Historia: Refuting Callback Reachability with Message-History Logics

SHAWN MEIER, University of Colorado Boulder, USA
SERGIO MOVER, LIX, École Polytechnique, CNRS, Institut Polytechnique de Paris, France
GOWTHAM KAKI, University of Colorado Boulder, USA
BOR-YUH EVAN CHANG*, University of Colorado Boulder & Amazon, USA

This paper considers the callback reachability problem — determining if a callback can be called by an event-driven framework in an unexpected state. Event-driven programming frameworks are pervasive for creating user-interactive applications (apps) on just about every modern platform. Control flow between callbacks is determined by the framework and largely opaque to the programmer. This opacity of the callback control flow not only causes difficulty for the programmer but is also difficult for those developing static analysis. Previous static analysis techniques address this opacity either by assuming an arbitrary framework implementation or attempting to eagerly specify all possible callback control flow, but this is either too coarse to prove properties requiring callback-ordering constraints or too burdensome and tricky to get right. Instead, we present a middle way where the callback control flow can be gradually refined in a targeted manner to prove assertions of interest. The key insight to get this middle way is by reasoning about the history of method invocations at the boundary between app and framework code — enabling a decoupling of the specification of callback control flow from the analysis of app code. We call the sequence of such boundary-method invocations *message histories* and develop message-history logics to do this reasoning. In particular, we define the notion of an application-only transition system with boundary transitions, a message-history program logic for programs with such transitions, and a temporal specification logic for capturing callback control flow in a targeted and compositional manner. Then to utilize the logics in a goal-directed verifier, we define a way to combine after-the-fact an assertion about message histories with a specification of callback control flow. We implemented a prototype message history-based verifier called Historia and provide evidence that our approach is uniquely capable of distinguishing between buggy and fixed versions on challenging examples drawn from real-world issues and that our targeted specification approach enables proving the absence of multi-callback bug patterns in real-world open-source Android apps.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; **Software verification**; **Automated static analysis**; *Software safety*; *Publish-subscribe / event-based architectures*; • **Theory of computation** → *Modal and temporal logics*; **Logic and verification**; Separation logic; *Hoare logic*.

Additional Key Words and Phrases: event-driven frameworks, refuting callback reachability, callback control flow, framework modeling, goal-directed verification, backwards abstract interpretation, ordered linear logics, temporal logics, message history logics

*Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at the University of Colorado Boulder and is not associated with Amazon.

Authors' addresses: Shawn Meier, shawn.meier@colorado.edu, University of Colorado Boulder, Boulder, USA; Sergio Mover, sergio.mover@lix.polytechnique.fr, LIX, École Polytechnique, CNRS, Institut Polytechnique de Paris, Paris, France; Gowtham Kaki, gowtham.kaki@colorado.edu, University of Colorado Boulder, Boulder, USA; Bor-Yuh Evan Chang, evan.chang@colorado.edu, University of Colorado Boulder & Amazon, Boulder, USA.

2475-1421/2023/10-ART289
https://doi.org/10.1145/3622865

ACM Reference Format:
Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. 2023. Historia: Refuting Callback Reachability with Message-History Logics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 289 (October 2023), 30 pages. https://doi.org/10.1145/3622865

1 INTRODUCTION

The standard approach for creating user-interactive applications (apps) is with event-driven frameworks. In this programming model, a developer defines *callback* methods that the framework invokes at run time in response to asynchronous events (e.g., starting the application, clicking a button, or a background task finishing). Since the callbacks modify the state of the application, an unexpected order of callback invocations may lead to a bad application state and a subsequent crash. To fix such a crash, it is common for the developer to update the application to change or accommodate complex callback interactions. To help the developer verify fixes to the crashing app, we present in this paper a technique to reason about callback order and to develop a tool that can automatically prove such fixes correct.

As a specific example of an event-driven framework, we consider Android, a widely-used and complex mobile operating system. Figure 1 shows a stack trace for a reported crash in AntennaPod, a popular open-source podcast player. But this stack trace is not helpful to the developer because the cause and effect span multiple callback invocations. This stack trace shows that there was a `null` dereference in the `call` callback — but not how the reference became `null`. In particular, the reference must have been set to `null` in some previous callback invocation before this `call` callback invocation that is not visible in this stack trace. To make such reasoning even more difficult, the app itself may affect the order of callbacks through the invocation of methods defined by the framework API; we refer to calls from the app to framework API methods as *callins* by analogy to callbacks. In summary, the developer of the app needs to reason about the order in which the framework could invoke callbacks and how the app invokes callins to understand and fix this crash.

Understanding the order of callbacks in this event-driven programming model is not only challenging for the developer but is also a central challenge for a program verifier attempting to prove an app safe from crashes. The program verifier has access to the app code, but the framework code is unavailable for all intents and purposes. While this is true for most event-driven frameworks, it is particularly difficult in Android, which consists of thousands of API classes [Android Developers 2022b], evolves quickly, includes lots of native code, and varies by device with manufacturer customizations. One approach to the unavailable-framework problem is to analyze the app code assuming an arbitrary framework implementation. This design corresponds to analyzing each top-level callback (i.e., entry point into the app code) as a separate program with an application-only call graph [Ali and Lhoták 2012]. The advantage of this approach is that it is simple and general (i.e., it can over-approximate any framework implementation by assuming all callbacks can be invoked at any time in the event loop) and thus is the approach generally taken by industrial-scale analyzers for Android apps (e.g., [Distefano et al. 2019; Fuchs et al. 2009; Liang et al. 2013; Mariana Trench 2022]).

However, this most-over-approximate framework model is also wildly unrealistic. Without callback-ordering constraints, a verifier cannot possibly prove correct the accepted fix for Figure 1. Thus, many static analyzers for Android attempt to eagerly encode the callback control flow of core classes of the framework (e.g., the Activity Lifecycle [Android Developers 2022a] modeled by [Arzt et al. 2014; Blackshear et al. 2015b; Yang et al. 2015]). This approach also has some significant limitations. For one, it is not feasible to eagerly specify the callback control flow of thousands of framework classes individually — let alone callback control flow involving relationships between

```
java.lang.IllegalStateException: ...
Caused by: java.lang.NullPointerException: ...
  at de.danoeh.antennapod.fragment.ExternalPlayerFragment$$Lambda$4.call (Unknown Source:1193)
  ...
```

Fig. 1. A reported stack trace from a confirmed bug [Fietz 2018b] that crashed the AntennaPod Android app. We have elided multiple lines identifying Android framework methods (using ... for 14 elided lines in total).

multiple classes. On top of that, even specifying the behavior for a few framework components results in both soundness and precision issues [Cao et al. 2015; Meier et al. 2019; Wang et al. 2016].

The main observation of this paper is that although the most-over-approximate framework model is unrealistic, the application-only approach for analyzing event-driven apps is not completely hopeless either. In particular, if we decouple the framework modeling from the analysis of the app code, then the approach offers the appealing capability to gradually refine the possible callback control flow as needed and in a targeted manner. To get a sense, imagine a call graph with a "framework" node representing the event loop and outgoing edges to each callback entry (as well as edges from callback nodes to the callin nodes they invoke). We can now consider traces through this graph instantiating callback or callin nodes with object instances; we refer to such a callback or a callin instantiation generically as a *message*. Thus, we are interested in reasoning about *message histories* — sequences of messages obtained by call-return traces through this graph.

For any given framework implementation, not all message histories are realizable at run time. Thus, our key insight is to encode possible framework implementations by abstracting possible message histories. Crucially, this encoding and reasoning about message histories enables decoupling the specification of callback control flow from the abstract interpretation to compute an inductive program invariant. Specifically, in this paper, we make the following contributions:

- We define the notion of an application-only transition system that records messages and a message-history program logic (MHPL) to describe and reason about *boundary transitions* between the app and the framework independently of any specification of the framework (Section 3). In MHPL, we consider a backwards-from-error formulation that enables goal-directed reasoning from a state assertion in the app, and we observe that deriving infeasible initial message histories refutes callback reachability. To capture consumption and ordering in MHPL, message-history assertions are derived from a fragment of ordered linear logic [Polakow and Pfenning 1999a,b] — making use of an ordered linear implication.
- We formalize a callback control-flow temporal logic (CBCFTL) to specify *realizable message histories* — that is fully decoupled from any particular program logic (Section 4). This specification logic enables us to restrict possible callback control flow in a manner that is targeted and compositional. To capture possible traces, CBCFTL is a specialization of past-time linear temporal logic [Lichtenstein et al. 1985].
- We design an automated reasoning approach for the combination of MHPL assertions and CBCFTL specifications (Section 5). To utilize MHPL and CBCFTL together in a static verifier, we define an algorithm instantiating CBCFTL specifications with MHPL assertions into a single formula describing realizable message histories. We then use this encoding to answer queries about message-history entailment or whether a message history excludes the initial state with an off-the-shelf SMT solver.
- We empirically evaluate Historia, a prototype goal-directed verifier with MHPL and show that it can refute callback reachability assertions with succinct specifications of callback control flow in CBCFTL (Section 6). In particular, we applied it to distinguish between the

buggy and fixed versions from 5 real-world multi-callback issues from in-the-wild crashes of Android apps. Furthermore, we codified these 5 issues into bug patterns and evaluated the ability to use Historia to prove the absence of these bug patterns on 47 open-source apps containing over 2 million lines of code: 43% of the potentially buggy locations could be proven safe using Historia with no additional modeling of callback control flow and then on a sample of the remaining locations, half of them could be proven safe (or witnessed as buggy) with a small amount of additional modeling.

2 OVERVIEW

In this section, we illustrate how a developer could go deeper to diagnose and fix the bug causing the crash shown in Figure 1 (Section 2.1) and then demonstrate how our approach is able to prove the fixed version correct (Section 2.2).

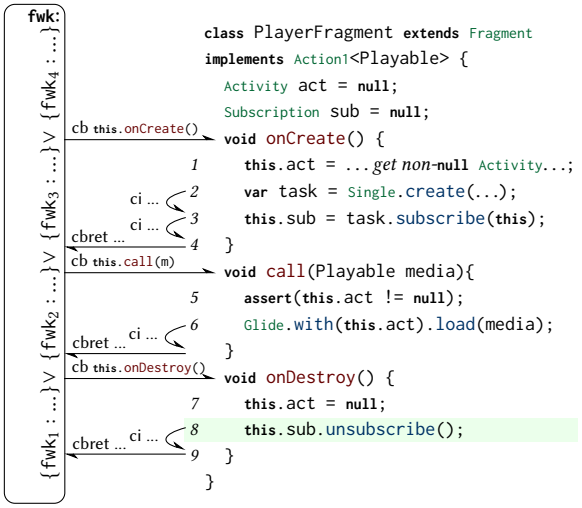
2.1 Using Message Histories to Distinguish Bugs from Fixes

We show a simplified version of an actual pull request submitted to fix the issue in Figure 2a. What distinguishes the buggy and fixed versions — without and with line 8, respectively — are the possible *message histories* (i.e., the possible sequences of callbacks and callins). In Figure 2b, we show a message history witnessing a crash in the buggy version, while in Figure 2c, we show the analogous message history in the fixed version. The key to distinguishing the buggy and fixed versions is determining if such message histories are *realizable* at run time.

Figure 2a shows part of the `PlayerFragment` class of the AntennaPod app that displays a user interface (UI) and loads some media for playing podcasts. Importantly, the app loads the media in a background task, a thread running asynchronously to the UI thread, to not block the user interface. The framework invokes the `onCreate` callback of a `PlayerFragment` object when initializing the user interface. At location 2, this callback invokes the `Single.create` callin (from the RxJava library) to start the background task that loads the media. The subsequent call to `task.subscribe(this)` at location 3 registers the `call` callback. At a later time, the framework will invoke the `call` callback. The `call` callback uses the `Glide` class to display the media on the `this.act` object. At any time, the framework can destroy the `PlayerFragment` (e.g., when the user navigates to another part of the app) and invoke the `onDestroy` callback, which at location 5, sets the field `this.act` to `null` to prevent memory leaks.

As noted above, Figure 2b shows a crashing message history, that is, a sequence callback entries (cb), callin calls (ci), and callback returns (cbret). The arguments and return values of each message (e.g., `f`, `t`, `s`, and `a`) represent run-time addresses of objects. The app crashes if the `call` callback is invoked after the `PlayerFragment` is paused. After the `f.onCreate()` callback invocation (i.e., messages 1–4), the framework disposes the user interface and invokes the `f.onDestroy()` callback, setting the `this.act` field to `null` (i.e., app transitions between messages 5–6). Then, the background process completes triggering the `call` callback (i.e., message 7). However, the `this.act` field is now `null` causing the call to `Glide.with(this.act)` to crash (represented by the `assert` on line 5 of Figure 2a).

The fixed version adds a call to `this.sub.unsubscribe()` at line 8, which unsubscribes the `call` callback preventing its invocation after `onDestroy`. In Figure 2c, we show a message history that, while similar looking to the crashing message history of Figure 2b, is *not realizable* with respect to the framework implementation. There is no execution that can generate the message history of Figure 2c because the callback at message 8 is removed by the added message 6 in the fixed version. Such a minimal difference between the "realizable crashing" message history and the "unrealizable safe" message history highlights the automated reasoning challenge in distinguishing between the buggy and fixed versions of the app.



(a) A patch adding line 8 fixing the crash [Fietz 2018a]. On the left side, we illustrate an application-only transition system with a location `fwk` representing the framework location inside the event loop with call and return edges ⟶ for callbacks. Callins are captured by self-loops as they affect framework state but are not on the event loop. We further indicate that the framework location `fwk` is annotated with an inductive invariant to prove the assertion safe (expanded later in Figure 3).

```
1 cb f.onCreate();
2   ci t = Single.create(...);
3   ci s = t.subscribe(f);
4 cbret f.onCreate();

5 cb f.onDestroy();
6 cbret f.onDestroy();

7 cb f.call(m);
    exn a == null;
```

(b) A realizable message history witnessing an exception (`exn`) in the buggy version.

```
1 cb f.onCreate();
2   ci t = Single.create(...);
3   ci s = t.subscribe(f);
4 cbret f.onCreate();

5 cb f.onDestroy();
6   ci s.unsubscribe();
7 cbret f.onDestroy();

8 cb f.call(m);
    exn a == null;
```

(c) An unrealizable message history witnessing an exception (`exn`) in the fixed app. This cannot happen at run time.

Fig. 2. The crash from Figure 1 arises in the buggy version of the code (without line 8) when a `null` value is passed to `Glide.with` from the `this.act` field at line 6 in the `call` callback. To prevent the crash, the developer creates the fixed version by adding `unsubscribe` at line 8. Message histories shown by sub-figures (b) and (c) show how a crash might be reached in both the buggy and fixed apps (without any reasoning about their realizability). What we prove is that the message history for the fixed version shown in (c) is unrealizable. In the message histories, single-letter identifiers represent run-time instances (e.g., `f` is an instance of `PlayerFragment`) and boxes are drawn around callback invocations.

