Message History Logics and Callback Control Flow Models for Automatic Event-Driven Application Analysis

by

Shawn Meier

B.A., Colorado School of Mines, 2008

M.S., University of Colorado, Boulder, 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2025

Committee Members:

Bor-Yuh Evan Chang, Chair

Gowtham Kaki

Sergio Mover

Fabio Somenzi

Ashutosh Trivedi

Meier, Shawn (Ph.D., Computer Science)

Message History Logics and Callback Control Flow Models for Automatic Event-Driven Application Analysis

Thesis directed by Prof. Bor-Yuh Evan Chang

In this dissertation, we address the problem of program analysis of event-driven applications to automatically prove safety properties. Event-driven applications refer to software that responds to external events such as user interface interactions, sensors, and more. Such applications are typically written against a common event-driven framework responsible for interfacing with the hardware and operating system. These event-driven frameworks offer an impressive array of functionality expediting the process of developing software, improving the security and privacy of the end user, and more. Due to the size and complexity of such event-driven frameworks as well as the hardware and operating system, it is often difficult for a developer to understand what the application and framework may do at runtime. This added complexity can easily result in unwanted application behavior known as defects or crashes.

Automatic program analysis is a broad range of techniques designed to assist developers both with proving the absence of defects or finding and explaining defects. However, such program analysis techniques have the same difficulties understanding the framework as a developer with event-driven applications. Due to the size, complexity and language features, the framework cannot be directly analyzed. Therefore, a program analysis needs a model of the event-driven framework behavior to be provided. Prior work has focused on manually modeling the behavior of such event-driven frameworks so that existing whole-program analysis techniques may be used. If these framework models are imprecise, the analysis will flag defects that do not exist. Similarly, an unsound model will cause an analysis to miss real defects.

The overall contribution of this dissertation centers around minimizing the effort required to model event-driven frameworks while maximizing the ability of event-driven program analysis to distinguish defects from correct code. Central to our contribution is that the interactions between the application and event-driven framework is the most effective location to describe framework behavior and abstract the exe-

cutions possible in an application. We call the interactions between the application and the framework the **message history**. Such message histories can be used to define the order of events that are possible in an event-driven framework and to effectively abstract application behavior.

In this dissertation, we first contribute a method of framework modeling to soundly capture the behaviors needed for sound and precise verification. These models are **targeted** meaning they capture only as much of the framework behavior as is needed to prove the safety property of interest. Any property not relevant to safety may be left unspecified resulting in a sound over-approximation of the framework and a syntactically simpler model. Next, we present a method of application-only program analysis that may prove a safety property using a framework model and application. Such an analysis may start with a sound over-approximation of all possible framework behavior and refine the model as needed. To support these framework models, we present two separate methods of validating framework models, with recorded message histories or simply by proving reachability of known defects or other program points. Finally, we can combine all of these techniques to automatically generate framework models through program synthesis. Given a large enough sample size of observed crashes or other logged behavior, we can automatically exclude unsound models. Then, through a process of carefully pruning the space of possible models, we can automatically select a sound model precise enough to prove a safety property. We evaluate each of these techniques on a corpus of defects found in real open-source Android applications demonstrating the practicality of our techniques.

Acknowledgements

I am deeply indebted to all those who have supported me throughout the journey of writing this dissertation. This work was possible because the support of my family, friends, and colleagues at every step of the way.

To my advisor, Evan Chang, whose unwavering guidance, insightful feedback, and steadfast support have been pivotal in navigating the challenges of this research. Your mentorship has been instrumental in my academic and personal growth. To my close collaborator, Sergio Mover, for his invaluable contributions, collaborative spirit, and the many stimulating discussions that have significantly enriched this work. Your partnership has been essential to my progress. Additionally, I extend my gratitude to Gowtham Kaki who helped immensely to get this work across the finish line.

Mentorship from the many members of the CU programming languages and verification group (CU-PLV) has been invaluable. Thanks to Devin Coughlin who took the time to make me feel welcome and help me get started with research. And thanks to Ashutosh Trivedi who's kind encouragment kept me moving forward. A special thanks to those I worked along side including Benno Stein, Nicholas Lewchenko, Christian Fontenot, Tianhan Lu, Souradeep Dutta, Aleks Chakarov, Taisa Kushner, Mateo Perez, Vishnu Murali, Sriram Sankaranarayanan, and more - for creating a supportive and dynamic research environment. Your technical assistance, encouragement, and camaraderie have been a source of strength and inspiration. Additionally, thanks to my friend Brian Cairns whose long discussions kept my goals grounded in problems faced by practicing software developers.

Lastly, to my family, your support has been the bedrock upon which this accomplishment stands. Thanks to my wonderful wife Chi who has spent many hours proofreading and arranging figures for my papers and dissertation. And to my mother, whose unwavering belief in my success has been a constant source of motivation.

Thank you to everyone who has contributed to my journey.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Event-driven programming [82, 90, 65] is a widely used programming model where an application repeatedly reacts to events from the environment or operating system. Events may contain a wide variety of information, for example: devices with user interfaces use events to display information and receive user input, sensors connected to computers report physical properties of the environment as events, or asynchronous tasks resulting in events for updates and completion. Such interaction with the external world makes programs extremely versatile but comes at the cost of added complexity. This added complexity makes testing and developing such software extremely difficult, often resulting in software defects. Software defects may come in the form of crashes, security vulnerabilities, slow performances, and more. This dissertation addresses the problem of automatically finding or proving the absence of such defects when software must interact with events.

In this programming model, applications are implemented as a set of callbacks, or methods that are invoked as external events occur. In event driven applications, the control flow between callbacks are not determined exclusively by the program code; instead, it is significantly shaped by the sequence and occurrence of external events. An event-driven framework is responsible for interfacing with the hardware and operating system to invoke these callbacks. Callback control flow is difficult for classical program verifiers because the event-driven framework cannot be analyzed directly. Event-driven frameworks use language features that are known to be difficult for program analysis, interact heavily with the operating system, and may depend on the hardware running an application. Therefore, the primary challenge with precise and automatic program analysis in event-driven applications is determining how control flows between callbacks.

Android is a useful example of event-driven programming since it is extremely common with over 3 billion active users [21] and 24,000 distinct Android devices (mostly phones) [53]. This dissertation explains the challenges and how effective program analysis may be applied using a crash and fix from the Antennapod Android application. The techniques we present are applicable to a wide variety of components common to event-driven frameworks such as background tasks, network, and more. This particular example focuses on the user interface component of the Android framework.

```
java.lang.IllegalStateException: Cannot execute task: the task is already running.
  at android.os.AsyncTask.execute(AsyncTask.java:576)
  at de.danoeh. ... .ConfirmationDialog$1.onClick(ConfirmationDialog.java:56)
  at com.android. ... .ButtonHandler.handleMessage(AlertController.java:167)
  ...
```

Figure 1.1: A crash and stack trace reported to the issue tracker of the Antennapod application [41]. The stack trace only lists a single event from the application – the `onClick` callback. The call site for `onClick` in the method `handleMessage` is not even defined in the development environment and is only linked with the application when it is installed on an Android phone. How can we capture when the `onClick` callback may occur and whether a potential fix prevents this issue?

The stack trace for the Antennapod application (Figure 1.1) gives a human-readable explanation of the methods involved in the crash. In particular, this crash is caused by `AsyncTask`, an Android utility for encapsulating long-running tasks. In order to avoid concurrent access to the memory used by these tasks, an `AsyncTask` instance may only be started without error `execute` once. For our explanation, `AsyncTask` may simply be thought of as an arbitrary object that can only be used once (i.e. a linear resource). Next, the stack trace lists the methods that were invoked at the time of the crash starting with `execute` at the top. The `execute` callin is called when the user clicked a button on a `ConfirmationDialog` invoking an `onClick` callback. Since the `onClick` callback is the only place in the Antennapod application that can start this background task, the crash must have occurred because `onClick` happened twice. Typically, a dialog disappears when a button is clicked, so why is this button clickable a second time? Then, once the developer has a possible fix, how can we determine whether this fix is correct?

It is common for teams developing software to test their programs by running on example inputs or have dedicated engineers to use the software, looking for such problems. However, such testing is expensive and cannot guarantee the software to be secure and free of crashes. Therefore, much research has gone into program analysis techniques that explore the control flow of a program systematically and represent what may happen at runtime. Abstract interpretation [29], model checking [28, 107], and Hoare logic [59] serve as examples. These techniques work by compactly representing sets of program memories and behaviors. The application may then be proven safe if these compact representations exclude defective states. Similar program analysis techniques [60, 131, 74] use such compact representations to efficiently discover defects. The relationship between these techniques is captured by Incorrectness Logic [93].

Despite the effectiveness of these program analysis techniques in representing vast spaces of program behavior and identifying defects efficiently, significant challenges are encountered with event-driven applications. The stack trace in Figure 1.1 shows that a framework class called `ButtonHandler` is responsible for invoking `onClick`. However, `ButtonHandler` is not included in the application or the development environment used to build such applications. `ButtonHandler` is part of the complex underlying software included with a cell phone. This class is only available after the application has been installed on the device. As a result, the developer of any program analysis technique needs to take an "educated guess" at how control flow leads to `onClick` from prior callbacks. Therefore, the developer of a program analysis is faced with the same dilemma as the application developer: under what conditions may a `onClick` callback be invoked twice? We will generally refer to the possible callbacks and their order as **callback control flow**.

Figure 1.2 illustrates more detail about how the Android framework and the fixed version of Antennapod interact and why this problem is so challenging. The left-hand side represents the framework and is currently displaying the `ConfirmationDialog`, asking if the user really wants to remove a podcast feed. On the right-hand side is the Antennapod application, which defines a callback for the "OK" button and a callback for the "Cancel" button using `makeButton`. These actions are represented by "*Remove_Action*" and "*Dismiss_Action*" respectively. When the user presses the "OK" button, the framework invokes `onClick` associated with the *Remove_Action*. The *Remove_Action* first causes the task to be executed, then the dialog is dismissed with `finish`, and the button is disabled with `setEnabled`. In the buggy version of the application, a second `onClick` of the *Remove_Action* occurs crashing the application. Intuitively, it seems like `finish` should prevent the second `onClick`, but the framework does not behave that way. Unlike `finish`, disabling the button with *OK_Button*.`setEnabled(false)` actually prevents the crash.

Unfortunately, the documentation does not fully specify the behavior of our example and many other behaviors. To make matters worse, there can be an unbounded number of actions, buttons and other "entities" that affect the callbacks. In order to determine if this application is safe, we need to reason about how the *OK_Button* relates to the *Remove_Action* to determine which action is prevented by `setEnabled(false)`. This interaction highlights the three key challenges of distinguishing a crash and a fix: (1) How can we accurately represent callback control flow with a framework model? (2) How can we distinguish bugs from correct application code using a framework model? That is, how can we perform a callback order aware, application-only analysis? (3) How can we make the process of constructing a framework model easier and less error-prone?

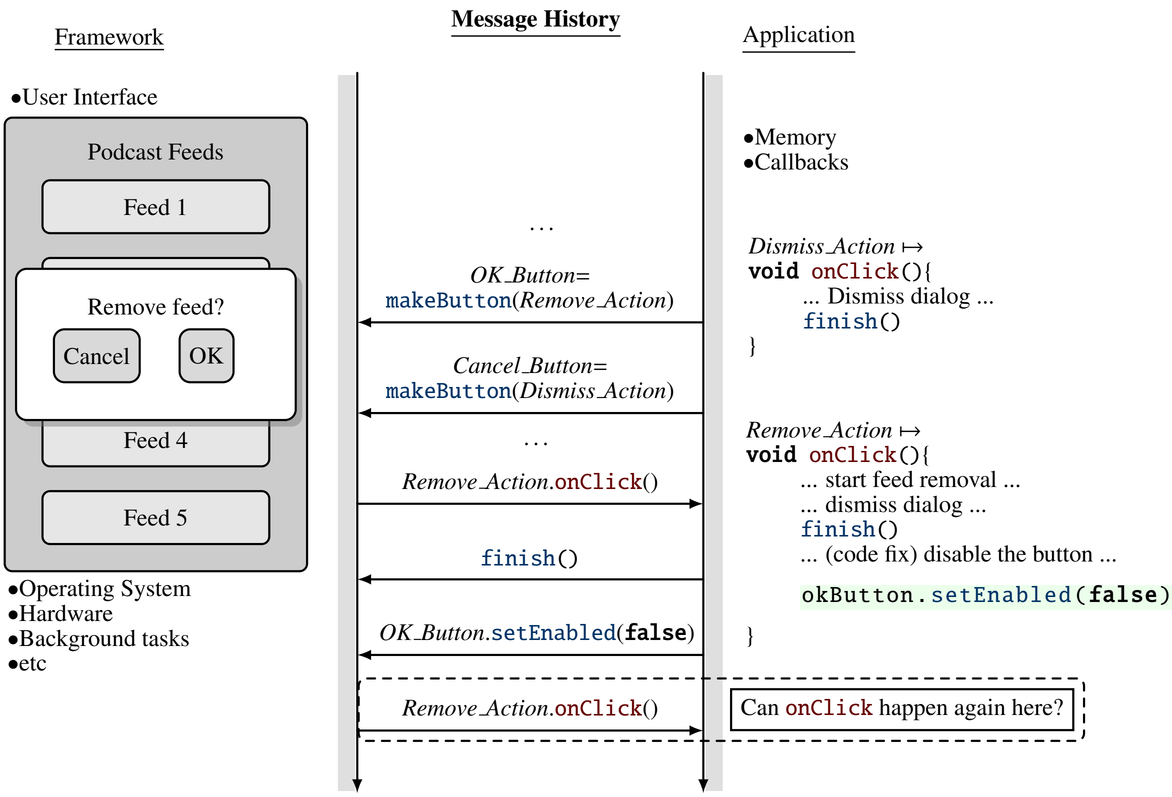

Figure 1.2: A sketch showing how the Antennapod application interacts with the event-driven Android framework. On the left-hand side, is the Android framework displaying the user interface confirming the user's intent to remove a podcast feed. The right-hand side represents a portion of the Antennapod application that defines two callbacks for the "OK" and "Cancel" buttons. In the center, is the message history capturing the interaction between the two. First, the application uses `makeButton` to display both the "OK" and "Cancel" buttons. This action also associates the *Remove_Action* with the "OK" button and *Dismiss_Action* with the "Cancel" button. When the user taps the "OK" button, the framework invokes the `onClick` with the *Remove_Action*. Which starts the removal of the feed and dismissing the dialog with finish. The crash occurs if the user taps the button a second time, quickly invoking another `onClick` with the *Remove_Action*. The developer fixes this application by adding `okButton.setEnabled(false)` to disable the button. But how do we know that this fix is correct? The behavior of `finish` and `setEnabled` are hidden in the framework and the *OK_Button* identifier used by `setEnabled` is not shared with the offending `onClick`.

The Difficulty of Representing the Callback Behavior of the Framework

Primarily, modeling the framework is difficult simply because it is huge. The `onClick` callback is just one of the more than 36,000 possible callbacks an Android application might utilize. There are hundreds of thousands of methods like `finish` and `setEnabled(false)` that can affect callback order. The framework is also continuously changing as new functionality is added. These changes mean that maintaining a framework model is extremely difficult.

One approach to representing the callback behavior is to assume 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 [1]. This approach effectively assumes that any callback may happen at any time with any arguments. 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 invoke at any time and with any arguments) and thus is the approach generally used by industrial-scale analyzers for Android apps (e.g., [47, 79, 86, 38]). However, such techniques often result in false alarms, caused by the analysis considering callback order that cannot happen at runtime. Such false alarms significantly affect the utility of any program verifier when used for real software development [38].

In order to reduce callback order relating to false alarms, many program analysis techniques for Android attempts to eagerly generate code that may be substituted for the framework. For example, the Activity Lifecycle [4] and buttons are modeled by detecting relationships between objects in the call graph and XML files using heuristics, then hard coding control flow [11, 18, 137]. This approach also has some significant limitations. It is not feasible to maintain such heuristics for over 36,000 callbacks in the Android framework. Such heuristics become especially hard when callback control flow involves relationships between multiple identifiers, like the buttons and their corresponding actions (e.g. imagine if an action is connected with multiple buttons). In practice, most analysis frameworks for Android do not accurately model the order callbacks may occur [133]. Additionally, it is extremely easy to miss components resulting in unreachable application code [25]. Both of these problems lead to unsound framework models that restricts the framework behavior too much, resulting in false negatives (i.e. true bugs that are not reported). Chapter 3 dives deeper into why such modeling approaches are problematic.

Ideally, one will have a completely sound and precise framework model that behaves exactly as the real framework. However, a completely precise model is impractical to build and difficult to analyze. Therefore, the central challenge is to build a model that is precise enough to prove safety in applications while being simple enough that the model is reasonable to build. An additional benefit to simpler models is that they improve the performance of the program analysis.

The Difficulty of Callback Order Aware, Application-Only Analysis

Representing the callback control flow is the primary challenge when analyzing an application without the framework implementation. Starting from a control flow graph, where all callbacks can happen at any time, we need to only consider control flow paths that are realizable under the framework. From the message history discussed earlier in Figure 1.2, it can be seen that restricting realizable message histories is at least as hard as context free reachability [111]. For a given button, each `onClick` invocation must be paired with a corresponding `makeButton`. In addition to the context free language of registering and receiving callbacks, identifiers for buttons, listeners, and more must be represented by the program analysis.

The next challenge is that not all application memories are possible under every sequence of callbacks. Since the application is designed to react to external events, the memory state will depend on what callbacks have occurred. For example, the Antennapod application memory will capture the effects of *Remove_Action* upon the user pressing "OK". Therefore, a program analysis needs to represent the state of the application memory with respect to what events have happened (i.e. we need a relational abstract domain between message histories and memory).

1.1 Thesis Statement

As hinted in the introduction with Figure 1.2, reasoning about the interaction between the application and the framework is the key to precise analysis of event-driven applications. Both callback and call-in invocations across this interface are referred to as **messages**, as they represent continuous communication between the application and the framework. The set of possible messages from the framework across this interface at any given time are determined by the **message history**. Message histories allow us to address our two key challenges from earlier: (1) The behavior of the framework can be represented as a **targeted** framework model, using the message history to determine when important callbacks may occur. For example, a model may state that if an `onClick` callback is invoked then the application must have shown the button and registered a listener with `makeButton` and not be in the disabled state as indicated by `setEnabled`. (2) Bugs may be distinguished from correct code by performing a message-history aware application-only program analysis. This analysis represents the callback-control flow precisely and soundly capturing application behavior. (3) The process of constructing framework models can be improved by first validating models against observed behavior to avoid unsoundness, and second, by learning a sufficient model based on the property to prove.

A framework model can be said to be **targeted** if it restricts only the framework behaviors that would lead a program verifier to reach a false alarm. For example, the relationship between `setEnabled(false)` to `onClick` is an important property to target. However, there are other callback order constraints that are less likely to be useful when proving safety. For example, a `Button` may only be used after the parent user interface has been created. Such a property is always true, but rarely contributes to a proof. Since the callback that creates the user interface usually invokes `makeButton`, the extra behavior of a button click happening earlier is excluded anyway. Similarly, there are a wide variety of other behaviors from the framework that, if possible, would not cause a crash in the application. The key insight of targeted modeling is that behaviors should only be restricted as needed. We find that applications typically only rely on an extremely small number of the possible sound constraints that could be placed on an event-driven framework.

For the second major challenge of analyzing event-driven applications precisely, this dissertation addresses the problem of **abstracting callback control** flow using message histories. Similar to how other program analysis compactly represents possible memory states of a program using a memory abstraction, we present a method of abstracting the history of messages to determine when paths through the callbacks are possible. This abstraction also relates the program memory reached under a given message history. The unbounded alphabet of messages between the application and framework is a key challenge in precisely representing a message history in a program analysis. These messages are unbounded due to the identifiers for buttons listeners and other entities involved in the runtime execution of the application. We show how to automatically determine properties of abstract message histories, such as emptiness or excludes initial.

As will be shown in chapter 4 and chapter 5, this approach to precisely abstracting message histories for verifying event-driven applications can distinguish defects and fixes that no other technique can address.

Due to the extremely large size of event-driven frameworks (over 36,000 possible callbacks in Android alone), we argue that framework models should be automatically generated from runtime data and the requirements of safety properties. This generation is a form of **abductive reasoning** since we are looking for an explanation that fits the observed runtime behavior and is sufficiently strong to prove the code correct. That is, any framework model that we prove safe should include behaviors reaching the crash in the buggy version of the application. A key part of abducing any hypothesis from data is to keep the hypothesis simple. Overly complex hypotheses can lead to over-fitting of the data. For this reason, targeting of the framework models is essential. Of the 36,000+ callbacks, we find that the vast majority can be ignored while modeling the framework. In fact, we find in our evaluation that most safety properties may be proven with around 3 callbacks restricted by the framework model (discussed in detail with section 4.4).

Therefore, this dissertation argues the following thesis statement:

> **Targeted** representations of **message histories** are essential to automatically **abduce** framework behavior for distinguishing defects from correct code in event-driven applications.

1.2 Contributions and Organization of Dissertation

The rest of this dissertation is organized as follows: First, we present an overview in chapter 2, with examples of each of the steps towards our goal of effective event-driven application analysis. Next, we present our contributions as individual chapters:

(1) Chapter 3 presents a method of reasoning about framework models based on the interface between the application and the framework. In this chapter, we examine whether it is possible to write framework models that are precise enough to exclude behavior that is not possible under a framework while soundly over-approximating real framework behavior. Since this work was performed before it was clear how such framework models may be used for static analysis, we develop a technique of predictive trace verification that explores if reordered events may reach a defect. In this chapter, we

demonstrate that precise models may be written while maintaining a high level of soundness. We evaluate the soundness of both our model and related models by recording the interactions between the application and framework for 1679 executions of 126 distinct Android applications.

(2) Chapter 4 presents a method for analyzing an event-driven application in isolation from the framework while abstracting message histories. We present this analysis by first defining an application-only transition system giving semantics to an application under an arbitrary framework model that may invoke any callback at any time. Next, we present a message-history program logic that abstracts message histories backwards from a suspected defect in order to prove that the defect may not be reached. This program logic is then extended with a domain specific framework modeling language called Callback Control Flow Temporal Logic (CBCFTL for short) that targets the properties of interest for proving the absense of a defect. CBCFTL formula may be automatically combined with abstract message histories and interpreted as SMT formula. We then implement these techniques in a tool named Historia and evaluate the effectiveness on real-world open source Android applications.

(3) Chapter 5 details a goal-directed approach to finding and proving that a message history reaches a location or defect under a given framework model. This is useful for both the development cycle, where a developer would like an explanation of the callbacks reaching a defect, and also for validating framework models. The earlier mentioned approach to validating framework models of recording message histories at runtime has significant drawbacks. Such instrumentation is invasive and can only run on a local device missing the behavior of the 23,999 other devices that could run the application. In contrast, a reachable location can be obtained from a variety of sources such as a stack trace.

(4) Chapter 6 finally puts the pieces together and takes a location of interest and abduces a framework model sufficiently strong to prove correctness, yet validates on a corpus of known reachable locations. The key to this approach is using the CBCFTL domain specific language to constrain the hypothesis space of framework models. This hypothesis space may further be constrained by the

observation that effects between messages are almost always connected (i.e. they share identifiers for buttons, listeners, etc). With this connectedness observation, we may then efficiently search the hypothesis space of framework models by following a specialized pointer analysis determining which messages may alias each other's arguments known as a message graph.

Chapter 2

Overview: Proving Properties Event-Driven Applications Using Message History Abstractions

This dissertation uses the Antennapod defect and the corresponding fix because it is reasonably simple and clearly depends on the order of the `onClick` callback. However, this defect illustrates many of the challenges associated with event-driven frameworks including the need for sound framework models and precise alias analysis. Our goal with this dissertation, is to be able to automatically distinguish between this kind of multi-callback defect and their associated fixes.

This overview section first describes the defective code and the accepted fix. Then, since it is difficult to talk about an analysis without being able to represent concrete executions of an application, we introduce a principled method of describing executions of the application and framework without relying on a framework implementation in section 2.1. With this new view of executions in mind, section 2.2 presents a domain specific language and technique for modeling the framework behavior precisely enough to be useful while not excluding possible behaviors. Putting these two pieces together, section 2.3 shows how we may prove properties in applications, such as the fix for the Antennapod crash using the framework model. With such a program analysis, we find that models are still hard to write due to the size and difficult behavior of event-driven frameworks such as Android. Therefore, section 2.4 shows how we may use the existence of known bugs, such as this Antennapod crash, to ensure that the model does not prevent an analysis from alarming on a potential defect involving similar callbacks. Finally, section 2.5 describes how we may use a target location as well as known defects (or other reachable locations), to automatically generate framework models using framework model synthesis.

As discussed in the introduction, an application is implemented as a set of callbacks that respond to external events. Figure 2.1 shows a simplified version of the code that caused the crash shown in Figure 1.1. Line 17 (highlighted in green) was added later to fix the crash. There are 3 callbacks, an `onCreate`, invoked when the dialog is started, and two `onClick` callbacks, corresponding to the removal action when the "OK" button is pressed and a dismiss action corresponding to when the "Cancel" button is pressed. The `RemoverActivity` is an object use to represent the state and user interface of the dialog shown in Figure 1.2. By extending the framework object `Activity`, the `RemoverActivity` can define callbacks, such as `onCreate`, and access callins for manipulating the user interface (e.g. `makeButton` for adding buttons and `finish` for dismissing the window). We will first describe the normal execution of the buggy application and then describe the crash and fixed application.

```
1  class RemoverActivity extends Activity implements OnClickListener{
2    Button okButton;
3    Button cancelButton;
4    FeedRemover remover;
5    void onCreate(){
6      // Set up the OK and Cancel Buttons
7      this.okButton = this.makeButton(this);
8      this.cancelButton = this.makeButton(new DismissAction());
9      // Create the object encapsulating the removal task
10     this.remover = new FeedRemover();
11   }
12   void onClick(){ //Remove Action
13     // Start the asynchronous removal of the podcast feed
14     this.remover.execute();
15     // Dismiss the dialog window
16     this.finish();
17     this.okButton.setEnabled(false);
18   }
19   class DismissAction extends OnClickListener{
20     void onClick(){
21       RemoverActivity.this.finish();
22     }};
23 }
```

Figure 2.1: An example defect and fix in the Antennapod application, where a background task may be started twice by the `execute` method. This results in a crash on Line 14. Each method in red is a callback (i.e. `onClick` and `onCreate`) that responds to external events. The error occurs because, counterintuitively, `finish` does not prevent a future call to `onClick`, allowing two `onClick` callbacks to occur. The fix that was accepted disables the button using `setEnabled(false)`, shown on Line 17. This example is challenging for program analysis because of the need to model the behavior of the `setEnabled(false)` method and the need of precise aliasing analysis.

Normal Application Execution

During execution of the application, the `onCreate` callback is invoked when the dialog is about to be shown. This callback creates an instance of the `FeedRemover` on Line 10, which encapsulates a background task to remove the podcast feed. When the application calls methods defined by the framework, we refer to them as **callins** to mirror the term callback since they are calling into the framework. The complex process of setting up a button, is merged into the `makeButton` callin (in the real framework this is multiple steps including allocation with `new Button()` and registering a listener with the callin `setOnClickListener`). The return value of `makeButton` is a reference to the button itself, that can be use for actions like disabling the button with `setEnabled(false)`. References to these buttons are stored in fields on the `RemoverActivity` class. There are two arguments to `makeButton`, the first is the `Activity`, representing the portion of the user interface displaying the button, and the second is the listener. A listener is a class with a callback that is invoked when an action occurs. For the buttons, there are two listeners, `RemoverActivity` and `DismissAction`. Such listeners are lambdas in the real code, but we have extracted the `DismissAction` into a named class for clarity of presentation. Additionally, the “Remove Action” has been merged with the `RemoverActivity` to simplify the explanation of proving the fix. These changes are simply for presentation purposes, the techniques we present work on the original versions as well.

