

ChimpCheck: Property-Based Randomized Test Generation for Interactive Apps

Edmund S.L. Lam
University of Colorado Boulder, USA
edmund.lam@colorado.edu

Peilun Zhang
University of Colorado Boulder, USA
peilun.zhang@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder, USA
evan.chang@colorado.edu

Abstract

We consider the problem of generating *relevant* execution traces to test rich interactive applications. Rich interactive applications, such as apps on mobile platforms, are complex stateful and often distributed systems where sufficiently exercising the app with user-interaction (UI) event sequences to expose defects is both hard and time-consuming. In particular, there is a fundamental tension between brute-force random UI exercising tools, which are fully-automated but offer low relevance, and UI test scripts, which are manual but offer high relevance. In this paper, we consider a middle way—enabling a seamless fusion of scripted and randomized UI testing. This fusion is prototyped in a testing tool called ChimpCheck for programming, generating, and executing property-based randomized test cases for Android apps. Our approach realizes this fusion by offering a high-level, embedded domain-specific language for defining custom generators of simulated user-interaction event sequences. What follows is a combinator library built on industrial strength frameworks for property-based testing (ScalaCheck) and Android testing (Android JUnit and Espresso) to implement property-based randomized testing for Android development. Driven by real, reported issues in open source Android apps, we show, through case studies, how ChimpCheck enables expressing effective testing patterns in a compact manner.

CCS Concepts • **Software and its engineering** → *Domain specific languages; Software testing and debugging;*

Keywords property-based testing, test generation, interactive applications, domain-specific languages, Android

ACM Reference Format:

Edmund S.L. Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: Property-Based Randomized Test Generation for Interactive Apps. In *Proceedings of 2017 ACM SIGPLAN International*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Onward! '17, October 25–27, 2017, Vancouver, Canada

© 2017 Association for Computing Machinery.

ACM ISBN 978-1-4503-5530-8/17/10...\$15.00

<https://doi.org/10.1145/3133850.3133853>

Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '17). ACM, New York, NY, USA, 20 pages.

<https://doi.org/10.1145/3133850.3133853>

1 Introduction

Driving Android apps to exercise relevant behavior is hard. Android apps are complex stateful systems that interact with not only the vast Android framework but also a rich environment ranging from sensors to cloud-based services. Even given a mock environment, the app developer must drive her app with a sufficiently large suite of user-interaction event sequences (UI traces) to test its ability to handle the multitude of asynchronous events dictated by the user (e.g., button clicks but also screen rotations and app suspensions).

Industrial practice of Android app testing is largely centered around two main techniques: automated test generation via brute-force random UI exercising (Android Monkey [6]) and lower-level scripting of UI test cases (Android Espresso [5], Robotium [17]). Monkey testing has the benefit of requiring very little development effort and offers a low-effort means to discover bugs in easily accessible regions of an app (that are environment independent). However, to achieve more relevant testing, the developer often has to rely on writing customized UI test scripts. By *relevance*, we mean using application-specific knowledge to be exercise the app-under-test in a more sensible way. For instance, to test a music-streaming service app, trying one failed login attempt is almost certainly sufficient. Then, to exercise the interesting part of the app requires using a test account to get past the login screen to check, for example, that the media-player behaves in a way that users expect for a music service. While rich library support (e.g., Android JUnit, Espresso, and Robotium) and IDE integration (Android Studio) can make custom UI scripting more manageable, implementing test cases one-at-a-time to cover all corner cases of an app is still a tedious and boilerplate process.

In the literature, advanced approaches for automating test generation has gained significant interest: model-based techniques (e.g., Android Ripper [3, 4], Dynodroid [12], evolutionary testing techniques (e.g., Evodroid [13]) and search-based techniques (e.g., Sapienz [14]). While each of these techniques easily out performs pure random techniques, the development of all of these techniques have almost been entirely focused on pure automation. Little attention has been

given to developing techniques that simplifies programmability and allowing higher-levels of customizability that empowers the test developer to inject her app-specific knowledge into the test generation technique. As a result, while these techniques offer effective automated solutions for testing generic functionality, they are unlikely to replace manual UI scripting because of their omission of the test developer’s human insight and app-specific knowledge. Pure automation is unlikely to create generators, for example, imbued with knowledge to get pass a dynamic single-sign-on page.

A New Paradigm for UI Testing The premise of this paper is that we cannot forsake human insight and app-specific knowledge. Instead, we must fuse scripted and randomized UI testing to derive *relevant* test-case generators. While improving and refining automated test generation techniques is indeed a fruitful endeavor, an equally important research thread is developing expressive ways to integrate human knowledge into these techniques.

As an initial demonstration of this fused approach, we present ChimpCheck, a proof-of-concept testing tool for programming, generating, and executing property-based randomized test cases for Android apps¹. Our key insight is that this tension between less relevant but automated and more relevant but manual can be eased or perhaps even eliminated by lifting the level of abstraction available to UI scripting. Specifically, we make *generators* of UI traces available to UI scripting, and we then discover that a brute-force random tester can simply be expressed as a particular generator. From a technical standpoint, ChimpCheck introduces property-based test case generation [8] to Android by making user-interaction event sequences (i.e., UI traces) first-class objects, enabling test developers to express (1) properties of UI traces that they deem to be relevant and (2) app-specific properties that are relevant to these UI traces. From a test developer’s perspective, ChimpCheck provides a high-level programming abstraction for writing UI test scripts and deriving app-specific test generators by integrating with advance test generation techniques—all from a single and simple programming interface. Furthermore, regardless of the underlying generation techniques used, this integrated framework generates a unified representation of relevant test artifacts (UI traces and generators), which effectively serves as both executable and human-readable specifications. To summarize, we make the following contributions:

- We formalize a core language of user-interaction event sequences or *UI traces* (Section 3). This core language captures what must be realized in a platform-specific test runner. In ChimpCheck, the execution of UI traces is realized by the ChimpDriver component built on top of Android JUnit and Espresso. This formalization provides the foundation for generalizing property-based

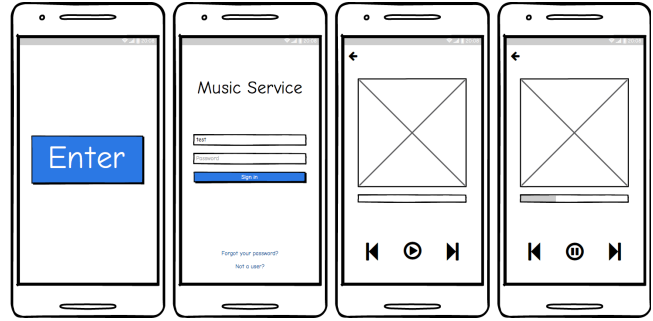


Figure 1. Testing a music service app requires, for example, (1) fusing fixture-specific flows with random interrupts and (2) asserting app-specific properties.

randomized test generation for interactive apps to other user-interactive platforms (e.g., iOS, web apps).

- Building on the formal notion of UI traces, we define *UI trace generators* that lifts scripting user-interaction event sequences to scripting sets of sequences—potentially infinite sets of infinite sequences, conceptually (Section 4). This lifting enables the seamless mix of scripted and randomized UI testing. It also captures the platform-independent portion of ChimpCheck that compiles to the core UI trace language. This component is realized by building on top of ScalaCheck, which provides a means of sampling from generators.
- Driven by case studies from real Android apps and real reported issues, we demonstrate how ChimpCheck enables expressing customized testing patterns in a compact manner that direct randomized testing to produce relevant traces and targets specific kinds of bugs (Section 5). Concretely, we show testing patterns like interruptible sequencing, property preservation, randomized UI exercising, and hybrid approaches can all be expressed as derived generators, that is, generators expressed in terms of the core UI trace generators.

In Section 7, we comment on how the ChimpCheck experience has motivated a vision for fused custom-scripting and automated-generation of interactive applications.

2 Overview: A Test Developer’s View

In this section, we introduce our fused approach and ChimpCheck by means of an example testing scenario from the app-test developer’s perspective. The purpose of this example is not necessarily to show every feature of ChimpCheck but rather to demonstrate the need to fuse app-specific knowledge with randomized testing.

Consider the problem of testing a music service app as shown in Figure 1. Testing this app requires using test fixtures for getting past user authentication and asserting properties of the user interface specific to being a music player. For concreteness, let us consider testing a particular user



¹ChimpCheck is available at <https://github.com/cuplv/ChimpCheck>.

```



1  val signinTraces =
2  Click(R.id.enter) *>>
3  Type(R.id.username, "test") *>>
4  { Type(R.id.password, "1234") *>>
5    Click(R.id.signin) *>>
6    assert(isDisplayed("Welcome")) } <+>
7  { Type(R.id.password, "bad") *>>
8    Click(R.id.signin) *>>
9    assert(isDisplayed("Invalid Password")) }
10
11 forAll(signinTraces) {
12   trace => trace.chimpCheck()
13 }

```

Figure 2. ChimpCheck focuses test development to specifying the skeleton of user interactions—here, the valid and invalid sign-in flows for the music service app from Figure 1.

story for the app: (1) Click on button Enter; (2) Type in test and 1234 into the username and password text boxes, respectively; (3) Click on button Sign in; and (4) Click on buttons  and  to try out starting and stopping the music player, respectively. Observe that the first few steps (Steps 1–3) describe setting up a particular test fixture to get to the “interesting part of the app,” while the last step (Step 4) finally gets to testing the app component of interest.

This description captures the user story that the test developer seeks to validate, but an effective test suite will likely need more than one corresponding test case to see that that this user-flow through the app is robust. With ChimpCheck, the test developer describes essentially the above script, but the script can be fused with fuzzing and properties to specify not a single test case but rather a family of test cases. For example, suppose the test developer wants to generate a family of test cases where

- A. The sign-in phase (Steps 1–3) is robust to interrupt events such as screen rotations.
- B. The state of  and  buttons in the user interface corresponds in the expected manner to the internal state of an `android.media.MediaPlayer` object.

A. Fusing Fixture-Specific Flows with Interrupts Testing requires sampling from valid and invalid scenarios. While invalid sign-ins are easily sampled by brute-force random techniques, the most reasonable means of testing valid sign-ins is to hard-code a test fixture account. While we would like a concise way of expressing such fixtures, we also want some way to generate variations that test the sign-in process with interrupt events (e.g., screen rotate, app suspend and resume) inserted at various points of the sign-in process. Such variations are important to test because interrupts often are sources of crashes (e.g., null pointer exceptions) as well as unexpected behaviors (e.g., characters keyed into a

text box vanishes after screen rotation). Developing such test cases one-at-a-time (via Espresso or Robotium [5, 17]) is too time consuming, and most test-generation techniques (via [4, 12–14]) would not be effective at finding valid sign-in sequences.

A key contribution of ChimpCheck is that it empowers the test developer to define the skeleton of the kind of UI traces of interest. Concretely in Figure 2, we specify not only a hard-coded sign-in sequence but variations of it that include interrupt actions that an app-user could potentially interleave with the sign-in sequence.

On line 1, the test developer defines a value `signinTraces` that is a *generator* of sign-in traces where a trace is a sequence of UI events that drives the app in some way. To define `signinTraces`, the test developer describes essentially the sign-in flow outlined earlier: (1) Click on button Enter with `Click(R.id.enter)` on line 2; (2) Type in test with `Type(R.id.username, "test")` on line 3 and type 1234 with `Type(R.id.password, "1234")` on line 4; and (3) Click on button Sign in with `Click(R.id.signin)` on line 5. In Android, the R class contains generated constants that uniquely identify user-interface elements of an app. Like an Espresso test developer today, we use these identifiers to name the user-interface elements.

The user-interaction events like `Click` and `Type` specify an individual user action in a UI trace. These events can be composed together with a core operator like `:>>` that represents sequencing or, as on line 6, with the operator `<+>` that implements a non-deterministic choice between its left operand (lines 4–5) and its right operand (lines 7–9). Thus, `signinTraces` does not represent just a single trace but a set of traces. Here, we union two sets of traces to get some valid sign-ins (lines 4–6) and some invalid ones (lines 7–9). As the test developer, we can check for the correctness of the sign-in scenarios with an `assert` for a welcome message in the case of the valid sign-ins (line 6) or for an “Invalid Password” error message in the case of the invalid ones (line 9). The `isDisplayed` expressions specify properties on the user interface that are then checked with `assert`.

Finally, the UI events are sequenced together with the `*>>` combinator rather than the `:>>` operator. The `*>>` combinator represents interruptible sequencing, that is, sequencing with some non-deterministic insertion of interrupt events. Because these generator expressions represent sets of traces rather than just a single trace, this interruptible sequencing combinator `*>>` becomes a natural drop in for the core sequencing operator `:>>`. And indeed, interruptible sequencing `*>>` is definable in terms of sequencing `:>>` and other core operators.

