View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Lifestate: Event-Driven Protocols and Callback Control Flow

Shawn Meier
University of Colorado Boulder, USA
shawn.meier@colorado.edu

Sergio Mover
École Polytechnique, Institute Polytechnique de Paris, Palaiseau, France
sergio.mover@lix.polytechnique.fr

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

Abstract

Developing interactive applications (apps) against event-driven software frameworks such as Android is notoriously difficult. To create apps that behave as expected, developers must follow complex and often implicit *asynchronous programming protocols*. Such protocols intertwine the proper registering of callbacks to receive control from the framework with appropriate application-programming interface (API) calls that in turn affect the set of possible future callbacks. An app violates the protocol when, for example, it calls a particular API method in a state of the framework where such a call is invalid. What makes automated reasoning hard in this domain is largely what makes programming apps against such frameworks hard: the specification of the protocol is unclear, and the control flow is complex, asynchronous, and higher-order. In this paper, we tackle the problem of specifying and modeling event-driven application-programming protocols. In particular, we formalize a core meta-model that captures the dialogue between event-driven frameworks and application callbacks. Based on this meta-model, we define a language called *lifestate* that permits precise and formal descriptions of application-programming protocols and the callback control flow imposed by the event-driven framework. Lifestate unifies modeling what app callbacks can expect of the framework with specifying rules the app must respect when calling into the framework. In this way, we effectively combine lifecycle constraints and typestate rules. To evaluate the effectiveness of lifestate modeling, we provide a dynamic verification algorithm that takes as input a trace of execution of an app and a lifestate protocol specification to either produce a trace witnessing a protocol violation or a proof that no such trace is realizable.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases event-driven systems, application-programming protocols, application framework interfaces, callbacks, sound framework modeling, predictive dynamic verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.1

Related Version An extended version of the paper is available at [33], `https://arxiv.org/abs/1906.04924`.

Supplement Material ECOOP 2019 Artifact Evaluation approved artifact available at `https://dx.doi.org/10.4230/DARTS.5.2.13`

Funding This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0263.

Acknowledgements Many thanks to Edmund S. L. Lam, Chance Roberts, and Chou Yi for help in gathering traces, as well as Alberto Griggio for a convenient tool for running tests. We also thank Aleksandar Chakarov, Maxwell Russek, the Fixr Team, and the University of Colorado Programming Languages and Verification (CUPLV) Group for insightful discussions, as well as the anonymous reviewers for their helpful comments.

© Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang;
licensed under Creative Commons License CC-BY

```
try { progress.dismiss(); } catch (IllegalArgumentException ignored) {} // race condition?
```

Figure 1 A protocol "fix" [10]. The `dismiss` call throws an exception if called in an invalid state.

1 Introduction

We consider the essential problem of checking that an application (app) programmed against an event-driven framework respects the required application-programming protocol. In such frameworks, apps implement *callback* interfaces so that the app is notified when an *event* managed by the framework occurs (e.g., a user-interface (UI) button is pressed). The app may then delegate back to the framework through calls to the application programming interface (API), which we term *callin* by analogy to callback. To develop working apps, the programmer must reason about hidden *callback control flow* and often implicit asynchronous programming protocols.

Couple difficult reasoning about the space of possible control flow between callbacks with insufficient framework documentation, and it is unsurprising to find some questionable "fixes" for protocol violations. In Figure 1, we show a snippet found on GitHub. The "race condition?" comment is quoted directly from the app developer. The same asynchronous, implicitly defined, control flow that make it difficult for the app developer to reason about his app is also what makes verifying the absence of such protocol violations hard.

In this paper, we focus on the problems of specifying event-driven protocols (i.e., specifying when the invocation of a callin in the app code causes a protocol violation) and modeling the callback control flow (i.e., modeling the possible executions of callbacks).

Lifecycle Automata are Insufficient for Modeling Callback Control Flow. *Lifecycle* automata are a common representation used to model callback control flow that is both central to Android documentation [1, 39] and prior Android analysis techniques – both static and dynamic ones (e.g., [5, 32, 8]). In Figure 2, we show a lifecycle automaton for the `Activity` class of the Android framework. The black, solid edges are the edges present in the Android documentation [1] showing common callback control flow. These edges capture, for example, that the app first receives the `onStart` callback before entering a cycle between the `onResume` and the `onPause` callbacks. But this clean and simple class-based model quickly becomes insufficient when we look deeper.

First, there are complex relationships between the callbacks on "related" objects. For

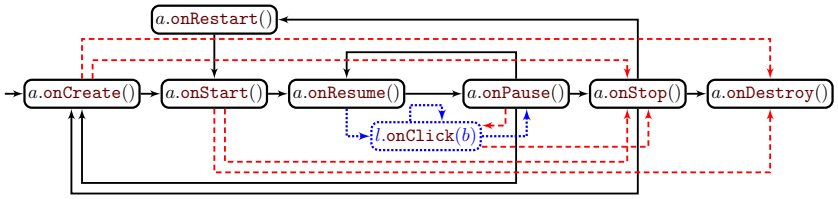


Figure 2 The `Activity` lifecycle automaton from the Android documentation [1] (shown with solid, black edges ⟶). To capture callback control flow between "related" objects, such component lifecycles are often instantiated and refined with additional callbacks from other objects, such as a `onClick` callback from the `OnClickListener` interface (shown with dotted, blue edges ⋯➤). But there are also less common callback control-flow paths that are often undocumented or easily missed, such as the additional edges induced by an invocation in the app code of the `finish` callin (shown as dashed, red edges - - ➤).

example, an `OnClickListener` object l with an `onClick` callback may be "registered" on a `View` object v that is "attached" to an `Activity` object a. Because of these relationships, the callback control flow we need to capture is somewhat described by modifying the lifecycle automaton for `Activity` a with the additional blue, dotted edges to and from `onClick` (implicitly for `OnClickListener` l) in Figure 2. This modified lifecycle encodes framework-specific knowledge that the `OnClickListener` l's `onClick` callback happens only in the "active" state of `Activity` a between its `onResume` and `onPause` callbacks, which typically requires a combination of static analysis on the app and hard-coded rules to connect callbacks on additional objects such as `OnClickListener`s to component lifecycles such as `Activity`. We refer to such callback control-flow models based on such refined lifecycle automatons as lifecycle++ models.

Second, there are less common framework-state changes that are difficult to capture soundly and precisely. For example, an analysis that relies on a callback control-flow model that does not consider the intertwined effect of a `finish` call may be unsound. The red, dashed edges represent callback control flow that are not documented (and thus missing from typical callback control flow models). Each one of these edges specifies different possible callback control flow that the framework imposes depending on *if and when* the app invokes the `finish` callin inside one of the `Activity`'s callbacks. Of course, the lifecycle automaton can be extended to include these red edges. However, this lifecycle automaton is now quite imprecise in the common case because it does not express precisely when certain callback control-flow paths are spurious (i.e., depending on where `finish` is not called). Figure 2 illustrates why developing callback control flow models is error prone: the effect of calls to `finish` are subtle and poorly understood.

It is simply too easy to miss possible callback control flow – an observation also made by Wang et al. [52] about lifecycle models. While lifecycle automata are useful for conveying the intuition of callback control flow, they are often insufficiently precise and easily unsound.

In this paper, we re-examine the process of modeling callback control flow. In prior work, modeling callback control flow was almost always a secondary concern in service to, and often built into, a specific program analysis where the analysis abstraction may reasonably mask unsound callback control flow. Instead, we consider modeling callback control flow independent of any analysis abstraction – we identify and formalize the key aspects to effectively model event-driven application-programming protocols at the app-framework interface, such as the effect of callin and callback invocations on the subsequent callback control flow, This first-principles approach enables us to *validate* callback control-flow soundness with real execution traces against the event-driven framework implementation. It is through this validation step that we discovered the red, dashed edges in Figure 2.

Contributions. We make the following contributions:

- We identify essential aspects of event-driven control flow and application-programming protocols to formalize a core abstract machine model λ_{life} (Section 3). This model provides a formal basis for thinking about event-driven frameworks and their application-programming protocols.
- We define a language for simultaneously capturing event-driven application-programming protocols and callback control flow called *lifestates*, which both *model* what callback invocations an app can expect from the framework and *specify* rules the app must respect when calling into the framework (Section 4). Intuitively, lifestates offer the ability to specify traces of the event-driven program in terms of an abstraction of the observable interface between the framework and the app. And thus, this definition leads to a methodology for empirically validating lifestate models against actual interaction traces.

- We define lifestate validation and dynamic lifestate verification. And then, we encode them as model checking problems (Section 5). Given an app-framework interaction trace and a lifestate model, validation checks that the trace is in the abstraction of the observable interface defined by the model. This validation can be done with corpora of traces recorded from any set of apps interacting with the same framework because, crucially, the lifestate model speaks only about the app-framework interface. Then, given a trace, dynamic lifestate verification attempts to prove the absence of a rearrangement of the recorded events that could cause a protocol violation. Rearranging the execution trace of events corresponds to exploring a different sequence of external inputs and hence discovering possible protocol violations not observed in the original trace.
- We implement our model validation and trace verification approach in a tool called Verivita and use it to empirically evaluate the soundness and precision of callback control flow models of Android (Section 6). Our results provide evidence for the hypotheses that lifecycle models, by themselves, are insufficiently precise to verify Android apps as conforming to the specified protocols, that model validation on large corpora of traces exposes surprising unsoundnesses, and that lifestates are indeed useful.

2 Overview: Specifying and Modeling Lifestates

Here, we illustrate the challenges in specifying and modeling event-driven application-programming protocols. In particular, we motivate the need for lifestates that permit specifying the intertwined effect of callin and callback invocations. We show that even if an app is buggy, it can be difficult to witness the violation of the Android application programming protocol. Then, more importantly, we show how an appropriate fix is both subtle to reason about and requires modeling the complex callback control flow that depends on the previous execution of not only the callbacks but also the callins.

Our running example (code shown in Figure 3) is inspired by actual issues in AntennaPod [16], a podcast manager with 100,000+ installs, and the Facebook SDK for Android [27]. The essence of the issue is that a potentially time-consuming background task is started by a user interaction and implemented using the `AsyncTask` framework class. Figure 3 shows buggy code that can potentially violate the application-programming protocol for

```
class RemoverActivity extends Activity {
 FeedRemover remover;
 void onCreate() {
1  Button button = ...;
2  remover = new FeedRemover(this);
3  button.setOnClickListener(
4   new OnClickListener() {
     void onClick(View view) {
5      remover.execute();   ⚠
     }
   });
 }
}
```

```
class FeedRemover extends AsyncTask {
 RemoverActivity activity;
 void doInBackground() {
  ... remove feed ...
 }
 void onPostExecute() {
  // return to previous activity
6  activity.finish();
 }
}
```

Figure 3 An example app that violates the protocol specified by the interaction of the Android framework components `AsyncTask`, `Button`, and `OnClickListener`. On line 5, `remover.execute()` (marked with ⚠) can throw an `IllegalStateException` if the `remover` task is already running.

AsyncTask. The `remover.execute()` call (marked with ⚠) throws an `IllegalStateException` if the *AsyncTask* t instance, pointed-to by `remover`, is already running. So a protocol rule for *AsyncTask* is that t`.execute()` cannot be called twice for the same *AsyncTask* t. The `IllegalStateException` type is commonly used to signal a protocol violation and has been shown to be a significant source of Android crashes [28].

In Figure 3, the `RemoverActivity` defines an app window that, on creation (via the *onCreate* callback), registers a click listener (via the `button.setOnClickListener(...)` call on line 3). This registration causes the framework to notify the app of a button click through the *onClick* method. When that happens, the *onClick* callback starts the `FeedRemover` asynchronous task (via the `remover.execute()` call on line 5). What to do asynchronously is defined in the *doInBackground* callback, and when the `FeedRemover` task is done, the framework delegates to the *onPostExecute* callback, which closes the `RemoverActivity` (via the call to `activity.finish()`).