2.2 Historia: Refuting Callback Reachability with Message-History Logics

Here, we provide an overview of our static analysis abstraction and our framework specification logic that enables refining paths through the *application-only transition system* by reasoning about the realizability of message histories. In Figure 2a, we illustrate an application-only transition system, which consists of app transitions from the app code augmented with *boundary transitions* to a single, distinguished framework location `fwk`. Boundary transitions represent places where the framework makes non-deterministic choices of callback invocations as well as return values of the callin invocations (formalized in Section 3.1). The application-only transition system has the benefit of being a sound framework model by default (i.e., without further specification, is the most-over-approximate framework model) but clearly admits many unrealizable paths. Our key insight is to internalize the concept of realizable message histories into the static analysis abstraction. This internalization of realizable message histories into the abstract domain enables decoupling the specification of possible callback control flow from the abstract interpretation to compute an inductive program invariant. Such an approach is in contrast with the ones that *eagerly* augment the interprocedural control flow graph with framework-specific control flow. At

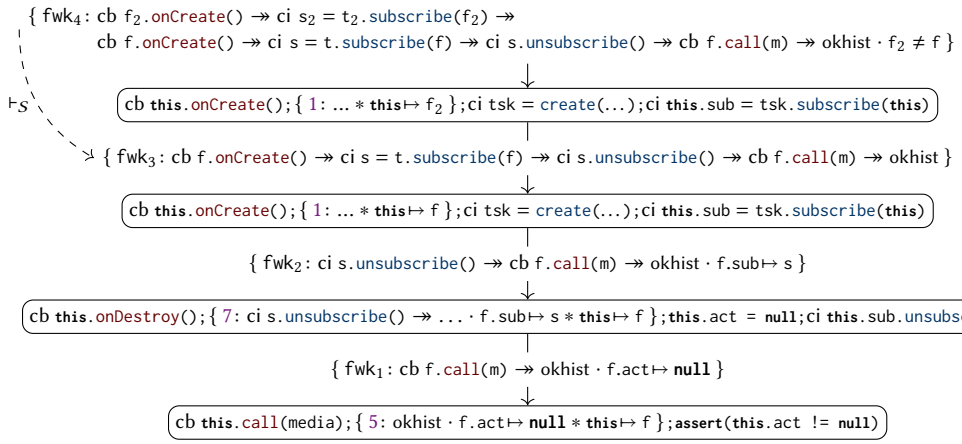


Fig. 3. An inductive invariant for the fixed app consisting of abstract message histories and abstract app states at each location in the application-only transition system (Figure 2a) that may reach the assertion failure. For brevity, this figure excludes less interesting transitions such as the case where the failing `call` invocation is preceded by another invocation of `call`. However, all transitions are considered by the verifier. The abstract message histories only show messages from the user provided specification for clarity. Note, there is only one framework location and that we use the subscripts (e.g., $\texttt{fwk}_1$, $\texttt{fwk}_2$) to indicate disjunctive elements of the abstract state at the framework location. With the specification of realizable message histories (Section 2.2.2) combined with the abstract message histories (Section 2.2.3), we can show that this invariant has reached a fixed point and that it excludes the initial state proving the assertion safe.

a high level, our approach shares some conceptual similarity with context-free language (CFL) reachability-based analysis [Reps 1998] in reasoning about realizability in the analysis but for imposing callback control flow instead of call-return semantics.

To describe our static analysis, we define Message-History Program Logic (MHPL), a program logic with an ordered linear implication for capturing assumptions about future messages. Then, to enable specifying callback-control flow, we introduce Callback-Control Flow Temporal Logic (CBCFTL), a past-time temporal logic for specifying constraints on the past message history given the present message. Finally, to automate reasoning about the realizability of message histories, we define an algorithm for instantiating CBCFTL specifications with MHPL assertions — combining the inferred assumptions from a backwards-from-error static analysis and specified callback-control–flow constraints. In the rest of this section, we demonstrate our technique by walking through the verification of the bug-fix from Figure 2a.

2.2.1 Message-History Program Logic (MHPL). Program analyses often compute invariants for each location in the program. Our approach relies on computing such an invariant that abstracts the message histories and application states that may reach the assertion failure using a novel message-history program logic (MHPL). If the invariant excludes the initial state of the program (e.g., the empty message history), then there is no way the assertion failure can be reached from the initial state, letting Historia prove that the assertion failure is impossible. Here, we demonstrate such a proof on our running example.

While analyzing the application, Historia maintains an invariant map, mapping each program location to its current invariant (represented as $\widehat{\Sigma}$ in Section 3.2). Individual locations in the application are labeled with line numbers, and the framework location representing the event loop is labeled with `fwk`. We show a representation of this invariant map for our example in Figure 3. In

a goal-directed, backwards-from-error formulation, the invariant map is initialized with the state $\{\ 5 : \text{okhist} \cdot \texttt{f.act} \mapsto \texttt{null} * \texttt{this} \mapsto \texttt{f}\ \}$ positioned right before the assertion failure. There are two parts of the abstract state (which we separate with a centered dot ·). On the right, there is an abstraction of the heap or store of the app for the assertion failure; in particular, it says an object `f` has a field `act` that points to `null` and `this` points to `f`. We use intuitionistic separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] to describe the relevant heap or store, but our approach is parameterized by essentially whatever logic one wishes to use to reason about the app state. The left component is more interesting — it is our abstraction of the possible message histories to reach this location ($\widehat{\omega}$ defined in Section 3.2). In particular, any execution that witnesses this assertion failure must have done so with a realizable message history, written okhist. Intuitively, okhist denotes the set of all realizable message histories as dictated by the framework. Note that okhist is not "top" or "true", which would concretize to all message histories — realizable or unrealizable.

Next, we compute the abstract message history at the location just before the `call` entry. This location is the `fwk` location (i.e., the framework event loop). We write the message from this transition as cb `f.call(m)` where `f` and `m` are symbolic variables corresponding to the values bound to the formal parameters `this` and `media` of `call`, respectively. To capture the effect of invoking `call`, the abstract message history is updated to cb `f.call(m)` $\twoheadrightarrow$ okhist at the abstract message history labeled by $\texttt{fwk}_1$ in Figure 3 (this abstract state also removes the `this` variable due to popping the stack). Intuitively, cb `f.call(m)` $\twoheadrightarrow$ okhist denotes any message history to which `f.call(m)` can be appended to obtain a realizable message history. Operationally, cb `f.call(m)` $\twoheadrightarrow$ okhist can be thought of as the set obtained by starting with the set of realizable message histories okhist, removing those that do not end with `f.call(m)`, and truncating the remaining ones to remove `f.call(m)` from the end.

Having computed an abstract state at a framework location ($\texttt{fwk}_1$ in Fig. 3), we can now check if the abstract state excludes the initial state; if it does not, then we have not yet found a proof that the assertion failure is unreachable. The message history abstraction of the initial state is simply a singleton set containing an empty message history. If the framework could invoke `f.call(m)` as the first callback, then `f.call(m)` alone is a realizable message history (i.e., `f.call(m)` $\in$ okhist). Consequently, cb `f.call(m)` $\twoheadrightarrow$ okhist is a set that contains an empty message history, hence it includes the initial state, prompting an alarm.

However, in reality the framework cannot invoke `f.call(m)` as the first callback (i.e., `f.call(m)` is not a realizable message history). Crucially, since the set of realizable message history okhist of the framework is not available (i.e., it is defined in the framework implementation), Historia can use separately-provided specifications of realizable message histories. The following informal specification requires `f.call(m)` to be preceded by the invocation of cb `s=t.subscribe(f)` for the message history to be realizable:

Informal Spec 1: "The framework may only invoke `call` if `subscribe` has been invoked in the past, and `unsubscribe` has not been invoked since `subscribe`."

With this targeted specification, cb `f.call(m)` $\twoheadrightarrow$ okhist no longer contains the empty message history and excludes the initial state. Historia therefore (correctly) does not raise an alarm at $\texttt{fwk}_1$.

This transformation of the abstract state from location ⑤ to $\texttt{fwk}_1$ is done by an abstract pre-transformer, which is applied repeatedly to all predecessor transitions of an updated state until reaching a fixed point. Applying the pre-transformer on the `onDestroy` callback results in the abstract state $\{\texttt{fwk}_2 : \text{ci } \texttt{s.unsubscribe()} \twoheadrightarrow \text{cb } \texttt{f.call(m)} \twoheadrightarrow \text{okhist} \cdot \ldots\}$ denoting the message histories that end in ci `s.unsubscribe()` followed by cb `f.call(m)`. Each new abstract state is added to the invariant at a location as a disjunctive clause (i.e., the `fwk` location has an invariant of the form $\{\texttt{fwk}_1 \vee \texttt{fwk}_2 \vee \ldots\}$). Continuing so on the buggy version from Figure 2a (and using the necessary specifications) would yield a state whose abstract message history is cb `f.onCreate()` $\twoheadrightarrow$ ci `s=t.subscribe(l)` $\twoheadrightarrow$

cb `l.call(m)` ↠ okhist. Note that this state includes the initial state raising an alarm, as it satisfies the constraint imposed by Informal Spec 1.

As messages are parametrized by symbolic variables, we consider an unbounded number of possible message instances at each predecessor step. Even if one restricts to one possible message instance for each callback method, considering all possible predecessor callbacks makes the proof search exponential. Fortunately, we are able to join abstract states by merging disjunctions, and merging is required to find a fixed point in most cases. Given a disjunction of abstract states, one disjunct may be merged with another if the first disjunction implies, or *entails*, the one it is being merged with. By merging new abstract states from backward transitions when possible, we can reach a fixed point on some paths being explored. In Figure 3, for example, $\texttt{fwk}_4$ is merged with $\texttt{fwk}_3$. No further backward transitions need to be explored from $\texttt{fwk}_4$ because they have been explored from $\texttt{fwk}_3$.

For presentation, Fig. 3 shows only a few of the transitions and abstract states computed for the running example. In practice, Historia computes the fixed point at the framework location after considering all backward transitions and shows that the resultant inductive invariant excludes the initial state (for the running example with the fix). In other words, it proves that no realizable message history reaches the assertion violation in the fixed version of AntennaPod.

2.2.2 Callback Control-Flow Temporal Logic (CBCFTL). CBCFTL is a novel language for formally writing specifications of realizable message histories. A specification written in CBCFTL consists of a conjunction of *history implications*. With no history implications, the CBCFTL specification places no restrictions on the realizable message history. Each additional history implication targets one message that the framework can control, such as the invocation of the callback `call` or `onCreate`. A history implication $\widehat{m} \mathrel{\square\!\!\rightarrow} \widetilde{\omega}$ says that whenever a message satisfying the target abstract message $\widehat{m}$ occurs, then the preceding message history must satisfy the temporal formula $\widetilde{\omega}$.[1] Temporal formulas are drawn from a syntactically restricted fragment of past-time linear temporal logic over finite traces. Such a structure for CBCFTL is natural for three reasons: (1) it allows the developer of the framework model to target callbacks or callins with history implications as needed, (2) history implications are compositional, and (3) specifications may be cleanly combined with abstract message histories to automatically check excludes-initial and entailment (as described in Section 2.2.3).

For Informal Spec 1, the history implication captures what must be true of the message history when cb `l.call(m)` is next. First, there must exist a subscription object `s` that was returned from invoking `subscribe`. This object `s` must be returned from the same invocation of `subscribe` that `l` was passed to registering the `call` callback. The subscription object `s` is the only parameter to the method `unsubscribe` which must not have come since the invocation of `subscribe`. Invoking `unsubscribe` on other objects will have no effect on the target `call`. All of this is captured by History Implication 1.

History Implication 1. For all objects `l` and `m`, if the framework invokes `l.call(m)`, then for some (subscription) object `s`, the message ci `s.unsubscribe()` has **N**ot happened **S**ince ci `s = _.subscribe(l)`.

cb `l.call(m)` $\mathrel{\square\!\!\rightarrow}$ ∃s. ci `s.unsubscribe()` **NS** ci `s = _.subscribe(l)`

Note that *since*, **S**, is the past-time dual of *until*, **U**, in LTL. The **NS** operator has a built-in "not" and restricts nesting to maintain decidability (discussed in Section 4). Underscore _ in the above specification is simply a shorthand for a locally existentially-quantified variable (i.e., "don't care").

A key feature of CBCFTL is that it handles quantified values such as the listener object `l` and the subscription object `s`. Since the history implication applies any time a `call` is invoked, the listener

[1]The $\widehat{m} \mathrel{\square\!\!\rightarrow} \widetilde{\omega}$ history implication can be seen as the first-order, past-time linear temporal logic formula $\square(\widehat{m} \rightarrow \mathbf{Y}\,\widetilde{\omega})$ that combines always (or globally) $\square$, implication $\rightarrow$, and yesterday (or previous) **Y** from past-time linear temporal logic (ptLTL).

object `l` is universally quantified. Reasoning about such quantifier alternation is often undecidable. However, the restrictions we have chosen for CBCFTL allow them to be combined with abstract message histories such that automated reasoning is feasible.

2.2.3 Combining Abstract Message Histories with Callback Control-Flow. Next, we consider how to interpret and automatically reason about the meaning of abstract message histories. Consider the abstract state just before the `call` callback with the abstract message history transition, cb `f.call(m)` ↠ okhist, which says that the next message must be cb `f.call(m)`. First, excludes-initial needs to be proven (i.e., this abstract state does not contain the initial state), and then, entailment checks if it should be merged with any equivalent or weaker abstract states. Both of these steps rely on a first-order logic encoding of abstract message histories that we explain here. We prove the resulting first-order logic encoding to be decidable for the *excludes-initial* judgment and computable in practice for the *entailment* judgment.

This encoding starts by combining the abstract state cb `f.call(m)` ↠ okhist with History Implication 1. The first step of combining abstract message histories with temporal formula is *instantiation* (i.e., the INSTANTIATE-YES judgment in Section 5). Intuitively, instantiation turns the "next" message from the abstract state into requirements on the message history so far. For convenience, the output of instantiation is represented by the same language of temporal formula as is used in the history implications. Temporal formula (1) shown below results from instantiating History Implication 1 on the abstract message history cb `f.call(m)` ↠ okhist. Note how fresh variables are introduced for the existentially quantified values, but the values from the `call` are retained.

$$\exists \texttt{s'}, \texttt{t'}.\ \text{ci}\ \texttt{s'.unsubscribe()}\ \mathbf{NS}\ \text{ci}\ \texttt{s'} = \texttt{t'.subscribe(f)} \tag{1}$$

Such a temporal formula may be converted to first-order logic and checked for excludes-initial and entailment via SMT solver as explained in Section 2.2.2.

This temporal formula excludes the initial state because there must exist a `subscribe` call in the message history (i.e., the judgment $\vdash_S \widehat{\omega}$ excludesinit from section Section 5). Here, we also see why targeted specification is desirable for performance reasons. Each history implication can add constraints that prevent states from being merged via entailment. The result of combining two sound history implications is always sound. However, such combinations may impact performance by increasing the abstract state disjunctions at a location.