ChimpCheck is built on top of ScalaCheck [16], extending this property-based test generation library with UI traces and integration with Android test drivers running emulators or actual devices. From a practical standpoint, by building on top of ScalaCheck, ChimpCheck inherits ScalaCheck’s

```
[1] Crashed after: Click(R.id.enter) :>> ClickHome
Stack trace: FATAL EXCEPTION: main, PID: 29302
java.lang.RuntimeException: Unable to start activity
...
[2] Failed assert isDisplayed("Welcome") after:
Click(R.id.enter) :>> Type(R.id.username,"test") :>>
Rotate :>> Type(R.id.password,"1234") :>>
Click(R.id.signin)
[3] Blocked after: Click(R.id.enter) :>> Rotate
```

Figure 3. Failed ChimpCheck test reports include the UI trace that leads to the crash or assertion failure.

test generation functionalities. The `signinTraces` generator is then passed to a `forall` call on line 12. The `forall` combinator comes directly from the `ScalaCheck` library that generically implements sampling from a generator. Tests are executed by invoking the `ChimpCheck` library operation `.chimpCheck()` on UI-trace samples. Here, UI trace samples are bound to `trace` and executed on line 12 (i.e., `trace.chimpCheck()`). When this operation is executed, it triggers off a run of the app on an actual emulated Android device and submits the trace `trace` (a trace sampled from `signinTraces`) to an associated test driver running in tandem with the app. When problems are encountered during each run, details of crashes or assertion failures are reported back to the user. Figure 3 shows examples of bug reports from `ChimpCheck`.

An important contribution of `ChimpCheck` is that all reports contain the actual user-interaction trace executed up to the point of failure. In an interactive application, a stack trace is severely limited because it contains only the internal method calls from the callback triggered by the last user interaction. This UI trace is essentially an executable and concise description of the sequence of UI events that led up to the failure. And thus, these executed UI traces are valuable in the debugging process as they can guide the developer in reproducing the failure and ultimately understanding the root causes.

B. Asserting App-Specific Properties Many problems in Android apps do not result in run-time exceptions that manifest as crashes but instead lead to unexpected behaviors. It is thus critical that the testing framework provide explicit support for checking for app-specific properties. To check for unexpected behaviors, the developer must write custom test scripts and invoke specific assertions at particular points of executing the Android app.

`ChimpCheck` provides explicit support for property-based testing, which enables the test developer to simultaneously express generators for describing relevant UI traces and assertions for specifying app-specific properties to check. In Figure 4, we show another UI trace generator `playStateTraces`

```
1 val playStateTraces =
2   Click(R.id.enter) :>> Type(R.id.username,"test") :>>
3   Type(R.id.password,"1234") :>> Click(R.id.signin) :>>
4   optional {
5     Click(R.id.play) *>> optional {
6       Sleep(Gen.choose(0,5000)) *>> Click(R.id.stop)
7     }
8   }
9
10 forall(playStateTraces) { trace =>
11   trace.chimpCheck {
12     (isClickable(R.id.play) ==> !mediaPlayerIsPlaying) /\
13     (isClickable(R.id.stop) ==> mediaPlayerIsPlaying)
14   }
15 }
```

Figure 4. `ChimpCheck` enables simultaneously describing the trace of relevant user interactions to drive to app to a particular state and checking properties on the resulting state. For the music service app from Figure 1, we check that the state of the \textcircled{P} (`R.id.play`) and \textcircled{S} (`R.id.stop`) toggle is consistent with the state of the `MediaPlayer` object.

for the same music-service app that focuses on testing that the UI state with the \textcircled{P} and \textcircled{S} toggle is consistent with the internal `MediaPlayer` object. Since the `signinTraces` from Figure 2 already test the sign-in process, these tests simply wire-in the test fixture to get past the sign-in screen into the music player with the plain sequencing operator `:>>` (lines 2–3). Past the sign-in screen, lines 4–8 implement the optional-cycling between the \textcircled{P} and \textcircled{S} music player states via `Clicking` the corresponding buttons. In this particular example, we insert a random idle of 0 to 5 seconds (`Sleep(Gen.choose(0,5000))`) between a cycle. Note that `Gen.choose(0,5000)` is not special to `ChimpCheck`; it is part of the `ScalaCheck` library to generate an integer between 0 and 5,000. Finally, on lines 10–15, we execute the actual test runs as before, but this time we supply a property expression (lines 12–13) comprising of two implication rules that asserts the expected consistency between the UI state and the underlying `MediaPlayer` state. Note that while `isClickable` is a built-in atomic predicate that accesses the Android run-time view-hierarchy, `mediaPlayerIsPlaying` is a predicate defined by the developer. Its implementation is simply a call to the `MediaPlayer` object’s `isPlaying` method.

3 User-Interaction Event Sequences

This section defines a core language of UI traces to understand what must be realized in a platform-specific test runner. We first discuss the syntactic constructs that make up this language (Section 3.1). Then, we present an operational semantics that gives these declarative symbols formal meaning

(Strings) $s \in \mathbb{S}$ (Integers) $n \in \mathbb{Z}$ (XY Coordinates) $XY ::= (n, n)$ (UI Identifiers) $id ::= n \mid s \mid XY$ $ID ::= id \mid *$
 (App Events) $a ::= \text{Click}(ID) \mid \text{LongClick}(ID) \mid \text{Swipe}(ID, XY) \mid \text{Type}(ID, s) \mid \text{Pinch}(XY, XY) \mid \text{Sleep}(n) \mid \text{Skip}$
 (Device Events) $d ::= \text{ClickBack} \mid \text{ClickHome} \mid \text{ClickMenu} \mid \text{Settings} \mid \text{Rotate}$ (UI Events) $u ::= a \mid d$
 (UI Traces) $\tau ::= u \mid \tau_1 :>> \tau_2 \mid \text{assert } \mathcal{P} \mid \tau ? \mid \mathcal{P} \text{ then } \tau \mid \bullet$ (Arguments) $args ::= n \mid s \mid args_1, args_2$
 (Properties) $\mathcal{P} ::= p(args) \mid !\mathcal{P} \mid \mathcal{P}_1 \Rightarrow \mathcal{P}_2 \mid \mathcal{P}_1 \wedge \mathcal{P}_2 \mid \mathcal{P}_1 \vee \mathcal{P}_2$

Figure 5. A language of UI events, UI traces, and properties. UI events u reify user interactions, and UI traces τ reify interaction sequences.

by defining the interactions between UI traces and an abstraction of the device and app in test (Section 3.2). This reification of implicit user interactions into explicit UI traces is crucial for fusing scripting and randomized testing. We finally discuss how an instance of these semantics are realized for testing Android apps (Section 3.3).

3.1 A Language of UI Traces and Properties

Figure 5 shows the syntax of the language of UI traces and properties. Basic terms of the language comprise of strings of characters (s) and integers (n). Specific points (XY coordinates) of the device display that renders the app are expressed by a 2-tuple of integers. UI elements can be identified (ID) with an integer identifier n , the display text of the element s , or the element’s XY coordinate. Additionally, our language includes a *UI element wild card* $*$, which represents a reference to some existing UI element without committing to a specific element. Existing testing frameworks for Android apps (e.g., Robotium and Espresso) enable developers to reference particular UI elements using integer identifiers (generated constants found in the R. `id` module of an Android app), display text, and XY coordinates. The UI element wild card $*$ is the lowest level or simplest example of fusing scripting and test-case generation (see Section 3.3 for further details).

App events (a) represent the UI events that a user can apply onto an app while staying within the app. Device events (d) are interrupt events that potentially suspends and resumes the app. UI traces τ are compositions of such events with some richer combinators: our example earlier introduced sequencing $\tau_1 :>> \tau_2$ and assertions `assert \mathcal{P}` , informally the try operator `$\tau ?$` represents an attempt on τ that will not halt the test (unless the app crashes or violates an assert), `\mathcal{P} then τ` represents a trace τ guarded by a property \mathcal{P} , while `\bullet` is the unit operator. The language of properties is a standard propositional logic with the usual interpretation. Predicates p can be either user-defined predicates (e.g., `MediaPlayer.isPlaying` in Figure 4) or built-in predicates that maps to known test operations provided by the Android framework (e.g., `isClickable`, `isDisplayed`). We discuss implementation of these predicates in Section 3.3.

(Device States) Φ (Crash Reports) \mathcal{R}
 (Oracles) $\Phi \vdash_{\text{enabled}} u \rightsquigarrow \Phi'$ $\Phi \vdash_{\text{disabled}} u$ $\vdash_{\text{idle}} \Phi$
 $\Phi \vdash_{\text{prop}} \mathcal{P}$ $\Phi \rightsquigarrow_{\text{mutate}} \Phi'$ $\Phi \rightsquigarrow_{\text{crash}} \mathcal{R}$
 (Results) $\omega ::= \text{succ} \mid \text{crash } \mathcal{R} \mid \text{fail } \mathcal{P} \mid \text{block } u$
 (Exec Events) $o ::= u \mid \bullet$ (Exec Traces) $\tau^o ::= o \mid \tau_1^o :>> \tau_2^o$

Figure 6. Oracle judgments abstract application and device-specific transitions.

What is critical here is not, for example, the particular app and device events but rather the reification of user interactions into UI traces τ .

3.2 A Semantics of UI Traces and Properties

Our main focus in this subsection is the operational semantics of a transition system we call Chimp Driver, that interprets UI events and submits commands to invoke the relevant UI action. That is, these semantics give an interpretation to reified user interactions.

Apps are complex, stateful event-driven systems. To abstract over the particulars of any particular app and its complex interactions with the underlying event-driven framework, we define Chimp Driver in terms of several *oracle* judgments (shown in Figure 6). These oracle judgments are then realized in an implementation on a per platform basis. In Section 3.3, we discuss how we realize these oracle judgments for Android apps. The device state Φ represents the consolidated state of the app and the (Android) device. The oracle judgment form $\Phi \vdash_{\text{enabled}} u \rightsquigarrow \Phi'$ describes a transition of the device state from Φ to Φ' on seeing UI event u . Recall that UI events u are primitive symbols of UI traces and they are ultimately mapped to relevant actions on the UI (e.g., `Click(ID)` to click on view with identifier ID). An event u is said to be *concrete* (denoted by `concrete(u)`) if and only if u does not contain an argument with the UI element wild card $*$ (i.e., all ID s are *ids*). The *enabled* oracle is only defined for concrete events. The judgment $\Phi \vdash_{\text{disabled}} u$ holds if an event u is not applicable in the current state Φ . Similarly, it is only defined for concrete events. We assume that these two judgments are mutually exclusive, formally:

$$\forall \Phi, \Phi', u \text{ (concrete}(u) \wedge \Phi \vdash_{\text{enabled}} u \rightsquigarrow \Phi' \Rightarrow \Phi \not\vdash_{\text{disabled}} u)$$

$$\forall \Phi, \Phi', u \text{ (concrete}(u) \wedge \Phi \vdash_{\text{disabled}} u \Rightarrow \Phi \not\vdash_{\text{enabled}} u \rightsquigarrow \Phi')$$

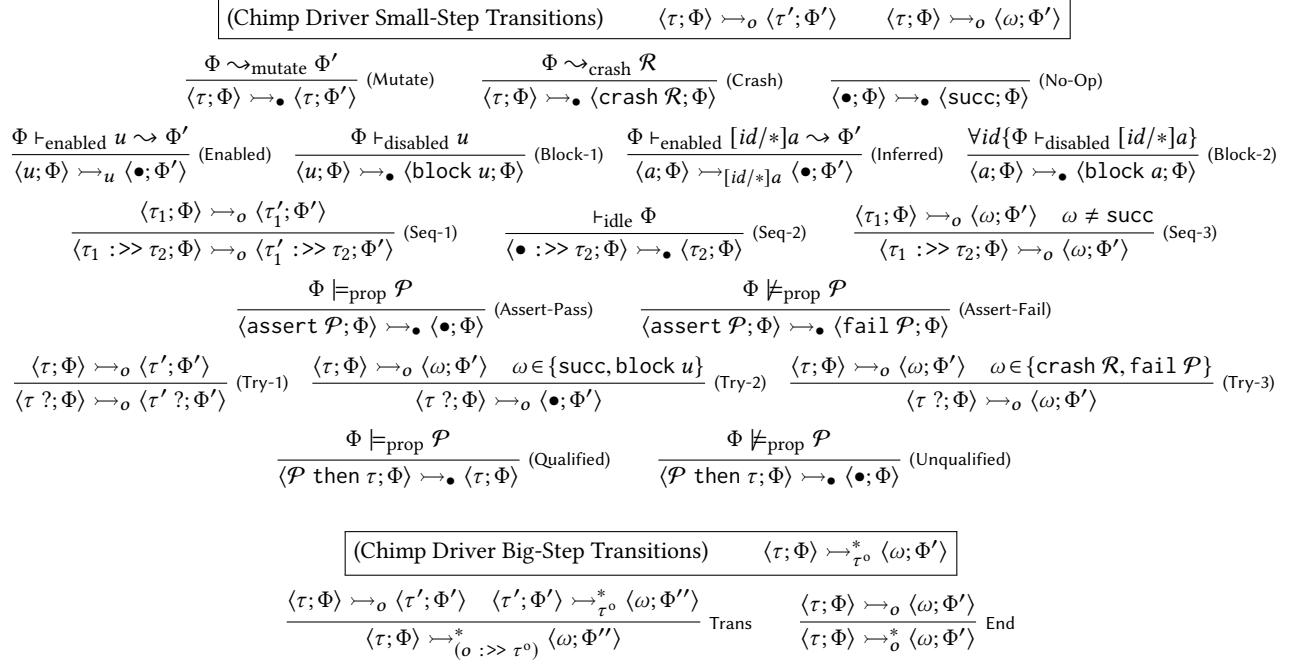


Figure 7. Chimp Driver is the operational semantics that defines the interpretation of UI traces τ in terms of the application and device-specific oracle judgments in Figure 6.