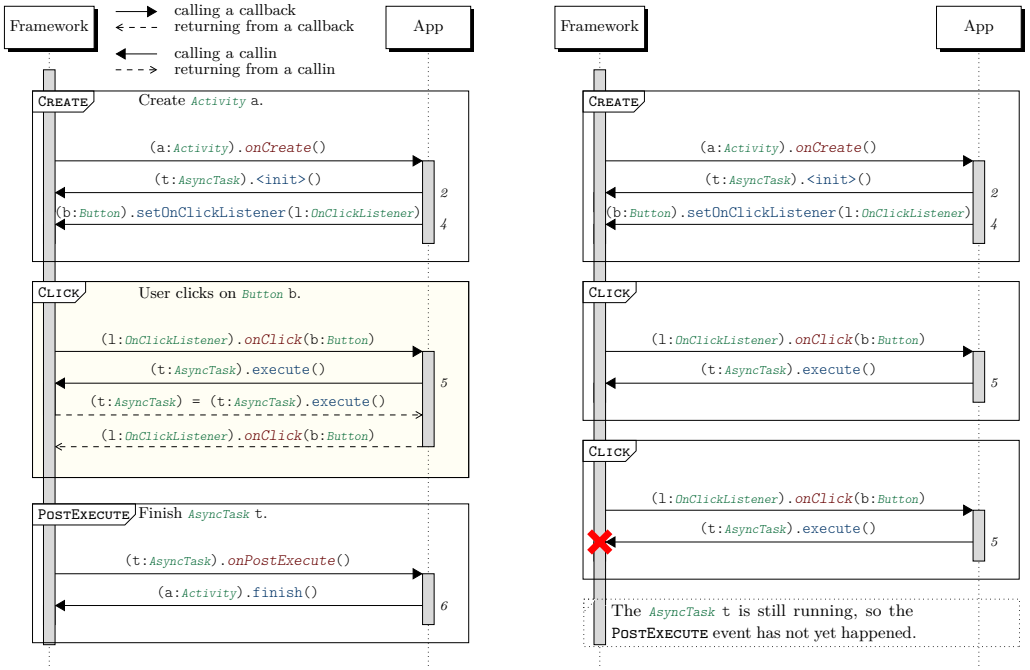
We diagram a common-case execution trace in Figure 4.a. Even though the app is buggy, the trace does not witness the protocol violation. The exception does not manifest because the user only clicks once (CLICK) before the `FeedRemover` task completes and generates the post-execute event (POSTEXECUTE). And so the `(t:`*AsyncTask*`).execute()` callin on the *AsyncTask* instance `t` is executed only once before the activity is closed (cf. the *onClick* and *onPostExecute* callbacks in Figure 3).

If typically the *Activity* is quickly destroyed after the button click, then seeing a protocol violation in a test is quite unlikely. However, it is possible to click a second time before the *AsyncTask* completes, thereby witnessing a protocol violation. We show this error trace in Figure 4.b: when the app invokes the callin `(t:`*AsyncTask*`).execute()` for the second time in the second CLICK event, the framework is in a state that does not allow this transition. We say that the callin invocation is *disallowed* at this point, and apps must only invoke allowed callins. While the original trace CREATE;CLICK;POSTEXECUTE does not concretely witness the protocol violation, it has sufficient information to predict the error trace CREATE;CLICK;CLICK. It may, however, be difficult to reproduce this error trace: the button must be pressed twice before the `activity.finish()` method is called by the POSTEXECUTE event destroying the *Activity*. But how can we predict this error trace from the original one?

2.1 Predict Violations from Recorded Interactions

We define the dynamic lifestate verification problem as predicting an error trace that (possibly) witnesses a protocol violation from a trace of interactions or proving that no such error trace exists. Concretely, the input to dynamic lifestate verification is an interaction trace like the one illustrated in Figure 4.a. These traces record the sequence of invocations and returns of callbacks and callins between the framework and the app that result from an interaction sequence. A recorded trace includes the concrete method arguments and return values (e.g., the instance `t` from the diagrams corresponds to a concrete memory address).

The main challenge, both for the app developer and dynamic lifestate verification, is that the relevant sequence of events that leads to a state where a callin is disallowed is *hidden* inside the framework. The developer must reason about the evolving internal state of the framework by considering the possible callback and callin interactions between the app and the framework to develop apps that both adhere to the protocol and behave intuitively. To find a reasonable fix for the buggy app from Figure 3, let us consider again the error trace shown in Figure 4.b. Here, the developer has to reason that the `(t:`*AsyncTask*`).execute()` callin is allowed as soon as `t` is initialized by the call to `(t:`*AsyncTask*`).<init>()` in the CREATE event and is *disallowed* just after the first call to `(t:`*AsyncTask*`).execute()` in the first CLICK



(4.a) A trace that does not witness a protocol violation since the callin `(t:AsyncTask).execute()` on `t` is executed only once.

(4.b) The CREATE;CLICK;CLICK interaction sequences witnesses the no-`execute`-call-on-already-executing-`AsyncTask` protocol violation.

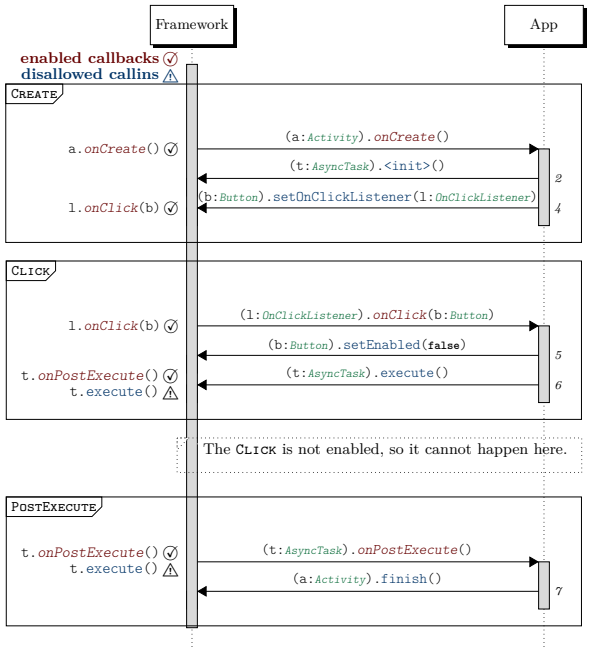
Figure 4 We visualize the interface between an event-driven framework and an app as a dialog between two components. With execution time flowing downwards as a sequence events, control begins on the left with the framework receiving an event. Focusing on the highlighted CLICK event in Figure 4.a, when a user clicks on the button corresponding to object `b` of type `Button`, the *onClick* callback is invoked by the framework on the registered listener `l`. For clarity, we write method invocations with type annotations (e.g., `(l:OnClickListener).onClick(b:Button)`), and variables `b` and `l` stand for some concrete instances (rather than program or symbolic variables). The app then delegates back to the framework by calling an API method to start an asynchronous task `t` via `(t:AsyncTask).execute()`. To connect with the app source code, we label the callins originating from the app timeline with the corresponding program point numbers in Figure 3. Here, we can see clearly a *callback* as any app method that the framework can call (i.e., with an arrow to the right ⟶), and a *callin* as any framework method that an app can call (i.e., with an arrow to the left ⟵). We show returns with dashed arrows (but sometimes elide them when they are unimportant).

event. That is, the developer must reason about what sequence of events and callins determine when a callin is allowed or disallowed. Since callins are invoked inside callback methods and callback methods are in turn invoked by the framework to notify the app of an event, the internal framework state determines what events can happen when and hence the callback control flow. In particular, the internal framework state determines when the CREATE and CLICK events are *enabled* (i.e., can happen) during the execution. Thus to properly fix this app, the developer must ensure that CREATE happens before a CLICK and then only a single CLICK happens before a POSTEXECUTE. How can the app developer constrain the external interaction sequence to conform to this property?

In Figure 5.a, we show a fix based on the above insight that is particularly challenging to verify. The fix adds line 5 that disables `Button button` to indicate when the task has already been started. Thus, this modified version does not violate the no-`execute`-call-on-already-

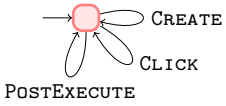
```
class RemoverActivity extends Activity {
 FeedRemover remover;
 void onCreate() {
1  Button button = ...;
2  remover = new FeedRemover(this);
3  button.setOnClickListener(
4   new OnClickListener() {
     void onClick(View view) {
+5    button.setEnabled(false);
6     remover.execute();   ⚠
     }
   });
 }
}

class FeedRemover extends AsyncTask {
 RemoverActivity activity;
 void doInBackground() {
  ... remove feed ...
 }
 void onPostExecute() {
  // return to previous activity
7  activity.finish();
 }
}
```

(5.a) A `button.setEnabled(false)` call prevents the user from clicking, triggering the *onClick* callback.

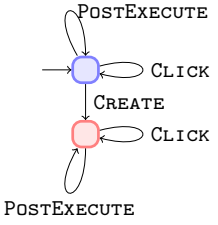


(5.b) The enabled callbacks and disallowed callins are shown along the CREATE;CLICK;POSTEXECUTE trace from the fixed app in 5.a.

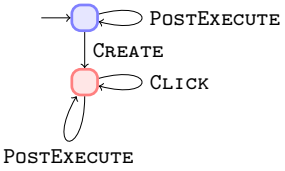
Figure 5 A fixed version of the app from Figure 3 that adheres to the application-programming protocol. The annotations in 5.b show that after the call to `(b:Button).setEnabled(false)`, the `l.onClick(b)` callback is no longer enabled, and thus the app can assume that the framework will not call `l.onClick(b)` at this point.



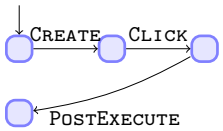
(6.a) False alarm on the trivially sound, unconstrained, "top" abstraction.



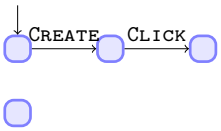
(6.b) False alarm on the *Activity* lifecycle-refined abstraction.



(6.c) False alarm on the lifecycle with the CLICK restricted to the active *Activity* state (as shown in Figure 2).



(6.d) Verified safe when we consider the effect of *Button*.setEnabled(...).



(6.e) This unsound abstraction is missing the POSTEXECUTE edge.

Figure 6 In previous works, models are generated for an application restricting the possible order of callbacks. In this figure, we show four sound abstractions with different levels of precision, indicating whether they can verify our fixed application 5.a, as well as one unsound abstraction.

executing-*AsyncTask* protocol on line 6. To reason precisely enough about this fix, we must know that the `button.setEnabled(false)` call changes internal framework state that prevents the *onClick* from happening again. Note that this need to reason about complex control flow arises from the interactions between just two framework types *Button* and *AsyncTask* – not to mention that these two are amongst the simplest framework types in Android. There is a clear need here for better automated reasoning tools to support the app developer.

Verivita Approach. Our dynamic verification approach explores all the possible sequences of interactions that can be obtained by replicating, removing, and reordering the events in a trace. By rearranging event traces, the algorithm statically explores different input sequences of events that a user interaction could generate. The algorithm applied to the CREATE;CLICK;POSTEXECUTE trace in Figure 4.a from the buggy app version indeed yields the error trace CREATE;CLICK;CLICK (shown in Figure 4.b). But more critically, our approach also makes it possible to prove that the fixed app version does not have any traces that violates the protocol (by rearranging CREATE;CLICK;POSTEXECUTE).

Central to our approach is capturing the essential, hidden framework state – tracking the set of enabled callbacks and the set of disallowed callins. Figure 5.b illustrates this model state along a trace from the fixed app. After the first CLICK, the application disables the button to prevent a second CLICK via the call to `(b:Button).setEnabled(false)`, which at that point removes `l.onClick(b)` from the set of enabled callbacks the framework can trigger.

Verivita addresses the dynamic verification problem by reducing it to a model checking problem. The model is a transition system with

(i) states abstracting the set of enabled callbacks and disallowed callins and
(ii) transitions capturing the possible replication, removing, and reordering of a given interaction trace.

The safety property of interest is that the transition system never visits a disallowed callin. How can we construct such a transition system that over-approximates concrete behavior while being precise enough to make alarm triage feasible? As alluded to in Section 1, lifestate specification is crucial here.

2.2 Specify Event-Driven Protocols and Model Callback Control Flow

In Figure 6, we illustrate the essence of callback control-flow modeling as finite-state automata that over-approximate rearrangements of the CREATE;CLICK;POSTEXECUTE trace shown in Figure 5.b. Automaton 6.a exhibits the trivially sound, unconstrained, "top" abstraction that considers all replications, removals, and reorderings of the interaction trace. This abstraction is the one that assumes all callbacks are always enabled. Since a possible trace in this abstraction includes two CLICK events, a sound verifier must alarm. Meanwhile, Automaton 6.b shows a refined abstraction encoding the Android-specific *Activity* lifecycle. The abstraction is framework-specific but application-independent and captures that the CREATE event cannot happen more than once. The abstraction shown by Automaton 6.b is also insufficient to verify the trace from the fixed app because two CLICK events are still possible.