The next abstract message history shown by the invariant map at $\texttt{fwk}_2$ adds the `unsubscribe` call, ci `s.unsubscribe()` ↠ cb `f.call(m)` ↠ okhist. The previously instantiated formula needs to be updated for the ci `s.unsubscribe()` message. That is, we must consider two cases: (1) this `unsubscribe` matches the `unsubscribe` from the previous step (deriving a contradiction), and (2) this `unsubscribe` is irrelevant to the previous step. Combining these cases into the temporal formula is referred to as *quotienting* (i.e., the QUOTIENT-NOT-SINCE judgment from Section 5). Additionally, if there was a history implication targeting `unsubscribe`, it would also need to be instantiated (since there is not, the INSTANTIATE-NO rule applies instead). Combining these steps results in temporal formula (2).

$$\exists \texttt{s'}.\ (\text{ci}\ \texttt{s'.unsubscribe()} \neq \text{ci}\ \texttt{s.unsubscribe()}) \wedge \text{ci}\ \texttt{s'.unsubscribe()}\ \mathbf{NS}\ \text{ci}\ \texttt{s'} = \_\texttt{.subscribe(f)} \tag{2}$$

We attempt to eagerly merge abstract states using *entailment*. While as noted above $\texttt{fwk}_4$ merges with $\texttt{fwk}_3$, the abstract states at $\texttt{fwk}_1$ and $\texttt{fwk}_2$ cannot be merged since they restrict the app heap differently. However for presentation, we illustrate entailment on the abstract message histories from $\texttt{fwk}_1$ and $\texttt{fwk}_2$, ignoring the app heap. To determine entailment, we algorithmically search for a message history represented by temporal formula (2) and not by temporal formula (1). If no such message history exists, then this disjunction has not progressed toward the initial state and can be dropped. This entailment holds: the added constraint (ci `s'.unsubscribe()` ≠ ci `s.unsubscribe()`) simplifies to `s'` ≠ `s` and does not add any message histories to the abstraction over those represented

by temporal formula (1). Note with the abstract app heap, a concrete app heap where `f.act` points to a non-null value is represented by $\texttt{fwk}_2$ but not $\texttt{fwk}_1$, so these states may not be merged overall.

3 MESSAGE-HISTORY PROGRAM LOGIC (MHPL)

In this section, we explain the process of proving an application safe by showing no realizable message history can reach the assertion failure. First, we define the notion of an application-only transition system that records a *message history* during execution (Section 3.1). Executions in this transition system, such as reaching the assertion failure, may be restricted based on whether they are realizable (e.g., with a user provided specification). The application-only transition system provides a concrete semantics for a message-history program logic (MHPL) to reason about realizable message histories by adding the message history ω to the concrete state as ghost state. Using MHPL, we can abstract message histories backwards from an assertion proving that no failing message history is realizable (Section 3.2).

3.1 An Application-Only Transition System with Message Histories

Figure 4 defines the syntax and semantics of a program that uses *boundary transitions* to provide semantics to an app absent of the hidden framework implementation. Conceptually, all framework code is merged within a single framework location `fwk` (as illustrated in Figure 2a from Section 2). Our semantics are non-deterministic when the framework chooses the arguments for a callback invocation or the return value for a callin. Execution simply "gets stuck" on an unrealizable boundary transition from the framework.

Boundary transitions b append a message to the message history ω capturing all interaction between the app and framework. A boundary transition is a crossing of the app-framework boundary via a callback invocation $\texttt{fwk} \dashv[\texttt{cb}\ md(\overline{x})]\mapsto \ell$ from the framework back to the app, a callback return $\ell \dashv[\texttt{cbret}\ x'\ md(\overline{x})]\mapsto \texttt{fwk}$ from the app into the framework, or a callin invocation $\ell \dashv[\texttt{ci}\ x'\ md(\overline{x})]\mapsto \ell'$ from the app into the framework and back. App transitions t represent app code, for example, consisting of standard operations like reading and writing to the application heap. A message history $\omega ::= \varepsilon \mid \omega; m$ is a sequence of messages with ε being the empty sequence. The application-only transition system is parametrized by a set of realizable message histories Ω representing actions possible under the real framework.

Method names md are a fully qualified and disambiguated name for method procedures. We assume we can identify a method as being an app (i.e., a callback) method or a framework (i.e., a callin) method based on the method identifier md (e.g., app methods that override a framework type are callbacks in the case of Android). The key part of the program state is recording a message history where messages m are instances of boundary transitions; that is, a callback invocation with bound values $\texttt{cb}\ md(\overline{v})$, a callback return $\texttt{cbret}\ v'\ md(\overline{v})$, or a callin invocation $\texttt{ci}\ v'\ md(\overline{v})$. Values, v, may be compared with equality, created by the app, or created by the framework.

Callbacks and callins use a sequence of parameters as program variables x and return a value; we write a sequence with an overline (e.g., $\overline{x}$ for a sequence of variables). For simplicity, we assume that variable scoping and shadowing is handled by translation to this language (e.g., via alpha-renaming). A callback return $\texttt{cbret}\ x'\ md(\overline{x})$ says that it returns the value in variable x' — for the corresponding callback $\texttt{cb}\ md(\overline{x})$; for simplicity, we assume an A-normal form where program expressions are evaluated in internal app transitions and bound to variable x' here. For a callin invocation $\texttt{ci}\ x'\ md(\overline{x})$, variable x' is the variable to bind the return value of the invocation.

We see transitions as control-flow edges between two program locations *loc*. A program location can be the framework location `fwk` or an app location ℓ. The framework location `fwk` represents all control locations inside the framework. A program p is then a set of boundary b or app transitions t for some unspecified syntax of app transitions. Conceptually, a program p is the control-flow

boundary transitions $b ::= \mathtt{fwk} \dashv[\mathrm{cb}\ md(\overline{x})]\!\vdash \ell \mid \ell \dashv[\mathrm{cbret}\ x'\ md(\overline{x})]\!\vdash \mathtt{fwk} \mid \ell \dashv[\mathrm{ci}\ x'\ md(\overline{x})]\!\vdash \ell'$

app transitions t message histories $\omega ::= \varepsilon \mid \omega; m$ realizable message histories Ω

method names md messages $m ::= \mathrm{cb}\ md(\overline{v}) \mid \mathrm{cbret}\ v'\ md(\overline{v}) \mid \mathrm{ci}\ v'\ md(\overline{v})$ values v

variables x program locations $loc ::= \mathtt{fwk} \mid \ell$ app locations ℓ

programs $p ::= \circ \mid p, b \mid p, t$ program states $\sigma ::= loc : \mu$

memories $\mu ::= \omega \cdot \kappa \cdot \rho$ app stores ρ

boundary stacks $\kappa ::= \circ \mid \kappa; k$ boundary activations $k ::= \mathrm{cb}\ md(\overline{v})$ app states $\varsigma ::= \ell : \rho$

$$\boxed{\langle \sigma, b\rangle \Downarrow^{\Omega} \sigma' \qquad \sigma \rightarrow_p^{\Omega} \sigma'}$$

C-CALLBACK-INVOKE

$$\frac{\omega; \mathrm{cb}\ md(\overline{v}) \in \Omega}{\langle \mathtt{fwk} : \omega \cdot \kappa \cdot \rho, \mathtt{fwk} \dashv[\mathrm{cb}\ md(\overline{x})]\!\vdash \ell\rangle \Downarrow^{\Omega} \ell : \omega; \mathrm{cb}\ md(\overline{v}) \cdot \kappa; \mathrm{cb}\ md(\overline{v}) \cdot \rho[\overline{x \mapsto v}]}$$

C-CALLBACK-RETURN

$$\frac{m = \mathrm{cbret}\ \rho(x')\ md(\overline{v}) \qquad \overline{v} = \overline{\rho(x)} \qquad \omega; m \in \Omega}{\langle \ell : \omega \cdot \kappa; \mathrm{cb}\ md(\overline{v}) \cdot \rho, \ell \dashv[\mathrm{cbret}\ x'\ md(\overline{x})]\!\vdash \mathtt{fwk}\rangle \Downarrow^{\Omega} \mathtt{fwk} : \omega; m \cdot \kappa \cdot \rho}$$

C-CALLIN-INVOKE

$$\frac{\overline{v} = \overline{\rho(x)} \qquad m = \mathrm{ci}\ v'\ md(\overline{v}) \qquad \omega; m \in \Omega}{\langle \ell : \omega \cdot \kappa \cdot \rho, \ell \dashv[\mathrm{ci}\ x'\ md(\overline{x})]\!\vdash \ell'\rangle \Downarrow^{\Omega} \ell' : \omega; m \cdot \kappa \cdot \rho[x' \mapsto v']}$$

C-APP-STEP

$$\frac{\langle \ell : \rho, t\rangle \Downarrow \ell' : \rho' \qquad t \in p \qquad \ell = \mathrm{pre}(t) \qquad \ell' = \mathrm{post}(t)}{\ell : \omega \cdot \kappa \cdot \rho \rightarrow_p^{\Omega} \ell' : \omega \cdot \kappa \cdot \rho'}$$

C-BOUNDARY-STEP

$$\frac{\langle \sigma, b\rangle \Downarrow^{\Omega} \sigma' \qquad b \in p}{\sigma \rightarrow_p^{\Omega} \sigma'}$$

initial program state $\sigma_{\mathrm{init}} = \mathtt{fwk} : \varepsilon \cdot \circ \cdot \rho_{\mathrm{init}}$ initial app store ρ_{init}

Fig. 4. An application-only transition system with boundary transitions and message histories. The message history ω component of the program state records the execution of boundary transitions b between the app and framework. We use the judgment $\sigma \rightarrow_p^{\Omega} \sigma'$ to represent a single step over either an app or boundary transition in the application. The transition system is parametrized by a set of realizable message histories $\omega \in \Omega$.

graph for each app callback augmented with boundary control-flow edges into and back from the framework location `fwk`.

A program state σ is a memory μ, at program location loc, which consists of a message history ω with a boundary stack κ and an app store ρ. A boundary stack κ is a stack ensuring that the message history consists of matching calls and returns. If we assume that callbacks may not be nested inside of callbacks, this stack may have at most one activation k. Like app transitions, the specific form of app stores ρ is unspecified, except we assume it supports looking up the value for an app variable $\rho(x)$ and initializing variables $\rho[x \mapsto v]$. An app state ς is then a pair of an app location ℓ and an app store ρ.

Boundary transitions b are particularly interesting as they capture the non-deterministic or unobserved behavior of the framework and record the action in the "ghost state" ω for the message history. The boundary transition judgment form $\langle \sigma, b \rangle \Downarrow^{\Omega} \sigma'$ says, "In program state σ, executing the boundary transition b results in an updated program state σ' and is realizable under realizable message histories Ω." This judgment form captures the realizable executions of boundary transitions. To execute a callback invocation transition $\mathtt{fwk} \dashv[\mathsf{cb}\ md(\overline{x})]\mapsto \ell$ via rule C-CALLBACK-INVOKE, the program state is at the framework location $\mathtt{fwk}$, values $\overline{v}$ are chosen for the callback parameters $\overline{x}$ non-deterministically (conceptually by the framework) and initialized in the app store $\rho[\overline{x \mapsto v}]$, and the callback activation $\mathsf{cb}\ md(\overline{v})$ is pushed on the boundary stack κ. Then to record this boundary transition execution, this callback message $\mathsf{cb}\ md(\overline{v})$ is appended onto the current message history ω. We want to capture that depending on the current message history ω, this callback invocation transition may not be realizable. This realizability of message histories is captured by checking if the new message history is realizable with $\omega; \mathsf{cb}\ md(\overline{v}) \in \Omega$.

Executing a callback return transition $\ell \dashv[\mathsf{cbret}\ x'\ md(\overline{x})]\mapsto \mathtt{fwk}$ via rule C-CALLBACK-RETURN is then the expected symmetric operation. The return value is read out of the app store $\rho(x')$, the callback activation $\mathsf{cb}\ md(\overline{v})$ is popped off the boundary stack, and control goes into the framework location $\mathtt{fwk}$. The premise $\overline{v} = \overline{\rho(x)}$ enforces that argument variables are not modified by the callback which simplifies the formalism. The callback return message $\mathsf{cbret}\ \rho(x')\ md(\overline{v})$ is similarly appended onto the current message history ω to record the execution of the callback return transition — and checked for realizability. Note that to connect the callback invocation with its return, the callback return message includes the method name md and actual arguments $\overline{v}$ from the callback activation.

The callin invocation transition $\ell \dashv[\mathsf{ci}\ x'\ md(\overline{x})]\mapsto \ell'$ via rule C-CALLIN-INVOKE is symmetric to callback invocation and return together, that is, the arguments for the callin $\overline{v}$ are read from the app store, and the callin return value from the framework v' is chosen non-deterministically (conceptually by the framework) and then bound to variable x'. Then, the callin message $\mathsf{ci}\ v'\ md(\overline{v})$ is appended onto the current message history ω and checked for realizability. It should be noted that C-CALLBACK-INVOKE, C-CALLBACK-RETURN, and C-CALLIN-INVOKE together capture that the framework cannot modify the app store ρ except through invoking callback methods. This formalizes one aspect of the so-called separate compilation assumption of Ali and Lhoták [2012], which considers the consequences of the assumption that framework is developed separately and compiled in the absence of the app. As a consequence of these semantics checking realizability at each boundary transition, every prefix of a realizable message history must also be realizable.

The application-only transition system is then given by the transition relation judgment form $\sigma \rightarrow_p^{\Omega} \sigma'$ that says, "Program state σ steps to σ' in program p by either a boundary transition b or app transition t under realizable message histories Ω." Straightforwardly, the C-APP-STEP and C-BOUNDARY-STEP rules simply state that we can either take a step with an app transition t or a boundary transition b in the program p (depending on the program location). The transition semantics of app transitions are left unspecified $\langle \varsigma, t \rangle \Downarrow \varsigma'$. We assume that app transitions themselves do not read or write framework state directly, as the app store ρ is separate from the framework state. And finally, concrete executions are given by the reflexive-transitive closure of this single-step transition relation from an initial program state σ_{init}. We write $\sigma \rightarrow_p^{*\Omega} \sigma'$ for the reflexive-transitive closure of $\sigma \rightarrow_p^{\Omega} \sigma'$.