Note that $\Phi \vdash_{\text{enabled}} u \rightsquigarrow \Phi'$ is possibly asynchronous and imposes no constraints that in fact the action corresponding to u has been executed. For this, we rely on the judgment $\vdash_{\text{idle}} \Phi$, that holds if the test app's main thread has executed all previous actions and is idling. The judgment $\Phi \models_{\text{prop}} \mathcal{P}$ holds if a given property \mathcal{P} holds in the current state Φ . Since \mathcal{P} is a fragment of propositional logic, its decidability depends on the decidability of the interpretations of predicates p . Since the language of properties is standard propositional logic, we omit detailed definitions. The judgment $\Phi \rightsquigarrow_{\text{mutate}} \Phi'$ describes a transition of the device from Φ to Φ' that is not a direct consequence of Chimp Driver operations. This corresponds to background asynchronous tasks that can be either part of the test app or other running tasks of the device. Note that $\vdash_{\text{idle}} \Phi$ does not imply that $\Phi \rightsquigarrow_{\text{mutate}} \Phi'$ is not possible but simply that the state of the main thread appears idle. This interpretation, of course, ultimately results in certain impreciseness in the reports extracted by Chimp Driver (as certain race conditions are indistinguishable), but such is a known and expected consequence of UI testing. Finally, $\Phi \rightsquigarrow_{\text{crash}} \mathcal{R}$ observes a transition of the device to a crash state, with a crash report \mathcal{R} (conceptually, a stack trace from handling the event the ends in a crash).

The state of the Chimp Driver is the pair $\langle \tau; \Phi \rangle$ comprising of the current UI trace τ and current device state Φ . Results ω are terminal states of this transition system and come in four forms: `succ` indicates a successful run, `crash \mathcal{R}` indicates a crash with report \mathcal{R} , `fail \mathcal{P}` indicates a failure to assert \mathcal{P} ,

and `block u` indicates that the Chimp Driver got stuck while attempting to execute event u . An auxiliary output of Chimp Driver is the actual concrete trace that was executed: τ^o is a sequencing ($:>>$) of primitive events u or the nullipotent (unit) event \bullet . It is the executed UI trace that reproduces the failure (modulo race-conditions).

Figure 7 introduces the operational semantics of Chimp Driver. Its small-step semantics is defined by the transitions $\langle \tau; \Phi \rangle \rightsquigarrow_o \langle \tau'; \Phi' \rangle$ and $\langle \tau; \Phi \rangle \rightsquigarrow_o \langle \omega; \Phi' \rangle$ that defines intermediate and terminal transitions respectively. Intuitively, τ and Φ are inputs of the transitions, while τ' , ω , Φ' together with the executed event o , are the outputs. The big-step semantics is defined by the transition $\langle \tau; \Phi \rangle \rightsquigarrow_{\tau^o}^* \langle \omega; \Phi' \rangle$ that exhaustively chains together steps of the small-step transitions. Outputs of the test run are τ^o , ω , and possibly observable (from the confines of the device framework) fragments of Φ' . The following paragraphs explains the purpose of each transition rule presented in Figure 7.

Mutate, Crash, and Unit The (Mutate) and (Crash) rules lift mutate and crash oracle transitions into Chimp Driver transitions. While the (Mutate) transition is transparent to Chimp Driver, (Crash) results in a terminal crash \mathcal{R} state. (No-Op) transits the unit event \bullet to a success state `succ`.

UI Events The four rules (Enabled), (Block-1), (Inferred), and (Block-2) define the UI event transitions. The (Enabled) rule defines the case when a concrete event u is successfully inserted into the device state, while (Block-1) defines the case when u is not enabled and the execution blocks. Note

that having block u as an output is an important diagnostic behavior of Chimp Driver, hence this blocking behavior is explicitly modeled in the transition system. The next two rules only apply to non-concrete events (containing $*$ as argument): the (Inferred) rule defines the case when a non-concrete event a is successfully concretized, during which the occurrence of $*$ in a is substituted by some UI identifier id (if it exists) such that the instance of a is an enabled event. This substitution is denoted by $[id/*]a$. Finally the rule (Block-2) defines the case when no enabled events can be inferred from instantiating $*$ to any valid UI identifier (i.e., all valid id 's are disabled), hence the transition system blocks.

UI Event Sequencing UI event sequencing ($\tau_1 :>> \tau_2$) is defined by three rules: the rule (Seq-1) defines a single-step transition on the prefix τ_1 to τ_1' . The rule (Seq-2) defines the case when the prefix trace is a unit event \bullet , during which the derivation can only proceed if the device is in an idle state (i.e., $\vdash_{\text{idle}} \Phi$). Finally the rule (Seq-3) defines the case when the prefix trace ends in some failure result ($\omega \neq \text{succ}$), during which the transition system terminates with ω as the final result. Importantly, note that the purpose of having $\vdash_{\text{idle}} \Phi$ as a premise of the (Seq-2) rule: this condition essentially enforces the execution of Chimp Driver *in tandem* with the test app. Particularly, this means that any event u in the prefix sequence τ_1 must have been executed before postfix τ_2 is attempted. We assume that the actual execution of these events are modeled by unspecified (Mutate) transitions and that the state of the device eventually reaches an idle state (i.e., $\vdash_{\text{idle}} \Phi$) once all pending events have been processed.

This semantics supports an intuitive, synchronous view of UI traces. For instance, consider the following UI trace:

```
assert count(0) :>> Click(R.id.cnt) :>>
assert count(1) :>> Click(R.id.cnt) :>>
assert count(2) :>> ...
```

In this example, $\text{count}(n)$ is a predicate that asserts that the UI element $R.\text{id.cnt}$ has so far been clicked n times. Notice that the correctness of this UI trace is predicated on the assumption that each $\text{Click}(R.\text{id.cnt})$ operation is actually synchronous and in tandem with the test app. This is modeled by the premise $\vdash_{\text{idle}} \Phi$ of the (Seq-2) rule, and we will discuss its implementation in the Chimp Driver for Android in the next section. The mainstream Android testing frameworks, such as Espresso and Robotium, have gone through great lengths to achieve this synchronous interpretation and publicly cite this behavior as a benefit of the frameworks.

Assertions and Try The (Assert-Pass) and (Assert-Fail) rules handle assertions. They each consult the property oracles $\Phi \models_{\text{prop}} \mathcal{P}$ and $\Phi \not\models_{\text{prop}} \mathcal{P}$ to result in a success (\bullet) or failure ($\text{fail } \mathcal{P}$), respectively. The *try* combinator ($\tau ?$) represents an attempt to execute trace τ that should not terminate the existing test (rule (Try-2)) unless τ results in a crash or

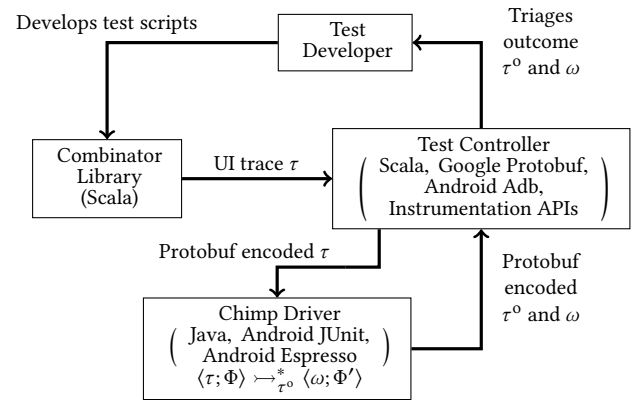


Figure 8. Implementing Chimp Driver for Android. Here, we depict the development and execution of a single UI trace τ . The Test Controller runs on the testing server, and Chimp Driver runs on the device. Both are Android-specific implementations.

failed assertion (rule (Try-3)). Rule (Try-1) simply defines intermediate transitions. From the test developer perspective, she writes ($\tau ?$) when she wants to suppress the blocking behavior within the trace τ and just move on to the rest of the UI sequence (if any). We discuss the motivation for this combinator in Section 5.3.

Conditional Events The (Qualified) and (Unqualified) rules handle cases of the combinator \mathcal{P} then τ . Informally, this combinator executes τ only if \mathcal{P} holds (i.e., $\Phi \models_{\text{prop}} \mathcal{P}$). This combinator represents the language's ability to express conditional interactions depending on the device's run-time state that are often necessary when an app's navigational behavior is not static (see Section 5.4 for examples).

3.3 Implementing Chimp Driver for Android

Here, we discuss one example instance of implementing Chimp Driver (along with the oracle judgments) abstractly defined in Section 3.2.

ChimpCheck Run-time Architecture Figure 8 illustrates the single-trace architecture of the ChimpCheck combinator library and the test driver (Chimp Driver) for Android. Here, we depict the scenario where the test developer programs single UI traces directly; in Section 4.3, we describe how we lift this architecture to sets of traces. The main motivation of our development work here is to implement a system that can be readily integrated with existing Android development practices.

The *combinator library* of ChimpCheck is implemented in Scala. Our reason for choosing Scala is simple and well-justified. Firstly, Scala programs integrate well with Java programs, hence importing dependencies from the underlying Android Java projects is trivial. For instance, it would be

most convenient if the test developer can reference static constants of the actual Android app in her test scripts, especially the unique identifier constants generated for the Android app (e.g., `Click(R.id.Btn)`). The second more important reason is Scala's advance syntactic support for developing domain-specific languages, particularly implicit values and classes, case classes and object definition, extensible pattern matching, and support for infix operators. The *test controller* module of ChimpCheck essentially brokers interaction between the test scripts written by the developer and actual runs of the test on an Android device (hardware or emulator). It is largely implemented in Scala while leveraging existing and actively maintained libraries: UI traces of the combinator library are serialized and handed off to Chimp Driver (see below) via Google's Protobuf library; communications between the test script and the Android device is managed by the Android testing framework, particularly Adb and the instrumentation APIs. The Chimp Driver module is the test driver that is deployed on the Android device. It is implemented as an interpreter that reads and executes the UI traces expressed by the developer's test script—leveraging on the Android JUnit test runner framework and the Android Espresso UI exercising framework. Most importantly, it implements the operational semantics defined in Figure 7.

Running UI Events in Tandem with the Test App As we mention in Section 3.2, for reliability of test scripts, the test driver runs in tandem with the test app. To account for this design, our operational semantics imposes the $\tau_{\text{idle}} \Phi$ restriction on sequencing transitions (rule (Seq-2) that applies to $\bullet : \gg \tau$). Our implementation realizes this semantics largely thanks to the design of the Espresso testing framework: primitive and concrete actions of our combinator library ultimately map to Espresso actions. For instance, the combinator `Click("Button1")` maps to the following Java code fragment that calls the Espresso library:

```
Espresso.onView(
    ViewMatchers.withText("Button1")
).perform(click());
```

We need not implement any additional synchronization routines because within the call to `perform`, Espresso embeds a sub-routine that blocks the control flow of the tester program until the test apps main UI thread is in an idle state. This work is essentially the implementation of the $\tau_{\text{idle}} \Phi$ condition in our operational semantics. Similarly, property assertions of our operational semantics are required to be exercised at the appropriate times, and this effect is realized by mapping our property assertions (i.e., rules (Assert-Pass), (Assert-Fail), (Qualified) and (Unqualified) for combinators `assert P` and `P then τ`) to Espresso assertions via its `ViewAssertion` library class.