Automaton 6.c shows a refined, lifecycle++ abstraction that considers the *Activity* lifecycle with additional constraints on an "attached" CLICK event. This abstraction is representative of the current practice in callback control-flow models (e.g., [5, 32, 8]). While Automaton 6.c restricts the CLICK event to come only after the CREATE event, the abstraction is still too over-approximate to verify that the trace from the fixed app is safe – two CLICK events are still possible with this model. But worse is that this model is still, in essence, a lifecycle model

that is constrained by Android-specific notions like `View` attachment, `Listener` registration, and the "live" portion of lifecycles. In existing analysis tools, such constrained lifecycle models are typically hard-coded into the analyzer.

We need a better way to capture how the application may affect callback control flow. In this example, we need to capture the effect of the callin `button.setEnabled(false)` at line 5 in Figure 5.a, which is the only difference with the buggy version in Figure 3. The modeling needs to be expressive to remove such infeasible traces and compositional to express state changes independently. Thus, the role of lifestate specification is to describe how the internal model state is updated by observing the history of intertwined callback and callin invocations. For example, we write

$(\ell_b : \mathit{Button}).\texttt{setEnabled}(\texttt{false}) \nrightarrow (\ell_l : \mathit{OnClickListener}).\mathit{onClick}(\ell_b : \mathit{Button})$ (for all ℓ_l, ℓ_b)

to model when $(\ell_b : \mathit{Button}).\texttt{setEnabled}(\texttt{false})$ is invoked, the click callback is *disabled* on the same button ℓ_b (on all listeners ℓ_l). Also, we similarly specify the safety property of interest

$(\ell_t : \mathit{AsyncTask}).\texttt{execute}() \nrightarrow (\ell_t : \mathit{AsyncTask}).\texttt{execute}()$ (for all ℓ_t)

that when $(\ell_t : \mathit{AsyncTask}).\texttt{execute}()$ is called on a task ℓ_t, it *disallows* itself. And analogously, lifestates include specification forms for *enabling callbacks* or *allowing callins*.

Lifestate uniformly models the callback control-flow and specifies event-driven application-programming protocols. The rules that enable and disable callbacks model what callbacks the framework can invoke at a specific point in the execution of the application, while the rules that disallow and allow callins specify what callins the application must invoke to respect the protocol. What makes lifestate unique compared to typestates [50] or lifecycle automata is this unification of the intertwined effects of callins and callbacks on each other.

The complexity of the implicit callback control flow is what makes expressing and writing correct models challenging. An issue whose importance is often under-estimated when developing callback control-flow models is how much the model faithfully reflects the framework semantics. How can we *validate* that a lifestate specification is a correct model of the event-driven framework?

Validating Event-Driven Programming Protocols. As argued in Section 1, a key concern when developing a framework model is that it must over-approximate the possible real behavior of the application. The "top" model as shown in Automaton 6.a trivially satisfies this property, and it may be reasonable to validate an application-independent lifecycle model like Automaton 6.c. However, as we have seen, verifying correct usage of event-driven protocols typically requires callback control-flow models with significantly more precision.

Automaton 6.d shows a correct lifestate-abstraction that contains an edge labeled PostExecute. We express this edge with the rule shown below:

$(\ell_t : \mathit{AsyncTask}).\texttt{execute}() \rightarrow (\ell_t : \mathit{AsyncTask}).\mathit{onPostExecute}()$ (for all ℓ_t)

This rule states that when $(\ell_t : \mathit{AsyncTask}).\texttt{execute}()$ is called, its effect is to enable the callback $(\ell_t : \mathit{AsyncTask}).\mathit{onPostExecute}()$ on the $\mathit{AsyncTask}$ ℓ_t.

If we do not model this rule, we obtain the abstraction in Automaton 6.e. The lifestate model is *unsound* since it misses the PostExecute edge.

The trace Create;Click;PostExecute shown in Figure 5.b is a witness of the unsoundness of the abstraction: Automaton 6.e accepts only proper prefixes of the trace (e.g., Create;Click), and hence the abstraction does not capture all the possible traces of the app.

We can thus use interaction traces to validate lifestate rules: a set of lifestate rules is valid if the abstraction accepts all the interaction traces. The validation applied to the abstraction shown in Automaton 6.e demonstrates that the abstraction accepts CREATE;CLICK as the longest prefix of the trace CREATE;CLICK;POSTEXECUTE. This information helps to localize the cause for unsoundness since we know that after the sequence CREATE;CLICK, the callback POSTEXECUTE is (erroneously) disabled.

The encoding of the abstraction from lifestate rules is a central step to perform model validation and dynamic verification. At this point, we still cannot directly encode the abstraction since the lifestate rules contain universally-quantified variables. How can we encode the lifestate abstraction as a transition system amenable to check language inclusion for validation, and to check safety properties for dynamic verification?

From Specification to Validation and Verification. Generalizing slightly, we use the term *message* to refer to any observable interaction between the framework and the app. Messages consist of invocations to and returns from callbacks and callins. The abstract state of the transition system is then a pair consisting of the *permitted-back messages* from framework to app and the *prohibited-in messages* from app to framework. And thus generalizing the example rules shown above, a lifestate specification is a set of rules whose meaning is,

> *If* the message history *matches* $\boldsymbol{r}$, *then* the abstract state is updated according to the specified *effect* on the set of permitted-back and prohibited-in messages.

There are many possible choices and tradeoffs for the matching language $\boldsymbol{r}$. As is common, we consider a regular expression-based (i.e., finite automata-based) matching language.

We exploit the structure of the validation and dynamic verification problem to encode the lifestate abstraction. In both problems, the set of possible objects and parameters is finite and determined by the messages recorded in the trace. We exploit this property to obtain a set of *ground* rules (rules without variables). We can then encode each ground rule in a transition system. Since the rule is ground, the encoding is standard: each regular expression is converted to an automaton and then encoded in the transition system, changing the permitted-prohibited state as soon as the transition system visits a trace accepted by the regular expression, which implicitly yields a model like automata 6.d.

Lifestate offers a general and flexible way to specify the possible future messages in terms of observing the past history of messages. It, however, essentially leaves the definition of messages and what is observable abstract. What observables characterize the interactions between an event-driven framework and an app that interfaces with it? And how do these observables define event-driven application-programming protocols and callback control flow?

2.3 Event-Driven App-Framework Interfaces

Lifestate rules are agnostic to the kinds of messages they match and effects they capture on the internal abstract state. To give meaning to lifestates, we formalize the essential aspects of the app-framework interface in an abstract machine model called λ_{life} in Section 3. This abstract machine model formally characterizes what we consider an *event-driven framework*. The λ_{life} abstract machine crisply defines the messages that the app and the framework code exchange and a formal correspondence between concrete executions of the program and the app-framework interface. We use this formal correspondence to define the semantics of the lifestate framework model, its validation problem, and protocol verification.

We do not intend for λ_{life} to capture all aspects of something as complex as Android; rather, the purpose of λ_{life} is to define a "contract" by which to consider a concrete event-driven

framework implementation. And thus, λ_{life} also defines the dynamic-analysis instrumentation we perform to record observable traces from Android applications that we then input to the Verivita tool to either validate a specification or verify protocol violations.

Preview. We have given a top-down overview of our approach, motivating with the dynamic protocol verification problem the need for having both a precise callback control-flow model and an event-driven protocol specification. We also presented how the lifestate language addresses this need capturing the intertwined effect of callins and callbacks. In the next sections, we detail our approach in a bottom-up manner – beginning with formalizing the λ_{life} abstract machine model. We show that, assuming such a model of execution, it is possible to provide a sound abstraction of the framework (i.e., no real behavior of the framework is missed by the abstraction) expressed with a lifestate model. We then formalize how we validate such models and how we use lifestates to verify the absence of protocol violations.

3 Defining Event-Driven Application-Programming Protocols

Following Section 2, we want to capture the essence of the app-framework interface with respect to framework-imposed programming protocols. To do so, we first formalize a small-step operational semantics for event-driven programs with an abstract machine model λ_{life}. The λ_{life} abstract machine draws on standard techniques but explicitly highlights enabled events and disallowed callins to precisely define event-driven protocols. We then instrument this semantics to formalize the interface of the event-driven framework with an app, thereby defining the traces of the observable app-framework interface of a λ_{life} program.

This language is intentionally minimalistic to center on capturing just the interface between event-driven frameworks and their client applications. By design, we leave out many aspects of real-world event-driven framework implementations (e.g., Android, Swing, or Node.js), such as typing, object-orientation, and module systems that are not needed for formalizing the dialogue between frameworks and their apps (cf. Section 2). Our intent is to illustrate, through examples, that event-driven frameworks could be implemented in λ_{life} and that λ_{life} makes explicit the app-framework interface to define *observable traces* consisting of *back-messages* and *in-messages* (Section 3.3).

3.1 Syntax: Enabling, Disabling, Allowing, and Disallowing

The syntax of λ_{life} is shown at the top of Figure 7.a, which is a λ-calculus in a let-normal form. The first two cases of expressions e split the standard call-by-value function application into multiple steps (similar to call-by-push-value [30]). The `bind` $\lambda\ v$ expression creates a thunk $\kappa = \lambda[v]$ by binding a function value λ with an argument value v. We abuse notation slightly by using λ as the meta-variable for function values (rather than as a terminal symbol). A thunk may be forced by direct invocation `invoke` κ – or indirectly via event dispatch.

▶ **Example 1** (Applying a Function). Let `t` be bound to an *AsyncTask* and *onPostExecute* to an app-defined callback (e.g., *onPostExecute* from Figure 5.a), then the direct invocation of a callback from the framework can be modeled by the two steps of binding and then invoking:

```
let cb = bind onPostExecute t in invoke cb
```

Now in λ_{life}, a thunk κ may or may not have the *permission* to be forced. Revoking and re-granting the permission to force a thunk via direct invocation is captured by the expressions `disallow` κ and `allow` κ, respectively. A protocol violation can thus be modeled by an application invoking a *disallowed* thunk.

$$
\begin{array}{rrll}
\text{expressions} & e \in \mathbf{Expr} ::= & \texttt{bind}\ v_1\ v_2 \mid \texttt{invoke}\ v \mid \texttt{disallow}\ v \mid \texttt{allow}\ v & \text{thunks and calls} \\
& & \mid \texttt{enable}\ v \mid \texttt{disable}\ v \mid \texttt{force}\ \kappa & \text{events and forcing} \\
& & \mid v \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \cdots & \text{other expressions} \\
\text{functions} & \lambda ::= & x\ \texttt{=>}_g\ e & \\
\text{packages} & g ::= & \texttt{app} \mid \texttt{fwk} & \\
\text{values} & v \in \mathbf{Val} ::= & x \mid \lambda \mid \kappa \mid () \mid \cdots \mid \texttt{thk} &
\end{array}
$$

$$
\begin{array}{llll}
\text{variables} & x \in \mathbf{Var} & \text{thunks} \quad \kappa \in \mathbf{Thunk} ::= \lambda[v] & \text{thunk stores} \quad \mu, \nu ::= \cdot \mid \mu;\kappa \\
\text{continuations} & k ::= \bullet \mid k \triangleright x.e \mid \kappa \mid k \gg \kappa & \text{states} \quad \sigma \in \mathbf{State} ::= \langle e, \mu, \nu, k \rangle \mid \mathsf{bad} &
\end{array}
$$

(7.a) The syntax and the semantic domains.

$$\boxed{\sigma \longrightarrow \sigma'}$$

$$\textsc{Enable}\ \frac{}{\langle \texttt{enable}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \kappa, \mu; \kappa, \nu, k \rangle} \qquad \textsc{Disable}\ \frac{}{\langle \texttt{disable}\ \kappa, \mu; \kappa, \nu, k \rangle \longrightarrow \langle \kappa, \mu, \nu, k \rangle} \qquad \textsc{Event}\ \frac{\kappa \in \mu}{\langle v, \mu, \nu, \bullet \rangle \longrightarrow \langle \texttt{force}\ \kappa, \mu, \nu, \kappa \rangle}$$

$$\textsc{Disallow}\ \frac{}{\langle \texttt{disallow}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \kappa, \mu, \nu; \kappa, k \rangle} \qquad \textsc{Allow}\ \frac{}{\langle \texttt{allow}\ \kappa, \mu, \nu; \kappa, k \rangle \longrightarrow \langle \kappa, \mu, \nu, k \rangle}$$

$$\textsc{Invoke}\ \frac{\kappa \notin \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle \texttt{force}\ \kappa, \mu, \nu, k \rangle} \qquad \textsc{InvokeDisallowed}\ \frac{\kappa \in \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k \rangle \longrightarrow \mathsf{bad}} \qquad \textsc{Bind}\ \frac{}{\langle \texttt{bind}\ \lambda\ v, \mu, \nu, k \rangle \longrightarrow \langle \lambda[v], \mu, \nu, k \rangle}$$

$$\textsc{Force}\ \frac{(x'\ \texttt{=>}_{g'}\ e')[v'] = \kappa}{\langle \texttt{force}\ \kappa, \mu, \nu, k \rangle \longrightarrow \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa \rangle} \qquad \textsc{Return}\ \frac{}{\langle v, \mu, \nu, k \gg \kappa \rangle \longrightarrow \langle v, \mu, \nu, k \rangle} \qquad \textsc{Finish}\ \frac{}{\langle v, \mu, \nu, \kappa \rangle \longrightarrow \langle v, \mu, \nu, \bullet \rangle}$$

$$\textsc{Let}\ \frac{}{\langle \texttt{let}\ x = e_1\ \texttt{in}\ e_2, \mu, \nu, k \rangle \longrightarrow \langle e_1, \mu, \nu, k \triangleright x.e_2 \rangle} \qquad \textsc{Continue}\ \frac{}{\langle v, \mu, \nu, k \triangleright x.e_2 \rangle \longrightarrow \langle [v/x]e_2, \mu, \nu, k \rangle}$$

(7.b) Semantics. Explicitly enable, disable, disallow, and allow thunks.

