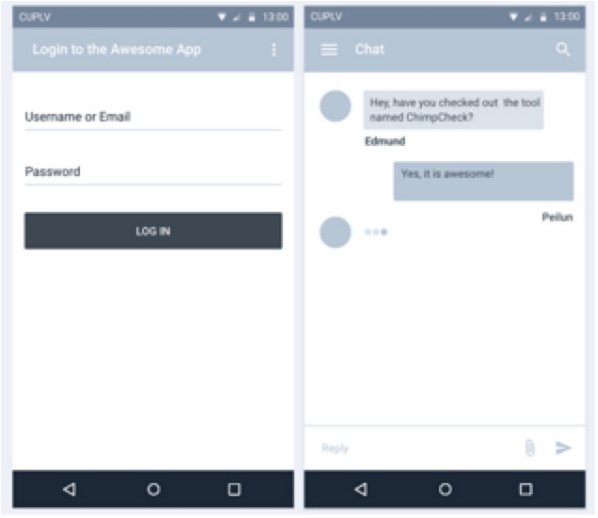

Figure 1: Testing a chat app requires, for example, (1) fusing fixture-specific flows with random interrupts and (2) asserting app-specific properties.

The alternative choice is the automated testing tools, which require little manual effort. The automated tools generate events whether randomly or based on some strategies to infer user interaction events. However, the automated test-generation techniques will fail at the login page for obvious reasons: they cannot proceed without guessing the correct combinations of authentic credentials. While there are many scripted or automated tools, there is no direct support from either scripted or automated testing tools to link the others with themselves. For developers, trying to make them work together is often difficult and error-prone.

This work presents a way to enable a fusion of scripted testing tools and automated testing tools.

```
1  val signinThenRandomTest =
```

[1] We have published a paper [5] on this project after submitting the early-stage ideas to the SRC. In this work, we present, from the developers' perspective, how to use the programming abstractions we implement in the Android platform based on the domain-specific language to expedite implementing test suites and also how to effectively address common issues in Android testing.

```
2    Type(R.id.username,"test_username") :>>
3    Type(R.id.password,"test_password") :>>
4    Click(R.id.login) :>> monkey 500
```

The ChimpCheck code above gives a basic illustration of how we fuse scripted testing and automated testing. Here, `Type` and `Click` in ChimpCheck describe that the user types in some text and click on some view, respectively. In this case, the user interface widgets are matched with the XML identifier built in the Android apps. `:>>` is the combinator that basically sequences two atomic user interaction. `monkey 500` is our implementation of mimicking the Android Monkey Exerciser that generates 500 random user interactions to exercise on the target apps.

In this work, we present this fusion of leveraging the high-level, embedded domain-specific language for defining generators that simulate user-interaction event sequences. As a proof of concept, we develop an Android-specific test runner and conduct experiments to exercise several open-source Android apps from GitHub. From the developers' perspective, we show how our testing tool enables the developers to effectively express test patterns and define sets of user interaction sequences, potentially infinite sets of sequences, with manageable manual effort. To summarize, we make the following contributions:

- Based on a domain-specific language of user-interaction event sequences (UI sequences), we reifies user interactions into syntax objects that developers can manipulate and also enable developers to conduct property-based test generations with our high-level programming abstractions.
- We lift scripting UI sequences to enable developers to write UI sequence generators that generate sets of sequences, potentially infinite sets, to exercise apps instead of implementing test cases one at a time.
- Having observed that developers' knowledge can help improve random testing significantly, we define insight-parametric generators that enable developers to embed their human insights into UI sequence generators. In this way, we allow developers to control generators to exercise app in a more relevant and effective way.

2 Uniqueness of the approach

2.1 Script User Interaction Sequences

Following is a sample user interaction sequence coded in ChimpCheck that tests the example chat app mentioned in Section 1.

```
1  val signinTraces =
2    Type(R.id.username,"test_username") :>>
3    Type(R.id.password,"test_password") :>>
4    Click(R.id.login) :>>
5    Type(R.id.chat_text, "How do you do?") :>>
6    Click(R.id.send_text)
7  forAll(signinTraces) {
8    trace => trace chimpCheck {
9      Assert(isDisplayed("How do you do?"))
10   }
11 }
```

This ChimpCheck code above simulates a user to type in her username and password to log in the app. After logging in, it types in "*How do you do*" in the text field and clicks on the "send" button. Here, `R.id.xxx` is the XML identifiers used in the Android framework to identify user interface widgets. For example, `R.id.username` matches the EditText widget on the login screen. The user-interaction events like `Click` and `Type` specify the corresponding user actions in a UI sequence. Those user-interaction events are connected with `:>>` that sequences two user-interaction events. `isDisplayed` accesses the current views on screen to see if the UI widget (e.g, texts, buttons) is shown on the screen and `Assert` checks if the properties hold.