Handling non-concrete events (i.e., events with the UI element wild card `*as` arguments), however, requires some

more care. To infer relevant UI elements, we need to access the app's run-time view hierarchy. In order to access the view hierarchy at the appropriate, current state of the app, our access to the view hierarchy must be guarded by a synchronization routine similar to the ones provided by the Android Espresso library.

Inferring Relevant UI Elements from the View Hierarchy We describe how we realize the rules (Inferred) and (Block-2) of the operational semantics. As noted above, implementing the inference property of the `*` combinator relies on accessing the test app's current view hierarchy. The view hierarchy is a run-time data structure that represents the current UI layout of the Android app. With it, we can enumerate most UI elements that are currently present in the app and filter down to the elements that are relevant to the current UI event combinator. For instance, for `Click(*)`, we want `*` to be substituted with some UI element *id* that is displayed and is clickable, while for `Type(*, s)`, we want a UI element *id* that is displayed and accepts user inputs. This filtering is done by leveraging Espresso's `ViewMatchers` framework, which provides us a modular way of matching for relevant views based on our required specifications. Our current implementation is sound in the sense that it will infer valid UI elements for each specific action type. However, it is not complete: certain UI elements may not be inferred, particularly because they do not belong in the test app's view hierarchy. For instance, UI elements generated by the Android framework's dialog library (e.g., `AlertDialog`) will not appear in an app's view hierarchy. Our current prototype will enumerate the default UI elements found in these dialog elements, but it does not attempt to extract UI elements introduced by user-customized dialogs.

Built-in and User-Defined Predicates Primitives of the properties of our language, particularly predicates, are implemented in two forms: *built-in* predicates are first-class predicates supported by the combinator library. For example, `isClickable(ID)` and `isDisplayed(ID)` are both combinators that accept *ID* as an argument, and they assert that the UI element corresponding to *ID* is clickable and displayed, respectively. Our current built-in predicate library includes the full range of tests supported by Espresso library's `ViewMatchers` class and their implementations are simply a direct mapping to the corresponding tests in the Espresso library.

Our library also allows the test developer to define her own predicates (e.g., `mediaPlayerIsPlaying` from Figure 4). The combinator library provides a generic predicate class that the developer can use to introduce instances of a predicate, such as

```
Predicate("mediaPlayerIsPlaying")
```

or extend the framework with her own case classes that represent predicate combinators. The current implementation

$$\begin{aligned}
& \text{(String Generators)} \mathcal{G}^S & \text{(Integer Generators)} \mathcal{G}^Z \\
& \text{(XY Coordinates)} \mathcal{G}^{XY} ::= (n, n) \mid (\mathcal{G}^Z, \mathcal{G}^Z) & \text{(UI Identifiers)} \mathcal{G}^{ID} ::= s \mid n \mid XY \mid * \mid \mathcal{G}^S \mid \mathcal{G}^Z \mid \mathcal{G}^{XY} \\
& \text{(App Events)} \mathcal{G}^a ::= \text{Click}(\mathcal{G}^{ID}) \mid \text{LongClick}(\mathcal{G}^{ID}) \mid \text{Type}(\mathcal{G}^{ID}, \mathcal{G}^S) \mid \text{Swipe}(\mathcal{G}^{ID}, \mathcal{G}^{XY}) \mid \text{Pinch}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) \mid \text{Sleep}(\mathcal{G}^Z) \\
& \text{(Trace Generators)} \mathcal{G} ::= \mathcal{G}^a \mid \text{Skip} \mid d \mid \mathcal{G} \text{ :>> } \mathcal{G}' \mid \mathcal{G} \text{ <+> } \mathcal{G}' \mid \mathcal{G} ? \mid \mathcal{P} \text{ then } \mathcal{G} \mid \text{repeat } n \mathcal{G} \mid \bullet
\end{aligned}$$

Figure 9. Lifting UI traces from Figure 5 to UI trace generators.

supports predicates with any number of call-by-name arguments of simple types (e.g., integers, strings). The developer is also required to provide Chimp Driver an operational interpretation of this predicate in the form of a boolean test method of the same name. A run-time call to this test operation is realized through reflection. The use of reflection is admittedly a possible source of errors in test suite itself because it circumvents static checking. To partially mitigate this problem, ChimpCheck’s run-time system has been designed to fail fast and to be explicit about such errors. Explicit language support like imposing explicit declarations of and pre-run-time checks on user-defined predicates could be introduced to further mitigate this problem, but such engineering improvements are beyond the scope of a research prototype.

Resuming the Test App After Suspending Device events like ClickBack, ClickHome, ClickMenu and Settings potentially map to actions that suspend the test app. Our operational semantics silently assumes that the app is subsequently resumed so that the rest of the UI test sequence can proceed as planned. However, implementing this behavior in Espresso, as well as Robotium, is currently not possible within the testing framework. Instead, to overcome this limitation of the underlying testing framework, we embed in our test controller a sub-routine that periodically polls the Android device on what app is currently displayed on the foreground. If this foreground app is determined to be one other than the test app, this sub-routine simply invokes the necessary commands to resume the test app. These polling and resume commands are invoked through Android’s command-line bridge (Adb) and this “kick back” subroutine is kept active until the UI test trace has been completed (with either success or failure).

4 UI Trace Generators

We now discuss lifting UI traces to *UI trace generators*. In Section 4.1, we define the language of generators, by (1) lifting from the symbols of UI traces and (2) introducing new combinators. Then in Section 4.2, we present the semantics of generators in the form of a transformation operation into the domain of sets of UI traces. Finally in Section 4.3, we present the full architecture of ChimpCheck, connecting generators to actual test runs and to test results.

4.1 A Language of UI Trace Generators

Figure 9 introduces the core language of trace generators. In essence, it is built on top of the language of traces (Figure 5), particularly extending the formal language in two ways: (1) extending primitive event arguments with generators and (2) a richer array of combinators, particularly non-deterministic choice and repetition. Our usual primitive terms (coordinates \mathcal{G}^{XY} and UI identifiers \mathcal{G}^{ID}) are now extended with string generators \mathcal{G}^S and integer generators \mathcal{G}^Z . Formally, we define string and integer generators as any subset of the string and integer domains, respectively. Hence app events \mathcal{G}^a can now be associated to sets of primitive values. Trace generators \mathcal{G} consists of this extension of app events (\mathcal{G}^a), the device events d , combinators lifted from UI traces (i.e., :>> , ? , and then), as well as two new combinators; $\mathcal{G} \text{ <+> } \mathcal{G}'$ ing a non-deterministic choice between \mathcal{G} and \mathcal{G}' and $\text{repeat } n \mathcal{G}$ representing the repeated sequencing of instances of \mathcal{G} for up to n times.

Notice that the combinators `optional` and `*>>` shown in the example in Figure 4 are not part of the core language introduced in Figure 9. The reason is that they are in fact *derivable* from combinators of the core language. For instance, `optional` \mathcal{G} can easily be derived as a non-deterministic choice between traces from \mathcal{G} or `Skip` (i.e., `optional` $\mathcal{G} \stackrel{\text{def}}{=} \mathcal{G} \text{ <+> } \text{Skip}$). *Derived* combinators are introduced to serve as syntactic sugar to help make trace generators more concise. More importantly, for more advance generative behaviors, derived combinators provide the main form of encapsulation and extensibility of the library. We will introduce more such derived combinators in Section 5 and demonstrate their utility in realistic scenarios.

4.2 A Semantics of UI Trace Generators

The semantics of generators are defined by $\text{Gen}[\mathcal{G}]$: given generator \mathcal{G} , the semantic function $\text{Gen}[\mathcal{G}]$ yields the (possibly infinite) set of UI traces where each trace τ is in the *concretization* [9] of \mathcal{G} (i.e., is an instance of \mathcal{G}). Figure 10 defines the semantics of UI trace generators, particularly the definition of $\text{Gen}[\mathcal{G}]$. For app events \mathcal{G}^a (e.g., $\text{Click}(\mathcal{G}^{ID})$, $\text{Type}(\mathcal{G}^{ID}, \mathcal{G}^S)$), the semantic function $\text{Gen}[\mathcal{G}^a]$ simply defines the set of app event instances with arguments drawn from the respective primitive generator domains (e.g., \mathcal{G}^{ID} , \mathcal{G}^S). For `Skip` and device events d , $\text{Gen}[\text{Skip}]$ and $\text{Gen}[d]$ are simply singleton sets. Similar to primitive app events, generators of the combinators :>> , ? and then define the sets of

$$\begin{aligned}
 \text{Gen}[\text{Click}(\mathcal{G}^{ID})] &\stackrel{\text{def}}{=} \{\text{Click}(id) \mid id \in \text{Gen}[\mathcal{G}^{ID}]\} \\
 \text{Gen}[\text{LongClick}(\mathcal{G}^{ID})] &\stackrel{\text{def}}{=} \{\text{LongClick}(id) \mid id \in \text{Gen}[\mathcal{G}^{ID}]\} \\
 \text{Gen}[\text{Type}(\mathcal{G}^{ID}, \mathcal{G}^S)] &\stackrel{\text{def}}{=} \left\{ \text{Type}(id, s) \mid \begin{array}{l} id \in \text{Gen}[\mathcal{G}^{ID}] \wedge \\ s \in \text{Gen}[\mathcal{G}^S] \end{array} \right\} \\
 \text{Gen}[\text{Swipe}(\mathcal{G}^{ID}, \mathcal{G}^{XY})] &\stackrel{\text{def}}{=} \left\{ \text{Swipe}(id, l) \mid \begin{array}{l} id \in \text{Gen}[\mathcal{G}^{ID}] \wedge \\ l \in \text{Gen}[\mathcal{G}^{XY}] \end{array} \right\} \\
 \text{Gen}[\text{Pinch}(\mathcal{G}_1^{XY}, \mathcal{G}_2^{XY})] &\stackrel{\text{def}}{=} \left\{ \text{Pinch}(l_1, l_2) \mid \begin{array}{l} l_1 \in \text{Gen}[\mathcal{G}_1^{XY}] \wedge \\ l_2 \in \text{Gen}[\mathcal{G}_2^{XY}] \end{array} \right\} \\
 \text{Gen}[\text{Sleep}(\mathcal{G}^Z)] &\stackrel{\text{def}}{=} \{\text{Sleep}(n) \mid n \in \text{Gen}[\mathcal{G}^Z]\} \\
 \text{Gen}[\text{Skip}] &\stackrel{\text{def}}{=} \{\text{Skip}\} \\
 \text{Gen}[d] &\stackrel{\text{def}}{=} \{d\} \\
 \text{Gen}[\mathcal{G}_1 \text{ :>> } \mathcal{G}_2] &\stackrel{\text{def}}{=} \left\{ \tau_1 \text{ :>> } \tau_2 \mid \begin{array}{l} \tau_1 \in \text{Gen}[\mathcal{G}_1] \wedge \\ \tau_2 \in \text{Gen}[\mathcal{G}_2] \end{array} \right\} \\
 \text{Gen}[\mathcal{G} \text{ ?}] &\stackrel{\text{def}}{=} \{\tau \text{ ?} \mid \tau \in \text{Gen}[\mathcal{G}]\} \\
 \text{Gen}[\mathcal{P} \text{ then } \mathcal{G}] &\stackrel{\text{def}}{=} \{\mathcal{P} \text{ then } \tau \mid \tau \in \text{Gen}[\mathcal{G}]\} \\
 \text{Gen}[\mathcal{G}_1 \text{ <+> } \mathcal{G}_2] &\stackrel{\text{def}}{=} \text{Gen}[\mathcal{G}_1] \cup \text{Gen}[\mathcal{G}_2] \\
 \text{Gen}[\text{repeat } n \mathcal{G}] &\stackrel{\text{def}}{=} \left\{ \text{:>>}_{i=1}^m \tau_i \mid \begin{array}{l} \tau_i \in \text{Gen}[\mathcal{G}] \wedge \\ 0 < m \leq n \end{array} \right\}
 \end{aligned}$$

Figure 10. Semantics of UI trace generators. Generators are interpreted as sets of UI trace that they generate.

their respective combinators. For non-deterministic choice $\mathcal{G}_1 \text{ <+> } \mathcal{G}_2$, $\text{Gen}[\mathcal{G}_1 \text{ <+> } \mathcal{G}_2]$ defines the union of $\text{Gen}[\mathcal{G}_1]$ and $\text{Gen}[\mathcal{G}_2]$. Intuitively, this means that its concrete traces can be drawn from either from concrete traces in \mathcal{G}_1 or \mathcal{G}_2 . Finally, $\text{Gen}[\text{repeat } n \mathcal{G}]$ defines the set containing trace sequences of τ_i of length m , where m is between zero and n and each τ_i are concrete instances of \mathcal{G} (denoted by $\text{:>>}_{i=1}^m \tau_i$).