Figure 7 λ_{life}, a core model of event-driven programs capturing *enabledness* of events and *disallowedness* of invocations.

The direct invocation expressions are mirrored with expressions for event dispatch. An `enable` κ expression *enables* a thunk κ for the external event-processing system (i.e., gives the system permission to force the thunk κ), while the `disable` κ expression *disables* the thunk κ.

▶ **Example 2** (Enabling an Event)**.** Let `t` be bound to an `AsyncTask` and `handlePostExecute` to an internal framework-defined function for handling a post-execute event, then enqueuing such an event can be modeled by the two steps of binding then enabling:

```
let h = bind handlePostExecute t in enable h
```

By separating function application and event dispatch into binding to create a thunk $\kappa = \lambda[v]$ and then forcing it, we uniformly make thunks the value form that can be granted permission to be invoked (via `allow` κ) or for event dispatch (via `enable` κ). The `force` κ expression is then an intermediate that represents a thunk that is forcible (i.e., has been permitted for forcing via `allow` κ or `enable` κ).

The remainder of the syntax is the standard part of the language: values v, variable binding `let` x `=` e_1 `in` e_2, and whatever other operations of interest $\cdots$ (e.g., arithmetic, tuples, control flow, heap manipulation). That is, we have made explicit the expressions to expose the app-framework interface and can imagine whatever standard language features in $\cdots$ in framework implementations. The values v of this expression language are variables x,

function values λ, thunks κ, unit (), and whatever other base values of interest $\cdots$. Two exceptions are that (1) the currently active thunk is available via the `thk` identifier (see Section 3.2) and (2) functions x `=>`$_g$ e are tagged with a package g (see Section 3.3).

3.2 Semantics: Protocol Violations

At the bottom of Figure 7.a, we consider an abstract machine model enriched with an *enabled-events* store μ, and a *disallowed-calls* store ν. These are finite sets of thunks, which we write as a list $\kappa_1;\cdots;\kappa_n$. The enabled-events store μ saves thunks that are permitted to be forced by the event loop, while the disallowed-calls store ν lists thunks that are *not* permitted to be forced by invocation. These thunk stores make explicit the event-driven application-programming protocol (that might otherwise be implicit in, for example, flag fields and conditional guards).

A machine state $\sigma : \langle e, \mu, \nu, k\rangle$ consists of an expression e, enabled events μ, disallowed calls ν, and a continuation k. A continuation k can be the top-level continuation $\bullet$ or a continuation for returning to the body of a `let` expression, which are standard. Continuations are also used to record the active thunk via κ and $k \gg \kappa$ corresponding to the run-time stack of activation records. These continuation forms record the active thunk and are for defining messages and the app-framework interface in Section 3.3. Since events occur non-deterministically and return to the main event loop, it is reasonable to assume that a state σ should also include a heap, and the expression language should have heap-manipulating operations through which events communicate. We do not, however, formalize heap operations since they are standard.

We define an operational semantics in terms of the judgment form $\sigma \longrightarrow \sigma'$ for a small-step transition relation. In Figure 7.b, we show the inference rules defining the reduction steps related to enabling-disabling, disallowing-allowing, invoking, creating, and finally forcing thunks. The rules follow closely the informal semantics discussed in Section 3.1. Observe that Enable and Allow both permit a thunk to be forced, and Disable and Disallow remove the permission to be forced for a thunk. The difference between Enable and Disable versus Allow and Disallow is that the former pair modifies the enabled events μ, while the latter touches the disallowed calls ν.

The Event rule says that when the expression is a value v and the continuation is the top-level continuation $\bullet$, then a thunk is non-deterministically chosen from the enabled events μ to force. Observe that an enabled event remains enabled after an Event reduction, hence λ_{life} can model both events that do not self-disable (e.g., the `Click` event from Section 2) and those that are self-disabling (e.g., the `Create` event). The Invoke rule has a similar effect, but it checks that the given thunk is not disallowed in ν before forcing. The InvokeDisallowed rule states that a disallowed thunk terminates the program in the bad state. And the Bind rule simply states that thunks are created by binding an actual argument to a function value.

The Force rule implements the "actual application" that reduces to the function body e' with the argument v' substituted for the formal x' and the thunk substituted for the identifier `thk`, that is, $[\kappa/\texttt{thk}][v'/x']e'$. To record the stack of activations, we push the forced thunk κ on the continuation (via $k \gg \kappa$). The Return and Finish rules simply state that the recorded thunk κ frames are popped on return from a Force and Event, respectively. The Return rule returns to the caller via the continuation k, while the Finish rule returns to the top-level event loop $\bullet$. The last line with the Let and Continue rules describe, in a standard way, evaluating let-binding.

A program e violates the event-driven protocol if it ends in the bad state from the initial state $\langle e, \cdot, \cdot, \bullet\rangle$.

▶ **Example 3** (Asserting a Protocol Property.)**.** The no-`execute`-call-on-already-executing-*AsyncTask* protocol can be captured by a **`disallow`**. We let `execute` be a framework function (i.e., tagged with **`fwk`**) that takes an *AsyncTask* `t`.

```
let execute = (t =>fwk disallow thk; ... let h = bind handlePostExecute t in enable h)
```

The `execute` function first disallows itself (via **`disallow thk`**) and does some work (via ...) before enabling the `handlePostExecute` event handler (writing $e_1; e_2$ as syntactic sugar for sequencing). The **`disallow thk`** asserts that this thunk cannot be forced again – doing so would result in a protocol violation (i.e., the bad state).

In contrast to an event-driven framework implementation, the state of a λ_{life} program does not have a queue. As we see here, a queue is an implementation detail not relevant for capturing event-driven programming protocols. Instead, λ_{life} models the external environment, such as, user interactions, by the non-deterministic selection of an enabled event.

3.3 Messages, Observable Traces, and the App-Framework Interface

To minimally capture how a program is composed of separate framework and app code, we add some simple syntactic restrictions to λ_{life} programs. Function values λ tagged with the **`fwk`** are framework code and the **`app`** tag labels app code. We express a framework implementation $\langle \mathbf{Fun}_{\mathbf{fwk}}, \lambda_{\mathbf{init}} \rangle$ with a finite set of framework functions $\mathbf{Fun}_{\mathbf{fwk}}$ and an initialization function $\lambda_{\mathbf{init}} \in \mathbf{Fun}_{\mathbf{fwk}}$. A program e uses the framework implementation if it first invokes the function $\lambda_{\mathbf{init}}$, and all the functions labeled as **`fwk`** in e are from $\mathbf{Fun}_{\mathbf{fwk}}$.

In a typical, real-world framework implementation, the framework implicitly defines the application-programming protocol with internal state to check for protocol violations. The Enable, Disable, Allow, and Disallow transitions make explicit the event-driven protocol specification in λ_{life}. Thus, it is straightforward to capture that framework-defined protocols by syntactically prohibiting the app from using **`enable`** κ, **`disable`** κ, **`allow`** κ, and **`disallow`** κ. Again, the enabled-event store μ and the disallowed-call store ν in λ_{life} can be seen as making explicit the implicit internal state of event-driven frameworks that define their application-programming protocols.

The app interacts with the framework only by "exchanging messages." The app-framework dialogue diagrams from Figures 4.a, 4.b, and 5.b depicts the notion of messages as arrows back-and-forth between the framework and the app. The framework invokes callbacks and returns from callins (the arrows from left to right), while the app invokes callins and returns from callbacks (the arrows from right to left). To formalize this dialogue, we label the observable transitions in the judgment form and define an *observable trace* – a trace formed only by these observable messages. Being internal to the framework, the Enable, Disable, Allow, and Disallow transitions are hidden, or unobservable, to the app.

In Figure 8, we define the judgment form $\sigma \xrightarrow{m} \sigma'$, which instruments our small-step transition relation $\sigma \longrightarrow \sigma'$ with message m. Recall from Section 2 that we define a callback as an invocation that transitions from framework to app code and a callin as an invocation from app to framework code. In λ_{life}, this definition is captured crisply by the execution context k in which a thunk is forced. In particular, we say that a thunk κ is a callback invocation $\mathsf{cb}\ \kappa$ if the underlying callee function is an app function (package **`app`**), and it is called from a framework function (package **`fwk`**) as in rule ForceCallback. The thk$(\cdot)$ function inspects the continuation for the running, caller thunk. The pkg$(\cdot)$ function gets the package of the running thunk.

Analogously, a thunk κ is a callin $\mathsf{ci}\ \kappa$ if the callee function is in the **`fwk`** package, and the caller thunk is in the **`app`** package via rule ForceCallin.

back-messages $m^{\mathbf{bk}} \in \Sigma^{\mathbf{bk}} ::= \mathsf{cb}\ \kappa \mid v = \mathsf{ciret}\ \kappa$ in-messages $m^{\mathbf{in}} \in \Sigma^{\mathbf{in}} ::= \mathsf{ci}\ \kappa \mid v = \mathsf{cbret}\ \kappa$

messages $m \in \Sigma ::= m^{\mathbf{bk}} \mid m^{\mathbf{in}} \mid \mathsf{dis}\ m^{\mathbf{in}} \mid \epsilon$ observable traces $\omega \in \Sigma^* ::= \epsilon \mid \omega m$

$\boxed{\sigma \xrightarrow{m} \sigma'}$

ForceCallback
$$\frac{(x' \texttt{=>}_{\texttt{app}} e')[v'] = \kappa \qquad \texttt{fwk} = \mathrm{pkg}(k)}{\langle \texttt{force}\ \kappa, \mu, \nu, k\rangle \xrightarrow{\mathsf{cb}\ \kappa} \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa\rangle}$$

ForceCallin
$$\frac{(x' \texttt{=>}_{\texttt{fwk}} e')[v'] = \kappa \qquad \texttt{app} = \mathrm{pkg}(k)}{\langle \texttt{force}\ \kappa, \mu, \nu, k\rangle \xrightarrow{\mathsf{ci}\ \kappa} \langle [\kappa/\texttt{thk}][v'/x']e', \mu, \nu, k \gg \kappa\rangle}$$

ReturnCallin
$$\frac{(x' \texttt{=>}_{\texttt{fwk}} e')[v'] = \kappa \qquad \texttt{app} = \mathrm{pkg}(k)}{\langle v, \mu, \nu, k \gg \kappa\rangle \xrightarrow{v = \mathsf{ciret}\ \kappa} \langle v, \mu, \nu, k\rangle}$$

ReturnCallback
$$\frac{(x' \texttt{=>}_{\texttt{app}} e')[v'] = \kappa \qquad \texttt{fwk} = \mathrm{pkg}(k)}{\langle v, \mu, \nu, k \gg \kappa\rangle \xrightarrow{v = \mathsf{cbret}\ \kappa} \langle v, \mu, \nu, k\rangle}$$

InvokeDisallowed
$$\frac{\kappa \in \nu}{\langle \texttt{invoke}\ \kappa, \mu, \nu, k\rangle \xrightarrow{\mathsf{dis\ ci}\ \kappa} \mathsf{bad}}$$

$\mathrm{thk}(\kappa) \stackrel{\text{def}}{=} \mathrm{thk}(k \gg \kappa) \stackrel{\text{def}}{=} \kappa \qquad \mathrm{thk}(k \triangleright x.e) \stackrel{\text{def}}{=} \mathrm{thk}(k) \qquad \mathrm{pkg}(k) \stackrel{\text{def}}{=} g \text{ if } (x \texttt{=>}_g e)[v] = \mathrm{thk}(k)$

Figure 8 The instrumented transition relation $\sigma \xrightarrow{m} \sigma'$ defines the app-framework interface and observing the event-driven protocol.