As shown, ChimpCheck provides a high-level programming abstraction for test developers to write user interaction events as concrete data structures. In this way, it provides the basis for utilizing property-based testing on interactive apps. Thus, ChimpCheck enables test developers to concisely express properties of user interaction sequences that they desire (e.g., how they want to test the app) and check app-specific properties (e.g. whether the button b is displayed or not).

2.2 Script Test Generators Instead of individual Test Cases

As we mention in Section 1, there are many asynchronous events (e.g. network transition, background counting down timer) that can happen at any time under the Android framework. A common problem in Android apps is the failure to handle the activity suspensions, (e.g. users exit the app, users reload the activity causing it to pause and resume). This failure often causes the app to crash with an `IllegalStateException` if the app does not make the correct assumption on asynchronous thread tasks. Failing to cancel the takes which could access attributes of activities at the wrong state (e.g. activities are paused). In addition, if a UI object, which does not survive the suspension and is not restored by the resumption, is accessed, it raises a `NullPointerException`.

For test developers, they have to develop test suites that include sufficient test cases that exercise apps' abilities to handle suspension and resumption. As previously stated, test suites must provide coverage at critical point of the apps, especially when there are asynchronous thread tasks.

With ChimpCheck, the development of sufficient test suites can be simplified by defining a new combinator that inserts suspend and resume events in addition to se-

```
def g_intr = Rotate <+> ClickHome <+> ClickMenu
def *>>(g_1: Gen[UIEvents], g_2: Gen[UIEvents]) = {
  val m: Integer = 3
  g_1 :>> repeat m g_intr :>> g_2
}
```

Figure 2: The Interruptible Sequencing Combinator is defined in terms of the ChimpCheck core language. `g_1` and `g_2` are two generators that generates user interactions events. The combinator sequences `g_1` and `g_2` and injects m interrupt events generated from the interrupt generator `g_instr`.

quencing to help developers test app's ability to handle suspend and resume operations.

Figure 2 shows how we define this combinator `*>>`, called interruptible sequencing combinator. We first define the interrupt generator `g_instr` that generates a random device system events (`<+>` means non-deterministic choosing between the two events). The events it generates have the same features that they all force the app to suspend current activity. The interruptible sequencing combinator `*>>` then inject those system events in between two UI sequence generator, which means that it sequences the two generators but after the first one is executed, it allows some system events to happen. This combinator takes one parameter m, which is the number of interrupt event that we want to trigger. We currently treat it optionally and set it to 3 by default as we have observed that in practice, two consecutive interrupt events are typically able to observe erroneous behaviors.

On execution of a ChimpCheck script that is coded with this interruptible sequencing combinator, random system events get injected so as test developers define the test script, she is actually coding **a set of UI sequences** instead of one single UI sequence. Actually, a single user interaction action (e.g. `Click("Button")`) is lifted automatically to a generator which is a set that only contains this single action. In contrast to defining the same test suites with scripted testing tools like Espresso, the developer now does not need to code test cases one by one but script one test generator and let it generate the test suite she wants.

Case study We use one open source apps, the Nextcloud Android app[2], as an example to show how we develop test generators. An issue reports[3] a crash of the Nextcloud app during a file selection routine when the user changes the orientation of the app (e.g. from portrait to landscape). The crash is the result of a null pointer exception on a reference to the file selector object, caused by the wrong assumption on the liveness of this object after suspension. Here we have developed test generators for Nextcloud to reproduce this crash with the following UI sequence generator.

```
⟨Login Sequence⟩ *>>
LongClick("Nextcloud.mp4") *>> Click("Move") *>>
⟨Other Operations⟩
```

The ChimpCheck code above describes a generator that types in the correct user login credentials, select one of the files in the list shown and press "Move" in the options menu and perform some other operations. In between two atomic user interaction events, random interrupt events are injected at the run time. Test developer use `*>>` to specify the points that interrupt events can happen and thus, by writing this one user interactions generators, a set of testing sequences are defined that add different interrupt events upon the basic functionalities testing. It is worth noting that one recent work [1] has also identified injecting interrupt events critical to testing apps. The advantage of our approach is that `*>>` , as a derived combinator, provides a basic demonstration of how more complicated test generators that satisfy test developers' testing target can be implemented in ChimpCheck and it is obvious to see that the implementation can be done easily with our programming abstractions.