When the user of the application clicks the “OK” button, the framework invokes the `onClick` callback on the instance of `RemoverActivity` which was registered as a listener on Line 7. By choosing the receiver of the `onClick` callback, the framework is indicating to the application whether “OK” or “Cancel” was just pressed. For this application (and most Android applications in general), dynamic dispatch is used to select the correct callback corresponding to an action. In this case, the `onClick` on Line 12 is invoked. Next, the invocation of `execute` starts the process of asynchronously removing the podcast feed using the `FeedRemover` created earlier. Finally, the dialog’s `Activity` is removed from the screen by calling the `finish` method.

As a rule, callbacks should always finish quickly to maintain user interface responsiveness. Removal of a podcast feed involves reading and writing to the system storage. This may take long enough to cause the user interface to appear “frozen” or unresponsive. Therefore, tasks that may run for longer than a few milliseconds should be done with such background tasks. We elide the specific code responsible for this background task since it is not relevant to the crash, but it has a separate callback `doInBackground` that is executed on a worker thread. This style of background task throws an error if executed twice to avoid race conditions (i.e. it is a linear resource that is "consumed" when `execute` is invoked). Similar to the "OK" button, the "Cancel" button invokes the `OnClickListener` passed to the earlier call to `makeButton`. However, in the case of the "Cancel" button it just dismisses the dialog.

Crashing Execution

Figure 2.2 shows how interactions between user interface (UI), event loop, framework interface, and app interact at runtime cause this crash. From the perspective of the user, the crash may manifest if the "OK" button is tapped twice in quick succession. These taps are handled by the UI component within the framework and eventually cause the framework interface component to invoke the `onClick` callback twice, causing the crash. To fully understand this defect, it is necessary to explain how the UI communicates to the other components using the event loop and framework interface.

The event loop component is key to understanding the crash. Event loops are used by nearly ever event-driven framework in order to simplify concurrency. In our example, the event loop ensures that all three callbacks are executed sequentially, regardless of when external events occur. For example, it is not possible for the `onClick` callback to occur while the `onCreate` callback is being invoked. Such sequential execution has the advantage that the developer does not need to consider interleaving of callbacks. Without an event loop, overlap of `onClick` and `onCreate` could result in a null pointer exception, if `execute` is called before the task is allocated. With an event loop, the `onClick` and `onCreate` callbacks appear to happen atomically simplifying development. However, the event loop can also complicate application behavior as we will see with this crash.

Figure 2.2 shows how the user interface and application interacts with the event loop to cause this crash. Both the user interface and event loop are part of the Android framework which interacts with the application through the framework interface. Messages within the framework are represented with arrows and labeled with human-readable descriptions. The earlier messages in the execution are at the top of the diagram, with time increasing as they go down. Messages between the framework interface and application consist of the method name and arguments as well as creation of new objects (this is the message history during execution of the application). Arguments are represented by italics letters (with subscripts for buttons and listeners). Callback invocations are labeled with `cb`. Callin invocations are labeled with `ci`. For example, the message b_{ok} = `ci` a.`makeButton`(a) indicates that the application called the `makeButton` method on the framework with the receiver a for the `RemoverActivity` where the button is displayed and the second argument is the listener object which is again the `RemoverActivity` (alternatively, the listener object is $l_{\text{dis}}$for the “Cancel” button).

First, the user interface indicates that the dialog is about to be shown, by queueing an event to create the `RemoverActivity`. When this event is processed, the `onCreate` callback is invoked on an instance a, setting up the buttons (b_{ok} and b_{cancel}), click listeners (a and l_{dis}), and background task (t), the same as normal operation. After invocation, the creation event is removed from the queue (shown by striking out the “Create a” in the queue) and the dialog is shown to the user. When the user taps the “OK” button, the user interface enqueues a “Click l_{rm}” event. As the first click event is being processed, the second user click occurs, adding another “Click l_{rm}” event to the queue. The call to `finish` enqueues an event to dismiss the dialog. But, the second click happens before the `finish` can take effect. As a result, of the second invocation of `onClick`, the callin `execute` is invoked a second time on the background task t.

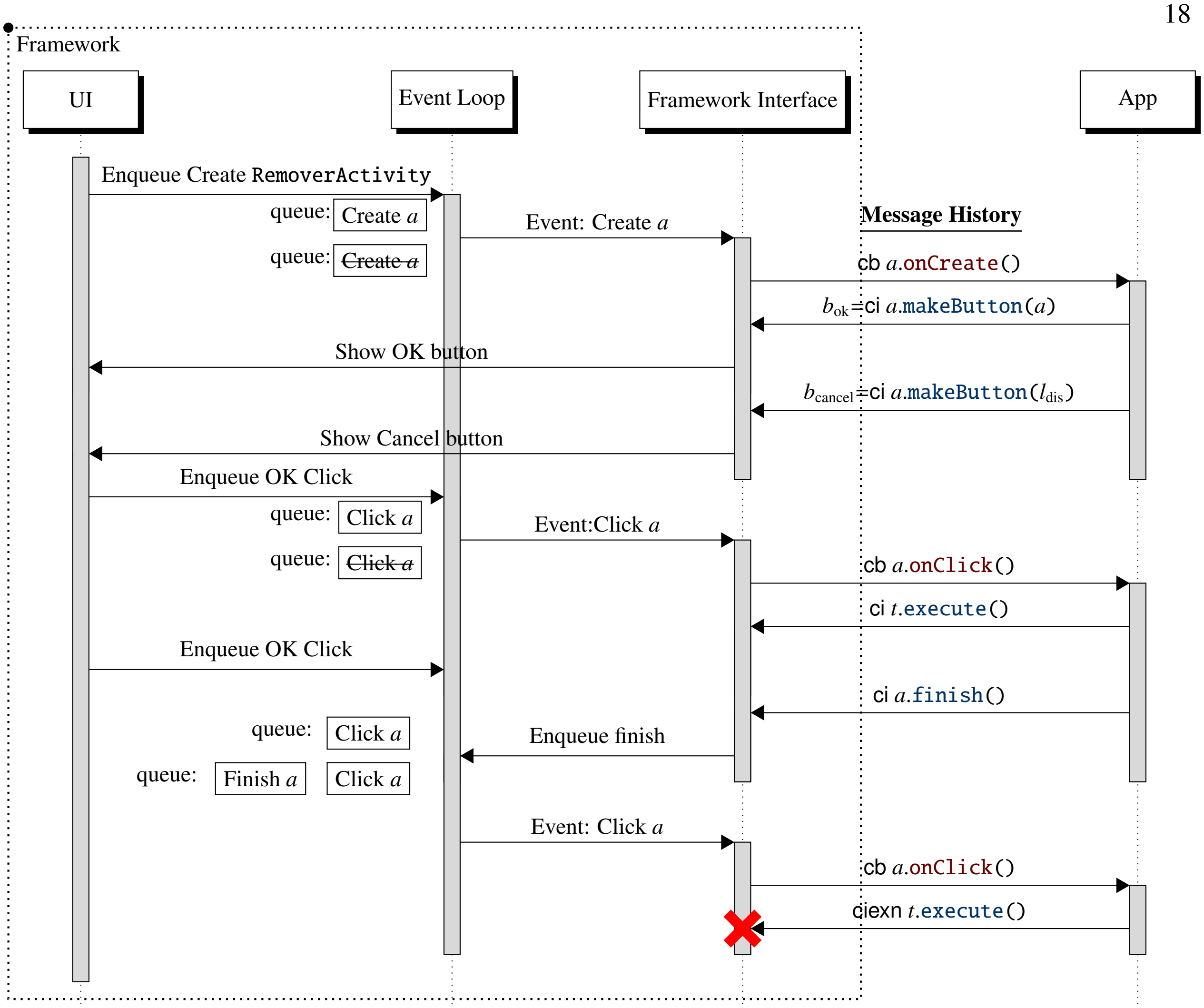

Figure 2.2: The sequence of execution causing the crash. We show three parts of the execution: the user interface (UI) which draws the buttons and dialog, the event loop which queues and executes events, and the application (App). The second click causing the error is possible because the "Finish" event is still in the queue and has not affected the "Click" event. In the fixed version, `setEnabled(false)` actively removes the corresponding "Click" events from the queue.

Fixing the Application

The accepted fix for this defect is to call `setEnabled(false)` (Line 17 of Figure 2.1) on the button to prevent a future `onClick` with the `RemoverActivity` as a receiver. Distinctly different from the behavior of the `finish` callin, `setEnabled(false)` will actually remove pending Click events from the queue.

This fix highlights two key challenges when understanding such a fix or attempting to automate proof that such a fix is safe in a program analysis. The first challenge is precisely reasoning about aliasing and relationship between objects: Since there is an arbitrary number of buttons and other UI objects, any model of the framework behavior needs to distinguish that only the click listeners associated with the disabled button can no longer happen. Specifically, in our example, disabling the "Cancel" button will not fix this issue. The second major challenge is that `finish` and `setEnabled` are just two of the hundreds of thousands of methods that an Android application may invoke. How can we hope to precisely understand the interactions among the tens of thousands of callbacks, hundreds of thousands of callins, and the event loop?

2.1 Describing Callback Control-Flow of an Application Without the Framework

The first contribution of this dissertation is a general method of representing concrete executions in an event-driven application, independent of the framework and event loop (described fully in chapter 3). Modeling and analyzing the queue, shown in Figure 2.2, would be extremely difficult. Various components of the framework may add, remove, or rearrange the events on the queue during normal execution. We call this concrete semantics LambdaLife. LambdaLife should be general enough to describe the execution of any application. For example, we do not want to "hard code" the relationship between the syntactic `onClick` methods, with the syntactic call site for `setEnabled`. Such hard coded solutions would require manual effort to analyze each new application. LambdaLife is the basis for the later contributions around modeling and analyzing event-driven applications.

Central to our approach is capturing the essential, hidden framework state—tracking the set of enabled callbacks and the set of disallowed callins. Figure 2.3 illustrates this model state along a trace from the fixed app. At the beginning of the execution, the cb a.`onCreate` callback is enabled, indicating that the framework may create the `RemoverActivity` at some point. Such callbacks may be thought of as suspended computations or "thunks", that may be executed non-deterministically at some point. Every time a callback is invoked or the application invokes a method on the framework, the model executes code that updates the enabled set. For the cb a.`onCreate()` callback, this code looks like "disable **`this`**`.onCreate()`". Next, the application creates the buttons and registers the listeners with `makeButton`, creating a new thunk representing the button, indicated by adding cb a.`onClick()` to the enabled set.

Next, LambdaLife models the framework execution, by non-deterministically choosing an enabled callback. We show the system choosing cb a.`onClick()`, but an equally valid execution could have been to choose cb l_{dis}.`onClick()`. Since buttons can be clicked an arbitrary number of times, we can think of the `onClick` thunk as re-enabling itself when executed. As a dual to the enabled set, we can think of `execute` disallowing itself by adding ciexn t.`execute()` to the disallowed set. If a method in the disallowed set is invoked, then the defect occurs.

A key difference with traditional thunks is that the thunks may be "disabled" before they execute, `setEnabled(false)` disable the cb a.`onClick()` thunk. To soundly model the framework, `finish` cannot disable the button.

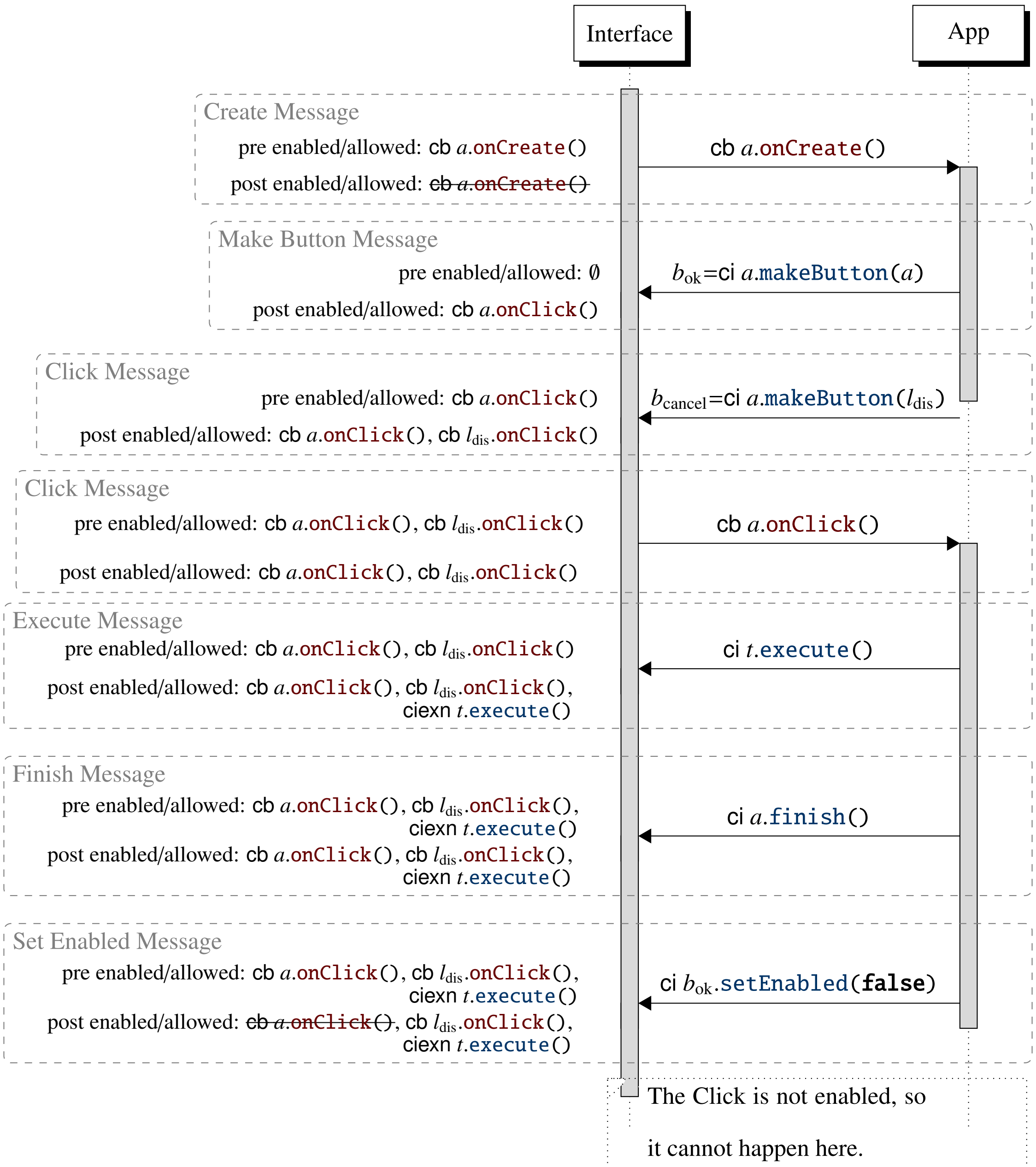

Figure 2.3: The key to modeling the framework is capturing the enabledness of each message paired with the corresponding identifiers. On the left-hand side of the figure, we list the set of enabled callbacks or the callbacks that may be non-deterministically selected to happen next. Each time a message occurs, our execution model updates the enabled and allowed set accordingly. Messages that were just disabled are indicated by being crossed out (e.g. cb a.`onCreate()` self-disables as indicated by ~~cb a.`onCreate()`~~ in the enabled set). Importantly, for the correct functioning of the fix, ci b_{ok}.`setEnabled(false)` disables cb a.`onClick()`. By reasoning about the enabled and allowed sets, we simplify the complex process of heap updates, native code, and more that manage the event loop.

The key observation is that the safety of the application does not depend on whether a button is displayed and not yet clicked or if the button has been clicked but is still in the queue, just whether the `onClick` callback may happen. Similarly, the initial state where the button has not been shown yet is the same as the state right after a `setEnabled(false)`, an `onClick` simply cannot happen. We refer to message histories that may be observed in a running application as **realizable message histories**. But how can we compactly represent realizable message histories without writing the code that enables callbacks?

2.2 Manually Modeling Event-Driven Protocols and Callback Control Flow

With the concrete transition system for an application, the next step is to be able to write down the behavior of the framework that determines the enabled set with a framework modeling language. A common starting point for a framework model is the documented `Activity` lifecycle [4, 105]. In Figure 2.4, we show a lifecycle automaton for the `Activity` class of the Android framework. The black, solid edges are the edges present in the Android documentation showing common callback control flow. However, the activity lifecycle alone is often too imprecise for program verification tasks. Therefore, many analysis techniques [11, 84, 17] will augment the lifecycle by iteratively finding other callbacks and attaching them to the lifecycle, such as `onClick` connected with the short dashed blue lines (for convenience we will refer to this as lifecycle++ later). Unfortunately, due to the dynamic nature of UI construction, it is extremely difficult to construct such a model that can precisely distinguish the bug and fix in our example.

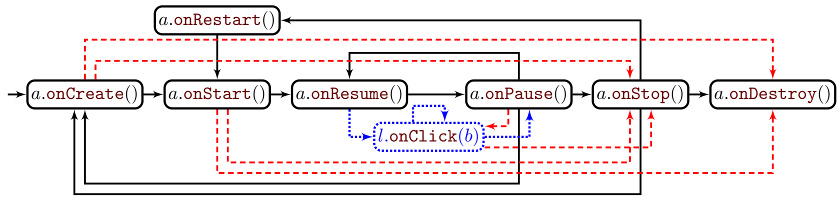

Figure 2.4: A common approach to modeling the Android framework is to use an automata of callbacks. Above, we show the Activity lifecycle automaton from the Android documentation [4] (solid black lines) augmented with the button onClick (short dashed blue lines). Unfortunately, such models are not precise enough to prove the fixed version of our example safe. Worse yet, the finish method can cause a variety of unexpected callback ordering including the edges shown as long dashed red lines.

In order to distinguish the bug and fixed version of our motivating example, we need a better framework modeling language. Such a framework modeling language should be designed to minimize the most common problems associated with framework modeling in general: (1) **The Framework Model May be Unsound**: a framework model can disable a callback when that callback is possible when running the application. As a result, existing defects, such as the buggy version of our example, may be hidden. (2) **The Framework Model May be Imprecise**: on the other hand, a framework model may enable a callback that cannot happen at runtime, which may result in not being able to prove that a correct application avoids a crash. (3) **The Framework Model Can be Undecidable**: later we introduce tools to automatically reason about these framework models. If the modeling language is too complex, then these tools will not perform well (both in runtime and distinguishability). To balance the needs of soundness, precision, and the need for decidability, we introduce the domain specific language Callback Control Flow Temporal Logic (CBCFTL).

CBCFTL is designed to determine the enabled set of callbacks and the allowed set of callins based on the message history. In this way, the model can relate the messages that enable or disable a callback to whether the callback can occur. Considering the onClick callback in both Figure 2.3 and Figure 2.2, we can see that the conditions for an onClick invocation are the same, regardless of the application implementation. These conditions are: (1) The button must be created and attached to a parent Activity. In our application this is done with the makeButton method, that takes the RemoverActivity (a subclass of Activity).

(2) A listener object sub-classing `OnClickListener` must be registered (e.g. with the first argument of `makeButton`). (3) The button must be enabled. That is, either `setEnabled(false)` has never been called or `setEnabled(true)` has been called more recently. We can model the behavior of the `onClick` callback by capturing these conditions.

It can be seen that messages leading to an enabled `onClick` callback fall into two broad categories: messages that affect whether `onClick` can happen at some point in the future and messages that have no effect on `onClick`. For example, calling `setEnabled(false)` has a very meaningful impact on the `onClick` callback. The vast majority of callbacks and callins have no effect. Invoking the `execute` method does not directly change the state of the framework, such that an `onClick` callback may be enabled or disabled. Therefore, CBCFTL is designed to capture the behavior of a small set of callbacks and callins at a time while leaving the majority of callbacks and callins unrestricted.

For this reason, CBCFTL captures one callback (or callin return) at a time and specifies the history under which it can happen using a fragment of past-time temporal logic [80]. We use the term back message to refer to callbacks and the return from callins as both may be captured in CBCFTL. A framework model in CBCFTL is a conjunction of history implications which each targets a single back message. A history implication capturing the creation and enabledness for a button's `onClick` listener is shown by History Implication 1. Full details of CBCFTL may be found in section 4.2.

History Implication 1. *(Sound `onClick`) For all listener objects l, if an `onClick` occurs, then it must be historically true that there exists an `Activity` a and `Button` b, such that Once (*`O` *) in the past `makeButton` was invoked and `setEnabled(false)` has Not been invoked Since (* `NS` *) `setEnabled(false)` or `setEnabled(false)` Has Not (*`HN` *) been invoked.*

$$\mathsf{cb}\ l.\mathtt{onClick}() \mathrel{\square\!\!\!\rightarrow} \mathtt{O}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l) \wedge$$

$$(\mathtt{HN}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{false}) \ \vee\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{false})\ \mathtt{NS}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{true}))$$

History Implication 1 is not specialized to the "OK" button or even this application. To illustrate why, consider the `onClick` defined by the `DismissAction`. In one case, the listener object is the same as the `Activity` representing the dialog, and in the other case, the listener object is an inner class. A framework modeling language needs to capture corner cases where such objects may or may not be aliased. Similarly, it is possible to register one click listener object to multiple buttons. For other components of the framework, it is even possible to register multiple listeners to the same event. All of these behaviors need to be possible to express in a framework modeling language.

Similar to the enabled set of callbacks, the return values of callins may be captured with the disallowed set. We call this a disallowed set because it is typically used to capture errors or return values that may cause errors (however, it may also capture arbitrary return values as well). History Implication 2 shows how the property that `execute` can only be invoked once may be written with CBCFTL. This shows that CBCFTL is versatile enough to capture typestate-like properties [127], as well as capture more complex safety properties with multiple interacting objects [91]. However, CBCFTL extends existing typestate work by allowing properties related to callbacks to be captured as well (e.g. we could specify that `isDestroyed` returns true only if Once in the past, the `onDestroy` callback was invoked).

History Implication 2. *(Disallow `execute`) For all `AsyncTask` objects t, if `execute` emits an exception, then Once (*O *) in the past, `execute` was invoked on t.*

$$\mathsf{ciexn}\ t.\mathtt{execute}() \ \boxminus\!\!\rightarrow \mathsf{O}\ \mathsf{ci}\ t.\mathtt{execute}()$$

The operator $\boxminus\!\!\rightarrow$ is used for history implication where the left-hand side is the target message with free variables (noted by $\widehat{m}$) and the right-hand side is the condition under which the message may occur (referred to as a temporal formula and noted by $\widetilde{\omega}$). The $\widehat{m} \boxminus\!\!\rightarrow \widetilde{\omega}$ 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) $\mathbf{Y}$ from past-time linear temporal logic (ptLTL).

On the right hand side of the $\boxminus\!\!\rightarrow$ operator, we have temporal formula capturing sets of message histories under which the target back message may occur. A temporal formula is a restricted past LTL formula consisting of temporal operators, conjunction, and disjunction. History Implication 1 shows each of these temporal operators: (1) Once ($\mathsf{O}\ \widehat{m}$) corresponds to a message required for the target to occur. For example, there must have been a `makeButton` to see an `onClick` callback. (2) Has Not ($\mathtt{HN}\ \widehat{m}$) corresponds to a message that disables a target back message. For example, `setEnabled(false)` prevents the `onClick` callback. (3) Finally, Not Since ($\widehat{m}_1$ $\mathtt{NS}$ $\widehat{m}_2$) corresponds to a toggling behavior where $\widehat{m}_1$ has not happened since $\widehat{m}_2$. For example, if `setEnabled(true)` has happened more recently than `setEnabled(false)`, then an `onClick` may occur if the other requirements are also met. Despite the apparent simplicity of these operators, this dissertation shows experimental evidence in chapter 4 and chapter 5 that they can capture a wide variety of event-driven components.

The “Next” temporal operator has been intentionally excluded from this language (or any other operator that would broadly affect messages not named in the formula). This is an intentional design choice based on the observation that most messages do not affect each other. From the perspective of designing framework modeling language well, we argue that the framework modeling language should clearly state how one named message affects another. A formula stating something like “after an `onCreate`, next there is an `onClick`”, unsoundly prevents the occurrence of unrelated callbacks between the two (e.g. if the user dismisses the application causing an `onDestroy` callback). There are a wide variety of possible representations that could have been used to capture sequences of messages (e.g. automata, context free grammars, etc). However, such representations can easily restrict unrelated messages by saying something like “`onClick` must come next”. This design philosophy reduces much of the unintentional unsoundness of framework models written in CBCFTL without significantly hindering the precision (as demonstrated by the experimental evaluation in section 4.4). However, unsound models may still be written so how can we build confidence that a model is correct?

2.2.1 Validation of Framework Models to test Soundness

Given the size and complexity of event driven frameworks, such as Android, it is important to be able to build confidence that a framework model is correct. Unfortunately, when writing a framework model, it is just as easy to make the same mistakes that a developer may make when determining how a given method behaves. For example, History Implication 3 corresponds to the earlier unsound assumption that calling the `finish` method can prevent a subsequent call to `onClick`. Similar to an application developer, the developer of a framework model may not have sufficient documentation to accurately model the behavior of an event-driven framework.

History Implication 3. *(Unsound* `onClick`*) For all listener objects l, if an* `onClick` *occurs, then it must be historically true that there exists an* `Activity` *a and* `Button` *b such that Once (*`O`*) in the past,* `makeButton` *was invoked on a returning b and, additionally,* `finish` *Has Not (*`HN`*) been invoked on a.*

$$\mathsf{ci}\ l.\mathtt{onClick}() \;\square\!\!\rightarrow \mathtt{O}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l) \wedge \mathtt{HN}\ \mathsf{ci}\ a.\mathtt{finish}()$$

However, the structure of these framework models means that message histories from any arbitrary application must be consistent. Therefore, if a message history is observed at runtime that shows the `onClick` callback after a `finish` callin, then the model can be observed to be unsound. For example, if we observe a message history of the form Figure 2.5 in an arbitrary running application, then History Implication 3 cannot be sound.

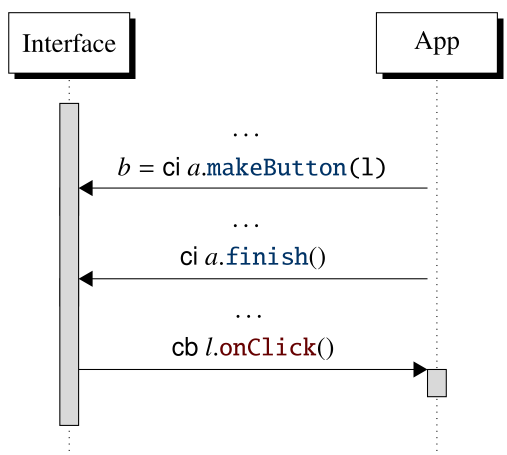

Figure 2.5: Any message history from any application that contains these three messages in this order shows History Implication 3 to be unsound. This could be from the message history observed in the crashing execution shown earlier (Figure 2.2), or it may be from a completely unrelated application.

In section 3.4 we perform this validation on the framework model for the most well-maintained open-source Android program analysis that models callback order, Flowdroid [11]. We find that this model is very unsound when it comes to precisely capturing callback order. Whereas this model has proven extremely useful for the purposes of taint analysis, it cannot be used to distinguish defects like our motivating example. Additionally, we show how useful models may be written while still validating on 1679 recorded message histories.

2.3 Refuting Callback Reachability with Message-History Logics

The next step is to explain the basis behind application only static analysis that uses these framework models to define realizable message histories. In Figure 2.6, 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. Complementing section 2.1 which describes the behavior of the framework, the application-only transition system describes the behavior of the application. Our key insight is to internalize the concept of realizable message histories into the static analysis abstraction when analyzing this application-only transition system. 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 a high level, our approach shares some conceptual similarity with context-free language (CFL) reachability-based analysis [111] 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 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 2.1.