▶ **Example 4** (Observing a Callback). Letting `handlePostExecute` be a framework function (i.e., in package `fwk`) and *`onPostExecute`* be an `app` function, the observable transition from the framework to the app defines the forcing of `cb` as a callback:

```
let onPostExecute = (t =>app ...) in
let handlePostExecute = (t =>fwk let cb = bind onPostExecute t in invoke cb) in
```

In the above, we focused on the transition back-and-forth between framework and app code via calls. Returning from calls can also be seen as a "message exchange" with a return from a callin as another kind of *back-message* going from framework code to app code (left-to-right in the figures from Section 2). We write a callin-return back-message $v = \mathsf{ciret}\ \kappa$ indicating the returning thunk κ with return value v. Likewise, a return from a callback is another kind *in-message* going from app code to framework code (right-to-left). We instrument returns in a similar way to forcings with the return back-message with ReturnCallin and the return in-message with ReturnCallback.

Finally to make explicit protocol violations, we instrument the InvokeDisallowed rule to record the disallowed-callin invocations. These rules replace the corresponding rules Force, Return, and InvokeDisallowed from Figure 7.b. For replacing the Force and Return rules, we elide two rules, one for each, where there is no switch in packages (i.e., $g' = \mathrm{pkg}(k)$ where g' is the package of the callee message). These "uninteresting" rules and the remaining rules defining the original transition relation $\sigma \longrightarrow \sigma'$ not discussed here are simply copied over with an empty message label ϵ.

Observable Traces and Dynamic-Analysis Instrumentation. As described above, the app-framework interface is defined by the possible messages that can exchanged where messages consist of callback-callin invocations and their returns. A possible app-framework interaction is thus a trace of such observable messages.

▶ **Definition 5** (App-Framework Interactions as Observable Traces). *Let* $\mathrm{paths}(e)$ *be the path semantics of* λ_{life} *expressions* e *that collects the finite sequences of alternating state-transition-state* $\sigma m \sigma'$ *triples according to the instrumented transition relation* $\sigma \xrightarrow{m} \sigma'$*. Then, an* observable trace *is a finite sequence of messages* $\omega\colon m_1 \ldots m_n$ *obtained from a path by dropping the intermediate states and keeping the non-*ϵ *messages. We write* $[\![e]\!]$ *for the set of the observable traces obtained from the set of paths,* $\mathrm{paths}(e)$*, of an expression* e*.*

states $\hat{\sigma} ::= \langle \hat{\mu}, \hat{\nu}, \omega \rangle \mid \mathsf{bad}\ \omega$ permitted-back $\hat{\mu} ::= \cdot \mid \hat{\mu}; m^{\mathbf{bk}}$ prohibited-in $\hat{\nu} ::= \cdot \mid \hat{\nu}; m^{\mathbf{in}}$

$\boxed{\hat{\sigma} \longrightarrow \hat{\sigma}'}$

$$\textsc{PermittedBack}\ \frac{m^{\mathbf{bk}} \in \hat{\mu} \quad \omega' = \omega m^{\mathbf{bk}} \quad \hat{\mu}' = \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega', \hat{\mu}) \quad \hat{\nu}' = \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega', \hat{\nu})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \langle \hat{\mu}', \hat{\nu}', \omega' \rangle}$$

$$\textsc{ProhibitedIn}\ \frac{m^{\mathbf{in}} \in \hat{\nu} \quad \omega' = \omega(\mathsf{dis}\ m^{\mathbf{in}})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \mathsf{bad}\ \omega'}$$

$$\textsc{PermittedIn}\ \frac{m^{\mathbf{in}} \notin \hat{\nu} \quad \omega' = \omega m^{\mathbf{in}} \quad \hat{\mu}' = \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega', \hat{\mu}) \quad \hat{\nu}' = \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega', \hat{\nu})}{\langle \hat{\mu}, \hat{\nu}, \omega \rangle \longrightarrow \langle \hat{\mu}', \hat{\nu}', \omega' \rangle}$$

Figure 9 This transition system defines an abstraction of the framework-internal state consistent with an observable trace ω with respect to a framework abstraction $\boldsymbol{S}$. The abstract state $\hat{\sigma}$ contains a store of permitted back-messages $\hat{\mu}$ and a store of prohibited in-messages $\hat{\nu}$, corresponding to an abstraction of enabled events and disallowed calls, respectively. The meaning of the framework abstraction $\boldsymbol{S}$ is captured by the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$, which determine how an abstract store changes on a new message.

An observable trace ω *violates* the event-driven application-programming protocol if ω ends with a disallowed dis message.

These definitions yield a design for a dynamic-analysis instrumentation that observes app-framework interactions. The trace recording in Verivita obtains observable traces ω like the app-framework dialogue diagrams in Section 2 by following the instrumented semantics $\sigma \xrightarrow{m} \sigma'$. Verivita maintains a stack similar to the continuation k to emit the messages corresponding to the forcings and returns of callbacks and callins, and it emits disallowed $\mathsf{dis}\ \kappa$ messages by observing the exceptions thrown by the framework.

4 Specifying Protocols and Modeling Callback Control Flow

Using λ_{life} as a concrete semantic foundation, we first formalize an abstraction of event-driven programs composed of separate app and framework code with respect to what is observable at the app-framework interface. This abstract transition system captures the possible enabled-event and disallowed-call stores internal to the framework that are consistent with observable traces, essentially defining a family of lifestate framework abstractions. Then, we instantiate this definition for a specific lifestate language that both specifies event-driven application-programming protocols and models callback control flow.

The main point in these definitions is that lifestate modeling of callback control flow can only depend on what is observable at the app-framework interface. Furthermore, the concrete semantic foundation given by λ_{life} leads to a careful definition of soundness and precision and a basis for model validation and predictive-trace verification (Section 5).

Abstracting Framework-Internal State by Observing Messages. In Figure 9, we define the transition system that abstracts the framework-internal state consistent with an observable trace ω. An abstract state $\langle \hat{\mu}, \hat{\nu}, \omega \rangle$ contains a store of *permitted back-messages* $\hat{\mu}$ and a store of *prohibited in-messages* $\hat{\nu}$. What the transition system captures are the possible traces consistent with iteratively applying a framework abstraction $\boldsymbol{S}$ to the current abstract state: it performs a transition with a back-message $m^{\mathbf{bk}}$ only if $m^{\mathbf{bk}}$ is permitted $m^{\mathbf{bk}} \in \hat{\mu}$, and a transition with an in-message $m^{\mathbf{in}}$ only if $m^{\mathbf{in}}$ is not prohibited $m^{\mathbf{in}} \notin \hat{\nu}$. The trace ω in an abstract state saves the history of messages observed so far. In the most general setting for modeling the event-driven framework, the transition system can update the stores $\hat{\mu}$ and $\hat{\nu}$ as a function of the history of the observed messages ω. These updates are formalized with

$$\text{parametrized messages} \quad \boldsymbol{m} ::= \mathsf{cb}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{ciret}\ \lambda[\boldsymbol{p}] \mid \mathsf{ci}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{cbret}\ \lambda[\boldsymbol{p}]$$

lifestate rules $\quad \boldsymbol{s} ::= \boldsymbol{r} \to \boldsymbol{m} \mid \boldsymbol{r} \nrightarrow \boldsymbol{m}$

lifestate abstractions $\quad \boldsymbol{S} ::= \cdot \mid \boldsymbol{s}\boldsymbol{S}$

trace matchers: regular expressions of parametrized messages $\quad \boldsymbol{r}$

symbolic variables $\ell \in \mathbf{SVar}$ $\quad$ parameters $\boldsymbol{p} \in \mathbf{SVar} \cup \mathbf{Val}$ $\quad$ binding maps $\theta ::= \cdot \mid \theta, \ell \mapsto v$

(10.a) A lifestate abstraction is a set of rules that permits ($\to$) or prohibits ($\nrightarrow$) parametrized messages $\boldsymbol{m}$.

$$\begin{aligned}
\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{bk}} \;\middle|\; \mathrm{consistent}_{\boldsymbol{S}}(\omega) \wedge \left(\neg\, \mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}}) \wedge (\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}}) \vee m^{\mathbf{bk}} \in \hat{\mu})\right) \right\} \\
\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{in}} \;\middle|\; \mathrm{consistent}_{\boldsymbol{S}}(\omega) \to \left(\neg\, \mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{in}}) \wedge (\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{in}}) \vee m^{\mathbf{in}} \in \hat{\nu})\right) \right\} \\
\mathrm{permit}_{\boldsymbol{S}}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \to \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\
\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \nrightarrow \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\
\mathrm{consistent}_{\boldsymbol{S}}(\omega) &\stackrel{\text{def}}{=} \forall m \in \Sigma, (\mathrm{permit}_{\boldsymbol{S}}(\omega, m) \leftrightarrow \neg\, \mathrm{prohibit}_{\boldsymbol{S}}(\omega, m))
\end{aligned}$$

$$\hat{\mu}^{\mathrm{init}}_{\boldsymbol{S}} \stackrel{\text{def}}{=} \mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\epsilon, \Sigma^{\mathbf{bk}}) \qquad \hat{\nu}^{\mathrm{init}}_{\boldsymbol{S}} \stackrel{\text{def}}{=} \mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\epsilon, \emptyset)$$

(10.b) Semantics of a lifestate framework abstraction. The store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$ find rules from $\boldsymbol{S}$ that match the given trace ω and update the store according to a consistent binding θ from symbolic variables to values.

Figure 10 Lifestate is a language for simultaneously specifying event-driven protocols and modeling callback control flow in terms of the observable app-framework interface.

the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ that define an abstraction $\boldsymbol{S}$ of the event-driven framework describing both its application-programming protocol and its callback control flow. A framework abstraction $\boldsymbol{S}$ also defines the initial abstract state $\langle \hat{\mu}^{\mathrm{init}}_{\boldsymbol{S}}, \hat{\nu}^{\mathrm{init}}_{\boldsymbol{S}}, \epsilon \rangle$ that contains the initial (abstract) state of the stores of the permitted back-messages and prohibited in-messages.

The semantics $[\![\boldsymbol{S}]\!]$ of a framework abstraction $\boldsymbol{S}$ is the set of observable traces of the transition system defined in Figure 9 instantiated with $\boldsymbol{S}$. We get sequences of states from the transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$, read the observable trace ω from the final state, and form a set of all such observable traces.

▶ **Definition 6** (Soundness of a Framework Abstraction)**.**

(1) *A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a λ_{life} program e if $[\![e]\!] \subseteq [\![\boldsymbol{S}]\!]$;*

(2) *A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a framework implementation $\langle \mathbf{Fun_{fwk}}, \lambda_{\mathbf{init}} \rangle$ if and only if $\boldsymbol{S}$ is sound for every possible program e that uses the framework implementation $\langle \mathbf{Fun_{fwk}}, \lambda_{\mathbf{init}} \rangle$.*

The possible observable traces of a framework abstraction $\boldsymbol{S}$ is slightly richer than observable λ_{life} traces in that callback-return messages ($v = \mathsf{cbret}\ \kappa$) may also be prohibited (in addition callin-invocations $\mathsf{ci}\ \kappa$). Prohibiting callback-return messages corresponds to specifying a protocol where the app yields an invalid return value. If we desire to capture such violations at the concrete level, it is straightforward to extend λ_{life} with "return-invalid" transitions by analogy to InvokeDisallowed transitions.

Finally, if $\boldsymbol{S_1}$ and $\boldsymbol{S_2}$ are sound specifications, we say that $\boldsymbol{S_1}$ is *at least as precise* as $\boldsymbol{S_2}$ if $[\![\boldsymbol{S_1}]\!] \subseteq [\![\boldsymbol{S_2}]\!]$.

A Lifestate Abstraction. We arrive at lifestates by instantiating the framework abstraction $\boldsymbol{S}$ in a direct way as shown in Figure 10. To describe rules independent of particular programs or executions, we *parametrize messages* with symbolic variables $\ell \in \mathbf{SVar}$. The definition of the parametrized messages $\boldsymbol{m}$ is parallel to the non-parameterized version but using

parameters instead of simply concrete values v. We call a message $\boldsymbol{m}$ *ground* when it does not have symbolic variables (from **SVar**), and we distinguish the ground and parameterized messages by using normal m and bold $\boldsymbol{m}$ fonts, respectively. For example, the parametrized callback-invocation message $\mathsf{cb}\ \lambda[\ell]$ specifies that a callback function λ is invoked with an arbitrary value from **Val**. The variable ℓ can be used across several messages in a rule, expressing that multiple messages are invoked with, or return, the same value.