3.2 Refuting Callback Reachability with Message-History Program Logic

The ultimate aim of MHPL is to prove statically that a program assertion cannot fail. We start with the *error condition*, $\widehat{\sigma}$, an abstract state just before the assertion such that the assertion may fail (e.g.,

`f.act`$\mapsto$ `null` from the running example in Section 2 representing app memories where there exists a framework object with a null `act` field). We refute the reachability of the error condition with the judgment form $\vdash_p^S \widehat{\sigma}$ unreach. This judgment form is read as, "No concrete program state satisfying the abstract state $\widehat{\sigma}$ is reachable in program p with realizable message history specification S." We use CBCFTL, defined in Section 4, to abstract the set of reachable message histories Ω in the concrete semantics. We use the judgment $\omega \models S$ to say that a message history ω is captured by a specification S and note the set of message histories in a specification as Ω_S (i.e., the concretization of a specification S is the set of message histories Ω_S). Since the specification is an input to our algorithm, we assume that S is a sound abstraction of realizable message histories (i.e., $\Omega_S \subseteq \Omega$ for the set of realizable message histories Ω in the concrete semantics).

Our proof technique works in a goal-directed manner: we over-approximate the set of the states that may reach the given error condition $\widehat{\sigma}$ with a program-state invariant, $\widehat{\Sigma}$. If the initial program state σ_{init}: `fwk`: $\varepsilon \cdot \circ \cdot \rho_{\text{init}}$ is excluded from the program state-invariant, $\widehat{\Sigma}$, then the location of $\widehat{\sigma}$ cannot be reached with any concrete state satisfying abstract state error condition $\widehat{\sigma}$. For some abstract state $\widehat{\sigma}$, the *excludes-initial* judgment, $\vdash_S \widehat{\sigma}$ excludesinit, holds only if the concretization of $\widehat{\sigma}$ must not contain the initial state σ_{init}. While an abstract state $\widehat{\sigma}$ may be excludes-initial in any of its components (e.g., the abstract app store), the particularly interesting component here is its abstract message history $\widehat{\omega}$. Thus, in subsequent sections, we focus in on the excludes-initial judgment on abstract message histories. Excludes-initial for message histories, $\vdash_S \widehat{\omega}$ excludesinit, holds if ε is not in the concretization of $\widehat{\omega}$.

3.2.1 An Abstract Semantics with Message Histories. In Figure 5, we define MHPL, which abstracts the application-only transition system from Section 3.1, to derive refutations with respect to message histories. To abstract messages $\widehat{m}$, we replace concrete values with symbolic variables $\hat{x}$. Symbolic variables are existentially quantified across each part of the abstract program state with an assignment, θ. That is, a concrete state σ satisfies the concretization relation of an abstract state, $\widehat{\sigma}$ (i.e., $\sigma \models \widehat{\sigma}$) if a θ exists such that each part of σ satisfies the concretization relation with each part of $\widehat{\sigma}$ (e.g., $\omega \cdot \theta \models_S \widehat{\omega}$).

An abstract message history $\widehat{\omega}$ captures the set of message histories reaching a given program location *loc* under the realizable message history specification S. An abstract message history can be okhist, which corresponds to all realizable message histories under S. Note that since we only care about *realizable* message histories, we do not include a $\top$ abstract message history corresponding to *all* message histories. Since our logic explores backwards, it adds constraints on *future* boundary transitions to the abstract message history $\widehat{\omega}$ as they are encountered. The key is to see this constraint as an ordered linear implication on the right $\widehat{m} \twoheadrightarrow \widehat{\omega}$, which informally says, "For all messages satisfying $\widehat{m}$, appending that message to the current message history implies that the new message history satisfies $\widehat{\omega}$." In the middle part of Figure 5, we give a precise concretization relation between a message history with an assignment $\omega \cdot \theta$ and an abstract message history: $\omega \cdot \theta \models_S \widehat{\omega}$.

The rest of an abstract program state is straightforward. We do not care specifically about the form of abstract app stores, except that like concrete app stores, we need a way to look up a (symbolic) value for a variable and to initialize variables. To do that, we use intuitionistic separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] to indicate arbitrary store $\top$, separating-conjunction of two stores $\widehat{\rho}_1 * \widehat{\rho}_2$, a singleton points-to or cell for program variables $x \mapsto \hat{x}$, an infeasible store $\bot$, or a disjunction of stores $\widehat{\rho}_1 \vee \widehat{\rho}_2$, which we usually consider in disjunctive normal form. An abstract app state $\widehat{\varsigma} ::= \ell : \widehat{\rho} \mid \cdots$ is then an abstract app store and location ℓ. An abstract memory $\widehat{\mu} ::= \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \mid \cdots$ is then a product of an abstract message history $\widehat{\omega}$, an abstract boundary stack $\widehat{\kappa}$, and an abstract store $\widehat{\rho}$ — or a disjunction of such products. An abstract program state $\widehat{\sigma} ::= loc : \widehat{\mu}$ is

abstract messages $\widehat{m} ::= \mathrm{cb}\ md(\overline{\hat{x}}) \mid \mathrm{ci}\ \hat{x}'\ md(\overline{\hat{x}}) \mid \mathrm{cbret}\ \hat{x}'\ md(\overline{\hat{x}})$ symbolic variables $\hat{x}$

assignments $\theta ::= \circ \mid \theta[\hat{x} \mapsto v]$ abstract message histories $\widehat{\omega} ::= \mathsf{okhist} \mid \widehat{m} \twoheadrightarrow \widehat{\omega}$

realizable message histories specs S abstract app stores $\widehat{\rho} ::= \top \mid \widehat{\rho}_1 * \widehat{\rho}_2 \mid x \mapsto \hat{x} \mid \cdots \mid \bot \mid \widehat{\rho}_1 \vee \widehat{\rho}_2$

abstract app states $\widehat{\varsigma} ::= \ell : \widehat{\rho}$ abstract memories $\widehat{\mu} ::= \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \mid \bot \mid \widehat{\mu}_1 \vee \widehat{\mu}_2$

abstract program states $\widehat{\sigma} ::= loc : \widehat{\mu}$ abstract boundary stacks $\widehat{\kappa} ::= \top \mid \widehat{\kappa}_1 \bullet \mathrm{cb}\ md(\overline{\hat{x}})$

abstract program-state invariants $\widehat{\Sigma} ::= \circ \mid \widehat{\Sigma}, \widehat{\sigma}$

$$\boxed{\omega \cdot \theta \models_S \widehat{\omega}}$$

$\omega \cdot \theta \models_S \mathsf{okhist}$ iff $\omega \models S$ $\omega \cdot \theta \models_S \widehat{m} \twoheadrightarrow \widehat{\omega}$ iff $m \cdot \theta \models \widehat{m}$ implies $\omega; m \cdot \theta \models_S \widehat{\omega}$ and $\omega; m \models S$

$\boxed{\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}}$

A-CALLBACK-INVOKE

$$\frac{}{\vdash \{\mathtt{fwk} : \mathrm{cb}\ md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}\ \mathtt{fwk}\ \text{-}[\mathrm{cb}\ md(\overline{x})]\!\!\rightarrow \ell\ \{\ell : \widehat{\omega} \cdot \widehat{\kappa} \bullet \mathrm{cb}\ md(\overline{\hat{x}}) \cdot \widehat{\rho} * * \overline{x \mapsto \hat{x}}\}}$$

A-CALLBACK-RETURN

$$\frac{\widehat{\rho} = \widehat{\rho}' * x' \mapsto \hat{x}' * * \overline{x \mapsto \hat{x}}}{\vdash \{\ell : \mathrm{cbret}\ \hat{x}'\ md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \bullet \mathrm{cb}\ md(\overline{\hat{x}}) \cdot \widehat{\rho}\}\ \ell\ \text{-}[\mathrm{cbret}\ x'\ md(\overline{x})]\!\!\rightarrow \mathtt{fwk}\ \{\mathtt{fwk} : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}}$$

A-CALLIN-INVOKE

$$\frac{\widehat{\rho} = \widehat{\rho}' * * \overline{x \mapsto \hat{x}}}{\vdash \{\ell : \mathrm{ci}\ \hat{x}'\ md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}\ \ell\ \text{-}[\mathrm{ci}\ x'\ md(\overline{x})]\!\!\rightarrow \ell'\ \{\ell' : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} * x' \mapsto \hat{x}'\}}$$

$\boxed{\widehat{\Sigma} \vdash_S b}$

A-BOUNDARY-STEP

$$\frac{\widehat{\Sigma}(\mathrm{post}(b)) \vdash_S \widehat{\sigma} \quad \vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\} \quad \widehat{\sigma}' \vdash_S \widehat{\Sigma}(\mathrm{pre}(b))}{\widehat{\Sigma} \vdash_S b}$$

$\boxed{\widehat{\Sigma} \vdash t}$

A-APP-STEP

$$\frac{\mathrm{app}(\widehat{\Sigma}(\mathrm{post}(t))) \vdash \widehat{\varsigma} \quad \vdash \{\widehat{\varsigma}'\}\ t\ \{\widehat{\varsigma}\} \quad \widehat{\varsigma}' \vdash \mathrm{app}(\widehat{\Sigma}(\mathrm{pre}(t)))}{\widehat{\Sigma} \vdash t}$$

$$\boxed{\widehat{\Sigma} \vdash^S_p \widehat{\sigma} \qquad \vdash^S_p \widehat{\sigma}\ \mathsf{unreach}}$$

A-INDUCTIVE

$$\frac{\widehat{\sigma} \vdash_S \widehat{\Sigma}(\mathrm{loc}(\widehat{\sigma})) \qquad \widehat{\Sigma} \vdash_S b \ \text{ for all } b \in p \qquad \widehat{\Sigma} \vdash t \ \text{ for all } t \in p}{\widehat{\Sigma} \vdash^S_p \widehat{\sigma}}$$

A-REFUTE

$$\frac{\widehat{\Sigma} \vdash^S_p \widehat{\sigma} \qquad \vdash_S \widehat{\Sigma}(\mathtt{fwk})\ \mathsf{excludesinit}}{\vdash^S_p \widehat{\sigma}\ \mathsf{unreach}}$$

Fig. 5. Refuting callback reachability with a MHPL. We abstract the application-only transition system with Hoare triples over app and boundary transitions and an abstract program state invariant $\widehat{\Sigma}$. The location of an abstract state is noted with $\mathrm{loc}(\widehat{\sigma})$ and looking up the state at a location in the invariant is noted with $\widehat{\Sigma}(loc)$. Executing backwards, an abstract message history $\widehat{\omega}$ becomes conditional in messages observed in the future execution. An abstract realizable message history S is parameter and is the abstract analogue of the concrete set of realizable message histories Ω.

simply an abstract memory $\widehat{\mu}$ at a program location *loc*. For abstract boundary stacks $\widehat{\kappa}$, we consider arbitrary boundary stacks $\top$, or appending a boundary activation $\widehat{\kappa} \bullet \mathsf{cb}\ md(\overline{\hat{x}})$. Finally, we consider abstract program-state invariants $\widehat{\Sigma}$ to be a set of abstract program states $\widehat{\sigma}$, which we also treat as a map from locations *loc* to the abstract state $\widehat{\sigma}$ at that location (i.e., $\widehat{\Sigma}(loc) = \widehat{\sigma}$ iff $\widehat{\sigma} = loc\colon \widehat{\mu}$ and $\widehat{\sigma} \in \widehat{\Sigma}$).

We describe the abstract semantics of boundary transitions b as Hoare triples $\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}$, except that we are interested in backwards over-approximating triples instead of forwards. That is, we read the judgment as, "If there is an execution of the boundary transition b to a post-state satisfying $\widehat{\sigma}$, the pre-state of that execution satisfies $\widehat{\sigma}'$" (Lemma 3.1).

Lemma 3.1 (Hoare triple soundness). *If* $\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}$ *and* $\langle \sigma', b\rangle \Downarrow^{\Omega} \sigma$ *such that* $\sigma \models_S \widehat{\sigma}$ *and* $\Omega \subseteq \Omega_S$, *then* $\sigma' \models_S \widehat{\sigma}'$.

The abstract semantics rules for boundary transitions b follow closely their concrete counterparts (assuming a structural rule for disjunction of memories $\widehat{\mu}$). The A-CALLBACK-INVOKE rule captures computing the pre-condition of the callback invocation transition $\mathtt{fwk} \dashv[\mathsf{cb}\ md(\overline{x})]\mapsto \ell$ and shows moving from an assertion on the abstract boundary stack $\widehat{\kappa} \bullet \mathsf{cb}\ md(\overline{\hat{x}})$ to a hypothetical next message in the abstract message history $\mathsf{cb}\ md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega}$. In detail, it first asserts that the post-app memory has bindings for the callback parameters $\widehat{\rho} * * \overline{x \mapsto \hat{x}}$ and has the corresponding callback activation on top of the boundary stack $\widehat{\kappa} \bullet \mathsf{cb}\ md(\overline{\hat{x}})$. Then, we drop the parameter bindings and pop the callback activation. Finally, we update the abstract message history with the abstract message corresponding to the callback invocation, $\mathsf{cb}\ md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega}$. As an example from Section 2, the abstract state $\{\ 5\colon \mathsf{okhist} \cdot \mathtt{f.act} \mapsto \mathtt{null} * \mathtt{this} \mapsto \mathtt{f}\ \}$ just after the entry of `call` produces the pre-state $\{\ \mathtt{fwk}_1\colon \mathsf{cb}\ \mathtt{f.call(m)} \twoheadrightarrow \mathsf{okhist} \cdot \mathtt{f.act} \mapsto \mathtt{null}\ \}$ at the framework location which proceeds `call`.

Continuing to mirror the abstract semantics, the A-CALLBACK-RETURN rule pushes a hypothetical callback message on the boundary stack corresponding to the callback that would have just returned in the concrete execution. Similar to abstract callback invoke (A-CALLBACK-INVOKE), abstract callback return (A-CALLBACK-RETURN) and abstract callin invoke (A-CALLIN-INVOKE) add hypotheticals to the abstract message history. The main difference is how each updates the abstract app store. For A-CALLBACK-RETURN, the return value and its relationship to other symbolic variables is unknown, therefore we ensure that the separation logic domain has materialized return values and arguments for the callback via $\widehat{\rho}' * x' \mapsto \hat{x}' * * \overline{x \mapsto \hat{x}}$. For example, the post state from the running example is transferred over the return of the `onDestroy` callback to produce the pre-state $\{\ 9\colon \mathsf{cbret}\ \mathtt{f_2.onDestroy()} \twoheadrightarrow \mathsf{cb}\ \mathtt{f.call(m)} \twoheadrightarrow \mathsf{okhist} \cdot \ldots\ \}$. Note that we elide the return value here, as `onDestroy` is `void`. The value $\mathtt{f}_2$ may or may not alias `f`, as there is a case split from the separation logic materialization (Figure 3 from Section 2 shows just the aliased case).

Finally, A-CALLIN-INVOKE removes a program variable from the post app store corresponding to the return value and introduces fresh symbolic variables to the pre-store bound to the arguments of the callin invoke. For example, the post state $\{\ 3\colon \ldots \twoheadrightarrow \mathsf{okhist} \cdot \mathtt{task} \mapsto \mathtt{t} * \mathtt{this} \mapsto \mathtt{f}\ \}$ transferred over the `create` call creates the pre-state $\{\ 2\colon \mathsf{ci}\ \mathtt{t} = \mathtt{create(\ldots)} \twoheadrightarrow \ldots \twoheadrightarrow \mathsf{okhist} \cdot \mathtt{this} \mapsto \mathtt{f}\ \}$.