2.3.1 An Application-Only Transition System to Define Executions Restricted by Message Histories

As discussed in the introduction, event-driven applications typically come as partial programs missing code that invokes the callbacks. This code is often also missing the implementation for callins. The goal of this section is to show how such a partial program is converted into a form that a program verifier can use. This application-only transition system is inspired by the work on application-only call graphs [1, 2] which present a reasonable set of assumptions about the operation of frameworks in order to build call graphs while not involving the complexity of the framework.

What makes the application-only transition system unique is that the code of the application is preserved in a way that enables program analysis that cares about the order of commands (i.e. flow-sensitive program analysis). Conceptually, the application only transition system is a control flow graph of the application where all nodes in the framework have been merged into a single Fwknode. The entry and exit of callbacks and callins are augmented with boundary transitions that append to the message history. 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 subsection 4.1.1).

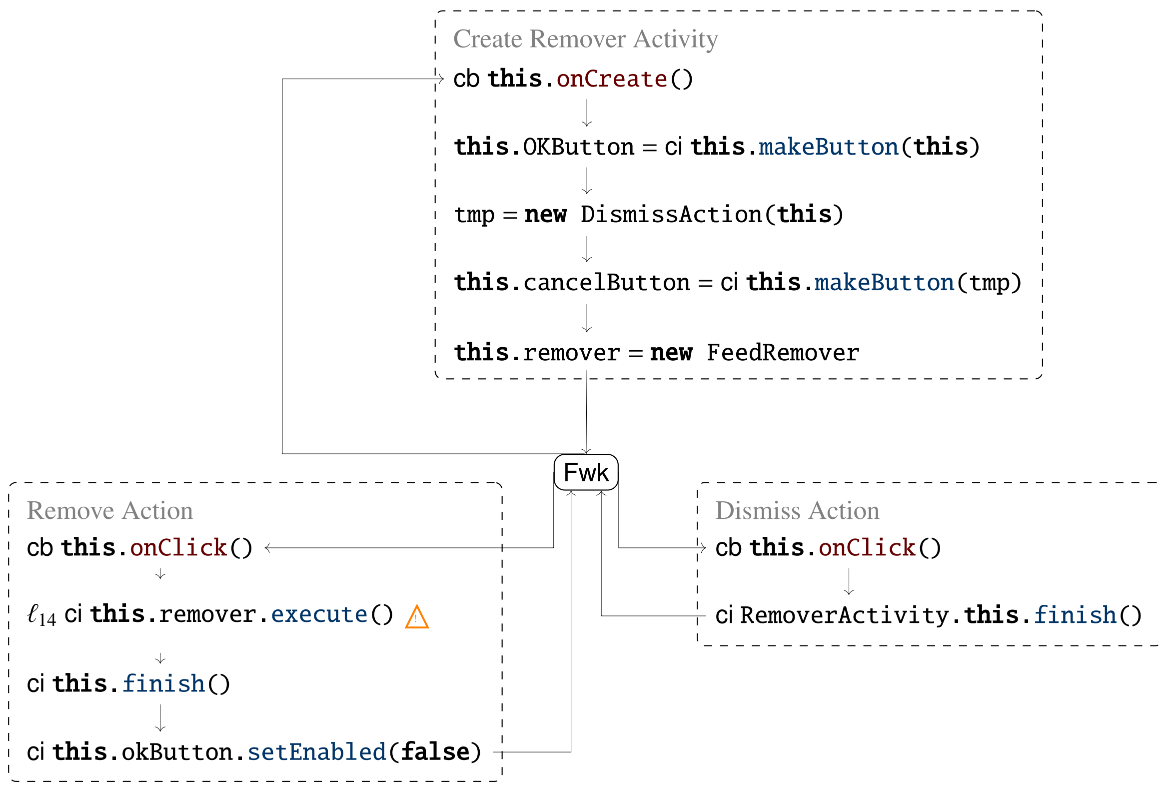

Figure 2.6: An application-only transition system constructed from our motivating example. Each callback is connected to a central framework location and boundary transitions are added at the invocation of callbacks and callins as well as their returns.

As these transitions occur, they have the effect of writing their message to the message history ghost state discussed earlier. 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). Each boundary transition is assumed to be possible until explicitly disabled by the framework model. The next challenge is to perform a static analysis that can utilize the CBCFTL framework model and the application only transition system to prove safety.

2.3.2 Message History Program Logic (MHPL) Enables Static Reasoning about Message Histories

Program analyses often compute invariants for each location in the program. MHPL specifically computes the invariant that over-approximates all states that may reach the assertion failure. This technique was inspired by goal-directed analysis [17, 16] which proves safety if all control flow paths that might reach the failure are contradictory (i.e. no concrete states can exist at a location that reaches the failure). However, unlike goal-directed analysis, event-driven programs rarely derive a direct contradiction. Instead, 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, and our program logic has proven that the assertion failure is impossible. Here, we demonstrate such a proof on our running example.

While analyzing the application, MHPL maintains an invariant map, mapping each program location to its current invariant (represented as $\widehat{\Sigma}$ in subsection 4.1.2). 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 2.8. In a goal-directed, backwards-from-error formulation, the invariant map initializes with the abstract state $\{\mathsf{ciexn}\ t.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist} \cdot \mathtt{this} \mapsto a * a.\mathtt{remover} \mapsto t\}$ which is positioned right before the assertion failure. There are two parts of the abstract state (which we separate with a center dot $\cdot$). On the right, there is an abstraction of the heap or store of the app for the assertion failure; in particular, it says that the **this** variable points to some object a and, separately, a has a field remover that points to a task t. We use intuitionistic separation logic [64, 112] 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 subsection 4.1.2). 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. The full starting abstract message history, $\mathsf{ciexn}\ t.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist}$, indicates the realizable message histories such that execute is dissallowed and is invoked next.

Fwk_4 {cb a_2.`onCreate()` $\twoheadrightarrow$ b_4 = ci a_2.`makeButton(`a_2`)` $\twoheadrightarrow$ b_3 = ci a_2.`makeButton(`l_2`)` $\twoheadrightarrow$
cb a.`onCreate()` $\twoheadrightarrow$ b_1 = ci a.`makeButton(`a`)` $\twoheadrightarrow$ b_2 = ci a.`makeButton(`l_1`)` $\twoheadrightarrow$
cb a.`onClick()` $\twoheadrightarrow$ ci t.`execute` $\twoheadrightarrow$ b_1.`setEnabled(false)` $\twoheadrightarrow$
cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist}

$\vdash$

$\boxed{\text{cb } \texttt{this.onCreate();}\ldots}$

Fwk_3 {cb a.`onCreate()` $\twoheadrightarrow$ b_1 = ci a.`makeButton(`a`)` $\twoheadrightarrow$ b_2 = ci a.`makeButton(`l_1`)` $\twoheadrightarrow$
cb a.`onClick()` $\twoheadrightarrow$ ci t.`execute` $\twoheadrightarrow$ b_1.`setEnabled(false)` $\twoheadrightarrow$
cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist}

$\boxed{\text{cb } \texttt{this.onCreate();}\ldots}$

Fwk_2 {cb a.`onClick()` $\twoheadrightarrow$ci t.`execute` $\twoheadrightarrow$ b_1.`setEnabled(false)` $\twoheadrightarrow$
cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist $\cdot$ a.okButton $\mapsto b_1 \ast a$.remover $\mapsto t$}

$\boxed{\begin{array}{c}\text{cb } \texttt{this.onClick();}\text{ ci } \texttt{remover.execute();} \\ \text{ci } \texttt{this.finish();}\text{ ci } \texttt{okButton.setEnable(false)}\end{array}}$

Fwk_1 { cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist $\cdot$ a.remover $\mapsto t$}

$\boxed{\text{cb } \texttt{this.onClick()}}$

ℓ_{14} {ciexn t.`execute()` $\twoheadrightarrow$ okhist $\cdot$ **this** $\mapsto a \ast a$.remover $\mapsto t$}

$\boxed{\text{ci } \texttt{remover.execute()}}$

Figure 2.7: 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 2.6) 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., Fwk_1, Fwk_2) to indicate disjunctive elements of the abstract state at the framework location. With the specification of realizable message histories (section 2.2) combined with the abstract message histories (subsection 2.3.3), we can show that this invariant has reached a fixed point and that it excludes the initial state proving the assertion safe.

Intuitively, ciexn t.`execute()` $\twoheadrightarrow$ okhist denotes any message history to which ciexn t.`execute()`

can be appended to obtain a realizable message history. Operationally, ciexn t.`execute()` $\twoheadrightarrow$ 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 cb a.`onClick()`, and truncating the remaining ones to remove cb a.`onClick()` from the end. Then, to capture the effect of invoking `onClick`, the abstract message history to updates cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist at the abstract message history labeled by Fwk_1 in Figure 2.8 (this abstract state also removes the **`this`** variable due to popping the stack).

Having computed an abstract state at a framework location (Fwk_1 in Fig. 2.8), 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. If the framework could invoke l.`onClick()` as the first callback, then l.`onClick()` alone is a realizable message history (i.e., cb l.`onClick()` $\in$ okhist). Consequently, cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ 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 l.`onClick()` as the first callback (i.e., cb l.`onClick()` 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), MHPL can use separately-provided specifications of realizable message histories. History Implication 1 requires cb l.`onClick()` to be preceded by the invocation of b = ci a.`makeButton(`l`)` for the message history to be realizable (as well as a prior `execute` on the same task t).

This transformation of the abstract state from ℓ_{14} to 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 `onClick` callback results in the abstract state Fwk_2{cb a.`onClick()` $\twoheadrightarrow$ ci t.`execute` $\twoheadrightarrow$ b_1.`setEnabled(`**`false`**`)` $\twoheadrightarrow$ cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist $\cdot$ a.`okButton` $\mapsto b_1 * a$.`remover` $\mapsto t$} denoting the message histories that end in the corresponding messages. 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 {Fwk_1{...} $\vee$ Fwk_2{...} $\vee$ ...}). Continuing so on through the application-only transition system (and using the necessary specifications) would yield an abstract state at Fwk_4 with two `onCreate` callbacks, two `onClick` callbacks with their associated callins interleaved.

Note that this state includes the initial state raising an alarm, as it satisfies the constraint imposed by History Implication 1. This execution corresponds to `onCreate` being invoked on the same `RemoverActivity` twice creating two separate buttons. Subsequently, the `makeButton` callin registers the listener on both buttons. Since `setEnabled(false)` only disables the most recent button stored in the field `okButton`, this means there may still be a clickable button visible. To remove this alarm, we add History Implication 7 restricting `onCreate` to only be invoked once.

History Implication 4. *(`onCreate` only once) For all `Activity` objects a, if `onCreate` was invoked, then `onCreate` Has Not (*`HN`*) been invoked in the past on a.*

$$\mathsf{cb}\ a.\mathtt{onCreate()} \mathrel{\square\!\!\rightarrow} 0\ \mathsf{cb}\ a.\mathtt{onCreate()}$$

As messages are parameterized 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 in the number of callbacks. 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 2.8, for example, Fwk_4 is merged with Fwk_3 since the first `onClick` cannot be the same instance of `RemoverActivity` due to History Implication 7. No further backward transitions need to be explored from Fwk_4 because they have been explored from Fwk_3.

We automate MHPL in an open-source tool called Historia which computes the fixed point at the framework location. Historia considers 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 crashing `execute` in the fixed version of AntennaPod. For presentation, Figure 2.8 shows only a few of the transitions and abstract states computed for the running example. As can be seen by this simplified example, reasoning about abstract message histories and CBCFTL can be extremely complex, how can we automate reasoning about abstract message histories?

2.3.3 Combining Abstract Message Histories and Framework Models for Automated Reasoning

Next, we consider how to interpret and automatically reason about the meaning of abstract message histories. As shown previously, we need to check two things: whether an abstract state excludes the initial state and whether one abstract state is contained in another (entailment). Efficient automation of checking excludes initial and entailment is crucial to verifying real applications due to the large number of abstract states for even small applications. Such efficiency is achieved by an encoding such that message histories are representable in Extended Effectively Propositional (Extended EPR) logic [73, 96].

Consider the abstract state just before the `onClick` callback with the abstract message history transition, $\mathsf{cb}\ a.\texttt{onClick()} \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute()} \twoheadrightarrow \mathsf{okhist}$. This abstract message history means that the next message must be $\mathsf{cb}\ a.\texttt{onClick()}$ and then, after that, $\mathsf{ciexn}\ t.\texttt{execute()}$. Then, the abstract message history matches any prefix of those two histories that is represented by the CBCFTL framework model. First, excludes-initial needs to be proven (i.e., this abstract state does not contain the initial state), and then, entailment checks if an abstract state may be merged with existing 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 $\mathsf{cb}\ a.\texttt{onClick()} \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute()} \twoheadrightarrow \mathsf{okhist}$ with History Implication 2 (`execute` error), History Implication 1 (`onClick`), and History Implication 7 (`onCreate`). The first step of combining abstract message histories with temporal formula is **instantiation** (i.e., the INSTANTIATE-YES judgment in section 4.3). 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.

First, we apply the effect of the `execute` history implication. We know that $\mathsf{cb}\ a.\texttt{onClick()} \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute()} \twoheadrightarrow \mathsf{okhist}$ matches message histories with an erroring `execute` appended at the end. Additionally, History Implication 2 says that if an error is emitted from `execute`, then once in the past `execute` must have been called. We can combine these two and get message histories where two things must be true:

(1) Once in the past, `execute` must have been called, i.e. $0\ \mathsf{ci}\ t.\mathtt{execute}()$

(2) Appending $\mathsf{cb}\ a.\mathtt{onClick}()$ to the message history is realizable, i.e. $\mathsf{cb}\ a.\mathtt{onClick}() \twoheadrightarrow \mathsf{okhist}$.

At this point, we need to consider two cases, the first where $\mathsf{cb}\ a.\mathtt{onClick}() = 0\ \mathsf{ci}\ t.\mathtt{execute}()$ in which case we no longer need an `execute` once in the past. However, these are clearly not equal as they are different methods so the requirement of $0\ \mathsf{ci}\ t.\mathtt{execute}()$ remains. Alternatively, an abstract message history of $\mathsf{ci}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{ciexn}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{okhist}$ would be equivalent to okhist. Encoding steps where messages are trivially non-equal are extremely common since these specifications tend to talk about small groups of interacting messages (as discussed in section 2.2). This non-equality helps significantly with performance allowing the formulas representing abstract message histories to remain relatively small.

We can then apply the instantiation on the message $\mathsf{cb}\ a.\mathtt{onClick}()$ and the `onClick` HistoryImplication 1 resulting in the following two requirements:

(1) okhist

(2) $0\ \mathsf{ci}\ t.\mathtt{execute}() \land 0\ b = \mathsf{ci}\ a.\mathtt{makeButton}(a)\ \land$

$(\mathtt{HN}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{false})\ \lor\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{false})\ \mathtt{NS}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{true}))$

For explanation purposes, we can simplify the second requirement since the application cannot emit a $\mathtt{setEnabled}(\mathbf{true})$ (this is the case since $\mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{false})\ \mathtt{NS}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{true})$ implies that once in the past $\mathsf{ci}\ b\mathtt{setEnabled}(\mathbf{true})$ was invoked). So the message histories must now have an `execute`, a `makeButton`, and not have a $\mathtt{setEnabled}(\mathbf{false})$. That is, message histories must satisfy the following temporal formula: $0\ \mathsf{ci}\ t.\mathtt{execute}() \land 0\ b = \mathsf{ci}\ a.\mathtt{makeButton}(a)\ \land \mathtt{HN}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathbf{false})$.

Intuitively, we can see that this temporal formula must exclude the initial state since an `execute` and a `makeButton` must have come earlier. However, the verification process described in section 2.3 generates numerous such formula requiring that excludes initial be automatic. For this reason, we convert such formula to first order logic within the extended EPR logic fragment (discussed earlier).

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.

The sub formula $\mathsf{ciexn}\ t.\mathtt{execute()}$ may be encoded in extended EPR by representing a concrete message history as functions from a totally ordered uninterpreted sort to messages as follows:

$$\exists t \in \mathbf{Val}, i \in [0, \mathtt{len}], m \in \mathbf{Msg}.\mathsf{hist}(i) = m \wedge \mathsf{msgname}(m) = \text{“}\mathsf{ci}\mathtt{execute}\text{”} \wedge \mathsf{msgargs}(m, \text{“receiver”}) = t$$

In this encoding, the uninterpreted functions hist,msgname, and msgargs are used to represent the concrete message history. Messages $m \in \mathbf{Msg}$ and values $t \in \mathbf{Val}$ are uninterpreted sorts. Indices are represented using another uninterpreted sort but with axioms making a total order that defines the special zero value as well as less than. Similarly, the other parts of the formula may be encoded (negating the Has Not and converting to negation normal form). The full explanation of this encoding may be found in subsection 4.2.2.

Once we have such a formula encoded for each part, excludes initial can be automatically determined by checking the satisfiability of “[FOL encoding of CBCFTL and Abstract State]$\wedge \mathtt{len} = 0$”. If this formula is unsatisfiable, then the abstract state must exclude the initial, empty message history.

Entailment of Abstract States

The same first order logic encoding may be used to determine entailment of abstract states such as the abstract states labeled Fwk_4 and Fwk_3. If the formula “[FOL encoding of CBCFTL and Abstract StateFwk_4] $\wedge$ ¬[FOL encoding of CBCFTL and Abstract StateFwk_3]” is unsatisfiable then the abstract state Fwk_3 entirely contains abstract state Fwk_4. While this negation does make the entailment check no longer in the extended EPR fragment of logic, we find that it performs well in section 4.4 Such an encoding allows abstract states to be eagerly merged during verification significantly improving performance. As hinted at when we were discussing that most messages do not need to be instantiated or updated, the next section answers the question “How can we improve both the performance and difficulty of writing models?”.

2.3.4 Targeted Framework Modeling

We hinted earlier that most messages in the framework do not interact with one another. This fact has enormous benefits for both the difficulty of writing models and the performance of analysis. An added bonus of the design of CBCFTL and our application only analysis is that it is always sound to assume that two messages do not interact with one another. As a result, the goal of framework modeling is not to capture all behavior of the event-driven framework. Rather, the framework model only needs to restrict just barely enough behavior to prove safety properties of interest. We call such a model a **targeted** model because it targets the properties that the application relies on for safety.

Another requirement for an `onClick` callback indicating that a button has been clicked is that the button appears on a visible window on the user interface. The CBCFTL History Implication 5 soundly captures this requirement by saying "If there is an `onClick` then the parent `Activity` must have had the `onCreate` method invoked". However, since `makeButton` is also required for an `onClick` and almost always appears in the `onCreate`, such a framework model is rarely useful for verifying safety properties.

History Implication 5. *(Sound but Un-Targeted* `onClick`*) For all listener objects l, if an* `onClick` *callback occurs, then there must exist an* `Activity` *object a, such that* `makeButton` *was invoked on a with the first argument l and* `onCreate` *was invoked on a.*

$$\mathsf{ci}\ l.\mathtt{onClick}()\ \square\!\!\rightarrow 0\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l)\ \wedge 0\ \mathsf{cb}\ a.\mathtt{onCreate}()$$

As we will find in section 4.4, most properties resulting from messages such as `onClick` and `onCreate` do not actually affect the correctness of an application. This evaluation shows that in real applications, we typically only need to write framework models that match 2 to 5% of the messages possible in realistic Android applications leaving the rest abstract.

2.4 Discovery and Explanation of Event-Driven Defects with History-Aware Incorrectness Logic

An interesting observation about our technique for proving properties in event-driven applications is that it can be extremely precise. In fact, with some reasonable modifications, we can change the direction of the soundness and create a program analysis technique that finds and proves the message history reaching a location or defect.

Such a technique still depends on the precision framework model but even that can end up being useful. Here we give an overview of the techniques presented in chapter 5 to find and prove that a message history reaches a location or defect.

The primary use for this history aware incorrectness logic is to validate framework models without the need of recording message histories. Any sound framework model should represent a message history reaching any point in a program that is reachable when running under the real framework. For example, the unsound History Implication 3 can not allow the crash shown by Figure 1.1 to be reachable. This is because the original buggy application would actually be correct if `finish` prevented future `onClick` callbacks. However, simply witnessing the defect gives us a hint as to what properties of the framework can be relied upon for safety.

While useful for validating framework models, recording message histories is extremely difficult as the instrumentation tends to alter the behavior of applications simply by changing the timing of callbacks. These timing alterations often result in not recording as much of the application behavior as would be needed to fully validate models. Known reachable locations for validation may be obtained easily for applications in a number of ways. For example, by observing a crash like the one in Figure 1.1, logging or print statements in various locations, sampling profilers, or even locations known to not be dead code.

A secondary reason for a technique to find and prove that a message history reaches a defect is to find and explain defects for developers. If the framework model is precise enough, then the technique presented in this section will find a message history that must reach the defect under a given framework model.

2.4.1 Goal-directed Incorrectness Logic

Much of this section mirrors how we would prove a location safe as discussed with the Historia work earlier. The key difference is that the invariant map now represents only states that must step to the error location and cause the failure. In order to find and prove that a sequence of transitions reaches a particular defect, we first start at the location highlighted by the stack trace in the buggy application Figure 1.1. The

error condition is similar to the proof over the safe application from earlier since the crashing line is unchanged. Then, we accumulate abstractions of states that must step to the error condition. If one of the accumulated states is the **initial state** of the application, then we have found an execution history reaching the defect.

$$\mathsf{Fwk}_3\ \{\mathsf{cb}\ a.\texttt{onCreate}() \twoheadrightarrow b_1 = \mathsf{ci}\ a.\texttt{makeButton}(a) \twoheadrightarrow l_{dis} = \mathsf{new} \twoheadrightarrow b_2 = \mathsf{ci}\ a.\texttt{makeButton}(l_1) \twoheadrightarrow t = \mathsf{new} \twoheadrightarrow$$
$$\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ci}\ t.\texttt{execute}() \twoheadrightarrow a.\texttt{finish}() \twoheadrightarrow$$
$$\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot \mathsf{irrelevant}\}$$

$$\downarrow$$

$$\boxed{\mathsf{cb}\ \textbf{this}.\texttt{onCreate}();\ldots}$$

$$\downarrow$$

$$\mathsf{Fwk}_2\ \{\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ci}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{ci}\ a.\texttt{finish}()$$
$$\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot a.\texttt{okButton} \mapsto b_1\ * a.\texttt{remover} \mapsto t * \mathsf{irrelevant}\}$$

$$\downarrow$$

$$\boxed{\mathsf{cb}\ \textbf{this}.\texttt{onClick}(); \mathsf{ci}\ \texttt{remover}.\texttt{execute}(); \mathsf{ci}\ \textbf{this}.\texttt{finish}()}$$

$$\downarrow$$

$$\mathsf{Fwk}_1\ \{\ \mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot a.\texttt{remover} \mapsto t * \mathsf{irrelevant}\}$$

$$\downarrow$$

$$\boxed{\mathsf{cb}\ \textbf{this}.\texttt{onClick}()}$$

$$\downarrow$$

$$\ell_{14}\ \{\mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot \textbf{this} \mapsto a * a.\texttt{remover} \mapsto t * \mathsf{irrelevant}\}$$

$$\downarrow$$

$$\boxed{\mathsf{ci}\ \texttt{remover}.\texttt{execute}()}$$

Figure 2.8: The invariant map used to prove that a message history reaches the defect in the buggy version of our example. This invariant map largely follows the same pattern but now only represents concrete states that must step to the error location. The main changes from the verification are that the new calls must be precisely tracked. For example, with Historia heap cells could be soundly summarized (dropped from the formula) which is not possible here. Upon reaching Fwk_3 we have found an abstract state that must include the initial state and represents a message history reaching the defect.

The error condition for our motivating example requires the `outer` field of some `RemoverActivity` a to point to some `FeedRemover` t and the execution history to end in a call to $t.\texttt{execute}()$. Specifically, we use the following as the error condition $\ell_{14}\{\mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot \textbf{this} \mapsto a * a.\texttt{remover} \mapsto t * \mathsf{irrelevant}\}$. For the memory, $\textbf{this} \mapsto a * a.\texttt{remover} \mapsto t$ represents the **this** variable pointing to a and the `outer` field pointing to the task t. Standard separating conjunction $*$ combines such memory locations with disjoint domains. We use the keyword $\mathsf{irrelevant}$ to represent the portion of memory that cannot be read along the path to the crash. This memory could have any value and the crash would still occur. Appending $\mathsf{ciexn}\ t.\mathtt{execute()}$ with the symbol $\twoheadrightarrow$ to okhist yields a constraint that must be read as following: "the history resulting from appending an exception from the `execute` callin to a realizable history must also be realizable". In general, this is not true as, given any realizable execution history (i.e., a legal trace) of the application in which a `FeedRemover` object t is created, appending a call to $t.\mathtt{execute()}$ at the end is not guaranteed to generate an exception. For the exception to manifest, there must have been an earlier call to $t.\mathtt{execute()}$, i.e., the history should already contain a $\mathsf{ci}\ r.\mathtt{execute()}$ event. Similar to the verifier, we can use History Implication 2 to interpret the meaning as "Once in the past, `execute` must have been invoked on t". Given such a specification, the aim of the analysis is to build a witness execution that satisfies the specification and ends in a crash.

Starting with the error condition $\ell_{14}\{\mathsf{ciexn}\ t.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist} \cdot \mathbf{this} \mapsto a * a.\mathtt{remover} \mapsto t * \mathsf{irrelevant}\}$ before Line 14, we accumulate states that must step to the error condition in an **invariant map**. The invariant map is a mapping from program locations (before or after application lines or the framework location) to abstract states and abstract temporal orders. Further steps will continue to pick abstract states in the invariant map and update based on the command that comes just before. Multiple abstract states may exist at a single location and simply represent that a state matching any single one steps to the error from that location. In this way, we ensure that only states that must be able to reach the defect are added, maintaining the spirit of incorrectness logic.

Within the application only transition system, some transitions like the entry into callbacks depend on the overall order of execution and are referred to as **sequence sensitive**. Other transitions such as field reads and writes are **sequence agnostic** as their semantics entirely depend on the application state before their execution. Surprisingly, we must treat calls to **`new`** as **sequence sensitive** because it cannot return a value that was involved in a previous callin or callback. If a value has been passed from the framework to the application in the past, then the **`new`** command will never return that same value since some call to **`new`** in the framework must have previously created it. To simplify our semantics, we assume that there is no garbage collection and each invocation of **new** returns a fresh value. In the next sub-section, we describe the sequence agnostic memory commands and in the following sub-section, we describe the sequence sensitive commands.

2.4.2 Relevant Memory Abstraction

The application memory is split into two parts for abstraction, the first is the **relevant** memory or the locations that **may** affect control flow to the defect. In general, the relevant portion of the memory can be thought of as the "live memory" that will be read at some point between the current location and the defect. However, any memory locations read after the defect or memory cells that will be written are not relevant to proving reachability. By representing this irrelevant memory with a special symbol, we save the trouble of needing to derive a precise abstraction of the full error condition at the location of the defect. We represent application memories with standard separation logic formula extended with the special symbol irrelevant representing irrelevant memory. Our other major divergence from the standard separation logic is that all materialized heap cells must be discharged through field writes in order for a memory to be initial.

Our error condition $\ell_{14}\{\mathsf{ciexn}\ t.\texttt{execute()} \twoheadrightarrow \mathsf{okhist} \cdot \textbf{this} \mapsto a * a.\texttt{remover} \mapsto t * \mathsf{irrelevant}\}$ captures the relevant memory with $\textbf{this} \mapsto a * a$ since the only memory locations that may be read are the **this** variable and the corresponding `remover` field. Besides these memory locations, no other portion of memory could affect the error at this point as it cannot be read by **this** command and therefore can be represented by irrelevant in the formula. However, it is possible and likely for other disjoint portions of the memory to have been written and still exist.