A Trace Combinator? Or A Generator? One reasonable question that likely arises by now is the following, “Why do the combinators for non-deterministic choice (<+>) and repetition (repeat) not have counterparts in the language of UI trace (Figure 5)?” No doubt for uniformity, it appears very tempting to instead define all the combinators in the language of UI traces and then the simply lifting all constructs of the language into generators (as done for :>> , ? and then and app events). Doing so however would introduce some amount of semantic redundancy. For instance, having $\tau_1 \text{ <+> } \tau_2$ as a UI trace combinator would introduce the following two rules in our operational semantics of Chimp Driver (Figure 7):

$$\frac{}{\langle \tau_1 \text{ <+> } \tau_2; \Phi \rangle \mapsto \bullet \langle \tau_1; \Phi \rangle} \text{(Left)} \quad \frac{}{\langle \tau_1 \text{ <+> } \tau_2; \Phi \rangle \mapsto \bullet \langle \tau_2; \Phi \rangle} \text{(Right)}$$

While these rules do offer a richer “dynamic” (from the perspective of the single-trace semantics) form of non-deterministic choice, they provide no additional expressivity to the overall testing framework since the non-deterministic choice behavior is already modeled by the generator semantics (Figure 10). Similarly, the repetition (repeat) generator faces the same semantic redundancy if it is introduced as a trace combinator. But conversely, are the UI trace combinators (i.e., u , :>> , \bullet , assert , ? and then) that we have introduced

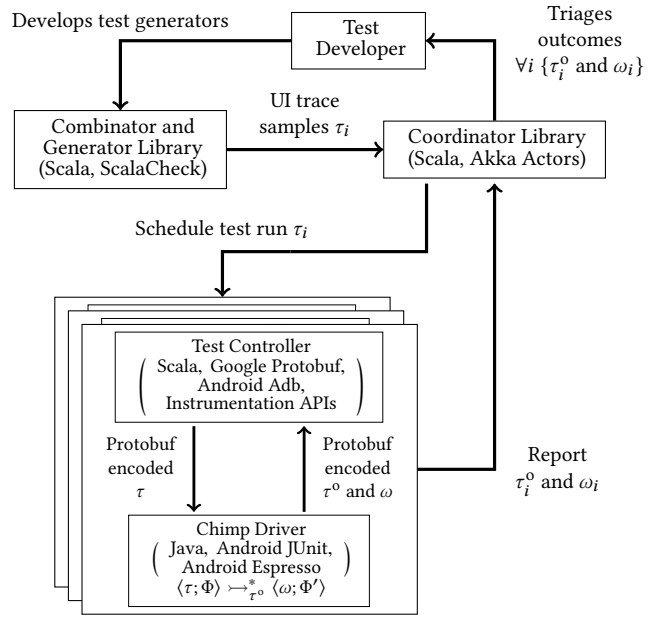


Figure 11. Architecture of ChimpCheck test generation. The test developer focuses on writing ChimpCheck UI trace generator scripts. At run time, the underlying ScalaCheck library samples UI traces from these UI trace generators. The test coordinator issues each UI trace sample to a unique Android emulator, which independently exercises an instance of the test app as dictated by the UI trace. The outcome of executing each UI trace is, in the end, reported back to the test developer.

truly necessarily part of that language? In fact, they each are indeed necessary: primitive UI events u directly interacts with the device state Φ hence must inevitably be part of the UI trace language. We need a fundamental way to express sequences of events, hence we have $\tau_1 \text{ :>> } \tau_2$ with \bullet as the unit operator. Finally, notice that the remaining combinators each in its own way explicitly interacts with the device state: $\text{assert } \mathcal{P}$ conducts a run-time test \mathcal{P} on the device, what prefix of $\tau \text{ ?}$ to be executed can only be determined at run-time by inspecting the device state Φ , and deciding to execute τ in \mathcal{P} then τ depends on the run-time test \mathcal{P} . In essence, symbols of the UI trace language should necessarily require run-time interpretation from the device or app state, while the language of generators may contain combinators that can be “statically compiled” away. This strategy of language and semantics minimization between traces and their generators would continue to make sense as ChimpCheck gets extended.

4.3 Implementing UI Trace Generators

Figure 11 illustrates the entire test generation and execution architecture of ChimpCheck. The test developer now writes UI trace generators rather than a single UI trace (as natively

in Espresso or Robotium). To generate concrete instances of traces from these generators, we have implemented the generators on top of the ScalaCheck [16] library—Scala’s active implementation of property-based test generation [8]. Since the testing framework now potentially needs to manage multiple test executions, the ChimpCheck library includes a coordinator library (called Mission Control) that coordinates the test executions: it coordinates the testing efforts across multiple instances of the test controller and Android devices (physical or emulated), scheduling test runs of traces τ_i across the test execution pool, and consolidating concretely executed trace τ_i^o and test result ω_i from executing trace τ_i . This coordinator library is developed with Scala’s high-performance Akka actor library.

Initializing Device State Our testing framework architecture assumes that every run of the new test $\langle \tau; \Phi \rangle \xrightarrow{\tau^o}^* \langle \omega; \Phi' \rangle$, is done starting from some initial app state Φ . To achieve this, the default schedule loop of the coordinator library conservatively re-installs the test app onto the device instance on which the test is to be executed. In general, this re-installation is the only way to fully guarantee that the test app has been started from a fresh state. However, this default re-install routine can be omitted at the test developer’s discretion. Similarly, the developer can specify if devices (for emulators only) should be re-initialized before starting each test run. Regardless of these initialization choices, the developer is expected to treat each test run in isolation, and a single UI trace should be defined to be self-contained, for instance, including the encoding of time delays for instrumenting the invocation of a specific time-idle based behavior in the app.

Apart from these global configurations, explicit support to control these initialization conditions are out-of-scope of the current prototype, though we postulate possible future work that involve extending the language and combinator library to allow the developer to specify these initialization conditions as first-class expressions. This would allow the developer to define more concise initialization sequences to stress test her app’s ability to handling unfavorable app start-up conditions.

UI Trace Generators with ScalaCheck The ChimpCheck Generator library is implemented on top of the ScalaCheck Library, an implementation of property-based test generation, QuickCheck [8]. This design choice has proven to be extremely beneficial for our current and anticipated future development efforts of ChimpCheck: rather than developing randomized test generation from scratch, we leverage on ScalaCheck’s extensive library support for generating primitive types (e.g., strings and integers denoted by \mathcal{G}^S and \mathcal{G}^Z , respectively). This library support includes many utility functions from generating arbitrary values of the respective domains, to uniform/weighted choosing operations. The ScalaCheck library is also highly extensible, allowing

developers to extend these functionalities to user-defined algebraic datatypes (e.g., trees, heaps) and in our case, extending test generation functionalities to UI trace combinators. This approach not only offers the ChimpCheck developer compatible access to ScalaCheck library of combinators, it makes ChimpCheck reasonably simple to maintain and incrementally developed.

As highlighted earlier in Figures 2 and 4, to generate and execute UI trace test cases, ChimpCheck relies on the ScalaCheck library combinator `forAll` to sample and instantiate concrete UI traces, while our `chimpCheck` combinator embeds the sub-routines that coordinates execution of the tests (as illustrated in Figure 11). The following illustrates the general form of the Scala code the developer would write to achieve this:

```
forAll( $\mathcal{G}$ : Gen[EventTrace]) {  $\tau_i$ : EventTrace =>
   $\tau_i$  chimpCheck {  $\mathcal{P}$ : Prop }
}
```

UI traces τ_i are objects of class `EventTrace` and are sampled and instantiated by the `forAll` combinator. The test operation is

```
 $\tau_i$  chimpCheck {  $\mathcal{P}$ : Prop }
```

where \mathcal{P} is the app property to be tested at the app state obtained by exercising the app with τ_i . The `chimpCheck` combinator implements the entire test routine, formally

$$\langle \tau_i : \gg \text{assert } \mathcal{P}; \Phi \rangle \xrightarrow{\tau_i^o}^* \langle \omega_i; \Phi' \rangle .$$

In addition to ScalaCheck’s standard output on total number of tests passed/failed, the ChimpCheck libraries generate log streams containing the information for reproducing failures, namely the concrete executed trace τ_i^o and test result ω_i .

5 Case Studies: Customized Test Patterns

In this section, we demonstrate the utility of UI trace generators as a higher-order combinator library. We present four novel ways that a generator derived from the core language is applied to address a specific issue in Android app testing. For each, we discuss real, third-party reported issues in open-source apps that motivate the conception of this generator.

5.1 Exceptions on Resume

A common problem in Android apps is the failure to handle suspend and resume operations. These failures are most commonly exhibited as app crashes caused by (1) *illegal state exceptions*, when an app’s suspend/resume routines do not make correct assumptions on a subcomponent’s life-cycle state, or (2) as *null pointer exceptions*, typically when an app’s suspend/resume routines wrongly assumes the availability of certain object resources that did not survive the suspend operation or was not explicitly restored during the resume operation. From a software testing perspective, the Android

$$\mathcal{G}_1 * \gg^m \mathcal{G}_2 \stackrel{\text{def}}{=} \mathcal{G}_1 : \gg \text{repeat } m \mathcal{G}_{\text{intr}} : \gg \mathcal{G}_2 \quad \text{where}$$

$$\mathcal{G}_{\text{intr}} \stackrel{\text{def}}{=} \text{ClickHome} \langle + \rangle \text{ClickMenu} \langle + \rangle \text{Settings} \langle + \rangle \text{Rotate}$$

Figure 12. The Interruptible Sequencing Combinator is defined in terms of the ChimpCheck core language. It sequences \mathcal{G}_1 and \mathcal{G}_2 but allows a finite number (m) of occurrences of interrupt events ($\mathcal{G}_{\text{intr}}$) in between.

app’s test suite should include sufficient test cases that exercises its suspend/resume routines. As illustrated in our example in Section 2, Android apps are stateful event-based systems, so conducting suspend and resume operations at different points (states) of an Android may result in entirely different outcomes. Test cases must provide coverage for suspend/resume at crucial points of the test app (e.g., login pages, while performing long background operations).

The Interruptible Sequencing Combinator To simplify the development of trace generator sequences that tests the app’s ability to handle interrupt events, we derive a specific combinator similar to sequencing but additionally inserts suspend and resume events in a *non-deterministic* manner.

Figure 12 shows how the *Interruptible Sequencing Combinator* is derived from the repetition (repeat) combinator: $\mathcal{G}_1 * \gg^m \mathcal{G}_2$ is defined as sequencing where we allow repeat $m \mathcal{G}_{\text{intr}}$ to be inserted between \mathcal{G}_1 and \mathcal{G}_2 . The interrupt generator $\mathcal{G}_{\text{intr}}$ denotes the non-deterministic choice between the various interrupt device events that triggers the suspending (and resuming) of the app. The combinator takes one parameter m , which is the maximum number of times we allow interrupt events to occur. Our current implementation treats this parameter optionally and defaults it to 3, as we have observed that in practice, such app crashes are typically reproducible within one or two consecutive interrupt events.

Case Studies We have observed numerous numbers of issue tracker cases on open Android GitHub projects that report failures that are exactly caused by this issue (illegal state exceptions or null pointer exceptions on app resume). One example is found in Tower², which is a popular open-source mobile ground control station app for UAV drones. A past issue³ of the Tower app documents a null-pointer exception that occurs when the user suspended and resumed the app from the app’s main map interface. This failure is caused by a wrong assumption that references to the map UI display fragment remain intact after the suspend/resume cycle.

Another example worth noting is the Nextcloud open-source Android app⁴, which provides rich and secured mobile access to data (documents, calendars, contacts, etc) stored (by

²DroidPlanner. Tower. <https://github.com/DroidPlanner/Tower>.

³Fredia Huy-Kouadio. FlightActivity NPE in onResume #1036. <https://github.com/DroidPlanner/Tower/issues/1036>. September 2, 2014.