We write $\mathsf{app}(\widehat{\sigma}) = \widehat{\varsigma}$ for the projection of an abstract program state $\widehat{\sigma}$ to an abstract app state $\widehat{\varsigma}$ that drops the abstract message history $\widehat{\omega}$ and abstract boundary stack $\widehat{\kappa}$ components. The app transitions are checked for being inductive in the analogous way with the A-APP-STEP rule defining the judgment form $\widehat{\Sigma} \vdash t$, which similarly depends on an abstract semantics for app transitions $\vdash \{\widehat{\varsigma}'\}\ t\ \{\widehat{\varsigma}\}$ and an entailment judgment for abstract app states $\widehat{\varsigma} \vdash \widehat{\varsigma}'$.

We then check that the abstract program-state invariants $\widehat{\Sigma}$ are inductive for executing backwards a given boundary transition b with the judgment form $\widehat{\Sigma} \vdash_S b$. This judgment says, "In abstract

program-state invariants $\widehat{\Sigma}$, executing boundary transition b backwards is inductive when constrained by realizable message histories defined by specification S." The A-BOUNDARY-STEP defines this judgment and captures the backwards over-approximation. Specifically, it depends on an entailment judgment $\widehat{\sigma} \vdash_S \widehat{\sigma}'$ that is parametrized by the realizable message histories specification S (i.e., it should satisfy the following soundness condition: if $\widehat{\sigma} \vdash_S \widehat{\sigma}'$ and $\sigma \models_S \widehat{\sigma}$, then $\sigma \models_S \widehat{\sigma}'$). To get the post- and pre-locations of a boundary transition b, we write $\mathrm{post}(b)$ and $\mathrm{pre}(b)$, respectively. Then, the rule chooses some $\widehat{\sigma}$ that over-approximates $\widehat{\Sigma}(\mathrm{post}(b))$ — i.e., $\widehat{\Sigma}(\mathrm{post}(b)) \vdash_S \widehat{\sigma}$, applies the abstract semantics for b — i.e., $\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}$, and checks that $\widehat{\Sigma}(\mathrm{pre}(b))$ over-approximates $\widehat{\sigma}'$ — i.e., $\widehat{\sigma}' \vdash_S \widehat{\Sigma}(\mathrm{pre}(b))$. This rule is the backwards over-approximating version of the usual Hoare rule of consequence. A similar judgment may be written for an app transition $\widehat{\Sigma} \vdash t$ (e.g., as in Blackshear et al. [2013]). For clarity, the semantics is written with explicitly materialized points-to for message arguments and return values (e.g., $* \overline{x \mapsto \hat{x}}$). If such values do not otherwise constrain the abstract state, they may be summarized into the top store $\top$ without precision loss.

LEMMA 3.2 (BOUNDARY-STEP SOUNDNESS). *If $\widehat{\Sigma} \vdash_S b$ and $\langle \sigma', b \rangle \Downarrow^{\Omega} \sigma$ such that $\sigma \models_S \widehat{\Sigma}(\mathrm{post}(b))$ and $\Omega \subseteq \Omega_S$, then $\sigma' \models_S \widehat{\Sigma}(\mathrm{pre}(b))$.*

To describe an inductive program invariant, we define the *may-witness* judgment form $\widehat{\Sigma} \vdash_p^S \widehat{\sigma}$ that says, "Abstract program-state invariants $\widehat{\Sigma}$ is inductive executing backwards from abstract program state $\widehat{\sigma}$ — at location $\mathrm{loc}(\widehat{\sigma})$ — in program p." The A-INDUCTIVE rule that defines this judgment form simply checks that each boundary transition b and each app transition t in program p are inductive.

LEMMA 3.3 (INDUCTIVE SOUNDNESS). *If $\widehat{\Sigma} \vdash_p^S \widehat{\sigma}$ and $\sigma' \rightarrow_p^{*\Omega} \sigma$ such that $\sigma \models_S \widehat{\sigma}$ and $\Omega \subseteq \Omega_S$, then $\sigma' \models_S \widehat{\Sigma}(\mathrm{loc}(\sigma'))$.*

Finally, to derive a refutation of reachability $\vdash_p^S \widehat{\sigma}$ unreach, the A-REFUTE rule says that we derive an inductive program invariant $\widehat{\Sigma}$ from $\widehat{\sigma}$ — i.e., $\widehat{\Sigma} \vdash_p^S \widehat{\sigma}$, and we derive that the program invariant at the entry location `fwk` excludes the initial (concrete) program state — i.e., $\vdash_S \widehat{\Sigma}(\mathtt{fwk})$ excludesinit.

THEOREM 3.4 (REFUTE SOUNDNESS). *If $\vdash_p^S \widehat{\sigma}$ unreach and $\sigma' \rightarrow_p^{*\Omega} \sigma$ such that $\sigma \models_S \widehat{\sigma}$ and $\Omega \subseteq \Omega_S$, then $\sigma' \neq \sigma_{\mathrm{init}}$.*

3.2.2 Abstract Interpretation with Message Histories. While we have described a checking system with the may-witness judgment form $\widehat{\Sigma} \vdash_p^S \widehat{\sigma}$, we can consider a direct approach to computing an inductive program invariant $\widehat{\Sigma}$ from an error condition $\widehat{\sigma}$ via a backwards abstract interpretation. The invariant map $\widehat{\Sigma}$ is initialized with the error condition just before the assertion ($\mathtt{okhist} \cdot \top \cdot \widehat{\rho}$ where $\widehat{\rho}$ negates the assertion condition) and $\bot$ at other locations. We then proceed with a standard worklist algorithm. When the invariant map is updated at a location, all transitions to that location are added to the worklist. Each transition in the worklist and abstract state at the post-location are processed by a transfer function (based on the Hoare triples defined above) producing a pre-condition that is joined into the invariant map. Pre-conditions at a location are eagerly merged with existing disjuncts both to avoid an updated state at a location and for efficiency. Merging is done automatically via the entailment check, $\widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \vdash_S \widehat{\omega}' \cdot \widehat{\kappa}' \cdot \widehat{\rho}'$. If a new pre-condition cannot be merged with an existing disjunct, it is added to the existing disjunctions. At the framework location `fwk`, all callback return boundary transitions, $\ell \dashv[\mathtt{cbret}\, x'\ md(\overline{x})]\mapsto \mathtt{fwk}$, are added to the worklist. We alarm if we cannot prove that the invariant at `fwk` excludes initial. If a fixed point is reached that excludes the initial state, $\vdash_S \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}$ excludesinit, then we have refuted the reachability of the assertion failure. Intuitively, we have now captured the abstract state at all locations that may step to the assertion failure, and excludes initial is proving that no message history can go from the initial state of the program to the assertion and fail.

Excludes-initial, $\vdash_S \widehat{\omega}$ excludesinit, and entailment of abstract message histories, $\widehat{\omega} \vdash_S \widehat{\omega}$ are automated via SMT (and described in Section 5). Existing techniques can combine the message history SMT encoding with other parts of the abstract state (e.g., using Piskac et al. [2013] for separation logic).

4 CALLBACK CONTROL-FLOW TEMPORAL LOGIC (CBCFTL)

In this section, we describe the Callback Control-Flow Temporal Logic (CBCFTL) that we use to express the specification S of the realizable message histories. We design CBCFTL as a compromise between the expressiveness required to specify callback control flow and the need of the abstract interpretation to automate judging excludes-initial ($\vdash_S \widehat{\omega}$ excludesinit) and entailment ($\widehat{\omega} \vdash_S \widehat{\omega}'$), which are parametric in the specification language used to express S.

As we observe in Section 2, a specification of realizable message histories must be able to express: (1) quantification over message values (e.g., the subscription object `s` from History Implication 1); and (2) constraints on what messages must have or have not happened in the past (e.g., `subscribe` or `unsubscribe`). These requirements suggest CBCFTL should be a linear temporal logic (LTL) [Manna and Pnueli 1992] interpreted over finite sequences (i.e., message histories) [Giacomo and Vardi 2013], with *first-order* quantification of message arguments, and *past-time* temporal operators [Lichtenstein et al. 1985]. In principle, the excludes-inital and entailment judgment could be reduced to checking the satisfiability of first-order LTL (FO-LTL) formulas, but it is undecidable [Song and Wu 2016] and is limited in ready-to-use implementations.

Instead, we restrict the CBCFTL syntax such that reasoning about message histories leading to a target message is decidable (i.e., with history implications consisting of a target message and temporal formula). In Section 5, we show that such a problem can be in turn reduced to the satisfiability of the fragment of *temporal formulas* of CBCFTL. We show how to use such a subproblem to decide the excludes-initial ($\vdash_S \widehat{\omega}$ excludesinit) judgment and to obtain a semi-algorithm for judging entailment ($\widehat{\omega} \vdash_S \widehat{\omega}'$). The syntactic restriction of CBCFTL carefully controls the use of features of the logic, such as negation and quantifier alternation, that complicate automated reasoning. In particular, the restrictions are such that we can encode a temporal formula $\widetilde{\omega}$ of CBCFTL in an equisatisfiable formula in the Extended Effectively Propositional (Extended EPR) logic [Korovin 2013; Padon et al. 2017]. This section first gives the syntax and semantics of CBCFTL and then explains the encoding of the temporal formula fragment in Extended EPR.

4.1 A Temporal Logic for Expressing Realizable Message Histories

Figure 6 describes the CBCFTL syntax and semantics. A CBCFTL specification S is a conjunction of *history implications* $s ::= \widehat{m} \boxdot\!\!\rightarrow \widetilde{\omega}$. Each history implication targets an abstract message, $\widehat{m}$, controlled by the framework (e.g., invocation of the `call` callback) and a temporal formula, $\widetilde{\omega}$, that must hold before the framework outputs that message. While not captured in the syntax, history implications s are closed formulas where the variables of the abstract message in the antecedent are implicitly universally quantified, and we assume that the temporal formula $\widetilde{\omega}$ in the consequent is quantified so that its free variables are a subset of the variables of $\widehat{m}$ (i.e., $\mathrm{fv}(\widetilde{\omega}) \subseteq \mathrm{fv}(\widehat{m})$ where $\mathrm{fv}(\cdot)$ yields the set of free variables of a formula). Section 5 will explain how abstract message histories combine with history implications leaving only temporal formula, which motivates this design.

Temporal formulas $\widetilde{\omega}$ include restricted versions of standard past-time temporal operators (**O** $\widetilde{m}$ for **O**nce, **HN** $\widetilde{m}$ for **H**istorically **N**ot, and $\widetilde{m}_2$ **NS** $\widetilde{m}_1$ for **N**ot **S**ince), equality and disequality between variables, and positive Boolean combinations. In particular, the temporal operators apply only to individual symbolic messages $\widetilde{m}$ and do not allow for explicit negations (although some negations are implicit in the history implication $\boxdot\!\!\rightarrow$ and in the temporal operators **HN** and **NS**). Symbolic messages $\widetilde{m}$ are essentially abstract messages $\widehat{m}$, except we allow for a local existential

quantification of variables $\exists \hat{x}.\widetilde{m}$, which is convenient for "don't care" arguments (e.g., the _ in History Implication 1). The structure of CBCFTL specifications S also limits the nesting of the temporal operators: past temporal operators are always nested in a future temporal operator ($\Box\!\!\rightarrow$), and the only future temporal operator is history implication $\widehat{m} \Box\!\!\rightarrow \widetilde{\omega}$. While not explicitly shown in the syntax, we also restrict quantifiers such that $\forall \hat{x}.\widetilde{\omega}$ may only contain conjunctions and disjunctions of $\mathbf{HN}\ \widetilde{m}$ and equality/disequality of symbolic variables.

The most interesting part of temporal formulas $\widetilde{\omega}$ are the temporal operators: $\mathbf{HN}\ \widetilde{m}$ states that $\widetilde{m}$ has historically not (i.e., has never) occurred in the past, $\mathbf{O}\ \widetilde{m}$ that $\widetilde{m}$ occurred at least once in the past, and $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1$ that $\widetilde{m}_2$ has not occurred since $\widetilde{m}_1$ occurred. The operators restrict standard past-time temporal operators so that for a given message $\widetilde{m}$, we either positively look back for the time $\widetilde{m}$ occurs or negatively rule out $\widetilde{m}$ at each time in the past. Thus, the $\mathbf{O}\ \widetilde{m}$ operator is directly the **O**nce operator from past-time LTL, while $\mathbf{HN}\ \widetilde{m}$ and $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1$ are syntactic restrictions for the appropriate negations within **H**istorically and Since (i.e., $\mathbf{HN}\ \widetilde{m} \stackrel{\text{def}}{=} \mathbf{H}\ \neg\widetilde{m}$ and $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1 \stackrel{\text{def}}{=} \neg\widetilde{m}_2\ \mathbf{S}\ \widetilde{m}_1$ in past-time LTL). Additionally, we allow standard boolean combinations of operators as well as equality of variables.