Upon jumping back from the entry of `onClick` to the exit of a previous `onClick`, the abstract state must capture whether this is the same click listener l and remover or different ones. In the case of different instances, they are materialized as new heap cells and constrained to be disaliased from the current relevant ones. These backwards steps continue through a previous `onClick` callback and then to the `onCreate` callback. Similarly, each subsequent step back must represent the portion of memory read and written in the relevant memory. Within the `onCreate` callback, the `remover` field is written. Proceeding backwards through the example in the aliased case, the `onCreate` callback sets the `outer` and `remover` fields getting to state 3 which includes the initial state indicating that a witness has been found.

2.4.3 Execution Sequence Abstraction

We abstract the sequence of the program execution such that the restrictions imposed by each future event are precisely captured. Such restrictions take the form of facts such as "execute must have thrown an exception in the future so another execute must have been invoked in the past on the same object" or "the instance of `FeedRemover` that throws the error was allocated in the future therefore nothing can currently reference the value". If we observe the requirements of such an abstraction, we can see that two important aspects must be captured: the order of events and their parameters.

In chapter 5, we show how the decision procedure using this fragment of temporal logic can prove inclusion of the initial state as well as determining emptiness of abstract message histories. CBCFTL targets a framework message such as `onClick` and restricts the message histories under which it could have happened using a history implication. We assume this definition to completely capture the ordering constraints on framework interactions. In the evaluation section, we constrain such behaviors as needed for the components we encounter.

Distinct from framework messages is the **`new`** command which returns a fresh address each time it is called. This message is distinct from framework messages because it not only constrains temporal ordering, but all values in the combined application and framework memories. To prove that any state reaching a given **`new`** command respects this behavior, we must constrain that it cannot have referenced the value returned by **`new`** before the **`new`** was actually invoked.

Later, when the abstract state containing two executes is reached, the state contains the initial empty trace because the error condition is satisfied. Then, if the abstract state also does not constrain the memory, the entire state includes the initial state. In our example, the following abstract state includes the initial state of the application finding the execution history reaching the defect:

$$\begin{aligned}\mathsf{Fwk}_3\ \{&\mathsf{cb}\ a.\mathtt{onCreate}() \twoheadrightarrow b_1 = \mathsf{ci}\ a.\mathtt{makeButton}(a) \twoheadrightarrow l_{\mathrm{dis}} = \mathsf{new} \twoheadrightarrow b_2 = \mathsf{ci}\ a.\mathtt{makeButton}(l_1) \twoheadrightarrow t = \mathsf{new} \twoheadrightarrow \\ &\mathsf{cb}\ a.\mathtt{onClick}() \twoheadrightarrow \mathsf{ci}\ t.\mathtt{execute}() \twoheadrightarrow a.\mathtt{finish}() \twoheadrightarrow \\ &\mathsf{cb}\ a.\mathtt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot \mathsf{irrelevant}\}\end{aligned}$$

The error condition is satisfied because we observed a previous call to `execute`, the `onClick` callback was registered with `setOnClickListener`, and values may be chosen for the calls to **`new`** such that memory is not referenced before allocation. Additionally, the memory abstraction irrelevant abstracts the initial, empty, and memory. By choosing satisfying assignments to the a,t, and b logic variables, we can reconstruct the history of execution reaching the defect as shown in Figure 2.2.

Extending the search to the abstract state where we have a two different activities that each invoke `onCreate` and then `onClick`, we can see how a lack of a temporal constraint on **`new`** commands causes unsoundness (i.e. our tool would say there is a path when there is none). Consider the fixed version of the application where `onClick` invokes `b.setEnabled(false)` where there cannot be a witness, without precisely capturing the behavior of new a state exists that includes initial. The following abstract state corresponds to two distinct `RemoverActivity` instances a and a' operating on the same `FeedRemover` task causing the exception by invoking `execute` twice. However, such an execution is impossible as both instances of `RemoverActivity` must have retrieved their `FeedRemover` from distinct calls to **`new`**. Without capturing the transitions new r and new r', there is nothing capturing that r and r' are disaliased and this state would erroneously include initial.

$$\begin{aligned}\mathsf{Fwk}_4\ \{&\mathsf{cb}\ a_2.\texttt{onCreate}() \twoheadrightarrow b_4 = \mathsf{ci}\ a_2.\texttt{makeButton}(a_2) \twoheadrightarrow b_3 = \mathsf{ci}\ a_2.\texttt{makeButton}(l_2) \twoheadrightarrow \\ &\mathsf{cb}\ a.\texttt{onCreate}() \twoheadrightarrow b_1 = \mathsf{ci}\ a.\texttt{makeButton}(a) \twoheadrightarrow b_2 = \mathsf{ci}\ a.\texttt{makeButton}(l_1) \twoheadrightarrow \\ &\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ci}\ t.\texttt{execute} \twoheadrightarrow b_1.\texttt{setEnabled}(\texttt{false}) \twoheadrightarrow \\ &\mathsf{cb}\ a.\texttt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\texttt{execute}() \twoheadrightarrow \mathsf{okhist}\}\end{aligned}$$

2.5 Abducing Targeted Models of Event-driven Frameworks For Verification

Given the difficulty in writing models by hand, the final contribution of this dissertation is a method of automatically generating framework models. Considering our motivating example, the primary challenge for a developer is to understand what behavior is possible while using a given callback or callin. Specifically for our example, whether `finish()` on the `Activity` or `setEnabled(false)` on the `Button` disables `onClick` preventing the defect. The same challenge applies to a program verifier since the specific behavior of the framework is hidden. This contribution specifically combines the verification and bug finding contributions from earlier with a technique to determine the best model based on observations of the framework's behavior. Specifically, the best framework model is the one that restricts framework behavior that would lead to a false alarm, does not restrict behaviors that may be possible, and is syntactically small. The Historia tool gives us a method of determining if a framework model restricts the framework behavior that leads to an alarm (true or false). Then, Wistoria provides the functionality to determine if that framework model is consistent with known reachable locations such as the crashing version of our application or the "Cancel" button's click listener. For example, crash reports like Figure 1.1 can indicate that framework models like History Implication 3 should be avoided. With enough reachable locations, a framework model will be sound with respect to the behavior of the real framework. Therefore, the challenge in this section is to effectively search the space of possible framework models.

We formulate the framework model abduction problem as an instance of programming-by-example, a program synthesis technique that finds a candidate program given a set of inputs and outputs. The inputs are the reachable and target issues and the outputs are either showing reachable issues to be reachable or proving the target issue unreachable. The key to the problem of programming-by-example is the need to resolve ambiguity over candidate programs [57]. For the problem of synthesizing framework models based on a finite number of reachable locations, the ambiguity is whether a given restriction on the framework behavior is sound. Reachable locations are necessarily weaker than the recorded runtime traces of other techniques. Weaker runtime data means that the hypothesis space of framework models needs to be smaller. We reduce the hypothesis space by considering **connected framework models** or framework models where the application can observe data flows explaining effects.

This technique takes three inputs: (1) A target location that the user would like to prove correct (i.e. unreachable), and (2) A corpus of known reachable locations in arbitrary applications that must remain reachable for a framework model to be sound. (3) A template framework model consisting of history implications containing holes as well as a set of messages that may be used when searching framework models. For simplicity we refer to a tuple containing an application and a location as an **issue** regardless of whether it is a crash, reachable location, or safe location. A **reachable issue** is a location that is known to be reachable when the application is running (i.e. a crash, log statement, sample from a profiler, etc). Similarly, we refer to an issue that is proven safe as a **unreachable issue**.

Upon successfully proving an issue unreachable, this technique will produce a framework model, that, if sound, proves the issue unreachable. If the issue target issue is reachable and there are a sufficiently large sample of known reachable locations, then this technique will not be able to produce a framework model and results in an alarm.

At the core of this technique, it can be thought of as an intelligent "syntactic enumeration" of candidate framework models. First, a candidate framework model is proposed and then it may be tested against the reachable and target issues. Candidate framework models are automatically rejected if they cannot be shown to be consistent with the issues. Even enumerating models around the size of our motivating example would be extremely computationally expensive. Therefore, the intelligence comes in with methods to avoid checking framework models that are not useful for proving the property safe. We first introduce a technique of enumerating CBCFTL formula with subsection 2.5.1 and then we further restrict the space of framework models by using a pointer analysis to avoid trivially false clauses.

2.5.1 Exploring the Space of Framework Models with Syntax Guided Synthesis

Figure 2.9 illustrates the process for enumerating framework models. At the top, the starting model of $\mathsf{ci}\ l.\texttt{onClick}() \;\Box\!\!\rightarrow\; \mathsf{O}\ b = \mathsf{ci}\ a.\texttt{makeButton}(l) \;\wedge\; \blacksquare$ represents the starting knowledge of the developer. That is, in order to have an `onClick` callback on a listener object l, then the button must have been created with that listener at some point with `makeListener`. Subsequently, our technique can do the hard part by figuring out what other conditions should be represented by $\blacksquare$. The formula that this technique substitutes for $\blacksquare$ automatically chooses between History Implication 1 and History Implication 3.

Each step shown in the figure is a choice of expanding a hole in a history implication with a temporal operator or boolean connective. For example, the second row of the figure shows a `finish` or `setEnabled` message with a "Once" operator. Each message contained within a temporal operator will also enumerate the aliasing possibilities with other messages in the history implication. These expansions also consider the other temporal operators and boolean connectives. For example, $\blacksquare \vee \blacksquare$ represents expansions where a logical or is next in the parse tree. We represent the other expansions with "...". Whenever a model with no holes is found, we can test the ability to prove the target issue and consistency with the reachable issues. For example, the candidate model containing `0 ci` a.`finish()` can be eliminated as unsound because it can unsoundly prove the buggy version of our example. However, it becomes clear that the space of models is huge even for a simple hole. With hundreds of thousands of messages possible in the Android framework and many containing multiple arguments, the space of models is intractable. How can we intelligently cut down this search space?

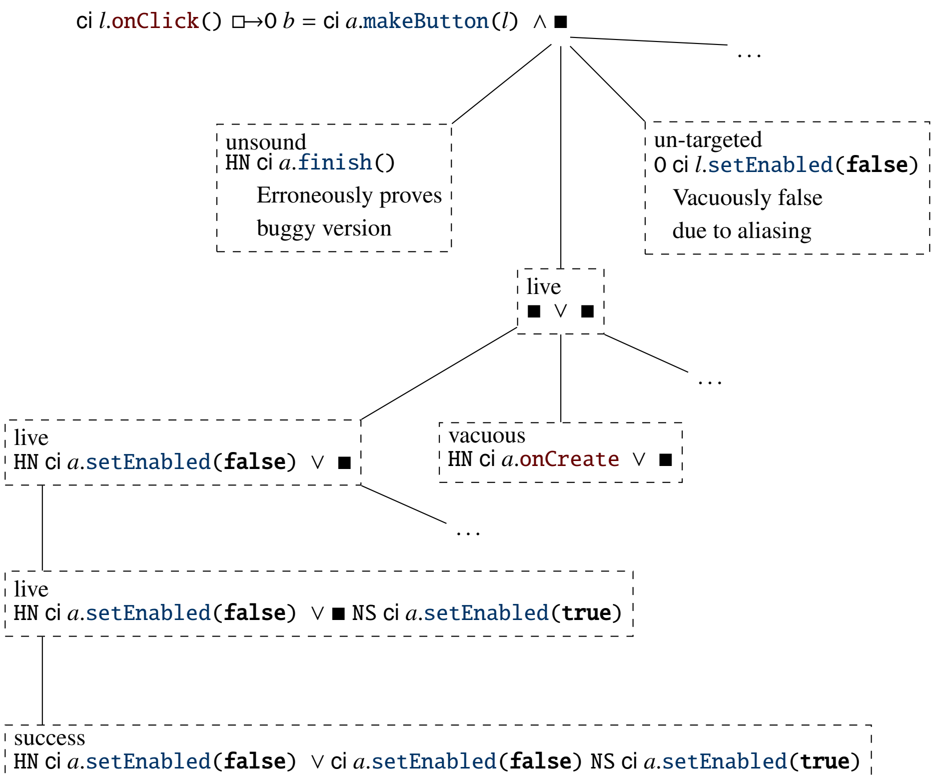

Figure 2.9: We enumerate the syntactic expansions of CBCFTL formula and test them against our corpus of reachable locations and the target location to verify. On each candidate model, the model may be: (1) Unsound if known reachable locations are not reachable under the model, (2) Un-targeted if a clause can never be true in the available issues, (3) Live if there is still a possibility of expanding, or (4) Successful if the model proves the location while not excluding known reachable locations.

To address the problem of searching the large space of possible framework models, we look at three major optimizations: (1) By only considering "connected" framework models, we avoid many types of vacuous models. (2) As a more sound optimization, we can compute over and under approximations of framework models containing holes subsection 2.5.3 and, (3) We can over-approximate the space of **targeted** framework models using message graphs subsection 2.5.4. As we will see in section 6.4, these techniques perform well on benchmarks from the Historia work.

2.5.2 Connected Framework Models

A key observation of this paper is that good framework models restricting the message history explain that restriction by sharing arguments. Since there may be an unbounded number of UI objects such as buttons, an action that can enable or disable an `onClick` message should **select** which button and click listeners are inovled. Such selection is done through arguments shared between messages like the listener object l or the button object b.

We can imagine a candidate history implication $\mathsf{ci}\ l.\texttt{onClick}() \;\square\!\!\rightarrow\; \mathsf{0}\ b = \mathsf{ci}\ a.\texttt{makeButton}(l) \;\wedge\; \texttt{HN}\ \mathsf{ci}\ b_2.\texttt{setEnabled}(\textbf{false})$ where the button b_2 used for `setEnabled(false)` is not necessarily the same as the one used for `makeButton`. It is possible to write an event-driven framework that prevents all button clicks if `setEnabled(false)` has been called on some button. However, such a behavior is impractical since an application typically only disables certain buttons at a time. For this reason, we simply assume that useful history implications always have some connection between messages that appear somewhere in the message history (i.e. not messages that are negated). Figure 2.10, shown below, specifically illustrates the connections for the button disabling history implication. In our experiments, we have not found a case where this assumption excludes a useful history implication.

$$\mathsf{cb}\ l.\mathtt{onClick}()\ \square\!\!\rightarrow \mathsf{O}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l) \land$$

$$(\mathtt{HN}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{\textbf{false}})\ \lor\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{\textbf{false}})\ \mathtt{NS}\ \mathsf{ci}\ b.\mathtt{setEnable}(\mathtt{\textbf{true}}))$$

Figure 2.10: A history implication is connected if there is a path between shared variables for every message starting at the target message. Such connected history implications ensure that a framework model explains the enabledness of a message. In this example, the connections explain which `onClick` listener is disabled when a given button is disabled.

2.5.3 Under and Over Approximation of Candidate Framework Models

The next optimization is over and under approximation of the framework models containing holes. These approximations are used to determine if a framework model with holes can possibly be expanded into a satisfying framework model. For example, the $\blacksquare$ in the formula $\mathsf{O}\ \mathsf{ci}\ l.\mathtt{onCreate}() \lor \blacksquare$ can not be expanded to a framework model able to prove the fixed version. This lower bound of imprecision is cause by the disjunction with `onCreate`. Since an `onCreate` always occurs during normal execution, this framework model can never sufficiently restrict the framework behavior in a way that proves the fixed version of our motivating example.

The formula on the right-hand side of the $\square\!\!\rightarrow$ of history implications are restricted to be in prenex normal form and negation normal form. That is, there is no arbitrary negation of formula, instead negation is pushed down into the individual temporal operators. This form allows us to easily compute under and over approximations of a history implication containing holes by replacing the hole with **`true`** or **`false`**, respectively. In our example, the formula $\mathtt{HN}\ \mathsf{ci}\ a.\mathtt{onCreate} \lor \blacksquare$ is labeled "vacuous" because $\mathtt{HN}\ \mathsf{ci}\ a.\mathtt{onCreate} \lor \mathtt{\textbf{false}}$ is too weak for proving the target issue. Nothing can constrain message histories more than **`false`**.

2.5.4 Targeting the Framework Models Using Message Graphs

A large space of framework models can be quickly eliminated by considering the possible aliasing between runtime messages within the applications in the issues. In the figure, we have the formula

$\mathsf{O}\ \mathsf{ci}\ l.\mathtt{setEnabled}(\mathtt{\textbf{false}})$ labeled as “un-targeted” because it can never be true in the available issues. If we consider the full history implication shown below, there is no message history where the temporal operator $\mathsf{O}\ \mathsf{ci}\ l.\mathtt{setEnabled}(\mathtt{\textbf{false}})$ can be true in either the buggy or fixed version of our example.

$$\mathsf{ci}\ l.\mathtt{onClick}() \mathrel{\square\!\!\rightarrow} \mathsf{O}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l)\ \wedge \mathsf{O}\ \mathsf{ci}\ l.\mathtt{setEnabled}(\mathtt{\textbf{false}})$$

The reason for this is subtle but important, it is not possible for the button object used by `setEnabled(false)` to alias the listener object passed to the first argument of `makeButton`. It can be seen that such an alias is not possible since one is of type `Button` and the other is of type `OnClickListener` which are not sub-classes of one another. Therefore, the formula $\mathsf{O}\ \mathsf{ci}\ l.\mathtt{setEnabled}(\mathtt{\textbf{false}})$ will be trivially equivalent to **`false`** in the target issue.

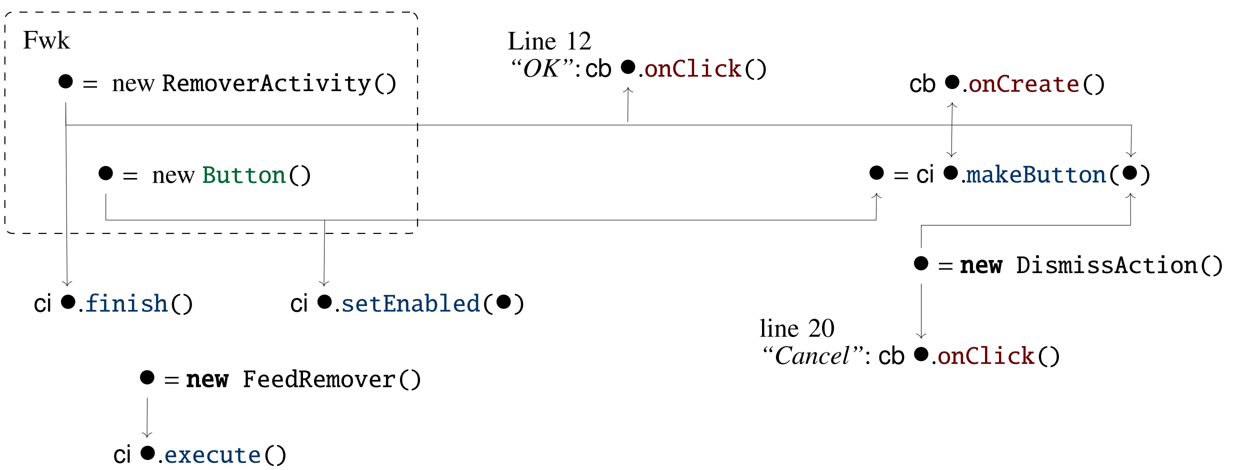

Figure 2.11: The message graph for the fixed version of Figure 1.1 shows each message that the application may emit at runtime (under any framework implementation) as nodes and edges between arguments if they may be aliased at runtime. We argue that messages that connect edges from existing messages in history implications are the most likely to be useful for proving a target issue.

In order to determine if arguments of messages may alias one another, we use a specialized pointer analysis called a message graph. This message graph over-approximates the aliasing between messages under any implementation of the framework (i.e. it uses the “top” model discussed earlier). Figure 2.11

shows the message graph computed from the applications in our example. Arrows in the graph flow from allocation sites (i.e. calls to new) to message arguments and return values. The meaning of this graph is that an argument in one message may only alias another message if they share an incoming allocation site. For example, the receiver of `onClick` can never alias `setEnabled`, meaning that we can determine the un-targeted history implication is always false before running the more precise and long-running Historia analysis.

We compute the message graph from the application combined with generated framework code which is suitable for an-off-the-shelf flow-insensitive pointer analysis. We can generate code that stands in for the allocation sites of the framework (shown in the Fwk box of Figure 2.11) and then artificially invoke callbacks such as `onCreate`. Finally, all allocation sites and callins write to a common framework location to allow flow of values through the framework to arguments of callbacks or the return value of callins. For our implementation, the final step is to run the SPARK [77] pointer analysis. The message graph is then defined by the pointer set at each program location that may emit a callback or callin message.

Chapter 3

Precisely Modeling Event-Driven Protocols and Callback Control Flow

In this section, we investigate the first steps of modeling event driven frameworks precisely enough to exclude crashes but soundly enough to accommodate observed behavior. Given a running application, we can record the interactions between the application and the framework then determine inclusion in a given framework model. Due to the difficulty of building an application-only static analysis, this chapter focuses on predictive trace verification to determine if a framework model is precise enough. Predictive trace verification records the interactions between an application and the framework in order to determine if it can be rearranged to find a crash. For example, Figure 3.1 shows an execution of the buggy version of our motivating example that does not crash. We can first ensure that framework models accomodate all such executions and second, we can duplicate and rearrange messages until a crash is observed.

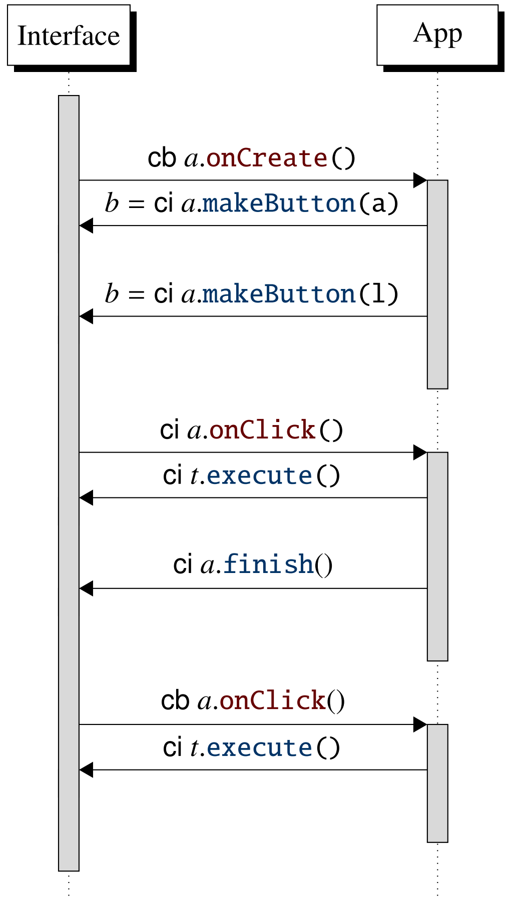

Figure 3.1: A non-crashing history of execution for the overview example 2.1. In this section, we show how models may be written that are precise enough to distinguish crashes and fixes among rearrangements of such executions while accommodating all observed framework behavior.

Following chapter 2, we want to capture the essence of the app-framework interface with respect to framework-imposed programming protocols while avoiding a specific representation of the event loop. To do so, we first formalize a small-step operational semantics for event-driven programs with an abstract machine model λ_{life} in section 3.1. The λ_{life} abstract machine draws on standard techniques, but explicitly highlights enabled events and disallowed callins to precisely define event-driven protocols. We then instrument this semantics to formalize the interface of the event-driven framework with an app, thereby defining the traces of the observable app-framework interface of a λ_{life} program.

We then use these semantics as a basis for, Lifestate, a precursor language to CBCFTL. Lifestate is presented in section 3.2. Recorded message histories with handwritten framework models can be used to check the model for soundness (validation) or be used in dynamic trace verification to find defects section 3.3. Finally, we evaluate these contributions by developing a sound framework model that can be used to dynamically verify applications but still validates on all recorded message histories.

This chapter includes portions of the ECOOP paper "Lifestate: Event-Driven Protocols and Callback Control Flow" [88].

3.1 Describing Callback Control-flow of an Application Without the Framework

We present a language that is intentionally minimalistic to center on capturing just the interface between event-driven frameworks and their client applications. By design, we leave out many aspects of real-world event-driven framework implementations (e.g., Android, Swing, or Node.js), such as typing, object-orientation, and module systems that are not needed for formalizing the dialogue between frameworks and their apps (cf. section 2.2). Our intent is to illustrate, through examples, that event-driven frameworks could be implemented in λ_{life} and that λ_{life} makes explicit the app-framework interface to define **observable traces** consisting of **back-messages** and **in-messages** (subsection 3.1.3).

$$\begin{array}{llll} \text{expressions} & e \in \textbf{Expr} ::= & \mathsf{bind}\ v_1\ v_2 \mid \mathsf{invoke}\ v \mid \mathsf{disallow}\ v \mid \mathsf{allow}\ v & \text{thunks and calls} \\ & & \mid \mathsf{enable}\ v \mid \mathsf{disable}\ v & \\ & & \mid \mathsf{force}\ \kappa & \text{events and forcing} \\ & & \mid v \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \cdots & \text{other expressions} \end{array}$$

functions $\lambda ::= x =>_g e$ packages $g ::= \mathsf{App} \mid \mathsf{Fwk}$ values $v \in \textbf{Val} ::= x \mid \lambda \mid \kappa \mid () \mid \cdots \mid \mathsf{thk}$

variables $x \in \textbf{Var}$ thunks $\kappa \in \textbf{Thunk} ::= \lambda[v]$ thunk stores $\mu, \nu ::= \cdot \mid \mu;\kappa$

continuations $k ::= \bullet \mid k \triangleright x.e \mid \kappa \mid k \gg \kappa$ states $\sigma \in \textbf{State} ::= \langle e, \mu, \nu, k\rangle \mid \mathsf{bad}$

(a) The syntax and the semantic domains.

$$\boxed{\sigma \to \sigma'}$$

$$\textsc{Enable}\ \frac{}{\langle \mathsf{enable}\ \kappa, \mu, \nu, k\rangle \to \langle \kappa, \mu;\kappa, \nu, k\rangle} \quad \textsc{Disable}\ \frac{}{\langle \mathsf{disable}\ \kappa, \mu;\kappa, \nu, k\rangle \to \langle \kappa, \mu, \nu, k\rangle} \quad \textsc{Event}\ \frac{\kappa \in \mu}{\langle v, \mu, \nu, \bullet\rangle \to \langle \mathsf{force}\ \kappa, \mu, \nu, \kappa\rangle}$$

$$\textsc{Disallow}\ \frac{}{\langle \mathsf{disallow}\ \kappa, \mu, \nu, k\rangle \to \langle \kappa, \mu, \nu;\kappa, k\rangle} \quad \textsc{Allow}\ \frac{}{\langle \mathsf{allow}\ \kappa, \mu, \nu;\kappa, k\rangle \to \langle \kappa, \mu, \nu, k\rangle}$$

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

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

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

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

Figure 3.2: λ_{life}, a core model of event-driven programs capturing **enabledness** of events and **disallowedness** of invocations.

3.1.1 Syntax: Enabling, Disabling, Allowing, and Disallowing

The syntax of λ_{life} is shown at the top of Figure 3.2a, which is a λ-calculus in a let-normal form. The first two cases of expressions e split the standard call-by-value function application into multiple steps (similar to call-by-push-value [75]). The bind λ v expression creates a thunk $\kappa = \lambda[v]$ by binding a function value λ with an argument value v. We abuse notation slightly by using λ as the meta-variable for function values (rather than as a terminal symbol). A thunk may be forced by direct invocation invoke κ—or indirectly via event dispatch. For example, let `t` be bound to an `OnClickListener` and `onClick` to an app-defined callback (e.g., Line 12 from Figure 2.1), then the direct invocation of a callback from the framework can be modeled by the two steps of binding and then invoking:

```
let cb = bind onClick l in invoke cb
```

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

The direct invocation expressions are mirrored with expressions for event dispatch. An enable κ expression **enables** a thunk κ for the external event-processing system (i.e., gives the system permission to force the thunk κ), while the disable κ expression **disables** the thunk κ. For example, let `l` be bound to an `OnClickListener` and `handleOnClick` to an internal framework-defined function for handling a post-execute event, then enqueuing such an event can be modeled by the two steps of binding then enabling:

```
let h = bind handleOnClick l in enable h
```

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

The remainder of the syntax is the standard part of the language:

values v, variable binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, and whatever other operations of interest $\cdots$ (e.g., arithmetic, tuples, control flow, heap manipulation). That is, we have made explicit the expressions to expose the app-framework interface and can imagine whatever standard language features in $\cdots$ in framework implementations. The values v of this expression language are variables x, function values λ, thunks κ, unit (), and whatever other base values of interest $\cdots$. Two exceptions are that (1) the currently active thunk is available via the thk identifier (see subsection 3.1.2) and (2) functions $x \Rightarrow_g e$ are tagged with a package g (see subsection 3.1.3).