⁴Nextcloud. <https://github.com/nextcloud/android>.

$$\mathcal{G} \text{ preserves } \mathcal{P} \stackrel{\text{def}}{=} \text{assert } \mathcal{P} : \gg \mathcal{G} : \gg \text{assert } \mathcal{P}$$

Figure 13. The Property-Preservation Combinator is defined by asserting a property \mathcal{P} before and after a given generator \mathcal{G} , and it tests if the property is preserved by UI traces sampled from \mathcal{G} .

paid users) in the company’s proprietary cloud data storage service. A recent issue⁵ reports a crash of the app during a file selection routine when the user re-orientes the app from portrait to landscape. This crash is the result of a null-pointer exception on a reference to the file selector UI object (OCFileListFragment), caused by the failure of the app’s resume operation to anticipate the destruction of the object after suspension.

We have developed test generators for Nextcloud and reproduced this crash (#448) with the following trace generator (with the login sequence omitted for simplicity):

```
⟨Login Sequence⟩ * >>
LongClick(R.id.linearlayout) * >>
Click("Move") * >> •
```

Note that other recent work [1] has also identified injecting interrupt events as being critical to testing apps. While our (current) implementation of $* \gg$ is less sophisticated than Adamsen et al. [1], the advantage of the ChimpCheck approach is that $* \gg$ is simply a derived combinator that can be placed alongside other scripted pieces (e.g., for the login sequence). The $* \gg$ provides a basic demonstration of how more complex test generators that address real app problems can be implemented in ChimpCheck.

5.2 Preserving Properties

An app’s proper functionality may very frequently depend on its ability to preserve important properties across certain state transitions that it is subjected to. For instance, it would be a very visible defect if key navigational buttons of the app vanish after user suspended and resumed the app. While this issue seems very similar to the previous (i.e., a failure caused by interrupt events), the distinction we consider here is that the issue does not result in app crashes. Hence, simply testing against interrupt events (via the $* \gg$ combinator) may not detect any faults. Since the decision of which UI elements should “survive” interrupt events is app specific, we cannot to fully-automate testing such property but instead derive customizable generators that allow the test developer to program such tests more effectively and efficiently.

The Property-Preservation Combinator Rather than writing boiler-plate assertions before and after specific events, we derive a generator that expresses the test sequences in a more general manner.

⁵Andy Scherzinger. FolderPicker - App crashes while rotating device #448. <https://github.com/nextcloud/android/issues/448>. December 13, 2016.

Figure 13 defines this generator. The Property-Preservation Combinator \mathcal{G} preserves \mathcal{P} asserts the (meta) property that \mathcal{P} is preserved across any trace instance of \mathcal{G} . For example, we can assert that "Button1" remains clickable across interrupt events (i.e., after the app resumes):

```
 $\mathcal{G}_{\text{intr}}$  preserves isClickable("Button1") where
 $\mathcal{G}_{\text{intr}} \stackrel{\text{def}}{=} \text{ClickHome} \langle + \rangle \text{ClickMenu} \langle + \rangle \text{Settings} \langle + \rangle \text{Rotate} .$ 
```

Case Studies We have observed many instances of this “vanishing UI element” scenario described above. Just to name a few popular apps in this list: Pokemap⁶, a supporting map app for the popular Pokémon Go game; c:geo⁷, a popular Geocaching app with 1M-5M installs from the Google Play Store as of August 23, 2017; and Tusky⁸, an Android client for a popular social-media channel Mastodon. Each app at some point of its active development contained a bug related to our given scenario. For Pokemap, issue #202⁹ describes a text display notifying your success in locating a Pokémon permanently disappears after screen rotation. For CGeo, issue #2424¹⁰ describes an opened download progress dialog is wrongly dismissed after screen rotation, leaving the user uncertain about the download progress. For Tusky, issue #45¹¹ states that replies typed into text inputs are not retained after screen rotation.

We found that we could reproduce issue #45 in Tusky with the following simple generator:

```
... :>> Type(R.id.edit_area, "Hi") :>>
Rotate preserves hasText(R.id.edit_area, "Hi")
```

5.3 Integrating Randomized UI Testing

While writing custom test scripts is often necessary for achieving the highest possible test coverage, black-box techniques like pure randomized UI testing [6] and model learning techniques [4, 12] are nonetheless important and an effective means in practice for providing basic test coverage. Industry-standard Android testing frameworks (e.g., Espresso and Robotium) provides little (or no) support for integrating with these test generation techniques, which unfortunately forces the test developer to use the various possible testing approaches in isolation.

The Monkey Combinators To demonstrate how black-box techniques can be integrated into our combinator library,

⁶Omkar Moghe. Pokemap. <https://github.com/omkarmoghe/Pokemap>.

⁷c:geo. <https://github.com/cgeo/cgeo>.

⁸Andrew Dawson. Tusky. <https://github.com/Vavassor/Tusky>.

⁹Andy Cervantes. Flipping Screen Orientation Issue - Pokemon Found Stops Being Displayed #202. <https://github.com/omkarmoghe/Pokemap/issues/202>. July 26, 2016.

¹⁰Ondřej Kunc. Download from map is dismissed by rotate #2424. <https://github.com/cgeo/cgeo/issues/2424>. January 22, 2013.

¹¹Julien Deswaef. When writing a reply, text disappears if app switches from portrait to landscape #45. <https://github.com/Vavassor/Tusky/issues/45>. April 2, 2017.

```
monkey n  $\stackrel{\text{def}}{=} \text{repeat } n (\mathcal{G}_{\text{Ms}} \langle + \rangle \mathcal{G}_{\text{intr}})$  where
 $\mathcal{G}_{\text{Ms}} \stackrel{\text{def}}{=} \text{Click}(\mathcal{G}^{XY}) ? \langle + \rangle \text{LongClick}(\mathcal{G}^{XY}) ?$ 
 $\langle + \rangle \text{Type}(\mathcal{G}^{XY}, \mathcal{G}^S) ? \langle + \rangle \text{Swipe}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) ?$ 
 $\langle + \rangle \text{Pinch}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) ? \langle + \rangle \text{Sleep}(\mathcal{G}^Z) ?$ 

relevantMonkey n  $\stackrel{\text{def}}{=} \text{repeat } n (\mathcal{G}_{\text{Gs}} \langle + \rangle \mathcal{G}_{\text{intr}})$  where
 $\mathcal{G}_{\text{Gs}} \stackrel{\text{def}}{=} \text{Click}(\ast) ? \langle + \rangle \text{LongClick}(\ast) ?$ 
 $\langle + \rangle \text{Type}(\ast, \mathcal{G}^S) ? \langle + \rangle \text{Swipe}(\ast, \mathcal{G}^{XY}) ?$ 
 $\langle + \rangle \text{Pinch}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) ? \langle + \rangle \text{Sleep}(\mathcal{G}^Z) ?$ 

 $\mathcal{G}_{\text{intr}} \stackrel{\text{def}}{=} \text{ClickHome} \langle + \rangle \text{ClickMenu} \langle + \rangle \text{Settings} \langle + \rangle \text{Rotate}$ 
```

Figure 14. Two implementations of the Monkey Combinator. The first (monkey) randomly applies user events on random XY coordinates on a device, mimicking exactly what the Android UI/Application Exerciser Monkey does. The next (relevantMonkey) is just slightly smarter—applying more relevant actions by accessing the device’s run-time view hierarchy and inferring relevant user events.

here we derive two generators monkey and relevantMonkey from existing combinators.

Figure 14 shows two implementations of generators for random UI event sequences. The first, called the monkey combinator, is similar to Android’s UI Exerciser Monkey [6] in that it generates random UI events applied to random locations on the device screen. The second combinator, called the relevantMonkey, generates random but more relevant UI events by relying on ChimpCheck’s primitive \ast combinator to infer relevant interactions from the app’s run-time view hierarchy. Having randomized test generators like these as combinators provides the developer with a natural programming interface to integrate these approaches with her own custom scripts. For instance, revisiting our example in Section 2 (or similarly for the Nextcloud app in Section 5.1), getting pass a login page is the hurdle to using a brute-force randomized testing, but we can implement the necessary traces to the media pages by simply the following generator:

```
Click(R.id.enter) :>>
Type(R.id.username, "test") :>>
Type(R.id.password, "1234") :>>
Click(R.id.signin) :>> relevantMonkey 50
```

This generator simply applies the relevant monkey combinator (arbitrarily for 50 steps) after the login sequence—thus generating multiple UI traces that randomly exercises the app functionalities after applying the login sequence fixture.

Case Studies Our preliminary studies have found that even for simple apps (e.g., Kistenstapeln¹², a score tracker

¹²Fachschaft Informatik. Kistenstapeln-Android. <https://github.com/d120/Kistenstapeln-Android>.

Table 1. We applied ChimpCheck’s relevantMonkey and the Android UI Exerciser Monkey to try to witness a known issue (#1) in *Kistenstapeln-Android*. We ran each exerciser 10 times for up to 5,000 UI events. The average number of steps is taken only over successful attempts in witnessing the bug.

UI Exerciser	Attempts	Witnessed		Steps to Bug (average n)
	(n)	(n)	(frac)	
relevantMonkey	10	10	1	289
Android Monkey	10	5	0.5	3177

for crate stacking game, and Contraction Timer¹³, a timer that tracks and stores contraction data), we require generating numerous UI event sequences from Android Monkey before we get acceptable coverage results. Preliminary experiments using the relevant monkey combinator (that accesses the view hierarchy) have shown promising results shown in Table 1. For the *Kistenstapeln* app, the relevant-monkey combinator witnesses the bug from issue #1¹⁴ in an order of magnitude less generated events than the Android UI Exerciser Monkey. Note that Android Monkey failed to witness the bug in under 5,000 events in half of the attempts.

We also note that these promising preliminary results are achieved with a simplistic, light-weight implementation: exactly the code in Figure 14, together with less than 200 lines of library code that implements view hierarchy access for the * combinator, developed within a span of two days, including time to learn the Android Espresso framework.

5.4 Injecting Custom Generators

In Section 5.3, we demonstrated how random UI testing techniques can be added to ChimpCheck as black-box generators. However in practice, many situations require more tightly coupled interactions between the custom scripts and the black-box techniques. For instance, many modern Android apps can contain features requiring user authentication with her account and such authentication procedures are often requested in an on-demand manner (only when user requests contents that requires two-factor authentication). Such dynamic behaviors makes it difficult or impractical to simply hard-code and prepend a custom login script as we did in the previous sections.

The Gorilla Combinator From the relevantMonkey combinator, we refine it into the gorilla combinator with the ability to inject customized scripting logic into randomized testing.

¹³Ian Lake. Contraction Timer. <https://github.com/ianhanniballake/ContractionTimer>.

¹⁴Tobias Neidig. Crash on timer-event on other fragment #1. <https://github.com/d120/Kistenstapeln-Android/issues/1>. March 19, 2015.

```
gorilla n G  $\stackrel{\text{def}}{=} \text{repeat } n (\mathcal{G} \text{ :>> } (\mathcal{G}_{\text{Ms}} \text{ <+> } \mathcal{G}_{\text{intr}})) \quad \text{where}$ 
 $\mathcal{G}_{\text{Ms}} \stackrel{\text{def}}{=} \text{Click}(\mathcal{G}^{XY}) ? \text{ <+> LongClick}(\mathcal{G}^{XY}) ?$ 
 $\text{ <+> Type}(\mathcal{G}^{XY}, \mathcal{G}^S) ? \text{ <+> Swipe}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) ?$ 
 $\text{ <+> Pinch}(\mathcal{G}^{XY}, \mathcal{G}^{XY}) ? \text{ <+> Sleep}(\mathcal{G}^Z) ?$ 
 $\mathcal{G}_{\text{intr}} \stackrel{\text{def}}{=} \text{ClickHome} \text{ <+> ClickMenu} \text{ <+> Settings} \text{ <+> Rotate}$ 
```

Figure 15. The Gorilla Combinator enriches the monkey combinators with an additional generator parameter \mathcal{G} . This generator is injected before every randomly sampled traces from \mathcal{G}_{Ms} or $\mathcal{G}_{\text{intr}}$.

Figure 15 shows the implementation of the gorilla combinator. It accepts an additional argument: a generator \mathcal{G} that is prepended before every randomized event ($\mathcal{G}_{\text{Ms}} \text{ <+> } \mathcal{G}_{\text{intr}}$, hence allowing the developer to “inject” custom *directives* to handle situations where a purely randomized technique may be mostly ineffective. For example, we can define a simple combinator that generates traces that randomly explores the app unless a login page is displayed. When the login page is displayed, it will append a hard-coded login sequence to effectively proceed through the login page:

```
val login = Click("Login") :>>
  Type("User", "test") :>>
  Type("Password", "1234") :>> Click("Sign-in")
gorilla 50 (isDisplayed("Login") then login)
```

Case Studies An example of a simple app with an authentication page is OppiaMobile¹⁵, a mobile learning app. We have developed test generators using the refined gorilla with a hard-coded login directive (as described above). In sample runs, we observed that the gorilla occasionally logs out and logs back in between unauthenticated and authenticated portions of the page. Though no bugs were found, such log in/out loops are very rarely tested and potentially hides defects.