2.3 Integrating Randomized Testing by Defining Generators

While writing scripts are often necessary for achieving the highest possible coverage, randomized testing techniques are also important and an effective means in practice for providing basic test coverage. Wide-used scripted testing tools like Espresso and Robotium provide little, if any, support for integrating with these test generation techniques. In this section, we present how randomized testing techniques can be integrated into the library easily.

Figure 3 shows two implementations of generators for random UI event sequences. The first one, named monkey, is similar to the Android Monkey Exerciser. It is built upon a generator named monkeyStep that generates random events that perform actions on random coordinates (`g_XY` is a generator that generates random XY coordinates) on the screen. What monkey does is basically generating n random user interactions that randomly perform some user actions on the screen. However, since it performs actions randomly based on locations, it may generate a lot of useless events like clicking on something that is not clickable. We observe that it may not be efficient to use monkey generator and we define the other generator called relevantMonkey. Instead of using locations, relevantMonkey use the wild card identifier $*$ provided with the language that infers from the view

[2]Nextcloud. https://github.com/nextcloud/android.

[3]Andy Scherzinger. FolderPicker - App crashes while rotating device #448. https://github.com/nextcloud/android/issues/448. December 13, 2016.

```
def monkeyStep = {
  Click(g_XY) <+> Type(g_XY, g_Str) <+> (...)
}
def monkey(n: Integer) = {
  repeat n monkeyStep
}
def relevantMonkeyStep = {
  Click(*) <+> Type(*, g_Str) <+> (...)
}
def relevantMonkey(n: Integer) = {
  repeat n relevantMonkeyStep
}
```

Figure 3: Two implementations of the Monkey Combinator. The first (monkey) randomly applies user events on random XY coordinates on a device. The relevantMonkey is just slightly smarter that it only performs valid user interactions (e.g. click on clickable views)

UI Exerciser	Attempts	Witnessed		Steps to Bug
	(n)	(n)	(frac)	(average n)
`relevantMonkey`	30	30	1	869
Android Monkey	30	14	0.47	3545

Table 1: We applied ChimpCheck's relevantMonkey and the Android UI Exerciser Monkey to try to witness a known issue in Kistenstapeln. We ran each exerciser 30 times for up to 5,000 UI events.

hierarchy to perform relevant actions (e.g. always click on clickable buttons). With the relevantMonkey generator implemented, we enable test developers to define scripted testing and automated test together.

```
Click(R.id.enter) :>>
Type(R.id.username,"test_account") :>>
Type(R.id.password,"test_password") :>>
Click(R.id.signin) :>> relevantMonkey 50
```

The ChimpCheck above show a simple fusion of scripted testing and automated testing. It simply does the login first and later applies relevant monkey that generates 50 random but relevant user interactions events to exercise the app.

Case Study We exercise our relevantMonkey combinator and the Android UI Exerciser Monkey to try to witness a known issue[4] in Kistenstapleln-Android[5]. We ran each exerciser 10 times for up to 5000 UI events. Preliminary experiments using the relevantMonkey combinator have shown promising results presented in Table 1. For the known issue we mentioned, the relevantMonkey combinator witnesses the bug with less generated events than the Android UI Exerciser Monkey. Android Monkey also failed to witness the bug in half of the attempts up to 5000 events generated while relevantMonkey always observe the bug mentioned in

[4]Tobias Neidig. Crash on timer-event on other fragment #1. https://github.com/d120/Kistenstapeln-Android/issues/1. March 19, 2015

[5]Fachschaft Informatik. Kistenstapeln-Android. https://github.com/d120/Kistenstapeln-Android.

```
def gorilla(n: Integer, g: Gen[UIEvent]) = {
  repeat n (g:>>relevantMonkeyStep)
}
```

Figure 4: The gorilla extends the relevantMonkey by taking an extra parameter, a user-defined generator. This generator that contains user's knowledge of the app is injected before every randomly user interaction generated from relevantMonkey

the Kistenstapleln app in less than 5000 events. It is also worth noting that these promising preliminary results are achieved with our simple implementation.

2.4 Script Insight-parametric Generator to Control UI Sequence Generator

In Section 2.2, we illustrate how a test developer can easily define generators. We observe that in practice, developers often want to gain more control over automated generated tests. For example, many modern Android apps have dynamic authentication system. A user is required to log in only when she is trying to access certain resources. Such dynamic behaviors make it difficult to simply prepend a login script like what we do in Section 2.1. Instead she may want conduct login when needed and then continue to exercise apps randomly

From the relevantMonkey combinator, we derive a new combinator, named gorilla (shown in Figure 4) with the ability to inject user-scripted logic into randomized testing. We define gorilla combinator by inserting an additional argument: a generator that gets prepended before every random user interaction event. In this way, gorilla enables the user to inject her insight of the app to the script that describes the main testing flow. We call it **insight-parametric generator** because we convert developer's knowledge of apps as a generator and use it as a parameter in the gorilla generator. Insight-parametric generator helps the user to control the user interaction generators and test in a more effective way.