3.1.2 Semantics: Protocol Violations

At the bottom of Figure 3.2a, we consider an abstract machine model enriched with an **enabled-events** store μ, and a **disallowed-calls** store ν. These are finite sets of thunks, which we write as a list $\kappa_1; \cdots; \kappa_n$. The enabled-events store μ saves thunks that are permitted to be forced by the event loop, while the disallowed-calls store ν lists thunks that are **not** permitted to be forced by invocation. These thunk stores make explicit the event-driven application-programming protocol (that might otherwise be implicit in, for example, flag fields and conditional guards).

A machine state $\sigma\colon \langle e, \mu, \nu, k\rangle$ consists of an expression e, enabled events μ, disallowed calls ν, and a continuation k. A continuation k can be the top-level continuation $\bullet$ or a continuation for returning to the body of a let expression, which are standard. Continuations are also used to record the active thunk via κ and $k \gg \kappa$ corresponding to the run-time stack of activation records. These continuation forms record the active thunk and are for defining messages and the app-framework interface in subsection 3.1.3. Since events occur non-deterministically and return to the main event loop, it is reasonable to assume that a state σ should also include a heap, and the expression language should have heap-manipulating operations through which events communicate. We do not, however, formalize heap operations since they are standard.

We define an operational semantics in terms of the judgment form $\sigma \to \sigma'$ for a small-step transition relation. In Figure 3.2b, we show the inference rules defining the reduction steps related to enabling-disabling, disallowing-allowing, invoking, creating, and finally forcing thunks. The rules follow closely the informal semantics discussed in subsection 3.1.1. Observe that Enable and Allow both permit a thunk to be forced, and Disable and Disallow remove the permission to be forced for a thunk. The difference between Enable and Disable versus Allow and Disallow is that the former pair modifies the enabled events μ, while the latter touches the disallowed calls ν.

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

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

A program e violates the event-driven protocol if it ends in the bad state from the initial state $\langle e, \cdot, \cdot, \bullet \rangle$. For example, the no-`execute`-call-on-already-executing-`AsyncTask` protocol can be captured by a $\mathsf{disallow}$. We let `execute` be a framework function (i.e., tagged with Fwk) that takes an `AsyncTask` `t`.

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

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

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

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

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

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

The app interacts with the framework only by "exchanging messages." The app-framework dialogue diagrams from Figures 2.2 and 2.3 depicts the notion of messages as arrows back-and-forth between the framework and the app. The framework invokes callbacks and returns from callins (the arrows from left to right), while the app invokes callins and returns from callbacks (the arrows from right to left). To formalize this dialogue, we label the observable transitions in the judgment form and define an **observable trace**—a trace formed only by these observable messages. Being internal to the framework, the Enable, Disable, Allow, and Disallow transitions are hidden, or unobservable, to the app.

In Figure 3.3, we define the judgment form $\sigma \xrightarrow[m]{} \sigma'$, which instruments our small-step transition relation $\sigma \to \sigma'$ with message m. We define a callback as an invocation that transitions from framework back-messages $m^{\mathbf{bk}} \in \Sigma^{\mathbf{bk}} ::= \mathsf{cb}\ \kappa \mid v = \mathsf{ciret}\ \kappa$ in-messages $m^{\mathbf{in}} \in \Sigma^{\mathbf{in}} ::= \mathsf{ci}\ \kappa \mid v = \mathsf{cbret}\ \kappa$

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

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

$$\textsc{ForceCallback}\quad \frac{(x' \Rightarrow_{\mathsf{App}} e')[v'] = \kappa \qquad \mathsf{Fwk} = \mathrm{pkg}(k)}{\langle \mathsf{force}\ \kappa, \mu, v, k\rangle \xrightarrow[\mathsf{cb}\ \kappa]{} \langle [\kappa/\mathsf{thk}][v'/x']e', \mu, v, k \gg \kappa\rangle}$$

$$\textsc{ForceCallin}\quad \frac{(x' \Rightarrow_{\mathsf{Fwk}} e')[v'] = \kappa \qquad \mathsf{App} = \mathrm{pkg}(k)}{\langle \mathsf{force}\ \kappa, \mu, v, k\rangle \xrightarrow[\mathsf{ci}\ \kappa]{} \langle [\kappa/\mathsf{thk}][v'/x']e', \mu, v, k \gg \kappa\rangle}$$

$$\textsc{ReturnCallin}\quad \frac{(x' \Rightarrow_{\mathsf{Fwk}} e')[v'] = \kappa \qquad \mathsf{App} = \mathrm{pkg}(k)}{\langle v, \mu, v, k \gg \kappa\rangle \xrightarrow[v = \mathsf{ciret}\ \kappa]{} \langle v, \mu, v, k\rangle}$$

$$\textsc{ReturnCallback}\quad \frac{(x' \Rightarrow_{\mathsf{App}} e')[v'] = \kappa \qquad \mathsf{Fwk} = \mathrm{pkg}(k)}{\langle v, \mu, v, k \gg \kappa\rangle \xrightarrow[v = \mathsf{cbret}\ \kappa]{} \langle v, \mu, v, k\rangle}$$

$$\textsc{InvokeDisallowed}\quad \frac{\kappa \in v}{\langle \mathsf{invoke}\ \kappa, \mu, v, k\rangle \xrightarrow[\mathsf{dis}\ \mathsf{ci}\ \kappa]{} \mathsf{bad}}$$

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

Figure 3.3: The instrumented transition relation $\sigma \xrightarrow[m]{} \sigma'$ defines the app-framework interface and observing the event-driven protocol.

to app code and a callin as an invocation from app to framework code. In λ_{life}, this definition is captured crisply by the execution context k in which a thunk is forced. In particular, we say that a thunk κ is a callback invocation $\mathsf{cb}\ \kappa$ if the underlying callee function is an app function (package App), and it is called from a framework function (package Fwk) as in rule ForceCallback. The thk(·) function inspects the continuation for the running, caller thunk. The pkg(·) function gets the package of the running thunk.

Analogously, a thunk κ is a callin $\mathsf{ci}\ \kappa$ if the callee function is in the Fwk package, and the caller thunk is in the App package via rule ForceCallin. For example, letting `handleOnClick` be a framework function (i.e., in package Fwk) and `onClick` be an App function, the observable transition from the framework to the app defines the forcing of `cb` as a callback:

```
let onClick = (l =>App ...) in
let handleOnClick = (l =>Fwk let cb = bind onClick l in invoke cb) in
```

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

Finally to make explicit protocol violations, we instrument the InvokeDisallowed rule to record the disallowed-callin invocations. These rules replace the corresponding rules Force, Return, and InvokeDisallowed from Figure 3.2b. For replacing the Force and Return rules, we elide two rules, one for each, where there is no switch in packages (i.e., $g' = \mathrm{pkg}(k)$ where g' is the package of the callee message). These "uninteresting" rules and the remaining rules defining the original transition relation $\sigma \to \sigma'$ not discussed here are simply copied over with an empty message label ϵ.

Observable Traces and Dynamic-Analysis Instrumentation.

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

Definition 1 (App-Framework Interactions as Observable Traces)**.** Let paths(e) be the path semantics of λ_{life} expressions e that collects the finite sequences of alternating state-transition-state $\sigma m \sigma'$ triples according to the instrumented transition relation $\sigma \xrightarrow[m]{} \sigma'$. Then, an **observable trace** is a finite sequence of messages $\omega : m_1 \ldots m_n$ obtained from a path by dropping the intermediate states and keeping the non-ϵ messages. We write $[\![e]\!]$ for the set of the observable traces obtained from the set of paths, paths(e), of an expression e.

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

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

3.2 Specifying Protocols and Modeling Callback Control Flow

CBCFTL, as presented in the overview, is a restricted form of Lifestate that is still able to express the properties we need and makes later decision procedures easier. Therefore, the explanation up until this point has focused on CBCFTL. Lifestate is an orthogonal method of capturing message histories. We have included it here for those who wish to dive deeper into the subject and understand the design of CBCFTL better.

The original design of lifestate was to capture the execution that leads to either an enable or disable of a callback. Figure 3.4 shows one half of the CBCFTL `onClick` history implication written in lifestate. On the left of the $\nrightarrow$or $\rightarrow$operator is a regular expression matching the conditions under which the target callback should be disabled or enabled respectively. The right-hand side is the target callback or callin. For disabling `onClick` the regular expression matches when `makeButton` was invoked and a `setEnabled(false)` has been invoked "most recently". "Most recently" is expressed with the regular expression $\mathsf{ci}\ b.\texttt{setEnabled}(\textbf{false}); (\neg\mathsf{ci}\ b.\texttt{setEnabled}(\textbf{true}) \land \cdot)$ combining the sequence operator ;, negation $\neg$, and an operator to match a single symbol $\cdot$.

$$b = \mathsf{ci}\ a.\texttt{makeButton}(l) \land \mathsf{ci}\ b.\texttt{setEnabled}(\textbf{false}); (\neg\mathsf{ci}\ b.\texttt{setEnabled}(\textbf{true}) \land \cdot) \nrightarrow \mathsf{cb}\ l.\texttt{onClick}()$$

Figure 3.4: Lifestate was the predecessor and inspiration for the more concise CBCFTL.

As we will see in the next chapter, CBCFTL is able to represent the properties that applications generally rely on while being much easier for both writing models and reasoning about. The most important reason Lifestate was replaced with CBCFTL is that the variables in Lifestate are implicitly quantified as "for all". Such quantification made abstracting the program state extremely difficult due to quantifier alternation. As a more minor reason, we found that certain patterns appeared in almost every Lifestate model such as the "most recently" pattern. Additionally, it was often difficult to for the developer of a framework model to avoid inconsistent models that both enabled and disabled a message simultaneously. The details of Lifestate are explained in this chapter, both because this was the language used for the experiments in section 3.4 and because certain properties could be expressed in Lifestate that are not possible in CBCFTL.

Lifestate for Specifying Protocols and Modeling Callback Control Flow

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

The main point in these definitions is that lifestate modeling of callback control flow can only depend on what is observable at the app-framework interface. Furthermore, the concrete semantic foundation given by λ_{life} leads to a careful definition of soundness and precision and a basis for model validation and predictive-trace verification (section 3.3).

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

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

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

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

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

Figure 3.5: This transition system defines an abstraction of the framework-internal state consistent with an observable trace ω with respect to a framework abstraction $\boldsymbol{S}$. The abstract state $\hat{\sigma}$ contains a store of permitted back-messages $\hat{\mu}$ and a store of prohibited in-messages $\hat{\nu}$, corresponding to an abstraction of enabled events and disallowed calls, respectively. The meaning of the framework abstraction $\boldsymbol{S}$ is captured by the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}$, which determine how an abstract store changes on a new message.

Abstracting Framework-Internal State by Observing Messages In Figure 3.5, we define the transition system that abstracts the framework-internal state consistent with an observable trace ω. An abstract state $\langle \hat{\mu}, \hat{\nu}, \omega \rangle$ contains a store of **permitted back-messages** $\hat{\mu}$ and a store of **prohibited in-messages** $\hat{\nu}$. What the transition system captures are the possible traces consistent with iteratively applying a framework abstraction $\boldsymbol{S}$ to the current abstract state: it performs a transition with a back-message $m^{\mathbf{bk}}$ only if $m^{\mathbf{bk}}$ is permitted $m^{\mathbf{bk}} \in \hat{\mu}$, and a transition with an in-message $m^{\mathbf{in}}$ only if $m^{\mathbf{in}}$ is not prohibited $m^{\mathbf{in}} \notin \hat{\nu}$. The trace ω in an abstract state saves the history of messages observed so far. In the most general setting for modeling the event-driven framework, the transition system can update the stores $\hat{\mu}$ and $\hat{\nu}$ as a function of the history of the observed messages ω. These updates are formalized with the store-update functions $\mathrm{upd}^{\mathbf{bk}}_{\boldsymbol{S}}(\omega, \hat{\mu})$ and $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ that define an abstraction $\boldsymbol{S}$ of the event-driven framework describing both its application-programming protocol and its callback control flow. A framework abstraction $\boldsymbol{S}$ also defines the initial abstract state $\langle \hat{\mu}^{\mathrm{init}}_{\boldsymbol{S}}, \hat{\nu}^{\mathrm{init}}_{\boldsymbol{S}}, \epsilon \rangle$ that contains the initial (abstract) state of the stores of the permitted back-messages and prohibited in-messages.

The semantics $[\![\boldsymbol{S}]\!]$ of a framework abstraction $\boldsymbol{S}$ is the set of observable traces of the transition system defined in Figure 3.5 instantiated with $\boldsymbol{S}$. We get sequences of states from the transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$, read the observable trace ω from the final state, and form a set of all such observable traces.

Definition 2 (Soundness of a Framework Abstraction)**.** (1) A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a λ_{life} program e if $[\![e]\!] \subseteq [\![\boldsymbol{S}]\!]$; (2) A framework abstraction $\boldsymbol{S}$ is a sound abstraction of a framework implementation $\langle \mathbf{Fun}_{\mathsf{Fwk}}, \lambda_{\mathbf{init}} \rangle$ if and only if $\boldsymbol{S}$ is sound for every possible program e that uses the framework implementation $\langle \mathbf{Fun}_{\mathsf{Fwk}}, \lambda_{\mathbf{init}} \rangle$.

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

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

$$\begin{gathered}\text{parametrized messages} \quad \boldsymbol{m} ::= \mathsf{cb}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{ciret}\ \lambda[\boldsymbol{p}] \mid \mathsf{ci}\ \lambda[\boldsymbol{p}] \mid \boldsymbol{p}' = \mathsf{cbret}\ \lambda[\boldsymbol{p}] \\ \text{lifestate rules} \quad \boldsymbol{s} ::= \boldsymbol{r} \to \boldsymbol{m} \mid \boldsymbol{r} \nrightarrow \boldsymbol{m} \\ \text{lifestate abstractions} \quad \boldsymbol{S} ::= \cdot \mid \boldsymbol{s}\boldsymbol{S} \\ \text{trace matchers: regular expressions of parametrized messages} \quad \boldsymbol{r} \\ \text{symbolic variables} \quad \hat{x} \in \mathbf{SVar} \qquad \text{parameters} \quad \boldsymbol{p} \in \mathbf{SVar} \cup \mathbf{Val} \qquad \text{binding maps} \quad \theta ::= \cdot \mid \theta, \hat{x} \mapsto v\end{gathered}$$

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

$$\begin{aligned}\mathrm{upd}^{\mathbf{bk}}_{S}(\omega, \hat{\mu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{bk}} \;\middle|\; \mathrm{consistent}_{S}(\omega) \wedge \left(\neg\, \mathrm{prohibit}_{S}(\omega, m^{\mathbf{bk}}) \wedge (\mathrm{permit}_{S}(\omega, m^{\mathbf{bk}}) \vee m^{\mathbf{bk}} \in \hat{\mu})\right) \right\} \\ \mathrm{upd}^{\mathbf{in}}_{S}(\omega, \hat{\nu}) &\stackrel{\text{def}}{=} \left\{ m^{\mathbf{in}} \;\middle|\; \mathrm{consistent}_{S}(\omega) \to \left(\neg\, \mathrm{permit}_{S}(\omega, m^{\mathbf{in}}) \wedge (\mathrm{prohibit}_{S}(\omega, m^{\mathbf{in}}) \vee m^{\mathbf{in}} \in \hat{\nu})\right) \right\} [1ex] \\ \mathrm{permit}_{S}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \to \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\ \mathrm{prohibit}_{S}(\omega, m) &\stackrel{\text{def}}{=} \exists \boldsymbol{r} \nrightarrow \boldsymbol{m} \in \boldsymbol{S}, \exists \theta, (\omega, \theta \models \boldsymbol{r}) \wedge \theta(\boldsymbol{m}) = m \\ \mathrm{consistent}_{S}(\omega) &\stackrel{\text{def}}{=} \forall m \in \Sigma, (\mathrm{permit}_{S}(\omega, m) \leftrightarrow \neg\, \mathrm{prohibit}_{S}(\omega, m))[1ex]\end{aligned}$$

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

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

Figure 3.6: Lifestate is a language for simultaneously specifying event-driven protocols and modeling callback control flow in terms of the observable app-framework interface.

A Lifestate Abstraction.

We arrive at lifestates by instantiating the framework abstraction $\boldsymbol{S}$ in a direct way as shown in Figure 3.6. To describe rules independent of particular programs or executions, we **parametrize messages** with symbolic variables $\hat{x} \in \mathbf{SVar}$. The definition of the parametrized messages $\boldsymbol{m}$ is parallel to the non-parameterized version but using parameters instead of simply concrete values v. We call a message $\boldsymbol{m}$ **ground** when it does not have symbolic variables (from **SVar**), and we distinguish the ground and parameterized messages by using normal m and bold $\boldsymbol{m}$ fonts, respectively. For example, the parametrized callback-invocation message $\mathsf{cb}\ \lambda[\hat{x}]$ specifies that a callback function λ is invoked with an arbitrary value from **Val**. The variable $\hat{x}$ can be used across several messages in a rule, expressing that multiple messages are invoked with, or return, the same value.

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

The semantics of lifestates is given by a choice of store-update functions $\mathrm{upd}_{\boldsymbol{S}}^{\mathbf{bk}}$ and $\mathrm{upd}_{\boldsymbol{S}}^{\mathbf{in}}$ in Figure 3.6b and the abstract transition relation $\hat{\sigma} \longrightarrow \hat{\sigma}'$ defined previously in Figure 3.2. The store-update functions work intuitively by matching the given trace ω against the matchers $\boldsymbol{r}$ amongst the rules in $\boldsymbol{S}$ and then updating the store according to the matching rules $\{\boldsymbol{s_1}, \ldots, \boldsymbol{s_n}\} \subseteq \boldsymbol{S}$.

To describe the store-update functions in Figure 3.6b, we write $\omega, \theta \models \boldsymbol{r}$ to express that a trace ω and a binding θ satisfy a regular expression $\boldsymbol{r}$. The definition of this semantic relation is standard, except for parametrized messages $\boldsymbol{m}$. Here, we explain this interesting case for when the trace ω and the binding θ satisfy the regular expression $\boldsymbol{m}$ (i.e., $\omega, \theta \models \boldsymbol{m}$):

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

A necessary condition for $\omega, \theta \models \boldsymbol{m}$ is, for example, that θ must assign a value to all the variables in $\boldsymbol{m}$, to get a ground message, and the message must be equal to the trace ω. Note that, if there is no such ground message for $\boldsymbol{m}$ with the binding θ, then $\omega, \theta \not\models \boldsymbol{m}$.

Now, the function $\mathrm{upd}_{\boldsymbol{S}}^{\mathbf{bk}}(\omega, \hat{\mu})$ captures how the state of the permitted back-messages store $\hat{\mu}$ changes according to the rules $\boldsymbol{S}$. As a somewhat technical point, a back-message can only be permitted if the rules $\boldsymbol{S}$ are **consistent** with respect to the given trace ω (i.e., $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$). The $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ predicate holds iff there are no rules that permits and prohibits m for the same message m and trace ω. Then, if the predicate $\mathrm{consistent}_{\boldsymbol{S}}(\omega)$ is true, the back-message $m^{\mathbf{bk}}$ must not be prohibited given the trace ω (i.e.,

$\neg\,\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}})$). Finally, if back-message $m^{\mathbf{bk}}$ is not prohibited, either it is permitted by a specification for this trace ω (i.e., $\mathrm{permit}_{\boldsymbol{S}}(\omega, m^{\mathbf{bk}})$) or it was already permitted in the current store $\hat{\mu}$ (i.e., $m^{\mathbf{bk}} \in \hat{\mu}$). The function $\mathrm{upd}^{\mathbf{in}}_{\boldsymbol{S}}(\omega, \hat{\nu})$ is similar, but it is defined for the prohibited in-messages store $\hat{\nu}$. An in-message $m^{\mathbf{in}}$ is prohibited first if the rules are not consistent. Then, if the rules are consistent, the in-message $m^{\mathbf{in}}$ must not be permitted by this trace, and either it is prohibited by a rule for this trace or the in-message was already prohibited in the current store $\hat{\nu}$. The auxiliary predicates $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ and $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ formally capture these conditions. The $\mathrm{permit}_{\boldsymbol{S}}(\omega, m)$ predicate is true iff there is a rule $\boldsymbol{r} \to \boldsymbol{m}$ in the specification $\boldsymbol{S}$ that permits a message $\boldsymbol{m}$ and a binding θ, such that the trace and the binding satisfy the regular expression ($\omega, \theta \models \boldsymbol{r}$), and the ground message permitted by the rule $\theta(\boldsymbol{m})$ is m. The $\mathrm{prohibit}_{\boldsymbol{S}}(\omega, m)$ predicate is analogous but for prohibit rules.

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

3.3 Dynamic Reasoning with Lifestates

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

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

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

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

The language $[\![S]\!]$ of a ground specification S can be represented with a finite transition system since the set of messages in S is finite, and lifestate rules are defined using regular expressions. We then pose the validation problem as a model checking problem that we solve using off-the-shelf symbolic model checking tools [26]. The transition system that we check is the parallel composition (i.e., the intersection of the languages of transition systems) of the transition system that accepts only the trace ω and the transition system $\hat{\sigma} \longrightarrow \hat{\sigma}'$ parametrized by the grounded lifestate abstraction S. The lifestate abstraction S is valid if and only if the composed transition system reaches the last state of the trace ω.

Dynamic Lifestate Verification

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

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

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

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

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

3.4 Empirical Evaluation

We implement our approach for Android in the Verivita tool that (i) instruments an Android app to record observable traces, (ii) validates a lifestate model for soundness against a corpus of traces, and (iii) assesses the precision of a lifestate model with dynamic verification. We use the following research questions to demonstrate that lifestate is an effective language to model event-driven protocols, and validation is a crucial step to avoid unsoundness.

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

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

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

Additionally, we considered the feasibility of continuous model validation. The bottom line is that we could validate 96% of the traces within a 6 minute time budget.

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

Table 3.1: Precision of callback control-flow models. The **sensitive callin** column lists protocol properties by the callin that crashes the app when invoked in a bad state. We collect a total of 50 traces from 5 applications with no crashes. The **sensitive** column lists the number of traces where the application invokes a sensitive callin. To provide a baseline for the precision of a model, we count the number of traces without a manually-confirmed real bug in the **verifiable** column. There are four columns labeled **verified** showing the number and percentage of verifiable traces proved correct using different callback control-flow models. The **lifestate** columns capture our contribution. The **lifecycle++** columns capture the current practice for modeling the Android framework. The **bad** column lists the number of missed buggy traces and is discussed further in **RQ2**.

RQ1: Lifestate Precision.

The bottom line of Table 3.1 is that lifestate modeling is essential to improve the percentage of verified traces to 83%—compared to 57% for lifecycle++ and 27% for lifecycle modeling.

Methodology. We collect execution traces from Android apps and compare the precision obtained verifying protocol violations with four different callback control-flow models. The first three models are expressed using different subsets of the lifestate language. The **top** model is the least precise (but clearly sound) model where any callback can happen at all times. The **lifecycle** model represents the most precise callback control-flow model that we can express only using callbacks only. For example, the lifecycle model of `Activity` would only follow the black lines in the figure shown by Figure 2.4. The **lifestate** model uses the full lifestate language, and hence also in-messages such as `setEnabled` and `makeButton`. It represents the most precise model that we can represent with lifestate. To faithfully compare the precision of the formalisms, we improved the precision of the lifecycle and lifestate models minimizing the false alarms from verification. And at the same time, we continuously run model validation to avoid unsoundnesses, as we discuss below in **RQ3**. As a result of this process, we modeled the behavior of several commonly-used Android classes, including `Activity`, `Fragment`, `AsyncTask`, `CountdownTimer`, `View`, `PopupMenu`, `ListView`, and `Toolbar` and their subclasses. Excluding similar rules for subclasses, this process resulted in a total of 167 lifestate rules.

We further compare with an instance of a **lifecycle++** model, which refines component lifecycles with callbacks from other Android objects. Our model is a re-implementation of the model used in FlowDroid [11] that considers the lifecycle for the UI components (i.e., `Activity` and `Fragment`) and bounds the execution of a pre-defined list of callback methods in the active state of the `Activity` lifecycle. Figure 2.4 shows an example of edges added by lifecycle++ with the `onClick` callback attached in the `Activity` lifecycle. In the lifecycle only model, callbacks like `onClick` may happen at any time.

To find error-prone protocols, we selected **sensitive callins**, shown in the first column of Table 3.1, that frequently occur as issues on GitHub and StackOverflow [42, 110, 7, 94, 123, 122]. We then specify the lifestate rules to allow and disallow the sensitive callins.

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

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

Discussion. In Table 3.1, we show the number of verified traces and the verified rates broken down by sensitive callins and different callback control-flow models—aggregated over all apps. As stated earlier, the precision improvement with lifestate is significant, essential to get to 83% verified. We also notice that the lifecycle model is only slightly more precise than the trivial top model (27% versus 22% verified rate). Even with unsoundnesses discussed later, lifecycle++ is still worse than the lifestate model, with 57% of traces proven.

Lifestate is also expressive enough to prove most verifiable traces—making manual triage of the remaining alarms feasible. We manually examined the 14 remaining alarms with the lifestate model, and we identified two sources of imprecision: (1) an insufficient modeling of the attachment of UI components (e.g., is a `View` in the `View` tree attached to a particular `Activity`?), resulting in 13 alarms; (2) a single detail on how Android options are set in the app's XML, resulting in 1 alarm. The former is not fundamental to lifestates but a modeling tradeoff where deeper attachment modeling offers diminishing returns on the verified rate while increasing the complexity of the model and verification times. The latter is an orthogonal detail for handling Android's XML processing (that allows the framework to invoke callbacks via reflection).

RQ2: Lifestate Generality.

The bottom line of Table 3.2 is that the lifestate model developed for **RQ1** as-is generalizes to provide precise results (with a verified rate of 86%) when used to verify traces from 121 previously unseen apps. This result provides evidence that lifestates capture general behaviors of the Android framework. While the lifecycle++ model verifies 76% of traces, it also misses 27 out of 64 buggy traces (i.e., has a 42% false-

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

Table 3.2: The table shows the precision results for the 1577 non-crashing traces that contained a sensitive callins from a total of 2202 traces that we collected from 121 distinct open source app repositories. We note that lifestate takes slightly longer than lifecycle; for this reason, lifestate performs slightly worse than lifecycle for execute. The bad column is 0 for models other than lifecycle++ because of continuous validation. Note that out of 64 total buggy traces, lifecycle++ missed 27 bugs (i.e., had a 42% false-negative rate).

negative rate).

Methodology. To get a larger corpus, we cloned 121 distinct open source apps repositories from GitHub that use at least one sensitive callin (the count combines forks and clones). Then, we generated execution traces using the Android UI Exerciser Monkey [5] that interacts with the app issuing random UI events (e.g., clicks, touches). We attempted to automatically generate three traces for each app file obtained by building each app.

Discussion. From Table 3.2, we see that the lifestate verified rate of 86% in this larger experiment is comparable with the verified rate obtained in **RQ1**. Moreover, lifestate still improves the verified rate with respect to lifecycle, which goes from 60% to 86%, showing that the expressivity of lifestate is necessary.