A model of a specification S is a (concrete) message history ω, which is a finite sequence of messages m. Message histories ω are zero-indexed by positions $i \in [0, \text{len}(\omega))$, and we write $\omega[i]$ for the message at position i in ω and $\text{len}(\omega)$ for the length of ω. A message history satisfies a history implication $\omega \models \widehat{m} \Box\!\!\rightarrow \widetilde{\omega}$ iff for all positions i in the message history (i.e., $i \in [0, \text{len}(\omega)]$), if a concrete message m with an assignment for its variables θ models $\widehat{m}$ and is $\omega[i]$, then the prefix of ω up to $i-1$ must satisfy the temporal formula $\widetilde{\omega}$ (under the assignment θ). A model for a temporal formula $\widetilde{\omega}$ is a tuple $\omega \cdot \theta \cdot i$ of a message history, an assignment, and a position in ω. The past-time temporal operators apply to the prefix of the message history ω up to (and including) position i,

$$\text{CBCFTL specification} \quad S ::= \text{true} \mid S_1 \wedge S_2 \mid s \qquad \text{history implication} \quad s ::= \widehat{m} \Box\!\!\rightarrow \widetilde{\omega}$$

$$\text{temporal formula} \quad \widetilde{\omega} ::= \mathbf{O}\ \widetilde{m} \mid \mathbf{HN}\ \widetilde{m} \mid \widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1 \mid \exists \hat{x}.\widetilde{\omega} \mid \forall \hat{x}.\widetilde{\omega} \mid \widetilde{\omega}_1 \wedge \widetilde{\omega}_2 \mid \widetilde{\omega}_1 \vee \widetilde{\omega}_2 \mid \hat{x}_1 = \hat{x}_2 \mid \hat{x}_1 \neq \hat{x}_2$$

$$\text{symbolic messages} \quad \widetilde{m} ::= \widehat{m} \mid \exists \hat{x}.\widetilde{m}$$

$$\boxed{\omega \models S} \quad \omega \models \widehat{m} \Box\!\!\rightarrow \widetilde{\omega} \quad \text{iff} \quad m \cdot \theta \models \widehat{m} \text{ such that } \omega[i] = m \text{ implies } \omega \cdot \theta \cdot i - 1 \models \widetilde{\omega}$$

$$\boxed{\omega \cdot \theta \cdot i \models \widetilde{\omega}} \qquad \omega \cdot \theta \cdot i \models \widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1 \quad \text{iff} \quad \exists j \in [0, i].\ \omega \cdot \theta \cdot j \models \widetilde{m}_1 \text{ and } \forall k \in (j, i].\ \omega \cdot \theta \cdot k \not\models \widetilde{m}_2$$

$$\omega \cdot \theta \cdot i \models \mathbf{O}\ \widetilde{m} \quad \text{iff} \quad \exists j \in [0, i].\ \omega \cdot \theta \cdot j \models \widetilde{m} \qquad \omega \cdot \theta \cdot i \models \mathbf{HN}\ \widetilde{m} \quad \text{iff} \quad \forall k \in [0, i].\ \omega \cdot \theta \cdot k \not\models \widetilde{m}$$

$$\boxed{\omega \cdot \theta \cdot i \models \widetilde{m}}$$

$$\omega \cdot \theta \cdot i \models \widehat{m} \quad \text{iff} \quad m \cdot \theta \models \widehat{m} \text{ and } \omega[i] = m \qquad \omega \cdot \theta \cdot i \models \exists \hat{x}.\widetilde{m} \quad \text{iff} \quad \exists v.\ \omega \cdot \theta[\hat{x} \mapsto v] \cdot i \models \widetilde{m}$$

Fig. 6. Syntax and semantics of callback control-flow temporal logic (CBCFTL). CBCFTL is a subset of first-order linear temporal logic (FO-LTL) that describes a set of realizable message histories. A CBCFTL specification S is a conjunction of *history implications* $\widehat{m} \Box\!\!\rightarrow \widetilde{\omega}$. Temporal formulas $\widetilde{\omega}$ include restricted versions of standard past-time temporal operators that apply only to individual symbolic messages $\widetilde{m}$ with limited negation **O** (**O**nce), **HN** (**H**istorically **N**ot), and **NS** (**N**ot **S**ince). While not shown in the syntax here, the subformula $\widetilde{\omega}$ of universal quantification $\forall \hat{x}.\widetilde{\omega}$ is further restricted to $\mathbf{HN}\ \widetilde{m}$ or the propositional forms to limit quantifier alternation.

and the relation is undefined for any position outside the valid range of indices of the message history ω (e.g., -1).

The temporal operators captures the looking back "positively" for a message (e.g., for $\mathbf{O}\ \widetilde{m}$, there must exist a $j \in [0, i]$ where the message at j models $\widetilde{m}$ — i.e., $\omega \cdot \theta \cdot j \models \widetilde{m}$) or "negatively" when ruling out one (i.e., for $\mathbf{HN}\ \widetilde{m}$, all the messages at $k \in [0, i]$ must not model $\widetilde{m}$). The temporal operator $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1$ is a more general version combining "positively" looking for $\widetilde{m}_1$ (i.e., $\exists j \in [0, i].\ \omega \cdot \theta \cdot j \models \widetilde{m}_1$) while "negatively" ruling out $\widetilde{m}_2$ (i.e., $\forall k \in (j, i].\ \omega \cdot \theta \cdot k \not\models \widetilde{m}_2$). All three temporal formula reference the judgment $\omega \cdot i \cdot m \models \widetilde{m}$ to match a symbolic message to a position i in a message history ω under a variable assignment θ.

4.2 Encoding Temporal Formula Into Extended EPR

Here, we describe how we encode temporal formula into Extended EPR [Korovin 2013; Padon et al. 2017], a decidable fragment of first-order logic. In brief, effectively propositional (EPR) is a first-order logic fragment where closed formulas converted into prenex normal form have the quantifier prefix $\exists\forall$ without any function symbols. Extended EPR adds function symbols as long as the quantifier alternation graph does not contain cycles. The quantifier alternation graph is a directed graph where the nodes are sorts and the edges are defined by functions (or $\forall x.\exists y.\ \ldots$) from the sort of the argument to the sort of the value.

To encode a temporal formula $\widetilde{\omega}$, we model message histories ω with uninterpreted functions over uninterpreted sorts. We use an uninterpreted function hist: **HistIdx** $\rightarrow$ **Msg** from history indices **HistIdx** to message instances **Msg**. To capture message instances, we use a function $\mathsf{msgname}$: **Msg** $\rightarrow$ **MsgName** from message instances to message names **MsgName** (i.e., representing the message kind, like cb, and the method name) and a function $\mathsf{msgargs}$: **Msg** $\rightarrow$ **ArgIdx** $\rightarrow$ **Val** from messages instances to arguments indices **ArgIdx** to values **Val** (i.e., representing the arguments of the message instance).

Then to describe ordering constraints on messages in a message history, we use a set of ordering axioms (referred to as ψ_{ax}). We use an uninterpreted function $\leq$: **HistIdx** $\rightarrow$ **HistIdx** $\rightarrow$ **Bool** and axiomatize a total ordering on **HistIdx** (like Padon et al. [2017]), as well as an axiom for zero (i.e., $\forall idx \in \mathbf{HistIdx}.\ 0 \leq idx$ where 0 is a variable). Argument indices **ArgIdx** are finite and bounded to the largest arity found in the framework methods. Message names **MsgName** are also finite and bounded by the framework interface definition. As such, we precisely represent the needed ordering constraints on messages in message histories.

Given the above, the encoding of temporal formula $\widetilde{\omega}$ is now direct. We can encode an abstract message $\widehat{m}$ (i.e., an unquantified symbolic message $\widetilde{m}$) at an index $idx \in \mathbf{HistIdx}$ using the hist, $\mathsf{msgname}$, and $\mathsf{msgargs}$ functions. To be able to encode the length of a message history, we introduce a distinguished variable $\mathtt{len}$. Then, we can encode the past-time temporal operators ($\mathbf{O}\ \widetilde{m}$, $\mathbf{HN}\ \widetilde{m}$, and $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1$) using the encoding of an abstract message at an index, 0, $\leq$, and $\mathtt{len}$. Here, we leverage the restriction that the temporal operators apply only to individual symbolic messages $\widetilde{m}$. With respect to Extended EPR, it is clear that the quantifier alternation graph from the function symbols is acyclic. And then, the encoding of temporal formula $\widetilde{\omega}$ described above stays in Extended EPR because of the careful control of negation to prevent introducing any $\forall\exists$ edges.

5 COMBINING ABSTRACT MESSAGE HISTORIES WITH CALLBACK CONTROL-FLOW

MHPL from Section 3 depends on two judgments, excludes initial $\vdash_S \widehat{\omega}$ excludesinit and entailment $\widehat{\omega} \vdash_S \widehat{\omega}'$. Excludes initial says that abstract message history, $\widehat{\omega}$, excludes the initial, empty message history. Message history $\widehat{\omega}$ entailing a second message history $\widehat{\omega}'$ says that all concrete traces abstracted by $\widehat{\omega}$ are also abstracted by $\widehat{\omega}'$. Both of these judgments depend on the CBCFTL specification from Section 4 for a definition of realizable message histories. For each abstract message history,

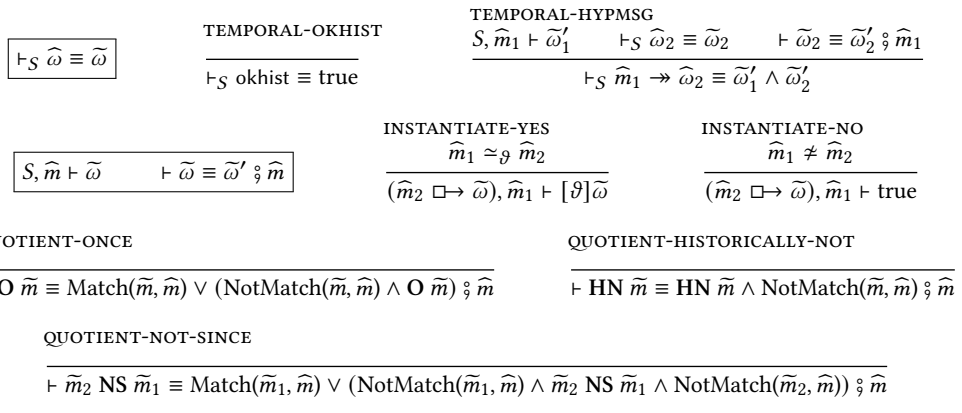


Fig. 7. Instantiating CBCFTL specifications S with abstract message histories $\widehat{\omega}$ from MHPL. The judgment form $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ says, "Under CBFTL specification S, an abstract message history $\widehat{\omega}$ is equivalent to a temporal formula $\widetilde{\omega}$." We can view this judgment as giving us an encoding into a temporal formula $\widetilde{\omega}$, the instantiation of a specification of realizable message histories S with a particular abstract message history $\widehat{\omega}$ to derive a description of the realizable message histories up to a program location.

we *combine* with the CBCFTL specification in order to avoid reasoning about the specification separately. In this section, we first show how to combine an abstract message history, $\widehat{\omega}$, with a specification, S, resulting in a single temporal formula, $\widetilde{\omega}$ (as we describe in Section 2.2.3). Second, we show how to compute excludes initial and entailment for temporal formula. Finally, we prove that defining these judgments in this way is sound.

The high-level intuition is that given an abstract message history $\widehat{\omega}$, we instantiate the specification of realizable message histories S with $\widehat{\omega}$ into a single temporal formula $\widetilde{\omega}$. Then, with this temporal formula $\widetilde{\omega}$, we can implement these judgments on abstract message histories via queries to an off-the-shelf SMT solver (using the encoding described at the end of Section 4). In Figure 7, we describe the judgment form $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ that captures this combining of $\widehat{\omega}$ and S into a single temporal formula $\widetilde{\omega}$.

As we see in Figure 7, the combining (or equivalent-to-a-temporal-formula) judgment form $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ is syntax-directed on the abstract message history $\widehat{\omega}$. Under the assumption of S, the abstract message history okhist is equivalent to the temporal formula true (rule TEMPORAL-OKHIST). For the ordered-implication abstract message history $\widehat{m} \twoheadrightarrow \widehat{\omega}$, intuitively, we want to hypothesize $\widehat{m}_1$ to derive any constraints from instantiating from S and to derive any constraints from "quotienting" the constraints from $\widehat{\omega}_2$ to "remove $\widehat{m}_1$ from the end". Instantiating and quotienting are captured by two helper judgments. The instantiate judgment form $S, \widehat{m} \vdash \widetilde{\omega}$ says, "In specification S, hypothesizing abstract message $\widehat{m}$, temporal formula $\widetilde{\omega}$ describe realizable message histories." And the quotient judgment form $\vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$ says, "Temporal formula $\widetilde{\omega}$ is equivalent to temporal formula $\widetilde{\omega}'$ with abstract message $\widehat{m}$ appended." We can now read the key TEMPORAL-HYPMSG rule: if hypothesizing $\widehat{m}_1$ in S yields temporal formula $\widetilde{\omega}'_1$, abstract message history $\widehat{\omega}_2$ is equivalent to temporal formula $\widetilde{\omega}_2$, and $\widetilde{\omega}_2$ is equivalent to temporal formula $\widetilde{\omega}'_2$ with $\widehat{m}_1$ appended, then the ordered-implication abstract message history $\widehat{m} \twoheadrightarrow \widehat{\omega}$ is equivalent to $\widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$.

Instantiation is the process of combining the hypothetical next message of an abstract message history with a history implication (e.g., Equation 1 in the running example from Section 2). For the instantiate judgment $S, \widehat{m} \vdash \widetilde{\omega}$, we show only the cases for single history implications $\widehat{m}_2 \boxdot\!\!\rightarrow \widetilde{\omega}$ where the hypothesized message $\widehat{m}_1$ either matches (INSTANTIATE-YES) or doesn't match

(INSTANTIATE-NO). The other cases for true and $S_1 \wedge S_2$ just yield true and the conjunction of the instantiations in S_1 and S_2, respectively. As $\widehat{m}_2 \boxdot\!\!\rightarrow \widetilde{\omega}$ implicitly binds the variables of $\widehat{m}_2$, we write $\widehat{m}_1 \simeq_\vartheta \widehat{m}_2$ for a matching up to a substitution ϑ from the variables of $\widehat{m}_2$ to the variables of $\widehat{m}_1$ and write $[\vartheta]\widetilde{\omega}$ for the capture-avoiding substitution with ϑ in $\widetilde{\omega}$. And we write $\widehat{m}_1 \not\simeq \widehat{m}_2$ for the case where $\widehat{m}_1$ cannot match $\widehat{m}_2$.

The quotient judgment form, shown in Figure 7, $\vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$ is syntax-directed on $\widetilde{\omega}$ to yield $\widetilde{\omega}'$. We show the quotienting judgments for the three temporal operators **O**, **HN**, and **NS**. Quotienting the other temporal formula $\widetilde{\omega}$ productions is straightforward. Quotienting once, $\mathbf{O}\ \widetilde{m}$, with the abstract message $\widehat{m}$ has two possibilities: (1) Match($\widetilde{m}, \widehat{m}$) — the abstract message is equivalent to the message in the "once" making the temporal formula equivalent to "true" and (2) NotMatch($\widetilde{m}, \widehat{m}$) — the abstract message is not equivalent to the message in the "once" leaving the temporal formula unchanged. Mirroring the quotienting of once, quotienting historically not, $\mathbf{HN}\ \widetilde{m}$, with the abstract message $\widehat{m}$ has two possibilities: (1) Match($\widetilde{m}, \widehat{m}$) — the abstract message is equivalent to the message in the "has never" making the temporal formula equivalent to "false" and (2) NotMatch($\widetilde{m}, \widehat{m}$) — the abstract message is not equivalent to the message in the "has never" leaving the temporal formula unchanged. The operator $\widetilde{m}_2\ \mathbf{NS}\ \widetilde{m}_1$ is a combination of the previous two: either the quotiented message matches the right-hand side becoming true, or it must not match the left-hand side.

We see that for quotienting with the temporal operators, we need an analogous encoding of match or doesn't match: the meta-level functions Match($\widetilde{m}, \widehat{m}$) and NotMatch($\widetilde{m}, \widehat{m}$) encode into propositional formula of a symbolic message $\widetilde{m}$ matching or not matching an abstract message $\widehat{m}$, respectively. Since the message names, *md*, and kinds are known, Match($\widetilde{m}, \widehat{m}$) and NotMatch($\widetilde{m}, \widehat{m}$) always result in equalities and disequalities of logic variables (e.g., Equation 2 in the running example from Section 2) or "false".