A lifestate abstraction $\boldsymbol{S}$ is a set of rules, and a *rule* consists of trace matcher $\boldsymbol{r}$ that when matched either *permits* ($\rightarrow$ operator) or *prohibits* ($\nrightarrow$ operator) a parametrized message $\boldsymbol{m}$. As just one possible choice for the matcher $\boldsymbol{r}$, we consider $\boldsymbol{r}$ to be a regular expression where the symbols of the alphabet are parametrized messages $\boldsymbol{m}$. In matching a trace ω to a regular expression of parametrized messages, we obtain a *binding* θ that maps symbolic variables from the parametrized messages to the concrete values from the trace. Given a binding θ and a message $\boldsymbol{m}$, we write $\theta(\boldsymbol{m})$ to denote the message $\boldsymbol{m'}$ obtained by replacing each symbolic variable ℓ in $\boldsymbol{m}$ with $\theta(\ell)$ if defined.

The semantics of lifestates is given by a choice of store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$ in Figure 10.b and the abstract transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$ defined previously in Section 4. The store-update functions work intuitively by matching the given trace ω against the matchers $\boldsymbol{r}$ amongst the rules in $\boldsymbol{S}$ and then updating the store according to the matching rules $\{\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}\} \subseteq \boldsymbol{S}$.

To describe the store-update functions in Figure 10.b, we write $\omega, \theta \models \boldsymbol{r}$ to express that a trace ω and a binding θ satisfy a regular expression $\boldsymbol{r}$. The definition of this semantic relation is standard, except for parametrized messages $\boldsymbol{m}$. Here, we explain this interesting case for when the trace ω and the binding θ satisfy the regular expression $\boldsymbol{m}$ (i.e., $\omega, \theta \models \boldsymbol{m}$):

$$\omega, \theta \models \boldsymbol{m} \quad \text{iff} \quad \omega = m \text{ and } \theta(\boldsymbol{m}) = m \text{ for some ground message } m$$

A necessary condition for $\omega, \theta \models \boldsymbol{m}$ is, for example, that θ must assign a value to all the variables in $\boldsymbol{m}$, to get a ground message, and the message must be equal to the trace ω. Note that, if there is no such ground message for $\boldsymbol{m}$ with the binding θ, then $\omega, \theta \not\models \boldsymbol{m}$. The full semantics of matching parametrized regular expressions is given in the extended version [33].

Now, the function $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ captures how the state of the permitted back-messages store $\hat{\mu}$ changes according to the rules $\boldsymbol{S}$. As a somewhat technical point, a back-message can only be permitted if the rules $\boldsymbol{S}$ are *consistent* with respect to the given trace ω (i.e., $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$). The $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ predicate holds iff there are no rules that permits and prohibits m for the same message m and trace ω. Then, if the predicate $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ is true, the back-message $m^{\mathbf{bk}}$ must not be prohibited given the trace ω (i.e., $\neg\, \mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}})$). Finally, if back-message $m^{\mathbf{bk}}$ is not prohibited, either it is permitted by a specification for this trace ω (i.e., $\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}})$) or it was already permitted in the current store $\hat{\mu}$ (i.e., $m^{\mathbf{bk}} \in \hat{\mu}$). The function $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ is similar, but it is defined for the prohibited in-messages store $\hat{\nu}$. An in-message $m^{\mathbf{in}}$ is prohibited first if the rules are not consistent. Then, if the rules are consistent, the in-message $m^{\mathbf{in}}$ must not be permitted by this trace, and either it is prohibited by a rule for this trace or the in-message was already prohibited in the current store $\hat{\nu}$. The auxiliary predicates $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ and $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ formally capture these conditions. The $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ predicate is true iff there is a rule $\boldsymbol{r} \rightarrow \boldsymbol{m}$ in the specification $\boldsymbol{S}$ that permits a message $\boldsymbol{m}$ and a binding θ, such that the trace and the binding satisfy the regular expression ($\omega, \theta \models \boldsymbol{r}$), and the ground message permitted by the rule $\theta(\boldsymbol{m})$ is m. The $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ predicate is analogous but for prohibit rules.

A key point is that the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ are defined only in terms of what is observable at the app-framework interface ω and stores of permitted back-messages $\hat{\mu}$ and prohibited in-messages $\hat{\nu}$. Lifestate abstractions $\boldsymbol{S}$ do not depend on framework or app expressions e, nor framework-internal state.

5 Dynamic Reasoning with Lifestates

Lifestates are precise and detailed abstractions of event-driven frameworks that simultaneously specify the protocol that the app should observe and the callback control-flow assumptions that an app can assume about the framework. The formal development of lifestates in the above offers a clear approach for *model validation* and *predictive-trace verification*. In this section, we define the model validation and verification problem and provide an intuition of their algorithms using the formal development in the previous sections. For completeness of presentation, we provide further details in the extended version [33].

Validating Lifestate Specifications. As documentation in a real framework implementation like Android is incomplete and ambiguous, it is critical that framework abstractions have a mechanism to validate candidate rules – in a manner independent of, say, a downstream static or dynamic analysis.

We say that a specification $\boldsymbol{S}$ is *valid* for an observable trace ω if $\omega \in [\![\boldsymbol{S}]\!]$. If a specification $\boldsymbol{S}$ is not valid for a trace ω from a program e, then $\boldsymbol{S}$ is not a sound abstraction of e.

We can then describe an algorithm that checks if $\boldsymbol{S}$ is a valid specification for a trace ω with a reduction to a model checking problem. Lifestate rules specify the behavior of an unbounded number of objects through the use of symbolic variables $\ell \in \mathbf{SVar}$ that are implicitly universally quantified in the language and hence describe an unbounded number of messages. However, as an observable trace ω has a finite number of ground messages, the set of messages that we can use to instantiate the quantifiers is also finite. Thus, the validation algorithm first "removes" the universal quantifier with the *grounding* process that transforms the lifestate abstraction $\boldsymbol{S}$ to a *ground abstraction* S containing only ground rules.

The language $[\![S]\!]$ of a ground specification S can be represented with a finite transition system since the set of messages in S is finite, and lifestate rules are defined using regular expressions. We then pose the validation problem as a model checking problem that we solve using off-the-shelf symbolic model checking tools [12]. The transition system that we check is the parallel composition (i.e., the intersection of the languages of transition systems) of the transition system that accepts only the trace ω and the transition system $\hat{\sigma} \longrightarrow \hat{\sigma}'$ parametrized by the grounded lifestate abstraction S. The lifestate abstraction S is valid if and only if the composed transition system reaches the last state of the trace ω.

Dynamic Lifestate Verification. Because of the previous sections building up to lifestate validation, the formulation of the dynamic verification is relatively straightforward and offers a means to evaluate the expressiveness of lifestate specification.

We define the set of sub-traces of a trace $\omega = \omega_1 \dots \omega_l$ as $Sub_\omega \stackrel{\text{def}}{=} \{\omega_1, \dots, \omega_l\}$, where $\omega' \in Sub_\omega$ if ω' is a substring of ω that represents the entire execution of a callback directly invoked by an event handler. We consider the set $[\![(\omega_1 + \dots + \omega_l)^*]\!]$ of all the traces obtained by repeating the elements in Sub_ω zero-or-more times and $\Omega_{\omega,e} \stackrel{\text{def}}{=} [\![e]\!] \cap [\![(\omega_1 + \dots + \omega_l)^*]\!]$ its intersection with the traces of the λ_{life} program e.

Given an observable trace ω of the program e (i.e., $\omega \in [\![e]\!]$), the *dynamic verification problem* consists of proving the absence of a trace $\omega' \in \Omega_{\omega,e}$ that violates the application-programming protocol. Since we cannot know the set of traces $[\![e]\!]$ for a λ_{life} program e (i.e., the set of traces for the app composed with the framework implementation), we cannot solve the dynamic verification problem directly. Instead, we solve an abstract version of the problem, where we use a lifestate specification $\boldsymbol{S}$ to abstract the framework implementation $\langle \mathbf{Fun}_{\mathtt{fwk}}, \lambda_{\mathbf{init}} \rangle$. Let $\Omega_{\omega,\boldsymbol{S}} \stackrel{\text{def}}{=} [\![\boldsymbol{S}]\!] \cap [\![(\omega_1 + \dots + \omega_l)^*]\!]$ be the set of repetitions of the trace ω that can be seen in the app-framework interface abstraction defined by $\boldsymbol{S}$.

Given a trace $\omega \in [\![e]\!]$ and a sound specification $\boldsymbol{S}$, the *abstract dynamic verification problem* consists of proving the absence of a trace $\omega' \in \Omega_{\omega,\boldsymbol{S}}$ that violates the application-programming protocol. If we do not find any protocol violation using a specification $\boldsymbol{S}$, then there are no violations in the possible repetitions of the concrete trace ω. Observe that the key verification challenge is getting a precise enough framework abstraction $\boldsymbol{S}$ that sufficiently restricts the possible repetitions of the concrete trace ω.

We reduce the abstract dynamic verification problem to a model checking problem in a similar way to validation: we first generate the *ground model* S from the lifestate model $\boldsymbol{S}$ and the trace ω. Then, we construct the transition system that only generates traces in the set $\Omega_{\omega,\boldsymbol{S}}$ by composing the transition system obtained from the ground specification S and the automaton accepting words in $[\![(\omega_1 + \ldots + \omega_l)^*]\!]$. This transition system satisfies a safety property iff there is no trace $\omega \in \Omega_{\omega,\boldsymbol{S}}$ that violates the protocol.

6 Empirical Evaluation

We implement our approach for Android in the Verivita tool that

(i) instruments an Android app to record observable traces,
(ii) validates a lifestate model for soundness against a corpus of traces, and
(iii) assesses the precision of a lifestate model with dynamic verification.

We use the following research questions to demonstrate that lifestate is an effective language to model event-driven protocols, and validation is a crucial step to avoid unsoundness.

RQ1 *Lifestate Precision.* Is the lifestate language adequate to model the callback control flow of Android? The paper hypothesizes that carefully capturing the app-framework interface is necessary to obtain precise protocol verification results.

RQ2 *Lifestate Generality.* Do lifestate models generalize across apps? We want to see if a lifestate model is still precise when used on a trace from a new, previously unseen app.

RQ3 *Model Validation.* Is validation of callback control-flow models with concrete traces necessary to develop *sound* models? We expect to witness unsoundnesses in existing (and not validated) callback control-flow models and that validation is a crucial tool to get sound models.

Additionally, we considered the feasibility of continuous model validation. The bottom line is that we could validate 96% of the traces within a 6 minute time budget; we discuss these results further in the extended version [33].

***RQ1*: Lifestate Precision.** The bottom line of Table 1 is that lifestate modeling is essential to improve the percentage of verified traces to 83% – compared to 57% for lifecycle++ and 27% for lifecycle modeling.

Methodology. We collect execution traces from Android apps and compare the precision obtained verifying protocol violations with four different callback control-flow models. The first three models are expressed using different subsets of the lifestate language. The *top* model is the least precise (but clearly sound) model where any callback can happen at all times, like in the Automaton 6.a in Section 2. The *lifecycle* model represents the most precise callback control-flow model that we can express only using back-messages, like in Automaton 6.b. The *lifestate* model uses the full lifestate language, and hence also in-messages like in the Automaton 6.d, to change the currently permitted back-messages. It represents the most precise model that we can represent with lifestate. To faithfully compare the precision of the formalisms, we improved the precision of the lifecycle and lifestate models minimizing the

false alarms from verification. And at the same time, we continuously run model validation to avoid unsoundnesses, as we discuss below in *RQ3*. As a result of this process, we modeled the behavior of several commonly-used Android classes, including `Activity`, `Fragment`, `AsyncTask`, `CountdownTimer`, `View`, `PopupMenu`, `ListView`, and `Toolbar` and their subclasses. Excluding similar rules for subclasses, this process resulted in a total of 167 lifestate rules.

We further compare with an instance of a *lifecycle++* model, which refines component lifecycles with callbacks from other Android objects. Our model is a re-implementation of the model used in FlowDroid [5] that considers the lifecycle for the UI components (i.e., `Activity` and `Fragment`) and bounds the execution of a pre-defined list of callback methods in the active state of the `Activity` lifecycle, similarly to the example we show in Figure 2. We made a best effort attempt to faithfully replicate the FlowDroid model (and discuss how we did so in the extended version [33]).

To find error-prone protocols, we selected *sensitive callins*, shown in the first column of Table 1, that frequently occur as issues on GitHub and StackOverflow [16, 41, 3, 36, 47, 46]. We then specify the lifestate rules to allow and disallow the sensitive callins.