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

The verified rate for the lifecycle model is higher in this larger corpus (60%) compared to the rate in **RQ1** (27%), and the precision improvement from the top abstraction is more substantial (60% to 30% versus 27% to 22%). This difference is perhaps to be expected when using automatically-generated traces that may have reduced coverage of app code and bias towards shallower, “less interesting” callbacks associated with application initialization instead of user interaction. In these traces, it is possible that UI elements were not exercised as frequently, which would result in more traces provable solely with the lifecycle specification. Since coverage is a known issue for the Android UI Exerciser Monkey [9]), it was critical to have some evidence on deep, manually-exercised traces as in **RQ1**.

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

Yet from the **RQ1** triage, we found 2 buggy apps out of 5 total. These apps were Puzzles [19] and SwiftNotes [27]. Puzzles had two bugs, one related to `AlertDialog.show` and one for `AlertDialog.dismiss`. Swiftnotes has a defect related to `AlertDialog.show`.

In the **RQ2** corpus, we found 7 distinct repositories with a buggy app (out of 121 distinct repositories) from 64 buggy traces (out of 2202). We were able to reproduce bugs in 4 of the repositories and strongly suspect the other 3 to also be buggy. Three of the buggy apps invoke a method on `Fragment` that requires the `Fragment` to be attached. This buggy invocation happens within unsafe callbacks. Audiobug [129] invokes `getResources`. NextGisLogger [92] and Kistenstapeln [30] invoke `getString`. We are able to reproduce the Kistenstapeln bug.

Interestingly, one of the apps that contain a bug is Yamba [50], a tutorial app from a book on learning Android [51]. We note that the Yamba code appears as a part of three repositories where the code was copied (we only count these as one bug). The tutorial app calls `AlertDialog.dismiss` when an `AsyncTask` is finishing and hence potentially after the `Activity` object used in the `AlertDialog` is not visible anymore. We found similar defects in several actively maintained open source apps where callbacks in an `AsyncTask` object were used either to invoke `AlertDialog.show` or `AlertDialog.dismiss`. These apps included OSM Tracker [55] and Noveldroid [115]. Additionally, we found this bug in a binary library connected with the PingPlusPlus android app [102]. By examining the output of our verifier, we were able to create a test to concretely witness defects in 4 of these apps.

RQ3: Model Validation. The plot in Figure 3.7 highlights the necessity of applying model validation: lifecycle++ based on a widely used callback control-flow model does not validate (i.e., an unsoundness is witnessed) on 58% of 2183 traces (and the validation ran out of memory for 19 out of the total 2202 traces).

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

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

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

Discussion: lifecycle++ Validation. From the first bar of the plot in Figure 3.7, we see that the lifecycle++ model validates only 42% of the total traces, while validation fails in the remaining cases (58%). The bar shows the number of traces that we validated for at least one step, grouping them by validation status and cause of validation failure. From our manual triage, we identified 4 different broad causes for unsoundness: i) **outside the active lifecycle**: the model prohibits the execution of a callback outside the modeled active state of the `Activity`; ii) **wrong lifecycle automata**: the model wrongly prohibits the execution of an `Activity` or `Fragment` lifecycle callback; iii) **wrong start of the `Fragment` lifecycle**: the model prohibits the start of the execution of the `Fragment` lifecycle; iv) **no components interleaving**:

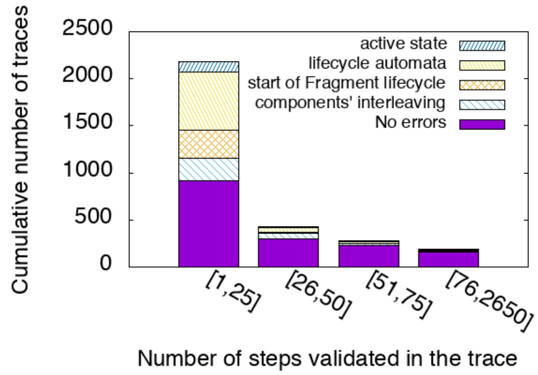

Figure 3.7: Results of the validation of the lifecycle++ model on all the traces. We plot the cumulative traces grouped by (intervals of) the number of steps validated. The number of traces are further divided into categories, either indicating that validation succeeded, "no errors," or the cause of failure of the validation process.

the model prohibits the interleaved execution of callbacks from different `Activity` or `Fragment` objects. The plot shows that the lifecycle++ model is not valid on 25% of the traces because it does not model the interleaving of components (e.g., the execution of callbacks from different `Activity` and `Fragment` objects cannot interleave) and the start of the `Fragment` lifecycle at an arbitrary point in the enclosing `Activity` object. With FlowDroid, such limitations are known and have been justified as practical choices to have feasible flow analyses [11]. But the remaining traces, 33% of the total, cannot be validated due other reasons including modeling mistakes. In particular, the FlowDroid model imprecisely captures the lifecycle automata (for both `Activity` and `Fragment`) and erroneously confines the execution of some callbacks in the active state of the lifecycle.

The other bars in the plot of Figure 3.7 show the number of traces we validated for more than 25, 50, and 75 steps, respectively. In the plot, we report the total number of steps in the execution traces that correspond to a callback or a callin that we either used in the lifestate or the lifecycle++ model, while we remove all the other messages. From such bars, we see that we usually detect the unsoundness of the lifecycle++ model "early" in the trace (i.e., in the first 25 steps). This result is not surprising since most of the modeling mistakes we found are related to the interaction with the lifecycle automata and can be witnessed in the first iteration of the lifecycle. We further discovered that the lifecycle++ model mostly validates shorter execution traces, showing that having sound models for real execution traces is more challenging.

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

Surprisingly, we identified and fixed several mistakes in our modeling of the `Activity` and `Fragment` lifecycle that are due to undocumented Android behaviors. An example of such behavior is the effect of `Activity.finish` and `Activity.startActivity` on the callback control flow for the `onClick` callback. It is unsound to restrict the enabling of `onClick` callbacks to the active state of the `Activity` lifecycle (i.e., between the execution of the `onResume` and `onPause` callbacks). This is the behavior represented with blue edges in Figure 2.4, what is typically understood from the Android documentation, and captured in the existing callback control-flow models used for static analysis.

We implemented a model where `onClick` could be invoked only when its `Activity` was running and found this assumption to be invalid on several traces. We inferred that the mistake was due to the wrong "bounding" of the `onClick` callbacks in the `Activity` lifecycle since in all the traces: i) the first callback that was erroneously disabled in the model was the `onClick` callback; and ii) the `onClick` callback was disabled in the model just after the execution of an `onPause` callback that appeared before in the trace, without an `onResume` callback in between (and hence, outside the active state of the `Activity`.) It turns out that both `finish` and `startActivity` cause the `Activity` to pause without preventing the pending `onClick` invocations from happening, as represented in the red edges connected to `onClick` in Figure 2.4. We validated such behaviors by writing and executing a test application and finding its description in several Stack Overflow posts [125, 124]. The fix for this issue is to detect the finishing state of the `Activity` and to not disable the `onClick` callback in this case.

3.5 Related Work

Several works [11, 17, 113, 117, 67, 109, 100, 114] propose different callback control-flow models. Many previous works, like FlowDroid [11] and Hopper [17], directly implement the lifecycle of Android components. While the main intention of these tools is to implement the lifecycle automata, in practice, they also encode some of the effects of callins invoked in the app code in an ad-hoc manner. For example, FlowDroid determines if and where a callback (e.g., `onClick`) is registered using a pre-defined list of callin methods and an analysis of the app call graph. Hopper implements the lifecycle callback control flow directly in a static analysis algorithm that efficiently explores the interleaving of Android components. In contrast, our work starts from the observation that reasoning about protocol violations requires capturing, in a first-class manner, the effects that invoking a callin has on the future execution of callbacks (and vice-versa).

Callback control-flow graphs [137] are graphs of callbacks generated from an application and a manually written model of the framework. Perez and Le [100] generate callback control-flow graphs with constraints relating program variables to callback invocations analyzing the Android framework. Such models can indirectly capture callin effects via the predicates on the program state. With lifestate, we carefully focus on what is observable at the app-framework interface so that lifestate specifications are agnostic to the internal implementation details of the framework. DroidStar [109] automatically learns a callback typestate automaton for an Android object from a developer-specified set of transition labels using both callbacks and callins symbols. Such automata specifically represent the protocol for a single object and, differently from lifestate, their labels are not parameterized messages. A callback typestate is thus a coarser abstraction than lifestate since it cannot express the relationships between different message occurrences that are required to describe multi-object protocols.

There exist other classes of framework models that represent different and complementary aspects of the framework than the callback control flow captured by lifestate. For example, Fuchs et al. [47] and Bastani et al. [13] represent the "heap properties" implicitly imposed by the framework. EdgeMiner [25] and Scandal [72] model the registration of callbacks. Droidel [18] also captures callback registration by modeling the reflection calls inside the Android framework code. Similarly, Pasket [68] automatically learns implementations of framework classes that behave according to particular design patterns.

While framework models have been extensively used to support static and dynamic analysis, not much attention has been paid to validating that the models soundly capture the semantics of the real framework. Wang et al. [133] recognized the problem of model unsoundness—measuring unsoundnesses in three different Android framework models. Unsoundnesses were found even using a much weaker notion of model validation than we do in this work. A significant advantage of lifestates is that we can validate their correctness with respect to any execution trace, obtained from arbitrary apps, because they speak generically about the app-framework interface.

There exist several programming languages for asynchronous event-driven systems, such as Tasks [46] and P [36]. In principle, such languages are general enough to develop event-driven systems such as Android. The purpose of our formalization is instead to provide a formalization that captures the app-framework interface.

The protocol verification problem for event-driven applications is related to typestate verification [91, 69, 45], but it is more complex since it requires reasoning about the asynchronous interaction of both callbacks and callins. Dynamic protocol verification is similar in spirit to dynamic event-race detection [84, 61, 15, 83], which predicts if there is an event data-race from execution traces. However, a lifestate violation differs from, and is not directly comparable to, an event data-race. A lifestate violation could manifest as a data race on a framework-internal field, but more commonly it results from encountering an undesirable run-time state within the framework.

3.6 Conclusion

We considered the problem of specifying event-driven application-programming protocols. The key insight behind our approach is a careful distillation of what is observable at the interface between the framework and the app. This distillation leads to the abstract notions of permitted messages from the framework to the app (e.g., enabled callbacks) and prohibited messages into the framework from the app (e.g., disallowed callins). Lifestate specification then offers the ability to describe the event-driven application-programming protocol in terms of this interface—capturing both what the app can expect of the framework and what the app must respect when calling into the framework. We evaluated our approach by implementing a dynamic lifestate verifier called Verivita and showed that the richness of lifestates are indeed necessary to verify real-world Android apps as conforming to actual Android protocols.

Chapter 4

Refuting Callback Reachability with Message History Logics

In this chapter, we explain the process of proving an application safe by showing no realizable message history can reach the assertion failure. Using the idea of enabledness of callbacks and callin returns, we can build an application-only static program analysis. The key benefit to this approach is that precise reasoning about application memory is now possible (as opposed to the dynamic verification in chapter 3).

First, we define the notion of an application-only transition system that records a **message history** during execution (subsection 4.1.1). This transition system builds upon the idea from LambdaLife that the enabled callbacks and callin returns may be predicted from the observable history. However, the key advantage to the application only transition system is that the original structure of an existing application is easier to preserve for analysis. 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 (subsection 4.1.2).

In support of MHPL, we also present a streamlined framework modeling language (as compared to Lifestate). This language is called Callback Control Flow Temporal Logic and is described in detail in section 4.2. Such streamlining of the framework modeling language allows a clean combination between abstract message histories and framework models that may be encoded as SMT formula. This encoding is presented in section 4.3. Finally, we evaluate these techniques by implementing them in a tool named

Historia and analyze real-world Android applications in section 4.4.

This chapter includes portions of the OOPSLA 2023 paper "Historia, Refuting Callback Reachability with Message History Logics" [89].

4.1 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 (subsection 4.1.1). 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 (subsection 4.1.2).

4.1.1 An Application-Only Transition System with Message Histories

Figure 4.1 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 4.1 from section 2.3). 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 $\mathsf{Fwk} \dashv\!\![\mathsf{cb}\; md(\overline{x})]\!\!\mapsto \ell$ from the framework back to the app, a callback return $\ell \dashv\!\![\mathsf{cbret}\; x'\; md(\overline{x})]\!\!\mapsto \mathsf{Fwk}$ from the app into the framework, or a callin invocation $\ell \dashv\!\![\mathsf{ci}\; x'\; md(\overline{x})]\!\!\mapsto \ell'$ 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 $\mathsf{cb}\, md(\overline{v})$, a callback return $\mathsf{cbret}\, v'\, md(\overline{v})$, or a callin invocation $\mathsf{ci}\, v'\, md(\overline{v})$. 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 $\mathsf{cbret}\, x'\, md(\overline{x})$ says that it returns the value in variable x' — for the corresponding callback $\mathsf{cb}\, md(\overline{x})$; 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 $\mathsf{ci}\, x'\, md(\overline{x})$, 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 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 $b ::= \mathsf{Fwk} \dashv\!\![\mathsf{cb}\, md(\overline{x})]\!\!\mapsto \ell \mid \ell \dashv\!\![\mathsf{cbret}\, x'\, md(\overline{x})]\!\!\mapsto \mathsf{Fwk} \mid \ell \dashv\!\![\mathsf{ci}\, x'\, md(\overline{x})]\!\!\mapsto \ell'$

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

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

variables x program locations $loc ::= \mathsf{Fwk} \mid \ell$ 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 $k ::= \mathsf{cb}\, md(\overline{v})$ app states $\varsigma ::= \ell : \rho$

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

C-CALLBACK-INVOKE

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

C-CALLBACK-RETURN

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

C-CALLIN-INVOKE

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

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 $\sigma_{\mathrm{init}} = \mathsf{Fwk} : \varepsilon \cdot \circ \cdot \rho_{\mathrm{init}}$ initial app store ρ_{init}

Figure 4.1: 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 $\omega \in \Omega$.

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 $\mathsf{Fwk} \dashv[\mathsf{cb}\, md(\overline{x})]\!\mapsto \ell$ via rule c-callback-invoke, the program state is at the framework location 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 $\ell \dashv[\mathsf{cbret}\, x'\, md(\overline{x})]\!\mapsto \mathsf{Fwk}$ 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 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 [1], 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 $\sigma \rightarrow_p^{\Omega} \sigma'$.

4.1.2 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. For example, ℓ_{14} $\{\mathsf{ciexn}\ t.\texttt{execute()} \twoheadrightarrow \mathsf{okhist} \cdot \texttt{this} \mapsto a * a.\texttt{remover} \mapsto t\}$ captures message histories where the callin `execute` emits an exception next and where the `remover` field and **`this`** variable point to objects t and a. We refute the reachability of the error condition with the judgment form $\vdash_p^S \widehat{\sigma}$ $\mathsf{unreach}$. 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." S is provided as input and represented by CBCFTL and is defined in section 4.2, 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 $\sigma_{\text{init}}\colon \mathsf{Fwk}\colon \varepsilon \cdot \circ \cdot \rho_{\text{init}}$ 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, $\vdash_S \widehat{\sigma}$ $\mathsf{excludesinit}$, 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, $\vdash_S \widehat{\omega}$ $\mathsf{excludesinit}$, holds if ε is not in the concretization of $\widehat{\omega}$.

4.1.2.1 An Abstract Semantics with Message Histories

In Figure 4.2, we define MHPL, which abstracts the application-only transition system from subsection 4.1.1, 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 error condition $\widehat{\sigma}$ 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 4.2, 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 θ is a mapping from symbolic variables in the abstract state to concrete values and abstract messages $\widehat{m} ::= \mathsf{cb}\, md(\overline{\hat{x}}) \mid \mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}}) \mid \mathsf{cbret}\, \hat{x}'\, md(\overline{\hat{x}})$ 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 $\widehat{\kappa} ::= \top \mid \widehat{\kappa_1} \bullet \mathsf{cb}\, md(\overline{\hat{x}})$

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
$$\frac{}{\vdash \{\mathsf{Fwk} : \mathsf{cb}\, md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}\ \mathsf{Fwk} \dashv[\mathsf{cb}\, md(\overline{x})]\mapsto \ell\ \{\ell : \widehat{\omega} \cdot \widehat{\kappa} \bullet \mathsf{cb}\, md(\overline{\hat{x}}) \cdot \widehat{\rho} * \ast\, \overline{x \mapsto \hat{x}}\}}$$

A-CALLBACK-RETURN
$$\frac{\widehat{\rho} = \widehat{\rho}' * x' \mapsto \hat{x}' * \ast\, \overline{x \mapsto \hat{x}}}{\vdash \{\ell : \mathsf{cbret}\, \hat{x}'\, md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \bullet \mathsf{cb}\, md(\overline{\hat{x}}) \cdot \widehat{\rho}\}\ \ell \dashv[\mathsf{cbret}\, x'\, md(\overline{x})]\mapsto \mathsf{Fwk}\ \{\mathsf{Fwk} : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}}$$

A-CALLIN-INVOKE
$$\frac{\widehat{\rho} = \widehat{\rho}' * \ast\, \overline{x \mapsto \hat{x}}}{\vdash \{\ell : \mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}}) \twoheadrightarrow \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho}\}\ \ell \dashv[\mathsf{ci}\, x'\, md(\overline{x})]\mapsto \ell'\ \{\ell' : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} * x' \mapsto \hat{x}'\}}$$

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

A-BOUNDARY-STEP
$$\frac{\widehat{\Sigma}(\mathrm{post}(b)) \vdash_S \widehat{\sigma} \qquad \vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\} \qquad \widehat{\sigma}' \vdash_S \widehat{\Sigma}(\mathrm{pre}(b))}{\widehat{\Sigma} \vdash_S b}$$

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

A-APP-STEP
$$\frac{\mathrm{app}(\widehat{\Sigma}(\mathrm{post}(t))) \vdash \widehat{\varsigma} \qquad \vdash \{\widehat{\varsigma}'\}\ t\ \{\widehat{\varsigma}\} \qquad \widehat{\varsigma}' \vdash \mathrm{app}(\widehat{\Sigma}(\mathrm{pre}(t)))}{\widehat{\Sigma} \vdash t}$$

$$\boxed{\widehat{\Sigma} \vdash_p^S \widehat{\sigma} \qquad \vdash_p^S \widehat{\sigma}\ \mathsf{unreach}}$$

A-INDUCTIVE
$$\frac{\widehat{\sigma} \vdash_S \widehat{\Sigma}(\mathrm{loc}(\widehat{\sigma})) \qquad \widehat{\Sigma} \vdash_S b \ \text{ for all } b \in p \qquad \widehat{\Sigma} \vdash t \ \text{ for all } t \in p}{\widehat{\Sigma} \vdash_p^S \widehat{\sigma}}$$

A-REFUTE
$$\frac{\widehat{\Sigma} \vdash_p^S \widehat{\sigma} \qquad \vdash_S \widehat{\Sigma}(\mathsf{Fwk})\ \mathsf{excludesinit}}{\vdash_p^S \widehat{\sigma}\ \mathsf{unreach}}$$

Figure 4.2: 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.

is used to ensure consistency between individual messages and other parts of the abstract state.

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 [64, 112] 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 $\widehat{\varsigma} ::= \ell \colon \widehat{\rho} \mid \cdots$ 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 $\widehat{\sigma} ::= loc \colon \widehat{\mu}$ is 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 $\widehat{\kappa} \bullet \mathsf{cb}\, md(\bar{\hat{x}})$. 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 1).

Lemma 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 $\mathsf{Fwk} \dashv\!\![\mathsf{cb}\, md(\bar{x})]\!\!\mapsto \ell$ and shows moving from an assertion on the abstract boundary stack $\widehat{\kappa} \bullet \mathsf{cb}\, md(\bar{\hat{x}})$ to a hypothetical next message in the abstract message history $\mathsf{cb}\, md(\bar{\hat{x}}) \twoheadrightarrow \widehat{\omega}$. In detail, it first asserts that the post-app memory has bindings for the callback parameters $\widehat{\rho} * \ast\, \overline{x \mapsto \hat{x}}$ and has the corresponding callback activation on top of the boundary stack $\widehat{\kappa} \bullet \mathsf{cb}\, md(\bar{\hat{x}})$. 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 2.3, the abstract state ℓ_{14} $\{\mathsf{ciexn}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot \mathbf{this} \mapsto a * a.\mathtt{remover} \mapsto t\}$ just after the invocation of the `call` callback will generate an abstract pre-state of $\mathsf{Fwk}_1\ \{\ \mathsf{cb}\ a.\mathtt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{okhist} \cdot a.\mathtt{remover} \mapsto t\}$ capturing that `onClick` happening next may reach the error.

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 $\widehat{\rho'} * x' \mapsto \hat{x}' ** \overline{x \mapsto \hat{x}}$. The overview example does not show a callback return for brevity, but an example would be the state just after the `onClick` in Figure 2.8 (note that we usually elide the $\top$ abstract message stack but include it here for the next step):

$$
\begin{array}{ll}
\mathsf{Fwk}_1\ \{ & \\
\quad \mathsf{cb}\ a.\mathtt{onClick}() \twoheadrightarrow \mathsf{ciexn}\ t.\mathtt{execute}() \twoheadrightarrow \mathsf{okhist} & \text{abstract message history} \\
\quad \cdot\ \top & \text{abstract boundary stack} \\
\quad \cdot\ a.\mathtt{remover} \mapsto t & \text{abstract app stores} \\
\} &
\end{array}
$$

From this state, a transfer over the return of the callback `onClick` produces an abstract pre-state representing the return happening next. There are three details elided in the overview: (1) The returns of callbacks are elided in order to keep the abstract state reasonable for the explanation (later, in section 4.3 it can be seen that messages that don't appear in the CBCFTL framework model may be elided soundly). (2) Only one aliasing case for the `RemoverActivity` is shown. Upon a jump back into the return of a callback such as `onClick`, each aliasing case with the return value and receiver must be considered for soundness. The easiest way to represent the aliasing possibilities is to create a fresh variable for any return value and the receiver object without equality constraints. In the abstract state above, we represent this fresh variable with a_2. (3) The call strings are not shown. Call strings are used to connect the arguments of a callback with the return value and to improve precision in the analysis (the latter improvement in precision is an existing technique [17]). The backwards transfer over the return results in an abstract pre-state that prepends a `onClick` to the abstract message history, pushes a `onClick` on the call string, and materializes the corresponding **`this`** variable.

$\ell_{17}\{$

cbret a_2.`onClick()` $\twoheadrightarrow$ cb a.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist abstract message history

$\cdot\ \top \bullet$ cbret a_2.`onClick()` abstract boundary stack

$\cdot\ a$.`remover` $\mapsto t *$ **`this`** $\mapsto a_2$ abstract app stores

$\}$

Finally, A-CALLIN-INVOKE is used to handle callins. In our example, the previous line in the application execution is Line 17`this.okButton.setEnabled(false);`. If a callin is assigned to a value, this judgment removes the program variable from the post app store. The callin `setEnabled` is **`void`**, so no program variable is removed. Next, fresh symbolic variables are added to the pre-state corresponding to the memory locations that were read by the callin. Since there is no `okButton` field, in the abstract state, it must be represented by the unconstrained memory. Therefore, the first step is to materialize the `okButton` field (i.e. add a a_2.`okButton` $\mapsto b_{\text{ok}}$ constraint on the memory). As a last step, A-CALLIN-INVOKE prepends the corresponding abstract message resulting in the following pre-state just before Line 17.

$\ell_{16}\{$

$\quad$ ci b_{ok}.setEnabled(false) $\twoheadrightarrow$ cbret a_2.onClick() $\twoheadrightarrow$ cb a.onClick() $\twoheadrightarrow$ ciexn t.execute() $\twoheadrightarrow$ okhist

$\quad \cdot \top \bullet$ cbret a_2.onClick()

$\quad \cdot\, a.\texttt{remover} \mapsto t * \textbf{this} \mapsto a_2 * a_2.\texttt{okButton} \mapsto b_{\text{ok}}$

$\}$

We write $\text{app}(\widehat{\sigma}) = \widehat{\varsigma}$ 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 $\text{post}(b)$ and $\text{pre}(b)$, respectively. Then, the rule chooses some $\widehat{\sigma}$ that over-approximates $\widehat{\Sigma}(\text{post}(b))$ — i.e., $\widehat{\Sigma}(\text{post}(b)) \vdash_S \widehat{\sigma}$, applies the abstract semantics for b — i.e., $\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}$, and checks that $\widehat{\Sigma}(\text{pre}(b))$ over-approximates $\widehat{\sigma}'$ — i.e., $\widehat{\sigma}' \vdash_S \widehat{\Sigma}(\text{pre}(b))$. 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 Thresher [16]). 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 2 (boundary-step soundness)**.** *If* $\widehat{\Sigma} \vdash_S b$ *and* $\langle \sigma', b \rangle \Downarrow^{\Omega} \sigma$ *such that* $\sigma \models_S \widehat{\Sigma}(\text{post}(b))$ *and* $\Omega \subseteq \Omega_S$*, then* $\sigma' \models_S \widehat{\Sigma}(\text{pre}(b))$.

To describe an inductive program invariant, we define the **may-witness** judgment form $\widehat{\Sigma} \vdash^S_p \widehat{\sigma}$ that says, "Abstract program-state invariants $\widehat{\Sigma}$ is inductive executing backwards from abstract program state $\widehat{\sigma}$ — at location $\text{loc}(\widehat{\sigma})$ — 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 (inductive soundness)**.** *If* $\widehat{\Sigma} \vdash^S_p \widehat{\sigma}$ *and* $\sigma' \rightarrow^{*\Omega}_p \sigma$ *such that* $\sigma \models_S \widehat{\sigma}$ *and* $\Omega \subseteq \Omega_S$*, then* $\sigma' \models_S \widehat{\Sigma}(\text{loc}(\sigma'))$.

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

Theorem 4 (refute soundness)**.** *If* $\vdash^S_p \widehat{\sigma}\ \mathsf{unreach}$ *and* $\sigma' \rightarrow^{*\Omega}_p \sigma$ *such that* $\sigma \models_S \widehat{\sigma}$ *and* $\Omega \subseteq \Omega_S$*, then* $\sigma' \neq \sigma_{\text{init}}$.

4.1.2.2 Abstract Interpretation with Message Histories

While we have described a checking system with the may-witness judgment form $\widehat{\Sigma} \vdash^S_p \widehat{\sigma}$, 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 ($\mathsf{okhist} \cdot \top \cdot \widehat{\rho}$ where $\widehat{\rho}$ negates the assertion condition) and $\perp$ 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,

ℓ $\dashv$[cbret x' $md(\overline{x})$]$\mapsto$ 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 4.3). Existing techniques can combine the message history SMT encoding with other parts of the abstract state [103].

4.2 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 2.3, a specification of realizable message histories must be able to express: (1) quantification over message values (e.g., the subscription object b from History Implication 1); and (2) constraints on what messages must have or have not happened in the past (e.g., `makeButton`, `setEnabled`, or `onCreate`). These requirements suggest CBCFTL should be a linear temporal logic (LTL) [85] interpreted over finite sequences (i.e., message histories) [52], with **first-order** quantification of message arguments, and **past-time** temporal operators [80]. 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 [119] 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 4.3, 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

($\vdash_S \widehat{\omega}$ $\mathsf{excludesinit}$) 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 [73, 96]. This section first gives the syntax and semantics of CBCFTL and then explains the encoding of the temporal formula fragment in Extended EPR.

4.2.1 A Temporal Logic for Expressing Realizable Message Histories

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

$$\text{temporal formula} \quad \widetilde{\omega} ::= \mathtt{O}\,\widetilde{m} \mid \mathtt{HN}\,\widetilde{m} \mid \widetilde{m}_2\,\mathtt{NS}\,\widetilde{m}_1 \mid \exists\hat{x}.\widetilde{\omega} \mid \forall\hat{x}.\widetilde{\omega} \mid \widetilde{\omega}_1 \land \widetilde{\omega}_2 \mid \widetilde{\omega}_1 \lor \widetilde{\omega}_2$$

$$\mid \hat{x}_1 = \hat{x}_2 \mid \hat{x}_1 \neq \hat{x}_2 \mid \text{true} \mid \text{false}$$

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

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

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

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

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

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

Figure 4.3: 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** $\widehat{m} \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}$. 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 $\mathtt{HN}\,\widetilde{m}$ or the propositional forms to limit quantifier alternation.