For our dynamic authentication example previously, we can test it by appending a hard-coded login sequence before every step of exercising it.

```
val login =
  Click("Login"):>> Type(R.id.userbox,"username"):>>
  Type(R.id.passbox,"password"):>> Click("Enter")
  gorilla 500 (isDisplayed("Login") then login)
```

Case Study We have tested our generators on an app with a dynamic authentication system, named OppiaMobile[6], a mobile learning app. We have developed test generators using the gorilla with a hard-coded login sequence (similar to the sequence above). We observed that the gorilla occasionally logs out and logs back in while randomly exercising on relevant UI

[6]Digital Campus. OppiaMobile Learning. https://github.com/DigitalCampus/oppia-mobile-android.

elements. Though no bugs were directly observed, state change between authenticated and unauthenticated are rarely tested in the normal testing scenario but it can happen a lot when a real user uses the app, to which test developers should pay attention.

3 Background and Related Work

3.1 Android UI Testing

Two largely distinct techniques make up the popular approaches to test Android apps from GUI exercising perspective: automated UI exercises and low-level script testing framework.

Built into the Android platform, Android Monkey generates and sends random user interaction events such as clicks, gestures, and system-level events. It generates a large set of events randomly with little manual effort needed. With nearly none knowledge of the apps, the Monkey is often time-consuming and exercises many irrelevant events of the apps (e.g. click on views that do not respond to click action). Advanced approaches for automated testing have been used to resolve this issues. There are tools based on model checking techniques (e.g. Android Ripper [2]), evolutionary testing techniques (e.g. Evodroid [6]), and search-based techniques (e.g. Sapienz [7]). They all use different sophisticated strategies to infer valid user interactions (e.g. click only on clickable UI element) and to increase code coverage. To use the tools mentioned above, developers often have to work around app-specific problems case by case, usually in awkward ways. For example, many automated tools, if not all, will fail at the login page for obvious reasons: they cannot proceed without guessing the correct combinations of authentic credentials. The test developers then have to modify their apps and test their apps without authentication system which makes the tested apps different from the real delivered apps.

The alternative choice is the low-level scripted testing tools (e.g. Espresso[3], Robotium [9]) that allow the test developers to manually describe the user interaction sequences (e.g. click button A then click button B). Since the developers take full control of the events executed, scenarios that require domain-specific knowledge can be easily tested or skipped (e.g. login pages, special motion gestures). But building a test suite that covers key features of the apps and meanwhile enough corner cases become more and more difficult as the complexity of the apps grows.

3.2 Property-based testing

Property-based testing has become popular and widely used in different kinds of programming scenarios since it was introduced in QuickCheck [4]. When using a property-based testing tool, the developer needs to describe the properties as predicates and the framework generate test cases to see if the properties are violated. With its popularity, there are many research efforts on property-based testing that we can leverage.

Our work here mostly utilizes ScalaCheck [8], a Scala language implementation of QuickCheck. It is the most widely used framework to automate property-based testing for Scala or Java programs. While ScalaCheck provides a lot of conveniences, its original target is to generate primitive test data (e.g. integers, floating numbers, and strings). Our work aims to generate user interactions events (e.g. click on a button, type in texts, swipe the slider bar) combined with asynchronous system events (e.g. screen rotation, suspend and resume app), which is not directly supported.

References

[1] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 83–93, New York, NY, USA, 2015. ACM.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[3] Android Developers. Testing UI for a single app. https://developer.android.com/training/testing/ui-testing/espresso-testing.html, 2016. Accessed: 2017-08-26.

[4] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279, 2000.

[5] E. S. L. Lam, P. Zhang, and B.-Y. E. Chang. Chimpcheck: Property-based randomized test generation for interactive apps. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 58–77, New York, NY, USA, 2017. ACM.

[6] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 599–609, New York, NY, USA, 2014. ACM.

[7] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 94–105, New York, NY, USA, 2016. ACM.

[8] R. Nilsson. ScalaCheck: Property-based testing for Scala. http://scalacheck.org/, 2015. Accessed: 2017-08-26.

[9] R. Reda. Robotium: User scenario testing for Android. http://www.robotium.org, 2009. Accessed: 2017-08-26.