This app also contains an information, modal dialog that is unfavorable for randomized testing techniques. In particular, the only way to navigate through this page is a lengthy scroll to the end of the UI view and hitting a button labeled “Select SD Card”. Attempts at random exercising often ends up stuck in this problematic page. To help the gorilla function more effectively, we injected a non-deterministic choice between the only two possible actions when this dialog page is visible: (1) proceed forward by scrolling down and click the button or (2) hit the device back navigation button. In Figure 16, we show the few lines that implements this injection of this application-specific way of getting past a particular modal dialog.

¹⁵Digital Campus. OppiaMobile Learning. <https://github.com/DigitalCampus/oppia-mobile-android>.

```

1 gorilla 100 {
2   isDisplayed("SD Card Access Framework") then {
3     { Swipe(R.id.scroll,Down) :>>
4       Click("Select SD Card") } <+> ClickBack
5   }
6 }

```

Figure 16. Getting past a modal dialog in the *OppiaMobile* app. This ChimpCheck generator says to do random UI exercising unless the app is showing the "SD Card Access Framework" page. In this case, inject the specific action to either scroll to the bottom of the page and click a specific button to continue or click the back button.

Finally, another application of *gorilla* is that from Android 6.0 (API level 23) onwards, device permissions (e.g., SD card usage, camera, location) are granted at run-time, as opposed to on installation. The app developer is free to decide how these *dynamic* permissions are requested. Typically, an app will use a modal dialog box with an acknowledgment and reject button. Dealing with these dynamic permissions is straight-forward with the *gorilla* combinator using code, for example, similar to Figure 16.

6 Related Work

Research in test-case generation for Android apps has largely focused on developing techniques and algorithms to automate testing while providing better code coverage than the industrial baseline, Android Monkey [6]. Evodroid [13] explores the adaptation of *evolutionary testing* methods [7] to Android apps. MobiGuitar [4] is a testing tool-chain that automatically reverse-engineers state-machine navigation models from Android apps to generate test cases from these models. It leverages a model-learning technique for Android apps called Android Ripper [3], which uses a stateful enhancement of *event-flow graphs* [15] to handle stateful Android apps. Dynodroid [12] also develops a model-based approach to generate event sequences. Similar to our approach in inferring relevant UI events, Dynodroid uses the app's view hierarchy to observe relevant actions. Sapienz [14] uses a multi-objective search-based technique to generate test cases with the aim of maximizing/minimizing several objective functions key to Android testing. Techniques described in this work has been successfully applied in an industrial strength tool, Majicke [11].

The works mentioned above each offer a steady advancement of automatic test-case generation. Given this focus of what's automatable, a perhaps unwitting result has been much less attention on the issue of *programmability* and *customizability* that we identify in this paper. To use these automatic test-case generation approaches, a test developer

often needs to work around the app-specific concerns in awkward ways. For instance, to deal with login screens, Amalfitano et al. [3, 4] assumes that the test developer provides a set of "initial" states of the app that are past the login screens, which thus allows the authors to focus only on the automatable aspects of the app. The test developer would then have to rely on other techniques (presumably scripting-based techniques) to exercise the app to these "initial" states. ChimpCheck offers the potential to fuse these automatic test-case generation techniques with scripted, app-specific behavior. Another example of this perhaps unwitting result can be found in Mao et al. [14]. This approach describes a test-case generation strategy that is inspired by genetic mutation. Though the "genes" can in principle be customized for specific apps, the authors chose to focus their experiments only on a set of genes that are known to be generic across all apps. Much less attention was given to the programmability or customizability question to, for example, empower the test developer to express her own customized genes. In the end, studies [2, 4, 12] on the saturation and coverage of automatic test-case generation for Android apps provide evidence for the ChimpClick claim—the need for human insight and app-specific knowledge to generate not just traces but *relevant* traces.

The most closely related work to ChimpCheck are a few pieces that consider some aspect of programmability in generating UI traces. Adamsen et al. [1] introduces a methodology that enriches existing test suites (Robotium scripts) by systematically injecting interrupt events (e.g., rotate, suspend/resume app) into key locations of test scripts to introduce what the authors refer to as *adverse conditions* that are responsible for many failures in real apps. Comparing to our approach, this work can be viewed as a more sophisticated implementation of the interruptible sequencing operator **>>* injected into the existing test suite in a fixed manner (replacing all *:>>*'s with **>>*). Our approach is complementary in that ChimpCheck provides the opportunity to fuse interruptible sequencing with other test-generation techniques in a user-specified manner, and we might adapt their approaches for systematic exploration of injected events and failure minimization to improve ChimpCheck's library combinators. Hao et al. [10] introduces a programmable framework for building dynamic analyses for mobile apps. A key component of this work is a programmable Java library for developing customized variants of random UI exercisers similar to the Android Monkey. Our work shares similar motivations but differs in that we provide a richer programmable interface (in the form of a combinator library), while their work provides stronger support for injecting code and logic for dynamic analysis.

Our approach leverages ideas from QuickCheck [8], a property-based test-generation framework for the functional programming language Haskell. Our implementation is built

on top of the ScalaCheck library [16], which Scala’s implementation of the QuickCheck test framework. A novel contribution of ChimpCheck over QuickCheck and ScalaCheck is the reification of user interactions as first-class objects (UI traces) so that UI trace generators can be defined and sampled from the ScalaCheck library.

7 Towards a Generalized Framework for Fusing Scripting and Generation

In this section, we discuss our vision of a general framework for creating effective UI tests by fusing scripting and generation. While the development of ChimpCheck has provided the first steps to this end, here we discuss steps that would take this approach of fusing scripting and generation to the next level. Specifically, we envision augmentations in two incremental fronts:

1. *Integrating* scripting with state-of-the-art automated test-generation techniques (Section 7.1), beyond pure randomized (monkey) exercising.
2. *Reifying* test input domains beyond just user interaction sequences (UI traces) (Section 7.2)

From lessons learnt from the conception and development ChimpCheck, we identify and distill key design principles and challenges that we need to overcome. Particularly, the key challenge for (1) is to formally define the semantics of interaction between the scripts that the developer writes and each specific test generation technique in which we fuse. For (2), the key challenges are developing general means of expressing multiple input domains (not only UI traces) and defining the means in which these input streams interact with one another. Generalizing reified input domains also introduces an opportunity for exploring an augmentation on state-of-the-art test generation techniques: generalizing them for generating not just UI traces but also other input sequences (e.g., GPS location updates, event-based broadcasts from other apps). In the following subsections, we will discuss ideas on how these challenges can be addressed.

7.1 Generalizing the Integration with State-of-the-Art Automated Generators

We now describe a design principle that will enable us to fuse ChimpCheck scripts with more advanced forms of automated test generation. Figure 17 illustrates two ways in which we have fused scripting and test generation in this paper. The top most diagram illustrates this interaction implemented by the monkey combinator (Section 5.3, Figure 14). In the diagram, we represent fragments of the UI traces from scripts written by the developer (\mathcal{G}_1 and \mathcal{G}_2) with rounded-boxes, while the fragment from a randomized exercising technique (monkey N) is represented with a cloud. Edges represent ordering between the UI trace fragments, as dictated by the $:>>$ combinator. This diagram shows a *coarse-grained* interaction between user scripts and test generator. Particularly,

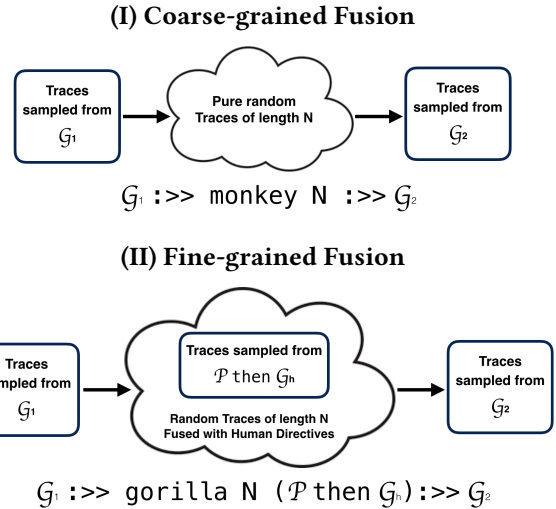


Figure 17. Two ways of using scripting and automated test generation: (I) the monkey combinator can be invoked within a script, but no user interaction is permitted while random exercising is done, hence coarse-grained fusion; (II) the gorilla combinator allows the test developer to inject human directives into random exercising, hence it is a higher-order combinator enabling finer-grained fusion.

even though the developer can fuse the two techniques, the expression `monkey N` relinquishes all control to the randomized exercising technique, other than the parameter N that dictates the maximum length of randomized steps. In general, this composition can be viewed as a “uni-directional” interaction, which enables a script to call the automated generator at a specific point of the UI trace.

Fine-Grained Fusion The gorilla (Section 5.4, Figure 15) realizes a more *fine-grained* interaction: shown in the lower diagram of Figure 17, the gorilla combinator refines the monkey combinator by further allowing the test developer to inject *directives* that supersedes randomized exercising. In this “bi-directional” interaction, the test developer can interact with the automated test generator by providing a script that is injected into the randomized exercising routine. In this instance, the developer injects \mathcal{P} then \mathcal{G}_h , customizing the randomized exercising by stating exceptional conditions (\mathcal{P}) in which a script (\mathcal{G}_h) should be used instead of pure randomized exercising. As demonstrated in Section 5.4, this combinator enhances randomized exercising by allowing the developer to guide the test generation process when needed. The key insight here is that the higher-order nature of the combinator library (a generator can be a parameter of another generator gorilla) has enabled us to express this fine-grained interaction that alternates control between user-directed scripts and the randomized exercising technique.

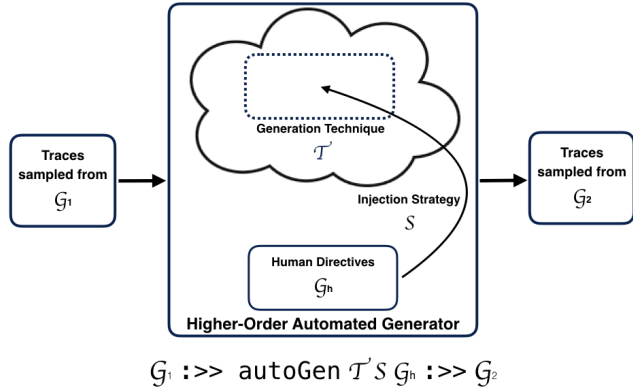


Figure 18. A higher-order combinator for generalized automated generators. Parameters \mathcal{T} is an expression that represents an instance of some automated test-generation technique, while \mathcal{G}_h is the higher-order generator representing human directives injected into \mathcal{T} . Finally \mathcal{S} specifies the injection strategy to be used.

Generalization From a broader perspective, the gorilla combinator is but just an instance of a higher-order automated generator, specifically for randomized exercising. We wish to apply this idea to other state-of-the-art automated test-generation techniques (e.g., model-based [3, 4], evolutionary testing [13], and search-based [14]). This work would provide a more generalized way for fusing automated test-generation techniques into the combinator library. Figure 18 illustrates this idea in the form of a combinator `autoGen`. Similar to the `gorilla` combinator, it is a higher-order combinator that accepts a generator \mathcal{G}_h . This generator represents the human directives to be injected into some instance of an automated generator, which is associated with this call to `autoGen`. To specify the instance of an automated generator, `autoGen` takes in another parameter \mathcal{T} , which is a new form of expression that defines automated test generators. The gorilla combinator can be refined as an instance of such an expression (i.e., `gorilla N`).

Parameter \mathcal{T} and \mathcal{G}_h do not yet completely define this generalization. Our definition of `gorilla N` earlier in this paper implements a very specific strategy for injecting \mathcal{G}_h into the randomized exercising technique. Particularly, it treats the semantics of randomized exercising as a transition system that appends a new user action (e.g., `click(ID)`) to a UI trace at each derivation step. The strategy of injection is simply to inject \mathcal{G}_h between every step. This strategy, we call *step-wise interleaving*, is effective for fusing with randomized exercising but not necessarily for other techniques. Hence, in order to enable more generality, we might need to introduce a third parameter \mathcal{S} that defines the *strategy of injection*.