Figure 4.3 describes the CBCFTL syntax and semantics. A CBCFTL specification S is a conjunction of **history implications** $s ::= \widehat{m} \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}$. 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 4.3 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 ($\mathtt{O}\ \widetilde{m}$ for **O**nce, $\mathtt{HN}\ \widetilde{m}$ for **H**istorically **N**ot, and $\widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1$ 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 $\boxminus\!\!\rightarrow$ 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., we can place a _ where a variable would go in a message). 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 ($\boxminus\!\!\rightarrow$), and the only future temporal operator is history implication $\widehat{m} \boxminus\!\!\rightarrow \widetilde{\omega}$. 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 $\mathtt{HN}\ \widetilde{m}$ and equality/disequality of symbolic variables.

The most interesting part of temporal formulas $\widetilde{\omega}$ are the temporal operators: $\mathtt{HN}\ \widetilde{m}$ states that $\widetilde{m}$ has historically not (i.e., has never) occurred in the past, $\mathtt{O}\ \widetilde{m}$ that $\widetilde{m}$ occurred at least once in the past, and $\widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1$ 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 $\mathtt{O}\ \widetilde{m}$ operator is directly the **O**nce operator from past-time LTL, while $\mathtt{HN}\ \widetilde{m}$ and $\widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1$ are syntactic restrictions for the appropriate negations within **H**istorically and **S**ince (i.e., $\mathtt{HN}\ \widetilde{m} \stackrel{\mathrm{def}}{=} \mathbf{H}\ \neg\widetilde{m}$ and $\widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1 \stackrel{\mathrm{def}}{=} \neg\widetilde{m}_2\ \mathbf{S}\ \widetilde{m}_1$ 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 $i \in [0, \mathrm{len}(\omega))$, and we write $\omega[i]$ for the message at position i in ω and $\mathrm{len}(\omega)$ for the length of ω. A message history satisfies a history implication $\omega \models \widehat{m} \boxminus\!\!\rightarrow \widetilde{\omega}$ iff for all positions i in the message history (i.e., $i \in [0, \mathrm{len}(\omega)]$), 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, 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 $\mathtt{O}\ \widetilde{m}$, 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 $\mathtt{HN}\ \widetilde{m}$, all the messages at $k \in [0, i]$ must not model $\widetilde{m}$). The temporal operator $\widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1$ 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.

4.2.2 Encoding Temporal Formula Into Extended EPR

Here, we describe how we encode temporal formula into Extended EPR [73, 96], 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 $\mathsf{hist}\colon \textbf{HistIdx} \to \textbf{Msg}$ from history indices **HistIdx** to message instances **Msg**. To capture message instances, we use a function $\mathsf{msgname}\colon \textbf{Msg} \to \textbf{MsgName}$ from message instances to message names **MsgName** (i.e., representing the message kind, like cb, and the method name) and a function $\mathsf{msgargs}\colon \textbf{Msg} \to \textbf{ArgIdx} \to \textbf{Val}$ 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 $\leq\colon \textbf{HistIdx} \to \textbf{HistIdx} \to \textbf{Bool}$ and axiomatize a total ordering on **HistIdx** (Similar to Paxos Made EPR [96]), as well as an axiom for zero (i.e., $\forall idx \in$ **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.

4.3 Combining Abstract Message Histories with Callback Control-Flow

MHPL from subsection 4.1.2 depends on two judgments, excludes initial $\vdash_S \widehat{\omega}\ \mathsf{excludesinit}$ 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 4.2 for a definition of realizable message histories. For each abstract message history, 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 subsection 2.3.3). 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 4.2). In Figure 4.4, 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 4.4, 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 $\widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$.

$$\boxed{\vdash_S \widehat{\omega} \equiv \widetilde{\omega}} \qquad \frac{}{\vdash_S \mathsf{okhist} \equiv \mathrm{true}}\ \text{TEMPORAL-OKHIST} \qquad \frac{S, \widehat{m}_1 \vdash \widetilde{\omega}'_1 \qquad \vdash_S \widehat{\omega}_2 \equiv \widetilde{\omega}_2 \qquad \vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \fatsemi \widehat{m}_1}{\vdash_S \widehat{m}_1 \twoheadrightarrow \widehat{\omega}_2 \equiv \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2}\ \text{TEMPORAL-HYPMSG}$$

$$\boxed{S, \widehat{m} \vdash \widetilde{\omega} \qquad \vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}} \qquad \frac{\widehat{m}_1 \simeq_\vartheta \widehat{m}_2}{(\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}), \widehat{m}_1 \vdash [\vartheta]\widetilde{\omega}}\ \text{INSTANTIATE-YES} \qquad \frac{\widehat{m}_1 \not\simeq \widehat{m}_2}{(\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}), \widehat{m}_1 \vdash \mathrm{true}}\ \text{INSTANTIATE-NO}$$

$$\frac{}{\vdash \mathsf{O}\ \widetilde{m} \equiv \mathrm{Match}(\widetilde{m}, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\ \widetilde{m}) \fatsemi \widehat{m}}\ \text{QUOTIENT-ONCE} \qquad \frac{}{\vdash \mathtt{HN}\ \widetilde{m} \equiv \mathtt{HN}\ \widetilde{m} \wedge \mathrm{NotMatch}(\widetilde{m}, \widehat{m}) \fatsemi \widehat{m}}\ \text{QUOTIENT-HISTORICALLY-NOT}$$

$$\frac{}{\vdash \widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1 \equiv \mathrm{Match}(\widetilde{m}_1, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}_1, \widehat{m}) \wedge \widetilde{m}_2\ \mathtt{NS}\ \widetilde{m}_1 \wedge \mathrm{NotMatch}(\widetilde{m}_2, \widehat{m})) \fatsemi \widehat{m}}\ \text{QUOTIENT-NOT-SINCE}$$

Figure 4.4: 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.

Instantiation is the process of combining the hypothetical next message of an abstract message history with a history implication. For example, after instantiating the `onClick` in overview subsection 2.3.3 we get the following temporal formula:

$$\texttt{O}\ \mathsf{ci}\ t.\texttt{execute}() \wedge \texttt{O}\ b = \mathsf{ci}\ a.\texttt{makeButton}(a)\ \wedge$$

$$(\texttt{HN}\ \mathsf{ci}\ b.\texttt{setEnable}(\textbf{false})\ \vee\ \mathsf{ci}\ b.\texttt{setEnable}(\textbf{false})\ \texttt{NS}\ \mathsf{ci}\ b.\texttt{setEnable}(\textbf{true}))$$

For the instantiate judgment $S, \widehat{m} \vdash \widetilde{\omega}$, we show only the cases for single history implications $\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}$ 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 $\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}$ 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 4.4, $\vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$ 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, $\texttt{O}\ \widetilde{m}$, 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, $\texttt{HN}\ \widetilde{m}$, 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 $\widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1$ 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 inequalities of logic variables. For example Match($\mathsf{ci}\ t.\texttt{execute}(), \mathsf{cb}\ a.\texttt{onCreate}()$) would be false since the names are different. On the other hand, Match($\mathsf{cb}\ a.\texttt{onClick}(), \mathsf{cb}\ a_2.\texttt{onClick}()$) is equivalent to $a = a_2$ since the names are the same.

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 4.2.

We define procedures for judging excludes-initial $\vdash_S \widehat{\omega}$ $\mathsf{excludesinit}$ 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 (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 6 (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 7 (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 section A.3.

4.4 Empirical Evaluation

The major 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 which could hide defects. 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 1.1. (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) $\texttt{getAct}^{[8]}$ [44] — 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) $\texttt{execute}^{[10]}$ [41] — the app calls `execute` twice on the same `AsyncTask` object, ending in an exception. (3) $\texttt{dismiss}^{[12]}$ [43] — the app calls `dismiss` on a `Dialog` constructed with an `Activity` that is currently in the "created" state, ending in an exception. (4) $\texttt{finish}^{\mathsf{null}}$ [87] — 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 `Activity` (we call "nullable" the fields that can be set to null in a callback). (5) $\texttt{subs}^{\mathsf{null}}$ [58] — 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 section A.4) 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 4.1 for refuting callback reachability assertions in Android apps. Historia uses Soot [130] for loading the compiled app and to implement the application only control flow graph construction (similar to [2] but augmented with boundary transitions as discussed in subsection 4.1.1). Historia implements the encoding of section 4.3, and uses the Z3 SMT solver [33] to check the satisfiability of temporal formulas (section 4.2). Historia further processes callbacks in parallel and pre-empts calls to Z3 when possible for performance. We ran our experiments using Chameleon Cloud [71] using an AMD EPYC 7763 and 256 GB of RAM.

Pattern		cb, ret (n)	ci (n)	Historia specs (n)	cb (n)	cb (%)	cbret (n)	cbret (%)	ci (n)	ci (%)	time (s)	depth (n)	res	no-order res	eager res
Bug	getAct[8]	9	15	3[6,9]	3	33	1	11	2	13	9	3	(!)	(!)	false-(✓)
	execute[10]	7	6	3[7,11]	2	29	0	0	2	33	12	3	(!)	(!)	(!)
	dismiss[12]	7	6	2[13]	1	14	1	14	1	17	18	4	(!)	(!)	false-(✓)
	finish[null]	7	9	3[14,15,7]	3	43	1	14	3	33	52	3	(!)	(!)	false-(✓)
	subs[null]	9	17	5[16,17,18,19,20]	3	33	0	0	5	29	24	3	(!)	(!)	false-(✓)
Fix	getAct[8]	9	16	3[6,9]	3	33	1	11	3	19	16	3	(✓)	false-(!)	(✓)
	execute[10]	7	7	3[7,11]	2	29	0	0	3	43	27	4	(✓)	false-(!)	false-(!)
	dismiss[12]	7	6	2[13]	1	14	1	14	1	17	79	6	(✓)	false-(!)	(✓)
	finish[null]	7	10	3[14,15,7]	3	43	1	14	4	40	1800	5	(5)	false-(!)	(✓)
	subs[null]	9	18	5[16,17,18,19,20]	3	33	0	0	6	33	150	5	(✓)	false-(!)	(✓)
	total	66	73	10	22	**33**	6	**9**	21	**29**					

Table 4.1: 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 section A.4), 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 2.8 shows 4 steps back). Historia alarms on all the bug versions (circled exclamation point), and refutes reachability of the bug assertion for 4 out of the 5 bug-fixes (circled check mark). In the last case, Historia explored up-to 5 callbacks before timing out at 30 min (circled five). For the comparison with ideal tools using the "no-order" and "eager" modeling approaches, some results are labeled false-check for a false proof and false-exclamation for a false alarm.

4.4.1 RQ1: Proving Event-Driven Patterns

In Table 4.1, 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 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 [22] and Flowdroid [11] 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 (ⓘ) 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`^{null} benchmark). For the Fix version of the `finish`^{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`^{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.

4.4.2 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 [39] 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.

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`[8]	558	24	1,655	261	47	97	17	200	36	105	22
`execute`[10]	155	31	1,669	0	0	2	1	153	99	92	18
`dismiss`[12]	291	38	1,853	43	15	208	71	40	14	102	21
`finish`[null]	31	8	323	3	10	3	10	25	81	29	6
`subs`[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 4.2: 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.

Results are reported in Table 4.2. 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 4.3 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 $\otimes$ in the result (res) column.

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`[8]	473	5,440									⊗	60
OpenVPN	`getAct`[8]	938	9,628	2	49	5	5	1	167	2	0	✓	5
Seafile	`getAct`[8]	2,464	19,562	2	54	2	9	0	68	0	1	✓	5
Syncthing	`getAct`[8]	709	5,573								3,600	⊗	9
Navit	`finish`[null]	279	2,424								3,600	⊗	54
Connectbot	`dismiss`[12]	19*	248*	1	0	0	0	0	2	1	754	true-(!)	4
BatteryBot	`getAct`[8]	171	2,470	1	7	4	4	2	55	2	1	✓	3
Antennapod	`getAct`[8]	3,906	2,3056								3,600	⊗	35

Table 4.3: We sampled 8 locations that could not be proven from Table 4.2 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 section A.4. *The Connectbot benchmark here was a timeout in Table 4.2; we manually removed callbacks to help find the alarm and understand the bug, while the benchmarks were unmodified.

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).

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`[10] , `finish`[null] , and `subs`[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 4.3, 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.

4.4.3 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.

4.5 Related Work

Depending on the analysis domain, precise models are often included for some components but elided for others. 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 [23].

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 [11, 54, 24]. Other program analysis tools will use the `Activity` lifecycle in addition to precise models of other user interface components for verifying user interface properties [137, 136, 101]. Framework precision with respect to objects used for concurrency such as `AsyncTask` and thread pools is often captured for race detection [137, 136, 62, 134] and other tools that detect concurrency issues [97]. 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 [136, 24, 99, 114]. The drawback to building the model directly into the analysis is that adding or updating behaviors [63] requires modifying the analysis itself. The most common approach to model callback orders is by generating an artificial `main` method [11, 98, 62, 10, 54]. 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 [17, 101] 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 [88, 133, 25]. Our approach can lessen the risk here by enabling a targeted approach to callback control-flow modeling to avoid modeling more than necessary.

4.6 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.

Chapter 5

Discovery and Explanation of Event-Driven Defects with History-Aware Incorrectness Logic

The previous section has established a technique for proving a safety property at a location in an event-driven application. In this chapter, we present a technique extending message history aware program logics to find the message history reaching a given location or defect. This technique is useful for two purposes: First, finding a message history that reaches a defect can be used to explain how a multi-callback crash happens. And, second, finding a message history reaching a known reachable location shows that a framework model is consistent. A framework model is unsound if it allows a known reachable location can be "proven" unreachable. However, an alarm from Historia on such a location does not guarantee reachability under a framework model due to abstraction of the program behavior. Therefore, a technique that proves reachability under a model is desirable. Later, we show how this technique may be used to ensure that the model resulting from framework model synthesis is consistent with observed reachable locations.

This chapter first presents a goal-directed incorrectness logic in section 5.1 that complements goal-directed verification and the design of our framework modeling language, Callback Control Flow Temporal Logic. Effectively, this technique is designed to accumulate an abstraction of concrete states that must step to the target location or crash. This backwards, goal-directed, style of reasoning pairs well with history implications that say "if a given message occurs now, here are the restrictions on the message history". Next, section 5.2 describes how the memory of the application may be split into a live portion and an irrelevant portion so that application memories reaching the target location may be efficiently represented. Conceptually, when the analysis starts at a program location of interest, it is best to be able to say ”any memory at this location" and then refine portions of the memory as they are read by the applicaiton. Finally, we show how the abstract message histories may be used in the context of a backwards incorrectness logic.

These contributions are then tested by implementing our techniques in a tool called WiStoria that takes a framework model and finds a message history consistent with that model reaching the target location. We evaluate WiStoria by finding message histories reaching representative examples of the nine most common runtime crashes developers face based on blog posts.

5.1 Goal-Directed Incorrectness Logic

In this section, we present a novel formulation of incorrectness logic that abstracts states in a program that must reach a defect of interest. Distinct from the normal formulation of incorrectness logic, we start at a location and abstraction of states representing a defect and work backwards through the program. The defect is proven reachable if the initial state is included in the abstraction.

We present our goal-directed semantics with a concrete transition system over commands and locations shown by the first part of Figure 5.1. The goal of these semantics is to structure the problem of finding an assertion violation as reachability of a specific target location in the program. For clarity of the formal semantics, we represent the error condition as a location with an unspecified state. It is always possible to compile an assertion to the reachability of a location. The error conditions presented in the overview and evaluation sections are the abstract state just before this compiled assertion. Such a compilation may be done with if statements for assertions over boolean expressions.

Programs p are represented by a set of transitions t. Each transition $\ell \dashv\! [c] \!\mapsto \ell'$ has a source location, a command, and a target location respectively. Locations ℓ are left abstract, but can be thought of as each location between commands. Execution starts at a special initial location ℓ_{init}. Concrete semantics are given by the small step semantics captured by the judgment form $\ell \colon \sigma \longrightarrow_p \ell' \colon \sigma'$. This logic consists of an unstructured control flow graph over program locations ℓ and a command language c which is defined in the next two sections of the paper. The c-step captures a concrete step that first chooses a transition in the program $\ell \dashv\! [c] \!\mapsto \ell' \in p$ and then executes the concrete step over the command $\langle \sigma, c \rangle \Downarrow^{\Omega} \sigma'$.Specific program states σ and commands c are left abstract for now and expanded in the next two sections along with the judgment defining the concrete semantics of commands $\langle \sigma, c \rangle \Downarrow^{\Omega} \sigma'$. We use $\ell' \colon \sigma' \longrightarrow^{*}{}_{p} \ell \colon \sigma$ to indicate the reflexive transitive closure of the step judgment form.

program $p ::= \circ \mid p, t$ transitions $b ::= \ell \dashv\!\![c]\!\!\mapsto \ell'$ locations ℓ initial location ℓ_{init}

program states σ initial program state σ_{init}

$$\boxed{\ell\colon \sigma \longrightarrow_p \ell' : \sigma' \qquad \ell\colon \sigma \models_S \widehat{\Sigma} \qquad \vdash_p \ell\ \textsf{reach} \qquad \widehat{\Sigma} \vdash^S \ell}$$

$$\frac{\ell \dashv\!\![c]\!\!\mapsto \ell' \in p \qquad \langle \sigma, c\rangle \Downarrow^{\Omega} \sigma'}{\ell\colon \sigma \longrightarrow_p \ell' : \sigma'}\ \text{C-STEP}$$

abstract program states $\widehat{\sigma}$ abstract state collections $\widehat{\Sigma} ::= \ell\colon \widehat{\sigma}, \mid \circ$

$$\ell\colon \sigma \models_S \ell\colon \widehat{\sigma}, \quad \text{iff} \quad \sigma \models_S \widehat{\sigma} \text{ or } \ell\colon \sigma \models_S \widehat{\Sigma}'$$

$$\frac{\ell\colon \widehat{\sigma}, \vdash^S \ell \qquad \vdash \widehat{\sigma}'\ \textsf{includesinit} \qquad \ell_{\text{init}}\colon \widehat{\sigma}' \in \widehat{\Sigma}}{\vdash_p \ell\ \textsf{reach}}\ \text{A-WITNESS}$$

$$\frac{\ell \dashv\!\![c]\!\!\mapsto \ell' \in p \qquad \ell'\colon \widehat{\sigma}' \in \widehat{\Sigma} \qquad \vdash [\widehat{\sigma}]\ c\ [\widehat{\sigma}'] \qquad \widehat{\Sigma} \vdash^S \ell''}{\ell\colon \widehat{\sigma}, \vdash^S \ell''}\ \text{A-COLLECT-TRANSITION}$$

Figure 5.1: A concrete and abstract goal-directed semantics of an unstructured control flow graph. We reduce the reachability of a failure to a location with an arbitrary state. A location ℓ is reachable if the judgment $\ell'\colon \sigma' \longrightarrow_p \ell\colon \sigma$ can reach some state at that location from the initial location ℓ_{init} and initial state σ_{init}. Reachability in our goal-directed incorrectness logic is captured by the judgment form $\vdash_p \ell\ \textsf{reach}$ which depends on the collect transition judgment form $\widehat{\Sigma} \vdash_p^S$ in order to abstract states reaching the location of interest. Individual abstract states at locations $\widehat{\sigma}$ are required to have a concretization $\sigma \models_S \widehat{\sigma}$ relation that defines the abstracted states. These semantics also rely on a judgment that can determine when the abstract state must include the initial state $\vdash_\Omega \widehat{\sigma}\ \textsf{includesinit}$.

We represent abstract program states with $\widehat{\sigma}$, and assume that there is a concretization relation $\sigma \models_S \widehat{\sigma}$ connecting abstract states to the concrete states they represent. To collect the abstract states at each location, we use $\widehat{\Sigma}$ to collect pairs of abstract program states and locations. Multiple abstract states may exist at a single location. Two abstract states at one location represents any state that models a single abstract state at that location. Concretization of a state with respect to the abstract state collection $\ell\colon \sigma \models_S \widehat{\Sigma}$ holds if the location is in the abstract state collection paired with an abstract program state such that the concretization holds.

Our program logic captures reachability with the judgment form $\vdash_p \ell\ \mathsf{reach}$ that holds when ℓ must be reachable. This judgment holds if three things are true, formalized in A-Witness: (1) An abstract state collection can be found such that there is some state at the target location and all other states must step to the target location. (2) There is some state $\widehat{\sigma}'$ in the abstract state collection paired with the initial location. (3) The abstract state $\widehat{\sigma}'$ must represent the initial state: $\vdash \widehat{\sigma}'\ \mathsf{includesinit}$.

Theorem 8 (Witness Soundness)**.** *If* $\vdash_p \ell\ \mathsf{reach}$ *then we have that* $\ell_{init}\colon \sigma_{init} \longrightarrow^*{}_p \ell\colon \sigma$ *for some* σ.

The judgment form $\widehat{\Sigma} \vdash^S \ell$ defines an abstract state collection $\widehat{\Sigma}$ such that each abstracted state must step to location ℓ under program p. This is defined inductively through the judgment a-collect-transition stating that if there is a location abstract state pair $\ell\colon \widehat{\sigma}$ in an abstract state collection reaching location ℓ'' (i.e. $\ell\colon \widehat{\sigma}, \vdash^S \ell''$) then, there must be: (1) a transition in the program such that it goes to a target location ℓ', (2) where the abstract state of the target location is $\widehat{\sigma}'$, (3) the under-approximate hoare-triple $\widehat{\Sigma} \vdash^S \ell$ holds on the post state generating the pre-state $\widehat{\sigma}$ and, (4) inductively, the abstract state collection holds on the target location ℓ'' (i.e. $\widehat{\Sigma} \vdash^S \ell''$). Soundness for this judgment form follows from the following lemma.

Lemma 9 (Collect Sound)**.** *If* $\widehat{\Sigma} \vdash^S \ell$ *and* $\ell'\colon \sigma' \models_S \widehat{\Sigma}$ *then* $\ell'\colon \sigma' \longrightarrow^*{}_p \ell\colon \sigma$ *for some* σ.

The judgment form representing the backwards incorrectness hoare-triple, $\vdash [\widehat{\sigma}]\ c\ [\widehat{\sigma}']$, says that given a post condition $\widehat{\sigma}'$ then all states represented by $\widehat{\sigma}$ must step to a post state in $\widehat{\sigma}'$ which is in the abstract state collection due to the a-collect-transition judgment. Over the next two sections, we expand on what we use for the program state σ and abstract program state $\widehat{\sigma}$. For now, we require that this judgment conforms to the following soundness condition: If $\vdash [\widehat{\sigma}]\ c\ [\widehat{\sigma}']$ and $\sigma \models \widehat{\sigma}$ then $\langle \sigma, c\rangle \Downarrow^{\Omega} \sigma'$ and $\sigma' \models \widehat{\sigma}'$.

The "includes initial" judgment form ($\vdash \widehat{\sigma}\ \mathsf{includesinit}$) holds if the concretization of the abstract state $\widehat{\sigma}$ must include the initial state σ_{init}. That is, $\sigma_{\text{init}} \models \widehat{\sigma}$. The next two sections define the specifics of this judgment in more detail.

5.2 Abstracting the Relevant Heap for Goal-Directed Reasoning

In the previous section, we started our analysis at the location of interest stating that "some state is feasible at the target location". However, starting this way is not so simple: we need to represent the **irrelevant** portions of the heap that are not read along the path to the error. At the target location discussed in the previous section, the entire memory is irrelevant. In contrast, **relevant** portions of the heap are those that can change the reachability of the defect. Consider for example the error-condition in the overview, where the `outer` and `remover` fields must be set at the error location, but it is impossible to guess this when starting the analysis without enumerating formula precisely representing the memory at the error location. Rather, we start our query at the target location, abstract the rest of the program memory, and constrain memory cells read and written by commands along the path to the defect.

memory commands $c ::= x = y.f \mid x.f = y \mid x = \mathsf{null} \mid \mathsf{assume}\ x = y \mid \mathsf{assume}\ x \neq y$ variables $x \mid y$

field names f values $v ::= \mathsf{null} \mid a$ addresses a memory location $n ::= x \mapsto v \mid a.f \mapsto v$

memories $\mu ::= \varepsilon \mid n \mid \mu \uplus \mu'$ initial empty memory ε

$\boxed{\mu[n]}$ $\mu[n] \stackrel{\text{def}}{=} \mu \uplus n$ iff $\mathsf{dom}(\mu) \cap \mathsf{dom}(n) = \emptyset$ $\mu \uplus n'[n] \stackrel{\text{def}}{=} \mu \uplus n$ iff $\mathsf{dom}(n) = \mathsf{dom}(n')$

$\boxed{\langle \mu, c\rangle \Downarrow^{\Omega} \mu'}$

C-FIELD READ

$$\frac{\mu(y) \neq \mathsf{null} \qquad \mu' = \mu[x \mapsto \mu(\mu(y).f)] \qquad x \neq y}{\langle \mu, x = y.f\rangle \Downarrow^{\Omega} \mu'}$$

C-FIELD WRITE

$$\frac{\mu(x) \neq \mathsf{null} \qquad \mu' = \mu[\mu(x).f \mapsto \mu(y)]}{\langle \mu, x.f = y\rangle \Downarrow^{\Omega} \mu'}$$

C-ASSUME EQ

$$\frac{\mu(x) = \mu(y)}{\langle \mu, \mathsf{assume}\ x = y\rangle \Downarrow^{\Omega} \mu}$$

C-ASSUME NEQ

$$\frac{\mu(x) \neq \mu(y)}{\langle \mu, \mathsf{assume}\ x \neq y\rangle \Downarrow^{\Omega} \mu}$$

C-NULL ASSIGN

$$\frac{}{\langle \mu, x = \mathsf{null}\rangle \Downarrow^{\Omega} \mu[x \mapsto \mathsf{null}]}$$

Figure 5.2: Concrete semantics for a java like heap manipulating language. Concrete memories consist of empty, memory locations n storing a variable or a field value, or a disjoint union of heaps $\mu \uplus \mu'$ (e.g. $v.f \mapsto v'' \uplus v'.f \mapsto v''$ implies $v \neq v'$).

We first present concrete semantics for a heap manipulating language consisting of field reads and writes, null, and assume commands, shown in Figure 5.2. As we will see in section 6.4, this fragment of the language is sufficient to express a wide range of defects, as well as find a wide variety of practical defects. Values v in this language consist of either null or some address a. Addresses are initially created by the new command discussed in the next section. Memory locations n are individual locations in the memory such as variables or fields on addresses. The program memory μ consists of an empty memory emp, memory locations n, or a disjoint union of memories $\mu \uplus \mu'$. By disjoint, we specifically mean that the keys for each memory location are separate, $\mu \uplus \mu'$ implies that $\mathsf{dom}(\mu) \cap \mathsf{dom}(\mu') = \emptyset$.

For convenience in the concrete semantics, we use short-hand notation for accessing and setting memory locations. A memory's domain $\mathsf{dom}(\mu)$ is the set of defined variables x and fields $a.f$. The notation $\mu[n]$ means to add the memory location n to the overall memory μ overwriting any other memory location with the same domain if it exists. We use the notation $\mu(x)$ to indicate that the memory must have some key x and that this expression evaluates to the corresponding value (e.g. $\mu(x) \neq \mu(y)$ becomes $\mu = \mu' \uplus x \mapsto v \uplus y \mapsto v'$ and $v \neq v'$). For C-FIELD READ, C-FIELD WRITE, and C-NULL ASSIGN we use relatively standard Java-like semantics. Accesses to fields require that the object be non-null. Assume expressions give our language control flow when combined with the unstructured control flow graph from the previous section. We restrict our language to assuming the equality or inequality of variables using C-ASSUME EQ and C-ASSUME NEQ,for simplicity of formalization.