With the ability to combine an abstract message history $\widehat{\omega}$ from MHPL with a CBCFTL specification of realizable message histories S via the $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ judgment, algorithms for judging excludes-initial and entailment via SMT queries become clear. Let us write $\vdash \widetilde{\omega} \equiv \psi$ for the encoding of a temporal formula $\widetilde{\omega}$ into a closed first-order formula ψ and use ψ_{ax} for the axioms encoding message histories from Section 4.

We define procedures for judging excludes-initial $\vdash_S \widehat{\omega}$ excludesinit as checking for the unsatisfiability of $\psi_{\text{ax}} \wedge \psi \wedge \texttt{len} = 0$ (where $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ and $\vdash \widetilde{\omega} \equiv \psi$), and entailment $\widehat{\omega} \vdash_S \widehat{\omega}'$ as checking for the unsatisfiability of $\psi_{\text{ax}} \wedge \psi \wedge \neg\psi'$ (where $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$, $\vdash \widetilde{\omega} \equiv \psi$, $\vdash_S \widehat{\omega}' \equiv \widetilde{\omega}'$, and $\vdash \widetilde{\omega}' \equiv \psi'$). The soundness of checking these judgments relies on the correctness of the combining judgment:

THEOREM 5.1 (CORRECT COMBINING OF MHPL AND CBCFTL). *(1) If $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ such that $\omega \cdot \theta \models_S \widehat{\omega}$, then $\omega \cdot \theta \models \widetilde{\omega}$. (2) If $\vdash_S \widehat{\omega} \equiv \widetilde{\omega}$ such that $\omega \cdot \theta \models \widetilde{\omega}$ and $\omega \models S$, then $\omega \cdot \theta \models_S \widehat{\omega}$.*

Note that we assume a well-formedness condition that no abstract message is vacuous (i.e., for any abstract message $\widehat{m}$ and any assignment θ, there exists a (concrete) message m such that $m \cdot \theta \models_S \widehat{m}$). The correctness of combining relies on correct instantiation and quotienting:

LEMMA 5.2 (CORRECT INSTANTIATING OF HISTORY IMPLICATIONS). *(1) If $S, \widehat{m} \vdash \widetilde{\omega}$ such that $\omega; m \models S$ and $m \cdot \theta \models \widehat{m}$, then $\omega \cdot \theta \models \widetilde{\omega}$. (2) If $S, \widehat{m} \vdash \widetilde{\omega}$ such that $\omega \models S$ and $m \cdot \theta \models \widehat{m}$ and $\omega \cdot \theta \models \widetilde{\omega}$, then $\omega; m \models S$.*

LEMMA 5.3 (CORRECT QUOTIENTING OF TEMPORAL FORMULAS). *(1) If $\vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$ such that $\omega; m \cdot \theta \models \widetilde{\omega}$ and $m \cdot \theta \models \widehat{m}$, then $\omega \cdot \theta \models \widetilde{\omega}'$. (2) If $\vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$ such that $\omega \cdot \theta \models \widetilde{\omega}'$ and $m \cdot \theta \models \widehat{m}$, then $\omega; m \cdot \theta \models \widetilde{\omega}$.*

Proofs for these statements may be found in the extended version [Meier et al. 2023b] Appendix C.

6 EMPIRICAL EVALUATION

As we discuss in Section 2, the challenge for a program verifier is to prove the safety of assertions that depend on the callback order, while avoiding unsound models of the framework. We hypothesize that: (1) thanks to the targeted callback control flow specification, Historia can prove safe assertions while avoiding unsound results when an assertion does not hold. And (2) Historia can be applied to real event-driven programs. We validate our hypotheses with the following research questions:

RQ1: Proving Assertions: Is it possible to write a targeted CBCFTL specification for Historia and prove safe assertions, while avoiding unsound framework models?

RQ2: Generalizability to Real-World Applications: Can Historia prove assertions on real-sized, complex, and widely used Android applications?

Bug Patterns. Checking for arbitrary assertions, such as safe `null` dereference, is not interesting as most safe assertions can be proven with an intra-callback analysis. So, to find assertion locations in Android apps that require callback control flow reasoning, we identified a set of problematic API usage *patterns*. We first searched bug reports of runtime crashes for popular open source Android apps satisfying *all* the following criteria: (a) The issue had a stack trace similar the one shown in Figure 1. (b) The issue accepted a fix that relies on the callback order. (c) The crash involved callbacks or callins from a set of commonly used Android objects (`Activity`, `Fragment`, `Dialog`, `View` objects such as buttons and menus, `AsyncTask`, and `Single`/`Maybe` from RxJava). Then, we classified the crashes according to 5 patterns of interaction between callbacks and callins. (1) $\texttt{getAct}^{[3]}$ [Fietz 2018a] — the Android method `getActivity` returns `null` if called on a `Fragment` that is not in the "created" state, and the app dereference such `null` pointer (`Activity` and `Fragment` objects are in the "created" state if the `onCreate` callback has been invoked, but the `onDestroy` has not). (2) $\texttt{execute}^{[5]}$ [Fietz 2015] — the app calls `execute` twice on the same `AsyncTask` object, ending in an exception. (3) $\texttt{dismiss}^{[7]}$ [Fietz 2016] — the app calls `dismiss` on a `Dialog` constructed with an `Activity` that is currently in the "created" state, ending in an exception. (4) $\texttt{finish}^{\texttt{null}}$ [Meier 2021] — the app dereference a field in an `onClick` callback, the same field can be set to `null` in the `onPause` callback, and the app call `finish` on the enclosing `Activty` (we call "nullable" the fields that can be set to null in a callback). (5) $\texttt{subs}^{\texttt{null}}$ [Hamster 2020] — the app dereferences a nullable field in a callback executed concurrently, such as `Runnable` `run`. We name the patterns with the main message involved in the crash, followed in subscript by an "exception property", specifying when the bug would manifest with throwing an exception. Such exception properties may be a CBCFTL history implication (referenced by number and listed in the extended version [Meier et al. 2023b] Appendix D) specifying when the framework returns an exception, a `null` value, or a nullable field dereference (indicated by `null`).

Implementation. Historia implements the backward abstract interpretation with message histories of Section 3 for refuting callback reachability assertions in Android apps. Historia uses Soot [Vallée-Rai et al. 1999] for loading the compiled app and to implement the application only control flow graph construction (similar to [Ali and Lhoták 2013] but augmented with boundary transitions as discussed in Section 3.1). Historia implements the encoding of Section 5, and uses the Z3 SMT solver [de Moura and Bjørner 2008] to check the satisfiability of temporal formulas (Section 4). Historia further processes callbacks in parallel and pre-empts calls to Z3 when possible for performance. We ran our experiments using Chameleon Cloud [Keahey et al. 2020] using an AMD EPYC 7763 and 256 GB of RAM.

6.1 RQ1: Proving Event-Driven Patterns

In Table 1, we evaluate the ability of Historia and the representative state of-the art framework modeling (*no-order*, *eager*) to prove safe fixes of the bug patterns, while correctly alarming on

Table 1. The rows of this table are split into Bug and Fix benchmarks for each pattern. We first list the number of callbacks, callback returns, or callins that could be captured by a history implication in the framework model. Next, we list the number of history implications (specs) written for the benchmark (listed in the extended version [Meier et al. 2023b] Appendix D), then, we show how many of the messages are in the specification. The depth captures how many times Historia needed to step backwards through a callback (e.g. Figure 3 shows 4 steps back). Historia alarms on all the bug versions (①), and refutes reachability of the bug assertion for 4 out of the 5 bug-fixes (⊘). In the last case, Historia explored up-to 5 callbacks before timing out at 30 min (⑤). For the comparison with ideal tools using the "no-order" and "eager" modeling approaches, some results are labeled false-⊘ and false-①.

Pattern				Historia									no-order	eager
		cb, ret (n)	ci (n)	specs (n)	cb (n)	cb (%)	cbret (n)	cbret (%)	ci (n)	ci (%)	time (s)	depth (n)	res	res	res
Bug	getAct[3]	9	15	$3^{[1,4]}$	3	33	1	11	2	13	9	3	①	①	false-⊘
	execute[5]	7	6	$3^{[2,6]}$	2	29	0	0	2	33	12	3	①	①	①
	dismiss[7]	7	6	$2^{[8]}$	1	14	1	14	1	17	18	4	①	①	false-⊘
	finish$^{\texttt{null}}$	7	9	$3^{[9,10,2]}$	3	43	1	14	3	33	52	3	①	①	false-⊘
	subs$^{\texttt{null}}$	9	17	$5^{[11,12\ 13,14,15]}$	3	33	0	0	5	29	24	3	①	①	false-⊘
Fix	getAct[3]	9	16	$3^{[1,4]}$	3	33	1	11	3	19	16	3	⊘	false-①	⊘
	execute[5]	7	7	$3^{[2,6]}$	2	29	0	0	3	43	27	4	⊘	false-①	false-①
	dismiss[7]	7	6	$2^{[8]}$	1	14	1	14	1	17	79	6	⊘	false-①	⊘
	finish$^{\texttt{null}}$	7	10	$3^{[9,10,2]}$	3	43	1	14	4	40	1800	5	⑤	false-①	⊘
	subs$^{\texttt{null}}$	9	18	$5^{[11,12\ 13,14,15]}$	3	33	0	0	6	33	150	5	⊘	false-①	⊘
	total	66	73	10	22	**33**	6	**9**	21	**29**					

instances containing the bug. For each one of the 5 bug patterns, we distilled a *Bug* and a *Fix* benchmark application from the real app code mentioned in the representative bug reports (slicing the app code to remove all the components and code non-necessary to reproduce the bug). The Bug version demonstrates the usage of the framework callbacks and callins causing the crash in the original application, while the Fix version applies the fix from the bug report. A sound analysis should always alarm on the Bug version.

We manually wrote a CBCFTL specification *sufficient* to prove the assertion safe for each fix, and then we run Historia with this specification (*specs* column) on both the Bug and Fix version. We compare Historia with the main framework modeling approaches, which either do not assume any callback ordering (*no-order*), or provide an *eager* modeling of the framework. Infer [Calcagno and Distefano 2011] and Flowdroid [Arzt et al. 2014] are used as representatives for the first and second approach, respectively. Of the 5 bug patterns, only the 4th and 5th patterns are supported by Infer and none are supported by Flowdroid. We note that Flowdroid is the only open source tool we could run in the *eager* category but does not natively support these properties. Therefore, in order to compare with the no-order model, the first three exception properties were reduced to a nullable field and checked with Infer (i.e., we manually wrote code that would throw a null pointer exception just before the actual exception was thrown). For the remaining two, we added nullability annotations on the affected fields (because Infer will not alarm on a null value from a field without this annotation). For the *eager* model, we manually examine the artificial main method generated by Flowdroid. This is a main method that should behave as the original app composed with the framework. We evaluate whether any sound and precise whole-program static analysis could prove the fix while alarming on the bug with this main method.

Discussion of the Results. Historia always (and correctly) alarms (①) on all the Bug versions, while either refutes (⊘) or does not terminate before exhausting a run-time budget of 30 minutes (⑤ result in the `finish`$^{\texttt{null}}$ benchmark). For the Fix version of the `finish`$^{\texttt{null}}$ benchmark, Historia still does not alarm, but provides the partial result proving that no program execution containing less than 5 callback invocations can reach the assertion. Interestingly, such a partial proof rules out the (abstract) execution Historia found when failing to refute the assertion in the Bug version for `finish`$^{\texttt{null}}$, which visits 4 callbacks (see the *depth* column). Targeted refinement of the framework specific control-flow specifications was required in each case for Historia to avoid false alarms.

Unsurprisingly, the no-order model results in false alarms on each fixed benchmark. Additionally, for the eager model, we found that in all but one case the artificial main method generated by Flowdroid rules out the sequence of callbacks reaching the real bug. In three of these cases, a callback that has to be executed to reach the bug was missing from the call graph. In one case, the main method over-constrained the callback order. For the remaining case, the eager model did not generate code that changed state when `setEnabled(false)` was invoked, disabling a button. Therefore, no program analysis could distinguish the state where `onClick` could not occur on that button.

6.2 RQ2: Generalizability to Real-World Applications

Next, we evaluate the generalizability of Historia by analyzing a set of 47 widely used applications containing over 2 million lines of code. These apps were found and retrieved from the F-Droid repository [F-Droid 2023] by filtering for apps updated in the last 2 years and that are more than 8 years old (rejecting obfuscated or otherwise difficult to inspect apps). We answer this question by searching for the five bug patterns and attempting to verify the 1090 locations found. First, we run Historia on each location with only the exception property, and then, we sample 8 locations that could not be proven for targeted specification refinement including timeouts and alarms.

We searched for the five patterns described in RQ1 using the application-only control flow graph. For the `execute` and `dismiss` patterns, we searched for the callins in the call graph. The remaining three patterns use an intraprocedural data flow analysis to find nullable values that were dereferenced. This value comes from either `getActivity` for the first pattern or a nullable field for the remaining patterns. The `finish` pattern looks for such dereference commands in the `onClick` callback when the `finish` method is used, and the `subs` pattern looks for dereferences in common concurrency callbacks.

Results are reported in Table 2. The first column lists the individual patterns while the second column lists the locations found for each. For scale, we list the thousands of lines of code contained by the apps the patterns were found in KLOC. We then report the number and percentage of the locations that Historia alarms on, timeouts on, and is able to prove safe. As the most common unsoundness in RQ1 was missing methods from the call graph, we compare the number of application methods found in the call graph of Historia and Flowdroid. A higher number of methods indicates more code is being analyzed.

Table 3 lists 8 randomly sampled locations from distinct apps that Historia could not prove without refinement. For each sample, we recorded the time required to write the CBCFTL specification in the "spec time" column. The time to understand the callbacks being specified is not included in the recorded time, as this would be required for any modeling approach. If it took more than an hour to run Historia or if we took more than an hour to write the specification time, we record a timeout ⊗ in the result (res) column. For perspective on the modeling difficulty, we list the number of messages that could be captured by the specification (the cb,ret and ci columns under Sample), as well as the total number of specifications as history implications we wrote (under the specs column) and the number and percentage of app messages that could be matched (the cb, cbret, and ci columns under Historia).

Table 2. Verifying usages of the multi-callback patterns among 47 open-source Android apps with only the exception property. We list the results by alarms where Historia finished but could not prove the property, timeouts where Historia took over half an hour, and safe where no further specification was needed. We list the number of app methods that are contained in the call graphs of both Historia and Flowdroid. There were 9 apps that timed out with Flowdroid. Among the 38 apps that Flowdroid could finish on, it found 28k application methods as compared to 70k application methods found by Historia.