We anticipate that defining \mathcal{S} is the main challenge of implementing this design principle because it is the key instrument that defines the semantics of the fusion between

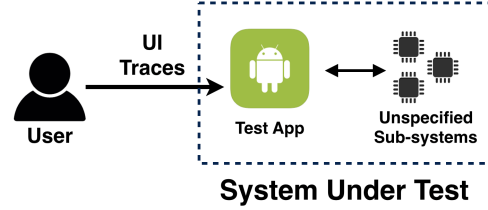


Figure 19. Reifying user interactions into UI traces—as done in the current ChimpCheck prototype.

automated generator and scripting. The applicability of a strategy \mathcal{S} to a generator technique \mathcal{T} would in general depend on \mathcal{T} 's fulfillment of certain properties (e.g., step-wise interleaving strategy will require \mathcal{T} to be a well-defined transition system). Developing a set of interfaces that implements this generalization will be the key engineering challenge. Our initial observation suggests that similar to randomized exercising, model-based techniques can exploit the step-wise interleaving strategy, though more investigation would be required to ascertain if that would be the most effective strategy. Search-based techniques, on the other hand, appear to permit a more sophisticated and specialized strategy: we can treat \mathcal{G}_h as a customized set of UI traces in which we want the technique's multi-objective search algorithm [14] to consider as basic UI trace fragments that it uses to generate test sequences. This essentially allows the developer to interact with the search algorithm—by fusing her own fragments of user interaction sequences (expressed by \mathcal{G}_h) into the search-based test generation technique.

Such “higher-orderness” tempts the obvious question: what happens if we inject an automated generator into another? It is unclear to us, at this moment, whether such interactions are useful or if they should be avoided. Developing an understanding of the semantics of such interactions will be a key challenge to address this question, as well as to enable us make the most sensible engineering choices.

7.2 Generalizing the Reification of Test-Input Domains

In this section, we discuss fusing test scripting and test generation to input domains beyond UI traces (user interaction sequences). To begin, we reflect on the original conception of ChimpCheck's domain-specific language: since we were interested in user interaction sequences as inputs to our testing efforts, we derived symbols that uniquely represent these user actions (e.g., `click(ID)`). We next define various combinators (e.g., `:>>`) that build more complex UI trace objects from atomic actions. Now we have means of expressing user interaction sequences (UI traces) as structured data, which can be manipulated and interpreted. This is the process of *reification*: concretizing implicit, abstract sequences of events into data structures that we can manipulate. For this particular case, we have reified the domain of user-interaction

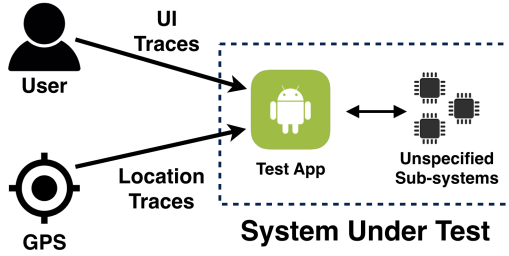


Figure 20. An example of a test environment with two reified domains, namely UI traces and location traces.

(UI) traces. Finally, the language of UI traces is lifted into the language of trace generators, hence giving us the means of generating and sampling UI traces.

Figure 19 illustrates an architectural diagram of the testing strategy for Android apps in ChimpCheck. Particularly, the system under test is the test app together with all other sub-systems that the app interacts with. Since these other interactions are not reified, test cases are agnostic to their existence. The edge between the user and test app represents the only input into the system under test and is essentially what we have reified into UI traces. By reifying UI traces, ChimpCheck is able to substitute an actual user with UI traces, and by lifting UI traces into the language of UI trace generators, the test developer has the means of expressing customized UI trace generators for their test apps.

Reifying Other Domains Our key observation is that, while user inputs (UI traces) is arguably the most important form of input for user event-driven apps, it is clearly not the only kind of inputs relevant to testing an app. Reifying other forms of input would enable us to use similar test-scripting and test-generation techniques to generate input sequences for testing the app. For instance, by subjecting location updates of the GPS module to the same reification process, we can derive the means of generating test cases that simulate the relevant location updates to the test app—in the same way we have done for UI traces in ChimpCheck. Figure 20 illustrates this new test environment that has two reified domains: UI traces from the user and location traces from the GPS module. This generalization introduces a new challenge: we now have two forms of inputs (UI and location traces), so how do they interact in terms of the syntax and semantics of this generalized language? Our initial observation is that at earlier stages of testing an app, it is still useful to derive test cases based on the sequential composition of elements from both domains (user actions and location updates). Such would be akin to expressing “laboratory” tests to test basic functionalities of the app with a controlled (discretized) sequence of events. As an example, let us assume the hypothetical reification of GPS location updates in the form of a primitive combinator `locUpdate(lg, la)` where `lg` and `la` are simulated inputs (longitude and latitude in

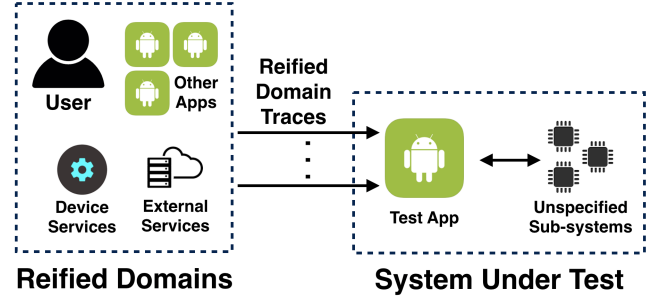


Figure 21. Generalized test environment where multiple domains (e.g., user input, inputs from device or external services, inputs from other apps) can be chosen as reified domains.

decimal degrees). The following expresses a very specific test sequence in which the the test developer uses ChimpCheck to inspect the app’s handling of a GPS location update after exercising the app past the login page:

```
Type(R.id.username, "test") ::>
Type(R.id.password, "1234") ::>
Click(R.id.login) ::> locUpdate(41.334, 2.1567)
```

While the above targets a specific test sequence, it is quite likely that at more advanced stages of testing, the test developer might be interested in subjecting the app to inputs that the correspond to UI traces interleaved with location update occurrences, for instance, to test the app’s handling of GPS location updates that interleaves with the user’s login sequence. A parallel composition operator would allow the test developer to express test generators of this nature, exemplified by the following (with a hypothetical parallel compose operator `||`):

```
{ Type(R.id.username, "test") ::>
  Type(R.id.password, "1234") ::>
  Click(R.id.login) } ||
locUpdate(41.334, 2.1567)
```

Note however, that this parallel composition is necessarily domain restricted. Other than composing UI traces with location traces, it might not make sense to allow the developer to parallel-compose streams of UI traces. The parallel composition represents an example of an extension of the language motivated by having multiple input reified domains. In general, it is likely that more such combinators relevant to and aimed at capturing idiomatic interactions between various input domains with be necessary and useful.

Generalization Figure 21 shows the generalization of this design principle. Particularly, we should assume that the system under test is subjected to inputs from any number of arbitrary reified domains. The relevant domains for reification are the same as the domains of unspecified sub-systems within the system under test, namely: (1) user inputs, (2)

inputs from device or external services, or (3) interactions with other Android apps. Depending on the focus of testing, the test developer should be allowed to decide which subsystems fall within the system under test and which should be explicitly subjected to reification. This design would allow her to express test generators seamlessly derived from any of the reified domains in a single language specification. Other (unreified) input domains would still interact with the test app as usual but are assumed not to be the main subjects of the test. In practice, it is likely that UI traces are typically given special attention (since Android apps are typically user driven), though in theory, UI traces can be uniformly treated as but one of the reified domains. That is, we can even entirely omit UI traces if it makes sense for the testing needs. An interesting effect of this generalization is that we now introduce traces of other domains (other than UI traces) to automated generator techniques, which so far has mostly been studied in the context of user-interaction sequences. While more studies are required to understand these new interactions with existing automated test generators, we believe that this also constitutes an opportunity to define automated generators in a more general manner: allowing us to generate input sequences not limited to just UI traces but also compositions of an arbitrary number of input domains.

We anticipate a number of exciting research opportunities and challenges to achieve this dimension of generalization in our testing framework: other than extending our combinator library with new reified domains, we must facilitate the possibility of allowing the user to extend the language with her own reified domains. We expect that a robust library will include reification from common domains (e.g., user interactions, GPS, WiFi), but the test developer would likely want to define, for instance, reified interfaces to her proprietary web service that her test app calls on during normal usage. This capability will entail designing programming interfaces that allow the test developer to implement the various required runtime obligations of her reified domains. Another interesting challenge would be to develop ways for the developer to express *domain-specific constraints* that governs the sampling strategy from test generators of each input domain. This ability is important because the notion of *relevance* of a test input sequence depends heavily on the input domain. For instance, a GPS location update sequence that instantaneously alternates between opposite ends of the Earth is likely to be an unrealistic input sequence. We envision that an effective test-scripting and generation library must include explicit support to enable the test developer to express such domain-specific omissions as constraints over the sampling strategies of the test-generation techniques. Finally, since test inputs are qualified from different domains, it makes sense to introduce explicit support for asserting certain meta-level properties. For example, a script must adhere to certain security policies and safety properties when

executing inter-app communication (e.g., a malicious app is present or a dependent app is not installed). Exploring such new language features will be critical to making the testing framework expressive and effective in practice.

Implementing the interfaces of new reified domains to testing framework is no doubt often a tedious endeavor. For instance, in the case of UI traces, we have to develop a mapping from UI trace atoms to Android Espresso method calls to realize the actual exercising on the test app. For the GPS module, we would have to develop a similar mapping from our reified domain of location updates to APIs in the `LocationManager` framework class of the Android framework. We believe that by encouraging the development of these mappings as library code that are accessible by highly reusable combinators, we ultimately provide more opportunities for reusability.

8 Conclusion

We considered the problem of exercising interactive apps to generate relevant user-interaction traces. Our key insight is a lifting of UI scripting from traces to generators drawing on ideas from property-based test-case generation [8]. In particular, we formalized the notion of user-interaction event sequences (or UI traces) as a first-class object with an operational semantics that is then implemented by a platform-specific component (Chimp Driver). First-class UI traces naturally lead to UI trace generators that abstract sets of UI traces. The sampling from UI trace generators is then platform-independent and can leverage a property-based testing framework such as ScalaCheck.

Driven by real issues reported in real Android apps, we demonstrated how ChimpCheck enables easily building customized testing patterns out of compositional components. The resulting testing patterns like interrupting sequencing, property preservation, brute-force randomized monkey testing, and a customizable *gorilla* tester seem broadly applicable and useful. Preliminary experiments provide evidence that simple specializations expressed in ChimpCheck can drive apps to witness bugs with many fewer events.

Generalizing from lessons learnt during development and experimentation of ChimpCheck, we have distilled two design principles, namely higher-order automated generators and custom reifications of test inputs. We believe that these design principles will serve as a guide to future development of highly relevant testing frameworks based on the humble idea of expressing property-based test generators via combinator libraries.

Acknowledgments

We thank the University of Colorado Fixr Team and the Programming Languages and Verification Group (CUPLV) for insightful discussions, as well as the anonymous reviewers

for their helpful comments. This material is based on research sponsored in part by DARPA under agreement number FA8750-14-2-0263 and by the National Science Foundation under grant number CCF-1055066. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

- [1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Software Testing and Analysis (ISSTA)*.
- [2] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the Saturation Effect in Automatic Random Testing of Android Applications. In *Mobile Software Engineering and Systems (MOBILESoft)*.
- [3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Automated Software Engineering (ASE)*.
- [4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015).
- [5] Android Developers. 2016. Testing UI for a Single App. <https://developer.android.com/training/testing/ui-testing/espresso-testing.html>. (2016).
- [6] Android Studio. 2010. UI/Application Exerciser Monkey. <https://developer.android.com/guide/components/activities.html>. (2010).
- [7] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. TestFul: An Evolutionary Test Approach for Java. In *Software Testing, Verification and Validation (ICST)*.
- [8] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*.
- [9] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL)*.
- [10] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Mobile Systems, Applications, and Services (MobiSys)*.
- [11] Yue Jia, Ke Mao, and Mark Harman. 2016. MaJiCKe: Automated Android Testing Solutions. <http://www.majicke.com>. (2016).
- [12] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *European Software Engineering Conference and Foundations of Software Engineering (ES-EC/FSE)*.
- [13] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Foundations of Software Engineering (FSE)*.
- [14] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Software Testing and Analysis (ISSTA)*.
- [15] Atif M. Memon. 2007. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.* 17, 3 (2007).
- [16] Rickard Nilsson. 2015. ScalaCheck: Property-based Testing for Scala. <http://scalacheck.org/>. (2015).
- [17] Renas Reda. 2009. Robotium: User scenario testing for Android. <http://www.robotium.org>. (2009).