To create a realistic trace corpus for *RQ1*, we selected five apps by consulting Android user groups to find those that extensively use Android UI objects, are not overly simple (e.g., student-developed or sample-projects apps), and use at least one of the sensitive callins. To obtain realistic interaction traces, we recorded manual interactions from a non-author user who had no prior knowledge of the internals of the app. The user used each app 10 times for 5 minutes (on an x86 Android emulator running Android 6.0) – obtaining a set of 50 interaction traces. With this trace-gathering process, we exercise a wide range of behaviors of Android UI objects that drives the callback control-flow modeling.

To evaluate the necessity and sufficiency of lifestate, we compare the verified rates (the total number of verified traces over the total number of verifiable traces) obtained using each callback control-flow model. We further measure the verification run time to evaluate the trade-off between the expressiveness of the models and the feasibility of verification.

Table 1 Precision of callback control-flow models. The *sensitive callin* column lists protocol properties by the callin that crashes the app when invoked in a bad state. We collect a total of 50 traces from 5 applications with no crashes. The *sensitive* column lists the number of traces where the application invokes a sensitive callin. To provide a baseline for the precision of a model, we count the number of traces without a manually-confirmed real bug in the *verifiable* column. There are four columns labeled *verified* showing the number and percentage of verifiable traces proved correct using different callback control-flow models. The *lifestate* columns capture our contribution. The *lifecycle++* columns capture the current practice for modeling the Android framework. The *bad* column lists the number of missed buggy traces and is discussed further in *RQ2*.

properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
sensitive callin	sensitive (n)	verifiable (n)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	bad (n)
`AlertDialog`											
dismiss	16	6	0	0	0	0	6	100	6	100	0
show	43	34	17	50	17	50	28	82	24	71	0
`AsyncTask`											
execute	4	4	0	0	4	100	4	100	0	0	0
`Fragment`											
getResources	10	10	0	0	0	0	10	100	4	40	0
getString	10	10	0	0	0	0	2	20	0	0	0
setArguments	19	19	1	5	1	5	19	100	13	68	0
total	102	83	18	22	22	27	69	83	47	57	0

Table 2 The table shows the precision results for the 1577 non-crashing traces that contained a sensitive callins from a total of 2202 traces that we collected from 121 distinct open source app repositories. We note that lifestate takes slightly longer than lifecycle; for this reason, lifestate performs slightly worse than lifecycle for execute. The bad column is 0 for models other than lifecycle++ because of continuous validation. Note that out of 64 total buggy traces, lifecycle++ missed 27 bugs (i.e., had a 42% false-negative rate).

properties	non-crashing traces		callback control-flow models								
			top		lifecycle		lifestate		lifecycle++		
sensitive callin	sensitive (n)	verifiable (n)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	verified (n)	(%)	bad (n)
AlertDialog											
dismiss	94	59	54	92	54	92	54	92	58	98	3
show	145	144	125	87	124	86	125	87	127	88	0
AsyncTask											
execute	415	415	0	0	415	100	412	99	262	63	0
Fragment											
getResources	156	155	89	57	89	57	128	83	116	75	0
getString	220	193	124	64	124	64	134	69	131	68	24
setArguments	456	456	59	13	108	24	437	96	435	95	0
startActivity	91	91	0	0	0	0	12	13	19	21	0
total	1577	1513	451	30	914	60	1302	86	1148	76	27

Discussion. In Table 1, we show the number of verified traces and the verified rates broken down by sensitive callins and different callback control-flow models – aggregated over all apps. As stated earlier, the precision improvement with lifestate is significant, essential to get to 83% verified. We also notice that the lifecycle model is only slightly more precise than the trivial top model (27% versus 22% verified rate). Even with unsoundnesses discussed later, lifecycle++ is still worse than the lifestate model, with 57% of traces proven.

Lifestate is also expressive enough to prove most verifiable traces – making manual triage of the remaining alarms feasible. We manually examined the 14 remaining alarms with the lifestate model, and we identified two sources of imprecision:

(1) an insufficient modeling of the attachment of UI components (e.g., is a *View* in the *View* tree attached to a particular *Activity*?), resulting in 13 alarms;

(2) a single detail on how Android options are set in the app's XML, resulting in 1 alarm.

The former is not fundamental to lifestates but a modeling tradeoff where deeper attachment modeling offers diminishing returns on the verified rate while increasing the complexity of the model and verification times. The latter is an orthogonal detail for handling Android's XML processing (that allows the framework to invoke callbacks via reflection).

***RQ2*: Lifestate Generality.** The bottom line of Table 2 is that the lifestate model developed for *RQ1* as-is generalizes to provide precise results (with a verified rate of 86%) when used to verify traces from 121 previously unseen apps. This result provides evidence that lifestates capture general behaviors of the Android framework. While the lifecycle++ model verifies 76% of traces, it also misses 27 out of 64 buggy traces (i.e., has a 42% false-negative rate).

Methodology. To get a larger corpus, we cloned 121 distinct open source apps repositories from GitHub that use at least one sensitive callin (the count combines forks and clones). Then, we generated execution traces using the Android UI Exerciser Monkey [2] that interacts with the app issuing random UI events (e.g., clicks, touches). We attempted to automatically generate three traces for each app file obtained by building each app.

Discussion. From Table 2, we see that the lifestate verified rate of 86% in this larger experiment is comparable with the verified rate obtained in *RQ1*. Moreover, lifestate still improves the verified rate with respect to lifecycle, which goes from 60% to 86%, showing that the expressivity of lifestate is necessary.

Critically, the lifecycle++ model does not alarm on 42% of the traces representing real defects. That is, we saw unsoundnesses of the lifecycle++ model manifest in the protocol verification client.

The verified rate for the lifecycle model is higher in this larger corpus (60%) compared to the rate in *RQ1* (27%), and the precision improvement from the top abstraction is more substantial (60% to 30% versus 27% to 22%). This difference is perhaps to be expected when using automatically-generated traces that may have reduced coverage of app code and bias towards shallower, "less interesting" callbacks associated with application initialization instead of user interaction. In these traces, it is possible that UI elements were not exercised as frequently, which would result in more traces provable solely with the lifecycle specification. Since coverage is a known issue for the Android UI Exerciser Monkey [4]), it was critical to have some evidence on deep, manually-exercised traces as in *RQ1*.

Bug Triage. We further manually triage every remaining alarm from both *RQ1* and *RQ2*. Finding protocol usage bugs was not necessarily expected: for *RQ1*, we selected seemingly well-developed, well-tested apps to challenge verification, and for *RQ2*, we did not expect automatically generated traces to get very deep into the app (and thus deep in any protocol).

Yet from the *RQ1* triage, we found 2 buggy apps out of 5 total. These apps were Puzzles [10] and SwiftNotes [13]. Puzzles had two bugs, one related to `AlertDialog.show` and one for `AlertDialog.dismiss`. Swiftnotes has a defect related to `AlertDialog.show`.

In the *RQ2* corpus, we found 7 distinct repositories with a buggy app (out of 121 distinct repositories) from 64 buggy traces (out of 2202). We were able to reproduce bugs in 4 of the repositories and strongly suspect the other 3 to also be buggy. Three of the buggy apps invoke a method on `Fragment` that requires the `Fragment` to be attached. This buggy invocation happens within unsafe callbacks. Audiobug [51] invokes `getResources`. NextGisLogger [35] and Kistenstapeln [14] invoke `getString`. We are able to reproduce the Kistenstapeln bug.

Interestingly, one of the apps that contain a bug is Yamba [20], a tutorial app from a book on learning Android [21]. We note that the Yamba code appears as a part of three repositories where the code was copied (we only count these as one bug). The tutorial app calls `AlertDialog.dismiss` when an `AsyncTask` is finishing and hence potentially after the `Activity` object used in the `AlertDialog` is not visible anymore. We found similar defects in several actively maintained open source apps where callbacks in an `AsyncTask` object were used either to invoke `AlertDialog.show` or `AlertDialog.dismiss`. These apps included OSM Tracker [22] and Noveldroid [44]. Additionally, we found this bug in a binary library connected with the PingPlusPlus android app [38]. By examining the output of our verifier, we were able to create a test to concretely witness defects in 4 of these apps.

***RQ3*: Model Validation.** The plot in Figure 11 highlights the necessity of applying model validation: lifecycle++ based on a widely used callback control-flow model does not validate (i.e., an unsoundness is witnessed) on 58% of 2183 traces (and the validation ran out of memory for 19 out of the total 2202 traces).

Methodology. We first evaluate the need for model validation by applying our approach to lifecycle++ and quantifying its discrepancies with the real Android executions.

Our first experiment validates the lifecycle++ model on all the traces we collected (bounding each validation check to 1 hour and 4 GB of memory). We quantify the necessity of model validation collecting for each trace if the model was valid and the length of the maximum prefix of the trace that the model validates. Since there are already some known limitations in the lifecycle++ model (e.g., components interleaving), we triage the results to understand if the real cause of failure is a new mistake discovered with the validation process.

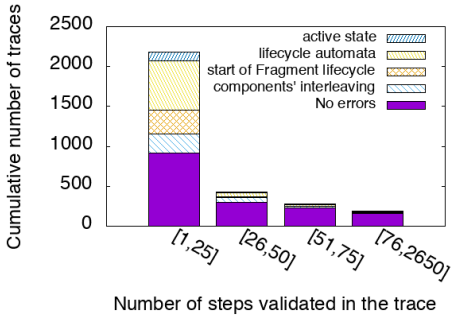


Figure 11 Results of the validation of the lifecycle++ model on all the traces. We plot the cumulative traces grouped by (intervals of) the number of steps validated. The number of traces are further divided into categories, either indicating that validation succeeded, "no errors," or the cause of failure of the validation process.

Our second experiment qualitatively evaluates the necessity of model validation to develop sound lifestate specifications. To create a sound model, we started from the empty model (without rules) and continuously applied validation to find and correct mistakes. In each iteration: we model the callback control flow for a specific Android object; we validate the current model on the entire corpus of traces (limiting each trace to one hour and 4 GB of memory); and when the model is not valid for a trace, we inspect the validation result and repair the specification. We stop when the model is valid for all the traces. We then collected the mistakes we found with automatic validation while developing the lifestate model. We describe such mistakes and discuss how we used validation to discover and fix them.

Discussion: lifecycle++ Validation. From the first bar of the plot in Figure 11, we see that the lifecycle++ model validates only 42% of the total traces, while validation fails in the remaining cases (58%). The bar shows the number of traces that we validated for at least one step, grouping them by validation status and cause of validation failure. From our manual triage, we identified 4 different broad causes for unsoundness:

i) *outside the active lifecycle*: the model prohibits the execution of a callback outside the modeled active state of the `Activity`;

ii) *wrong lifecycle automata*: the model wrongly prohibits the execution of an `Activity` or `Fragment` lifecycle callback;

iii) *wrong start of the* `Fragment` *lifecycle*: the model prohibits the start of the execution of the `Fragment` lifecycle;

iv) *no components interleaving*: the model prohibits the interleaved execution of callbacks from different `Activity` or `Fragment` objects.

The plot shows that the lifecycle++ model is not valid on 25% of the traces because it does not model the interleaving of components (e.g., the execution of callbacks from different `Activity` and `Fragment` objects cannot interleave) and the start of the `Fragment` lifecycle at an arbitrary point in the enclosing `Activity` object. With FlowDroid, such limitations are known and have been justified as practical choices to have feasible flow analyses [5]. But the remaining traces, 33% of the total, cannot be validated due other reasons including modeling mistakes. In particular, the FlowDroid model imprecisely captures the lifecycle automata (for both `Activity` and `Fragment`) and erroneously confines the execution of some callbacks in the active state of the lifecycle.

The other bars in the plot of Figure 11 show the number of traces we validated for more than 25, 50, and 75 steps, respectively. In the plot, we report the total number of steps in the execution traces that correspond to a callback or a callin that we either used in the lifestate

or the lifecycle++ model, while we remove all the other messages. From such bars, we see that we usually detect the unsoundness of the lifecycle++ model "early" in the trace (i.e., in the first 25 steps). This result is not surprising since most of the modeling mistakes we found are related to the interaction with the lifecycle automata and can be witnessed in the first iteration of the lifecycle. We further discovered that the lifecycle++ model mostly validates shorter execution traces, showing that having sound models for real execution traces is more challenging, which we discuss further in the extended version [33].

Discussion: Catching Mistakes During Modeling. We were able to obtain a valid lifestate specification *for over 99.9%* of the traces in our corpus. That is, we were able to understand and model the objects we selected in all but two traces.