Pattern				Historia							Flowdroid
	locations (n)	apps (n)	KLOC (n*1000)	alarm (n)	alarm (%)	timeout (n)	timeout (%)	safe (n)	safe (%)	K methods (n*1000)	K methods (n*1000)
`getAct`[3]	558	24	1,655	261	47	97	17	200	36	105	22
`execute`[5]	155	31	1,669	0	0	2	1	153	99	92	18
`dismiss`[7]	291	38	1,853	43	15	208	71	40	14	102	21
`finish`$^{\texttt{null}}$	31	8	323	3	10	3	10	25	81	29	6
`subs`$^{\texttt{null}}$	55	10	1,108	1	2	7	13	47	85	16	3
total	1090	47	2,058	308	28	317	29	465	43	121	28

Table 3. We sampled 8 locations that could not be proven from Table 2 and attempted to add CBCFTL specifications to prove them safe. These are listed by app name as all samples were chosen so the apps are unique; specific locations app versions and links may be found in the extended version [Meier et al. 2023b] Appendix D. *The Connectbot benchmark here was a timeout in Table 2; we manually removed callbacks to help find the alarm and understand the bug, while the benchmarks were unmodified.

Sample				Historia									
App	Pattern	cb,ret (n)	ci (n)	specs (n)	cb (n)	cb (%)	cbret (n)	cbret (%)	ci (n)	ci (%)	time (s)	res	spec time (m)
Vanilla Music	`getAct`[3]	473	5,440									⊗	60
OpenVPN	`getAct`[3]	938	9,628	2	49	5	5	1	167	2	0	✓	5
Seafile	`getAct`[3]	2,464	19,562	2	54	2	9	0	68	0	1	✓	5
Syncthing	`getAct`[3]	709	5,573								3,600	⊗	9
Navit	`finish`$^{\texttt{null}}$	279	2,424								3,600	⊗	54
Connectbot	`dismiss`[7]	19*	248*	1	0	0	0	0	2	1	754	true-!	4
BatteryBot	`getAct`[3]	171	2,470	1	7	4	4	2	55	2	1	✓	3
Antennapod	`getAct`[3]	3,906	2,3056								3,600	⊗	35

Discussion. Before manual refinement of the specification of the framework model, our tool was able to prove 43% of the locations safe, raise alarms on 28% and times out on 29%. We note that for `execute`[5] , `finish`$^{\texttt{null}}$, and `subs`$^{\texttt{null}}$, we get few alarms as developers appear to use these patterns defensively. Of the 8 samples, we found that we were able to correctly classify 4 locations within the hour budget of specification writing time (and always in within 5 minutes) and the hour budget of Historia run time (and from a few seconds to a few minutes). Of the 4 timeouts, one was from the specification writing time, and the rest were Historia taking more than an hour.

In the 3 cases where we were able to prove locations, the specifications ignored 95% or more of the boundary transitions in the app (i.e., no abstract message can match the majority of transitions in the applications). This highlights a performance benefit to targeted-refinement — although our analysis is unbounded in the worst case, in practice the majority of the messages that boundary transitions in the app can produce do not affect the encoded meaning of the message history, and most abstract states are immediately merged via entailment. Calling back to Section 5, the

specification ignoring most messages means that most of the time the quotient judgments result in an equivalent message history. Ignoring most messages allows most abstract states to be merged. With benchmarks containing hundreds to thousands of callbacks among thousands to tens of thousands of app methods and SMT calls for each new abstract pre-state, this means that we are avoiding the exponential explosion in the typical case.

It is also noteworthy that the application-only control flow graph used by Historia captures significantly more applications methods than Flowdroid. Among the applications that we could use Flowdroid to build call graphs for, it found 28K app methods. In these same apps, Historia found 70K app methods. This seems to reflect our observation from RQ1 that it is very challenging to capture all possible callbacks while eagerly modeling the framework and thus an argument for the targeted modeling approach of Historia.

6.3 Threats to Validity

The main threat to validity of our experiments is the shifting behavior and authors understanding of the Android framework for which we used as a case study. As noted in the Introduction, manual modeling of the Android framework is extremely difficult. Even though the application-only control flow graph appears more sound from these experiments, we found that it is possible to miss callbacks without a complete list of objects the framework can instantiate with reflection. To reduce the risk of an unsound call graph, we ensured that each location in RQ1 and a sampling of locations from RQ2 cannot be proven unreachable for any state (i.e., is reachable under *some* app state) unless the location appears to actually be unreachable through manual inspection.

7 RELATED WORK

Depending on the analysis domain, precise models are often included for some components but elided for others. As described in Section 1, a common approach, particularly in industrial Android analysis tools, is to use no model at all (i.e., the most over-approximate model). Verifiers with no callback order modeling have the advantage of performance and are the easiest to maintain but have a high false alarm rate requiring heuristic filtering [Calcagno et al. 2015]. Precision is added to the callback control flow models for a range of different domains of static analysis. Awareness of the Activity lifecycle and other user interface callbacks can improve taint analysis for security [Arzt et al. 2014; Calzavara et al. 2016; Gordon et al. 2015]. Other program analysis tools will use the `Activity` lifecycle in addition to precise models of other user interface components for verifying user interface properties [Perez and Le 2021; Yang et al. 2018, 2015]. Framework precision with respect to objects used for concurrency such as `AsyncTask` and thread pools is often captured for race detection [Hu and Neamtiu 2018; Wu et al. 2019; Yang et al. 2018, 2015] and other tools that detect concurrency issues [Pan et al. 2020]. For our experiments, we added precision for some callbacks from the `Activity` lifecycle, other user interface components, and objects for concurrency such as `AsyncTask`. The benefit of our compositional modeling approach is that components may be added on an as-needed basis as opposed to eagerly modeling a large portion of the framework.

Building the model of the framework directly into the program semantics used for the analysis has the advantage that the subsequent abstraction may be precisely chosen based on the modeled behavior. Many of the tools that build the model into the analysis capture UI elements, inter-component communication, and the stack like behavior of windows [Calzavara et al. 2016; Payet and Spoto 2014; Rountev and Yan 2014; Yang et al. 2018]. The drawback to building the model directly into the analysis is that adding or updating behaviors [Huang et al. 2018] requires modifying the analysis itself. The most common approach to model callback orders is by generating an artificial `main` method [Arzt and Bodden 2016; Arzt et al. 2014; Gordon et al. 2015; Hu and Neamtiu 2018; Pan et al. 2019]. An artificial main method has the advantage that modeling can be decoupled

from the program analysis by generating code that enforces a callback order to link with the application. When analyzing with such a main method, the normal abstraction used by the program analysis captures the callback control flow (e.g., through context sensitivity). The generation of main methods that can be abstracted precisely is a challenge. Capturing behavior such as arbitrary interleaving between callbacks (e.g., multiple simultaneous activities) can be difficult while avoiding language features that cause imprecision in analysis such as dynamic dispatch. We note that race detectors often combine some aspects of hard coding the callback control flow into the program semantics with utilizing an artificial main method (often for call graph construction). Automata and graph based approaches to modeling and abstraction [Blackshear et al. 2015a; Perez and Le 2021] are compositional and only rely on knowledge of relative order between callbacks. A difficulty with any modeling approach that eagerly models components is that the more components are modeled, the more likely the model is unsound. Unsoundness is common among any approach we listed that captures some callback order [Cao et al. 2015; Meier et al. 2019; Wang et al. 2016]. Our approach can lessen the risk here by enabling a targeted approach to callback control-flow modeling to avoid modeling more than necessary.

8 CONCLUSION

We have described a novel middle way for refuting callback reachability that enables a decoupling of the specification of callback control flow from the abstract interpretation to compute program invariants over an application-only transition system. This decoupling offers the appealing capability to gradually refine the possible callback control flow as needed and in a targeted manner to prove an assertion of interest, and it thus moves us past the false dichotomy of either using no modeling or eagerly modeling all callback control-flow constraints. The key innovation of our approach is an internalization of message histories into the analysis abstraction as a hypothetical (i.e., an ordered linear implication) to capture message histories up to a program location constrained by future messages and parametrized by a separate specification of realizable message histories. We then define a specification logic for callback control flow (CBCFTL) that carefully specializes past-time linear temporal logic so that we can utilize message-history program logic (MHPL) assertions together with CBCFTL specifications. Our evaluation provides evidence with a proof-of-concept implementation that our approach can refute callback reachability in challenging examples drawn from real-world issues among open-source apps.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive reviews, suggestions, and guidance throughout the revision process. We also cannot thank enough the members of the University of Colorado Programming Languages and Verification Group (CUPLV) for the helpful comments and support through the course of this work. Specific thanks to Chi Huynh for helping with paper formatting and running experiments, as well as Benno Stein for thoughtful discussions on goal-directed verification. This research was supported in part by the National Science Foundation under grants CCF-1619282, CCF-2008369, and AID/CIEDS project FARO.

DATA-AVAILABILITY STATEMENT

The full implementation and data used for this evaluation are available as an artifact on Zenodo [Meier et al. 2023a]. While the full experiments are resource-intensive computationally, a subset of the experiments may be run on an x86 Linux machine with 8GB of memory allocated to a Docker container.

REFERENCES

Karim Ali and Ondrej Lhoták. 2012. Application-Only Call Graph Construction. In *European Conference on Object-Oriented Programming (ECOOP)*, Vol. 7313. https://doi.org/10.1007/978-3-642-31057-7_30

Karim Ali and Ondrej Lhoták. 2013. Averroes: Whole-Program Analysis without the Whole Program. In *European Conference on Object-Oriented Programming (ECOOP)*, Vol. 7920. https://doi.org/10.1007/978-3-642-39038-8_16

Android Developers. 2022a. The Activity Lifecycle. https://developer.android.com/guide/components/activities/activity-lifecycle.

Android Developers. 2022b. Class Index (API level 32). https://developer.android.com/reference/classes.

Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the Android framework. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2884781.2884816

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

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: precise refutations for heap reachability. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2491956.2462186

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

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

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_33

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-319-17524-9_1

Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 2016. HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *European Symposium on Security and Privacy (EuroS&P)*. https://doi.org/10.1109/EuroSP.2016.16

Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Network and Distributed System Security (NDSS)*.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 4963. https://doi.org/10.1007/978-3-540-78800-3_24

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019). https://doi.org/10.1145/3338112

F-Droid. 2023. F-Droid - Free and Open Source Android App repository. https://www.f-droid.org. Accessed: 2023-01-30.

Martin Fietz. 2015. Feed Remover: Don't let the user cancel the ProgressDialog by mfietz - Pull Request #1306 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/pull/1306/files.

Martin Fietz. 2016. Fix dismiss IllegalArgumentException. https://github.com/AntennaPod/AntennaPod/issues/2148.

Martin Fietz. 2018a. 2855 Cancel UI media info update when fragment is destroyed by mfietz - Pull Request #2856 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/pull/2856/files.

Martin Fietz. 2018b. IllegalStateException after ExternalPlayerFragment was destroyed - Issue #2855 - AntennaPod/AntennaPod. https://github.com/AntennaPod/AntennaPod/issues/2855.

Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. 2009. *SCanDroid: Automated Security Certification of Android Applications*. Technical Report CS-TR-4991. University of Maryland, College Park.

Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. 2015. Information-Flow Analysis of Android Applications in DroidSafe. In *Network and Distributed System Security (NDSS)*.

Byte Hamster. 2020. Dispose loaders before setting controller to null. https://github.com/AntennaPod/AntennaPod/pull/4325.

Yongjian Hu and Iulian Neamtiu. 2018. Static Detection of Event-based Races in Android Apps. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3173162.3173173

Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2018. Understanding and detecting callback compatibility issues for Android applications. In *Automated Software Engineering (ASE)*. https://doi.org/10.1145/3238147.3238181

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/360204.375719

Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *USENIX*.

Konstantin Korovin. 2013. Non-cyclic Sorts for First-Order Satisfiability. In *Frontiers of Combining Systems (FroCoS)*, Vol. 8152. https://doi.org/10.1007/978-3-642-40885-4_15

Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 2013. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS)*. https://doi.org/10.1145/2516760.2516769

Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. 1985. The Glory of the Past. In *Logics of Programs*. https://doi.org/10.1007/3-540-15648-8_16

Zohar Manna and Amir Pnueli. 1992. *The temporal logic of reactive and concurrent systems - specification*. https://doi.org/10.1007/978-1-4612-0931-7

Mariana Trench. 2022. Mariana Trench. https://mariana-tren.ch/.

Shawn Meier. 2021. Fix null pointer exception when exiting terminal. https://github.com/connectbot/connectbot/pull/1016.

Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. 2019. Lifestate: Event-Driven Protocols and Callback Control Flow. In *European Conference on Object-Oriented Programming (ECOOP)*, Vol. 134. https://doi.org/10.4230/LIPIcs.ECOOP.2019.1

Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. 2023a. Historia: Refuting Callback Reachability with Message-History Logics (Artifact). https://doi.org/10.5281/zenodo.8331516.

Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. 2023b. Historia: Refuting Callback Reachability with Message-History Logics (Extended Version). https://arxiv.org/abs/2309.04464. (2023).

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *Proc. ACM Program. Lang.* 1, OOPSLA (2017). https://doi.org/10.1145/3140568

Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. 2020. Static asynchronous component misuse detection for Android applications. In *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/3368089.3409699

Linjie Pan, Baoquan Cui, Jiwei Yan, Xutong Ma, Jun Yan, and Jian Zhang. 2019. Androlic: an extensible flow, context, object, field, and path-sensitive static analysis framework for Android. In *Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3293882.3339001

Étienne Payet and Fausto Spoto. 2014. An operational semantics for Android activities. In *Partial evaluation and program manipulation, (PEPM)*. https://doi.org/10.1145/2543728.2543748

Danilo Dominguez Perez and Wei Le. 2021. Specifying Callback Control Flow of Mobile Apps Using Finite Automata. *IEEE Trans. Software Eng.* 47, 2 (2021). https://doi.org/10.1109/TSE.2019.2893207

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-39799-8_54

Jeff Polakow and Frank Pfenning. 1999a. Natural Deduction for Intuitionistic Non-communicative Linear Logic. In *Typed Lambda Calculi and Applications (TLCA)*, Vol. 1581. https://doi.org/10.1007/3-540-48959-2_21

Jeff Polakow and Frank Pfenning. 1999b. Relating Natural Deduction and Sequent Calculus for Intuitionistic Non-Commutative Linear Logic. In *Mathematical Foundations of Progamming Semantics (MFPS)*, Vol. 20. https://doi.org/10.1016/S1571-0661(04)80088-4

Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (1998).

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. https://doi.org/10.1109/LICS.2002.1029817

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

Fu Song and Zhilin Wu. 2016. On temporal logics with data variable quantifications: Decidability and complexity. *Inf. Comput.* 251 (2016). https://doi.org/10.1016/j.ic.2016.08.002

Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. 1999. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*.

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

Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *Conference on Software Testing, Validation and Verification, ICST*. https://doi.org/10.1109/ICST.2019.00035

Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Autom. Softw. Eng.* 25, 4 (2018). https://doi.org/10.1007/s10515-018-0237-6

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

Received 2023-04-14; accepted 2023-08-27