Surprisingly, we identified and fixed several mistakes in our modeling of the `Activity` and `Fragment` lifecycle that are due to undocumented Android behaviors. An example of such behavior is the effect of `Activity.finish` and `Activity.startActivity` on the callback control flow for the `onClick` callback. It is unsound to restrict the enabling of `onClick` callbacks to the active state of the `Activity` lifecycle (i.e., between the execution of the `onResume` and `onPause` callbacks). This is the behavior represented with blue edges in Figure 2, what is typically understood from the Android documentation, and captured in the existing callback control-flow models used for static analysis.

We implemented a model where `onClick` could be invoked only when its `Activity` was running and found this assumption to be invalid on several traces. We inferred that the mistake was due to the wrong "bounding" of the `onClick` callbacks in the `Activity` lifecycle since in all the traces:

i) the first callback that was erroneously disabled in the model was the `onClick` callback; and

ii) the `onClick` callback was disabled in the model just after the execution of an `onPause` callback that appeared before in the trace, without an `onResume` callback in between (and hence, outside the active state of the `Activity`.)

It turns out that both `finish` and `startActivity` cause the `Activity` to pause without preventing the pending `onClick` invocations from happening, as represented in the red edges connected to `onClick` in Figure 2. We validated such behaviors by writing and executing a test application and finding its description in several Stack Overflow posts [49, 48]. The fix for this issue is to detect the finishing state of the `Activity` and to not disable the `onClick` callback in this case.

7 Related Work

Several works [5, 8, 42, 45, 24, 40, 37, 43, 24] propose different callback control-flow models. Many previous works, like FlowDroid [5] and Hopper [8], directly implement the lifecycle of Android components. While the main intention of these tools is to implement the lifecycle automata, in practice, they also encode some of the effects of callins invoked in the app code in an ad-hoc manner. For example, FlowDroid determines if and where a callback (e.g., `onClick`) is registered using a pre-defined list of callin methods and an analysis of the app call graph. Hopper implements the lifecycle callback control flow directly in a static analysis algorithm that efficiently explores the interleaving of Android components. In contrast, our work starts from the observation that reasoning about protocol violations requires capturing, in a first-class manner, the effects that invoking a callin has on the future execution of callbacks (and vice-versa).

Callback control-flow graphs [53] are graphs of callbacks generated from an application and a manually written model of the framework. Perez and Le [37] generate callback control-flow graphs with constraints relating program variables to callback invocations analyzing the Android framework. Such models can indirectly capture callin effects via the predicates on the program state. With lifestate, we carefully focus on what is observable at the app-framework interface so that lifestate specifications are agnostic to the internal implementation details of the framework. DroidStar [40] automatically learns a callback typestate automaton for an Android object from a developer-specified set of transition labels using both callbacks and callins symbols. Such automata specifically represent the protocol for a single object and, differently from lifestate, their labels are not parameterized messages. A callback typestate is thus a coarser abstraction than lifestate since it cannot express the relationships between different message occurrences that are required to describe multi-object protocols.

There exist other classes of framework models that represent different and complementary aspects of the framework than the callback control flow captured by lifestate. For example, Fuchs et al. [19] and Bastani et al. [6] represent the "heap properties" implicitly imposed by the framework. EdgeMiner [11] and Scandal [29] model the registration of callbacks. Droidel [9] also captures callback registration by modeling the reflection calls inside the Android framework code. Similarly, Pasket [25] automatically learns implementations of framework classes that behave according to particular design patterns.

While framework models have been extensively used to support static and dynamic analysis, not much attention has been paid to validating that the models soundly capture the semantics of the real framework. Wang et al. [52] recognized the problem of model unsoundness – measuring unsoundnesses in three different Android framework models. Unsoundnesses were found even using a much weaker notion of model validation than we do in this work. A significant advantage of lifestates is that we can validate their correctness with respect to any execution trace, obtained from arbitrary apps, because they speak generically about the app-framework interface.

There exist several programming languages for asynchronous event-driven systems, such as Tasks [18] and P [15]. In principle, such languages are general enough to develop event-driven systems such as Android. The purpose of our formalization λ_{life} is instead to provide a formalization that captures the app-framework interface.

The protocol verification problem for event-driven applications is related to typestate verification [34, 26, 17], but it is more complex since it requires reasoning about the asynchronous interaction of both callbacks and callins. Dynamic protocol verification is similar in spirit to dynamic event-race detection [32, 23, 7, 31], which predicts if there is an event data-race from execution traces. However, a lifestate violation differs from, and is not directly comparable to, an event data-race. A lifestate violation could manifest as a data race on a framework-internal field, but more commonly it results from encountering an undesirable run-time state within the framework.

8 Conclusion

We considered the problem of specifying event-driven application-programming protocols. The key insight behind our approach is a careful distillation of what is observable at the interface between the framework and the app. This distillation leads to the abstract notions of permitted messages from the framework to the app (e.g., enabled callbacks) and prohibited messages into the framework from the app (e.g., disallowed callins). Lifestate specification then offers the ability to describe the event-driven application-programming protocol in

terms of this interface – capturing both what the app can expect of the framework and what the app must respect when calling into the framework. We evaluated our approach by implementing a dynamic lifestate verifier called Verivita and showed that the richness of lifestates are indeed necessary to verify real-world Android apps as conforming to actual Android protocols.

References

1 Android Developers. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.html, 2018.

2 Android Developers. UI/Application exerciser monkey. https://developer.android.com/studio/test/monkey.html, 2018.

3 Android Topeka. Crash if rotate device right after press floating action button #4 Topeka for Android. https://github.com/googlesamples/android-topeka/issues/4, 2015.

4 Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Hee Beng Kuan Tan, and Charlie Soh. Achieving High Code Coverage in Android UI Testing via Automated Widget Exercising. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2016. doi:10.1109/APSEC.2016.036.

5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594299.

6 Osbert Bastani, Saswat Anand, and Alex Aiken. Specification Inference Using Context-Free Language Reachability. In *Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676977.

7 Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for Android applications. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814303.

8 Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2015. doi:10.1145/2814270.2814293.

9 Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to Android framework modeling. In *State of the Art in Program Analysis (SOAP)*, 2015. doi:10.1145/2771284.2771288.

10 Chris Boyle. Simon Tatham's Puzzles. https://github.com/chrisboyle/sgtpuzzles/blob/658f00f19172bdbceb5329bc77376b40fe550fcb/app/src/main/java/name/boyle/chris/sgtpuzzles/GamePlay.java#L183, 2014.

11 Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In *Network and Distributed System Security (NDSS)*, 2015. URL: https://www.ndss-symposium.org/ndss2015/edgeminer-automatically-detecting-implicit-control-flow-transitions-through-android-framework.

12 Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *Computer-Aided Verification (CAV)*, 2014. doi:10.1007/978-3-319-08867-9_22.

13 Adrian Chifor. Swiftnotes. https://f-droid.org/en/packages/com.moonpi.swiftnotes/, 2015.

14 D120. Kistenstapeln. https://github.com/d120/Kistenstapeln-Android, 2015.

15 Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Programming Language Design and Implementation (PLDI)*, 2013. doi:10.1145/2491956.2462184.

16 Martin Fietz. FeedRemover: already running - issue #1304 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/issues/1304, 2015.

17 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. doi:10.1145/1348250.1348255.

18 Jeffrey Fischer, Rupak Majumdar, and Todd D. Millstein. Tasks: language support for event-driven programming. In *Partial Evaluation and Program Manipulation (PEPM)*, 2007. doi:10.1145/1244381.1244403.

19 Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated security certification of Android applications. Technical Report CS-TR-4991, University of Maryland, College Park, 2009.

20 Marko Gargenta. Yamba. https://github.com/learning-android/Yamba/blob/429e37365f35ac4e5419884ef88b6fa378c023f8/src/com/marakana/android/yamba/StatusFragment.java, 2014.

21 Marko Gargenta and Masumi Nakamura. *Learning Android*. O'Reilly Media, 2014.

22 Nicolas Guillaumin. OSMTracker for Android. https://github.com/nguillaumin/osmtracker-android/blob/d80dea16e456defe5ab62ed8b5bc35ede363415e/app/src/main/java/me/guillaumin/android/osmtracker/gpx/ExportTrackTask.java, 2015.

23 Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594330.

24 Jinseong Jeon, Kristopher K. Micinski, and Jeffrey S. Foster. SymDroid: Symbolic execution for Dalvik bytecode. Technical report, Department of Computer Science, University of Maryland, College Park, 2012.

25 Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *International Conference on Software Engineering (ICSE)*, 2016. doi:10.1145/2884781.2884856.

26 Pallavi Joshi and Koushik Sen. Predictive Typestate Checking of Multithreaded Java Programs. In *Automated Software Engineering (ASE)*, 2008. doi:10.1109/ASE.2008.39.

27 Vladislav Kaplun. Update RequestAsyncTask.java by kaplad - Pull Request #315 - facebook/facebook-android-sdk. https://github.com/facebook/facebook-android-sdk/pull/315, 2014.

28 Maria Kechagia and Diomidis Spinellis. Undocumented and unchecked: exceptions that spell trouble. In *Mining Software Repositories, (MSR)*, 2014. doi:10.1145/2597073.2597089.

29 Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. SCANDAL: Static analyzer for detecting privacy leaks in Android applications. *IEEE Mobile Security Technologies (MoST).*, 2017.

30 Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4), 2006. doi:10.1007/s10990-006-0480-6.

31 Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial Order Reduction for Event-Driven Multi-threaded Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016. doi:10.1007/978-3-662-49674-9_44.

32 Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In *Programming Language Design and Implementation (PLDI)*, 2014. doi:10.1145/2594291.2594311.

33 Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-Driven Protocols and Callback Control Flow (Extended Version). *CoRR*, abs/, 2019. arXiv:1906.04924.

34 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2008. doi:10.1145/1449764.1449792.

35 NextGis. NextGisLogger. https://github.com/nextgis/nextgislogger, 2017.

36 OneBusAway. IllegalStateException: Fragment BaseMapFragment not attached to Activity #570 OneBusAway. https://github.com/OneBusAway/onebusaway-android/issues/570, 2016.

37 Danilo Dominguez Perez and Wei Le. Predicate callback summaries. In *International Conference on Software Engineering (ICSE)*, 2017. doi:10.1109/ICSE-C.2017.95.

38 PingPlusPlus. Ping Plus Plus. https://github.com/PingPlusPlus/pingpp-android, 2017.

39 Steve Pomeroy. The Complete Android Activity/Fragment Lifecycle v0.9.0. https://github.com/xxv/android-lifecycle, 2014.

40 Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Cerný. DroidStar: callback typestates for Android classes. In *International Conference on Software Engineering (ICSE)*, 2018. doi:10.1145/3180155.3180232.

41 Red Reader. Crash during commenting #467 RedReader. https://github.com/QuantumBadger/RedReader/issues/467, 2017.

42 A. Rountev, D. Yan, S. Yang, H. Wu, Y. Wang, and H. Zhang. GATOR: Program analysis toolkit for Android. http://web.cse.ohio-state.edu/presto/software/, 2017.

43 Atanas Rountev and Dacong Yan. Static Reference Analysis for GUI Objects in Android Software. In *Code Generation and Optimization (CGO)*, 2014. doi:10.1145/2544137.2544159.

44 sh1ro. NovelDroid. https://github.com/sh1r0/NovelDroid/blob/f3245055d7a8bcc69a9bca278fbe890081dac58a/app/src/main/java/com/sh1r0/noveldroid/SettingsFragment.java, 2016.

45 Eric Smith and Alessandro Coglio. Android platform modeling and Android app verification in the ACL2 theorem prover. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2015. doi:10.1007/978-3-319-29613-5_11.

46 StackOverflow Post. Got exception: fragment already active. https://stackoverflow.com/questions/10364478/got-exception-fragment-already-active, 2012.

47 StackOverflow Post. Alertdialog creating exception in android. https://stackoverflow.com/questions/15104677/alertdialog-creating-exception-in-android, 2013.

48 StackOverflow Post. OnClickListener fired after onPause? https://stackoverflow.com/questions/31432014/onclicklistener-fired-after-onpause, 2015.

49 StackOverflow Post. Android: click event after Activity.onPause(). https://stackoverflow.com/questions/38368391/android-click-event-after-activity-onpause, 2016.

50 Robert E. Strom and Shaula Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.*, 12(1), 1986.

51 Matthias Urhahn. AudioBug. https://github.com/d4rken/audiobug, 2017.

52 Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for Android GUIs. In *State of the Art in Program Analysis (SOAP)*, 2016. doi:10.1145/2931021.2931026.

53 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static Control-Flow Analysis of User-Driven Callbacks in Android Applications. In *International Conference on Software Engineering (ICSE)*, 2015. doi:10.1109/ICSE.2015.31.