The key to our abstract semantics is splitting the relevant portion of the heap that may be read along the path to the defect from the irrelevant portion of the memory that will not be read. We use the symbol $\mathsf{irrelevant}$ to abstract the portion of the heap that is not read along some path to the target location. Any abstract memory location $\widehat{n}$ represents memory that may be read at some point along the path to the error.

Figure 5.3 shows the abstract domain for application memories. An abstract memory $\widehat{\mu}$ follows the same notation as standard separation logic and represents sets of concrete memories through the concretization relation $\mu \cdot \theta \models \widehat{\mu} \cdot \widehat{\pi}$. These abstract memories can be $\mathsf{irrelevant}$ in which case they represent any memory, a single memory cell $\widehat{n}$, or separating conjunction over abstract memories $\widehat{\mu} * \widehat{\mu}'$. The symbol "$\mathsf{irrelevant}$" concretizes to any abstract memory. However, we maintain an additional meaning: $\mathsf{irrelevant}$ may only abstract portions of memory such that the defect may be reached without reading one of the memory locations. Individual memory cells that **may** be read along the path to the error condition are represented by the symbol $\widehat{n}$ which consists of an abstract field and an abstract value or a variable an abstract value. We represent such live portions of memory with $\mathsf{live}(\widehat{\mu})$, which consists of every abstract memory location in the abstract memory. The liveness of materialized memory locations is enforced by the abstract semantics. We use $\mathsf{live}(\widehat{\mu})$ to represent the syntactic set of keys in the relevant portion of $\widehat{\mu}$ (i.e. memory locations represented by a memory location $\widehat{n}$ and not $\mathsf{irrelevant}$).

abstract memories $\widehat{\mu} ::= \mathsf{irrelevant} \mid \widehat{\mu} * \widehat{\mu}' \mid \widehat{n}$ abstract memory location $\widehat{n} ::= x \mapsto \hat{v} \mid \hat{v}.f \mapsto \hat{v}'$

abstract values $\hat{v} ::= \mathsf{null} \mid \hat{x} \mid \hat{y} \mid \ldots$ pure constraints $\widehat{\pi} ::= \hat{v} \neq \hat{v}' \mid \widehat{\pi} \wedge \widehat{\pi}'$

valuation $\theta ::= \circ \mid \theta[\hat{x} \mapsto v]$

$$\boxed{\mu \cdot \theta \models \widehat{\mu} \cdot \widehat{\pi}} \qquad \mu \cdot \theta \models \mathsf{irrelevant} \cdot \widehat{\pi} \quad \text{iff} \quad \theta \models_S \widehat{\pi}$$

$$\mu \uplus \mu' \cdot \theta \models \widehat{\mu} * \widehat{\mu}' \cdot \widehat{\pi} \quad \text{iff} \quad \mu \cdot \theta \models_S \widehat{\mu} \cdot \widehat{\pi} \text{ and } \mu' \cdot \theta \models_S \widehat{\mu}' \cdot \widehat{\pi} \qquad x \mapsto v \cdot \theta[\hat{v} \mapsto v] \models x \mapsto \hat{v} \cdot \widehat{\pi} \quad \text{iff} \quad \theta[\hat{v} \mapsto v] \models_S \widehat{\pi}$$

$$v.f \mapsto v' \cdot \theta[\hat{v} \mapsto v][\hat{v}' \mapsto v'] \models \hat{v}.f \mapsto \hat{v'} \cdot \widehat{\pi} \quad \text{iff} \quad \theta[\hat{v} \mapsto v][\hat{v}' \mapsto v'] \models_S \widehat{\pi} \qquad \boxed{\mathsf{live}(\widehat{\mu})}$$

$$\mathsf{live}(\mathsf{irrelevant}) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{live}(\widehat{\mu} * \widehat{\mu}') \stackrel{\text{def}}{=} \mathsf{live}(\widehat{\mu}) \uplus \mathsf{live}(\widehat{\mu}') \qquad \mathsf{live}(x \mapsto \hat{v}) \stackrel{\text{def}}{=} \{x\} \qquad \mathsf{live}(\hat{v}.f \mapsto \hat{v}') \stackrel{\text{def}}{=} \{\hat{v}.f\}$$

Figure 5.3: Abstract states representing sets of memories. Key to this abstraction is that the memory is divided into a relevant portion represented by abstract memory locations and an irrelevant portion.

We use $\widehat{\pi}$ to represent pure constraints consisting of inequality,conjunction, and top. Equal values are represented by simply using the same logic variable $\hat{v}$. Disjunction of abstract states is handled by the abstract state collections from section 5.1. Valuations, θ are used to ensure that abstract values in different parts of the abstract state concretize to the same value.

As we discussed in the previous section, the soundness condition for one of these triples ensures that any state in the pre-condition $\mu \cdot \theta \models_S \widehat{\mu} \cdot \widehat{\pi}$ must correspond to a concrete step to a post state in the post-condition $\mu' \cdot \theta' \models_S \widehat{\mu}' \cdot \widehat{\pi}'$. Specifically, these semantics hold to the following condition: If $\vdash [\widehat{\mu} \cdot \cdot]c\ [[\cdot' \cdot \widehat{\pi}]\widehat{\mu}']$ and $\mu \cdot \theta \models \widehat{\mu} \cdot t \cdot$hen $\langle \mu, c \rangle \Downarrow^{\Omega} \mu'$ and $\mu' \cdot \theta' \models [\cdot' \cdot \widehat{\pi}]\widehat{\mu}'$ for some θ'. The abstract semantics enforce this condition by requiring that every portion of the memory that is read by a command must be precisely represented with an abstract memory location $\widehat{n}$.

Since all the judgments over commands rely on the full footprint of the command being materialized in the abstract memory, the A-STRENGTHEN rule is used to add missing variables or heap cells. The structure of our rules ensures that precisely the footprint is represented by the post condition of a command in order to generate a pre-condition. For example, Fwk_1 from the example in section 2.3 does not have a `okButton` field but Fwk_2 does since this field was read by the `onClick` between. Our program logic assumes that we can algorithmically guess the aliasing relationships and strengthen the post condition to reference the required variables. If the guess is wrong, then the path will become unreachable and no witness can be found. If the guess is correct, then the witness may be found. In the practical implementation, we simply case split on all the aliasing possibilities and see which one works.

$$\boxed{\vdash [\widehat{\mu} \cdot \widehat{\pi}]\ c\ [\widehat{\mu'} \cdot \widehat{\pi'}]}$$

A-STRENGTHEN
$$\frac{\vdash [\widehat{\mu} \cdot \widehat{\pi}]\ c\ [\widehat{\mu''} \cdot \widehat{\pi''}] \qquad \widehat{\mu'} \cdot \widehat{\pi'} \vdash \widehat{\mu''} \cdot \widehat{\pi''}}{\vdash [\widehat{\mu} \cdot \widehat{\pi}]\ c\ [\widehat{\mu'} \cdot \widehat{\pi'}]}$$

A-FIELD READ
$$\frac{x \notin \mathsf{live}(\widehat{\mu})}{\vdash [\hat{y}.f \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \hat{y} \neq \mathsf{null} \wedge \widehat{\pi}]\ x = y.f\ [\hat{y}.f \mapsto \hat{x} * x \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \widehat{\pi}]}$$

A-FIELD WRITE
$$\frac{\hat{x}.f \notin \mathsf{live}(\widehat{\mu}) \qquad \widehat{\pi} = \widehat{\pi'} \wedge \bigwedge_{\hat{z}.f \in \mathsf{live}(\widehat{\mu})} \hat{z} \neq \hat{x}}{\vdash [x \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \hat{x} \neq \mathsf{null} \wedge \widehat{\pi}]\ x.f = y\ [\hat{x}.f \mapsto \hat{y} * x \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \widehat{\pi'}]}$$

A-NULL ASSIGN
$$\frac{x \notin \mathsf{live}(\widehat{\mu})}{\vdash [\widehat{\mu} \cdot \widehat{\pi}]\ x = \mathsf{null}\ [x \mapsto \mathsf{null} * \widehat{\mu} \cdot \widehat{\pi}]}$$

A-ASSUME EQ
$$\frac{}{\vdash [x \mapsto \hat{x} * y \mapsto \hat{x} * \widehat{\mu} \cdot \widehat{\pi}]\ \mathsf{assume}\ x = y\ [x \mapsto \hat{x} * y \mapsto \hat{x} * \widehat{\mu} \cdot \widehat{\pi}]}$$

A-ASSUME NEQ
$$\frac{}{\vdash [x \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \widehat{\pi} \wedge \hat{x} \neq \hat{y}]\ \mathsf{assume}\ x \neq y\ [x \mapsto \hat{x} * y \mapsto \hat{y} * \widehat{\mu} \cdot \widehat{\pi} \wedge \hat{x} \neq \hat{y}]}$$

$$\boxed{\theta \vdash \widehat{\mu} \cdot \widehat{\pi}\ \mathsf{includesinit}}$$

MEM-INCLUDES INIT
$$\frac{\theta \vdash \widehat{\pi}}{\theta \vdash \mathsf{irrelevant} \cdot \widehat{\pi}\ \mathsf{includesinit}}$$

Figure 5.4: Backward abstract semantics over our heap manipulating language. Key to these semantics is that the memory a command operates on is always in the "relevant" portion of the heap represented by abstract memory locations. The irrelevant memory represented by the symbol irrelevant can only contain memory locations that are never read or written after the current location. Under-approximation occurs in either the pre or post-state based on the A-STRENGTHEN judgment.

Once the post state has been strengthened to represent all the variables that are read by a command, we can apply the appropriate abstract step. In the assignment judgments A-FIELD READ, A-FIELD WRITE, and A-NULL ASSIGN, the memory is split into a relevant portion represented by individual abstract memory locations in the pre and post state and the rest of the memory represented by $\widehat{\mu}$. For each of the assignment operations, the assigned memory location must not be in the domain of the irrelevant portion of memory (e.g. $x \notin \mathsf{live}(\widehat{\mu})$ for the field read and $\hat{x}.f \notin \mathsf{live}(\widehat{\mu})$ for field write). The primary reason is to ensure that the post state is feasible even though the command only operates on the relevant portion of memory (e.g. the abstract memory $x \mapsto a * x \mapsto a'$ implies false so any concrete state in the pre-condition would violate the earlier stated condition). The secondary reason is that when a value is assigned by a command, it cannot be live in the pre-condition (any value in that memory location cannot be read after this command). Regardless of the value in the pre-condition, a written location takes on a specific value in the post condition (note that our command language does not allow reading and writing of a memory location within the same command). For A-FIELD WRITE, we must ensure that the object of the field write is not equal to all other instances of the same field in the abstract memory. Otherwise, constraints about logic variables being not equal may be lost at this step. Assume statements block if the condition does not hold but otherwise do not alter the memory state. For simplicity, we only have equality and inequality with assume statements.

As discussed in the previous section, we need a way to determine when the abstract state must include the initial state. With the way that our abstract state is formulated, we reach the initial state when no memory locations are constrained (i.e. the memory represented by irrelevant). Includes initial is sound if $\theta \vdash \widehat{\mu} \cdot \widehat{\pi}$ includesinit then $\varepsilon \cdot \theta \models \widehat{\mu} \cdot \widehat{\pi}$ which holds due to the concretization of "irrelevant".

5.3 Execution History Abstractions for Finding and Explaining Defects

The final part of discovering and explaining event-driven defects is to abstract the execution history capturing the behavior of the framework. This consists of capturing the transitions between the application and framework as well as describing the new command and the associated values. We first describe the background on message history program logics. Then we show how we can modify the message history program logics to prove an execution sequence reaches the defect. Finally, we show how to recover the execution sequence for the end user of our tool.

Background: Application Only Transition Systems and Message History Program Logics. This subsection is meant to give background and connect the notation we use in Figure 5.5 to the work on using message history program logics to prove safety properties [89]. Specifically, we give background on: (1) the application only transition system for analyzing the application without the framework implementation. (2) message history program logics for abstracting the callback control flow, and (3) callback control flow temporal logics for capturing the realizable message histories.

As discussed in the overview, an application only transition system is an unstructured control flow graph like the one described in section 5.1 but with the framework transitions removed. All the framework transitions are merged into a single framework location ℓ_{Fwk}. For each transition in or out of the framework, there is a boundary message command c_{msg} such as `cb this.onCreate()` to indicate the method called and the argument `this`. The transitions for the application itself are left unmodified.

It is assumed that a record of the sequence of boundary message commands and associated values is sufficient to determine if a given execution is or is not possible. This sequence is referred to as a **message history** ω and consists of a sequence of boundary message commands m with arguments and return variables swapped out for runtime values. We write Ω to represent the set of message histories that may be seen under some framework implementation, that is these message histories are **realizable**. We assume that a judgment form $\langle \mu_{\mathrm{read}}, c_{\mathrm{msg}} \rangle \Downarrow \langle \mu_{\mathrm{read}} \uplus \mu_{\mathrm{write}}, m \rangle$ exists such that μ_{read} is precisely the memory ready by the boundary transition and μ_{write} is precisely the memory written added to the memory that was read. The details of such a judgment are straightforward. The memory read is the variables used as arguments for a callin (e.g. `tmp` in the line `tmp.execute()` or the return value of a callback). Written memory is the variables assigned by a transitions (e.g. arguments to callbacks such as the `this` variable or return values from callins). Each of these values are then captured by the message m along with the name of the method and the type of boundary transition.

Message history program logics abstract the history of boundary transitions one message at a time. Starting at the target location, we abstract the set of realizable message histories with the symbol okhist. Then, as the analysis traverses boundary message commands that emit messages, the messages are "removed" from the end using $\widehat{m} \twoheadrightarrow \widehat{\omega}$. The meaning of this is that it abstracts every message history such that appending some message m abstracted by $\widehat{m}$ is still realizable. As we will see, this abstraction is also sufficient for under-approximating the execution history reaching the defect.

To define the realizable message histories, the Historia paper introduces Callback Control Flow Temporal Logic (CBCFTL). CBCFTL is a first-order temporal logic designed to be used as a domain specific language for writing down the message histories that are realizable between a running application and framework. The specific structure of CBCFTL is designed to maintain performance when deciding properties about abstract message histories. A definition of framework behavior in CBCFTL consists of a set of history implications. Each history implication is marked by a $\square\!\!\to$ operator. On the left side of the $\square\!\!\to$ operator is a target message, a callback or callin. The right side contains a formula matching when that callback or callin may occur. Since such history implications are only used by the analysis as transitions are encountered, the framework may be modeled as needed rather than developing a full model up front.

A definition of realizable framework behaviors may be combined with an abstract message history to decide when a particular concrete message history may be modeled. Given some abstract message history $\widehat{m} \twoheadrightarrow \widehat{\omega}$ and a history implication with a left hand side message matching $\widehat{m}$, then $\widehat{\omega}$ must be consistent with the right hand side. For example, the abstract message history from the error condition in the overview $\mathsf{ciexn}\ r.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist}$ combines with the history implication $\mathsf{ciexn}\ t.\mathtt{execute()}\ \square\!\!\to$ $\mathsf{O}\ \mathsf{ci}\ t.\mathtt{execute()}$ to represent the message histories that are realizable (i.e. okhist) and where Once in the past, $\mathsf{ci}\ t.\mathtt{execute()}$ was invoked. The resulting formula may then be over-approximated by an encoding into the extended effectively propositional logic [73, 96] fragment of first order logic. The message history itself is represented by a total order over a message type and other uninterpreted functions are used to capture the message type, name, and arguments. In our example it becomes $\exists t \in \mathtt{Addr}, i \in [0, \mathtt{tracelen}].\omega[i] =$ $\mathsf{ci}\ t.\mathtt{execute()}$. This resulting formula may then be given to a SMT solver to determine if it excludes the initial state. Similarly, an abstract state with two `execute` calls,

$\mathsf{ciexn}\ t.\mathtt{execute()} \twoheadrightarrow \mathsf{ciexn}\ r.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist}$ may be encoded such that any trace is abstracted if and only if $t = r$.

However, it is important to note that the above formula is **over-approximate** in that the behavior of the framework represented by okhist is elided. Additionally, this work cannot precisely represent the memory as it does not restrict the framework from returning a value that is subsequently also returned by an application call to **new**.

Precisely Abstracting Framework Behavior with Execution Histories.

Our key observation in this section is that we may modify the message history program logic into an **execution history program logic** and precisely represent the execution history reaching the defect. In section 5.2 we observed that the memory read along the path to the defect should be precisely represented. Here, we observe that the key to under-approximation of the execution history is to precisely represent the messages and invocations of **new** from the current location to the target location. For many of the needed operations, the semantics of the message history program logic are sufficiently precise to be used as an under-approximation. In this section, we focus on showing how the addition of a new command to the abstract message history turning it into an **abstract execution history** is sufficient for our fragment of the Java language.

Figure 5.5 presents a concrete semantics that focuses on the message histories and the new command. We refer to any command that needs to be abstracted with respect to the execution history a **sequence command** c_{s}. For our fragment of the language, these are the **new** command and boundary message commands c_{msg}. A message command is a command at the boundary between the app and framework where we assume arguments are read or written on the boundary of a method invocation resulting in a message m. The program state σ extends the memory discussed in the previous section with a message history. The judgment form $\langle \sigma, c_{\mathrm{s}} \rangle \Downarrow^{\Omega} \sigma'$ captures the evaluation of sequence commands. With the judgment c-new there must be a value v bound to the target variable x in the post state and this value cannot have appeared in the memory or message history previously. Message commands are captured with c-msg cmd which differs the semantics of a message command to the judgment $\langle \mu_{\mathrm{read}}, c_{\mathrm{msg}} \rangle \Downarrow \langle \mu_{\mathrm{read}} \uplus \mu_{\mathrm{write}}, m \rangle$. This judgment reads variables in μ_{read} corresponding to a callback return value or arguments for a callin and writes variables in μ_{write} corresponding to arguments of a callback or the return value from a callin. The values in the combined memory $\mu_{\mathrm{write}} \uplus \mu_{\mathrm{read}}$ corresponds precisely to the values in m.

sequence commands $\quad c_{\mathrm{s}} ::= x = \textsf{new} \mid c_{\mathrm{msg}}$ $\qquad$ boundary message commands $\quad c_{\mathrm{msg}}$

messages $\quad m$ $\qquad$ message history $\quad \omega ::= \varepsilon \mid \omega; m$ $\qquad$ program states $\quad \sigma ::= \mu \cdot \omega$

$$\boxed{\langle \sigma, c_{\mathrm{s}} \rangle \Downarrow^{\Omega} \sigma'}$$

C-NEW

$$\frac{\omega \in \Omega \qquad v \notin \mu \qquad v \notin \omega}{\langle \mu \cdot \omega, x = \textsf{new} \rangle \Downarrow^{\Omega} \mu[x \mapsto v] \cdot \omega}$$

C-MSG CMD

$$\frac{\langle \mu_{\mathrm{read}}, c_{\mathrm{msg}} \rangle \Downarrow \langle \mu_{\mathrm{read}} \uplus \mu_{\mathrm{write}}, m \rangle \qquad \omega; m \in \Omega}{\langle \mu \uplus \mu_{\mathrm{read}} \cdot \omega, c_{\mathrm{msg}} \rangle \Downarrow^{\Omega} \mu[\mu_{\mathrm{write}}] \uplus \mu_{\mathrm{read}} \cdot \omega; m}$$

Figure 5.5: Concrete semantics capturing the state of the framework in the ghost variable ω representing the history of commands that can indicate framework state as well as the precise behavior of new.

$$\text{abstract program states} \quad \widehat{\sigma} ::= \widehat{\mu} \cdot \widehat{\omega} \cdot \widehat{\pi} \qquad\qquad \text{abstract observable} \quad \widehat{e} ::= \mathsf{new}\,\hat{x} \mid \widehat{m}$$

$$\text{abstract execution histories} \quad \widehat{\omega} ::= \mathsf{okhist} \mid \widehat{e} \twoheadrightarrow \widehat{\omega}$$

$$\boxed{\omega \cdot \theta \models_{\Omega} \widehat{\omega} \qquad \sigma \cdot \theta \models_{\Omega} \widehat{\sigma}} \qquad\qquad \omega \cdot \theta \models_{\Omega} \mathsf{okhist} \quad \text{iff} \quad \omega \in \Omega$$

$$\omega \cdot \theta \models_{\Omega} \mathsf{new}\,\hat{x} \twoheadrightarrow \widehat{\omega} \quad \text{iff} \quad \theta(\hat{x}) \notin \omega \text{ and } \omega \cdot \theta \models_{\Omega} \widehat{\omega}$$

$$\omega \cdot \theta \models_{\Omega} \widehat{m} \twoheadrightarrow \widehat{\omega} \quad \text{iff} \quad m \cdot \theta \models_{S} \widehat{m} \text{ implies } \omega; m \cdot \theta \models_{\Omega} \widehat{\omega}$$

$$\mu \cdot \omega \cdot \theta \models_{\Omega} \widehat{\mu} \cdot \widehat{\omega} \cdot \widehat{\pi} \quad \text{iff} \quad \mu \cdot \theta \models [\cdot] \cdot \widehat{\pi\mu} \text{ and } \omega \cdot \theta \models_{\Omega} \widehat{\omega} \text{ and } \text{forall } \mathsf{new}\,\hat{x} \in \widehat{\omega} \text{ implies } \theta(\hat{x}) \notin \mu$$

A-NEW

$$\boxed{\vdash [\widehat{\sigma}]\; c_{\mathrm{s}}\; [\widehat{\sigma}']} \qquad \frac{x \notin \mathsf{live}(\widehat{\mu})}{\vdash [\widehat{\mu} \cdot \mathsf{new}\,\hat{x} \twoheadrightarrow \widehat{\omega} \cdot \widehat{\pi}]\; x = \mathsf{new}\; [\widehat{\mu} * x \mapsto \hat{x} \cdot \widehat{\omega} \cdot \widehat{\pi}]}$$

A-MSG CMD

$$\frac{\vdash [\widehat{\mu}_{\mathrm{read}} \cdot \widehat{\pi}] c_{\mathrm{msg}} [\widehat{\mu}_{\mathrm{read}} * \widehat{\mu}_{\mathrm{write}} \cdot \widehat{\pi}, \widehat{m}] \qquad \mathsf{live}(\widehat{\mu}_{\mathrm{write}}) \cap \mathsf{live}(\widehat{\mu}) = \emptyset}{\vdash [\widehat{\mu} * \widehat{\mu}_{\mathrm{read}} \cdot \widehat{m} \twoheadrightarrow \widehat{\omega} \cdot \widehat{\pi}]\; c_{\mathrm{msg}}\; [\widehat{\mu} * \widehat{\mu}_{\mathrm{read}} * \widehat{\mu}_{\mathrm{write}} \cdot \widehat{\omega} \cdot \widehat{\pi}]}$$

Figure 5.6: Abstract semantics capturing the suffix of execution that reaches a defect. When the abstract history models the initial, empty history, a witness has been found.

The concretization of abstract execution histories $\omega \cdot \theta \models_{\Omega} \widehat{\omega}$ defines the concrete message histories reaching the defect. First, the base case of okhist concretizes to every realizable message history in Ω. Next, the concretization of $\omega \cdot \theta \models_{\Omega} \mathsf{new}\,\hat{x} \twoheadrightarrow \widehat{\omega}$ holds if the value corresponding to $\hat{x}$ in θ is not in the message history and the message history is in the abstract message history $\widehat{\omega}$. Finally, $\omega \cdot \theta \models_{\Omega} \widehat{m} \twoheadrightarrow \widehat{\omega}$ has the definition discussed earlier with the background on message histories. This is connected with the memory of the application and the separation logic earlier with the final concretization judgment over the entire program state $\sigma \cdot \theta \models_{\Omega} \widehat{\sigma}$. Valuations θ ensure that the concrete values assigned to abstract values are consistent between the parts of the abstract state. Additionally, each $\mathsf{new}\,\hat{x}$ in the abstract execution history must restrict the memory such that values created in the future are not referenced.

The A-NEW judgment requires that the variable x points to some value $\hat{x}$ in the post state and combines a $\mathsf{new}\,\hat{x}$ with the abstract execution history in the post state. Similar to the judgments in section 5.2, we ensure that the variable is not in the live portion of the post-state. Based on the definition of concretization, this ensures that the value is not in the abstracted message history or memory. A-MSG CMD relies on an abstract message transfer $\vdash [\widehat{\mu}_{\text{read}} \cdot \widehat{\pi}]c_{\text{msg}}[\widehat{\mu}_{\text{read}} * \widehat{\mu}_{\text{write}} \cdot \widehat{\pi}, \widehat{m}]$ that relates an abstract message $\widehat{m}$ to the memory that was read and written. We assume a soundness condition for these abstract message transfers: If $\vdash [\widehat{\mu}_{\text{read}} \cdot \widehat{\pi}]c_{\text{msg}}[\widehat{\mu}_{\text{read}} * \widehat{\mu}_{\text{write}} \cdot \widehat{\pi}, \widehat{m}]$ and $\mu_{\text{read}} \cdot \theta \models \widehat{\mu}_{\text{read}} \cdot \widehat{\pi}$ then $\langle \mu_{\text{read}}, c_{\text{msg}} \rangle \Downarrow \langle \mu_{\text{read}} \uplus \mu_{\text{write}}, m \rangle$ and $\mu_{\text{read}} \uplus \mu_{\text{write}} \cdot \theta' \models \widehat{\mu}_{\text{read}} * \widehat{\mu}_{\text{write}} \cdot \widehat{\pi}$ and $m \cdot \theta \models_{\theta'} \widehat{m}$ for some θ'.

Theorem 10 (Abstract Sequence Command Sound)**.** *If* $\vdash [\widehat{\sigma}]\ c_s\ [\widehat{\sigma}]$ *such that* $\sigma \cdot \theta \models_{\Omega} \widehat{\sigma}$ *then* $\langle \sigma, c_s \rangle \Downarrow^{\Omega} \sigma'$ *and* $\sigma' \cdot \theta' \models_S \widehat{\sigma}$ *for some* θ'.

Similar to our discussion on the background, the includes initial of the abstract execution histories may be encoded into first order logic. To handle each new in the abstract execution history, we can keep a set of logic variables that must be created in the future. For each previous message, we add a conjunction that each argument must not be equal to each future value. Similarly, we add a constraint for each new value returned that it cannot be equal to any future new (i.e. new cannot return the same value twice). Deciding if an abstract execution history includes the initial state can be done precisely since the imprecision in the encoding discussed earlier is with respect to the prefix. An abstract execution history includes the initial state if and only if the encoded SMT formula is satisfiable when the message history length is zero.

Explanation of Event-Driven Defects.

Once an alarm is found where the abstract state includes the initial state, we can find an explanation of the sequence of events leading to the defect. A satisfying interpretation of the SMT encoding will have assignments to each logic variables effectively giving us an example θ as used by the concretization. From there, we may unroll the abstract execution history one message at a time substituting the logic variables retrieving an example of the boundary invocations, calls to new, and associated values that reaches the defect. If we assume that new can return an arbitrary value each time it is called as long as that value has not been returned before, and the code uses our fragment of Java, the execution sequence will correspond to a real execution reaching the defect.

5.4 Empirical Evaluation

In this section, we evaluate the ability of our techniques to find and explain defects in event-driven programs. We implement our techniques in a tool called WiStoria which targets the Android platform as a case study for event-driven programming models. In particular, we evaluate WiStoria on the following questions:

(1) **RQ1 Encoding Common Runtime Problems**: Is our abstract domain able to represent the states under which common runtime issues in Android applications occur?

(2) **RQ2 Producing Realistic Execution Histories**: Are we able to produce realistic message histories for these problems explaining how a particular Android application reaches the problem?

5.4.1 Benchmarks

To evaluate WiStoria, we construct a benchmark of common bugs encountered by developers when working with Android. We chose these benchmarks by combining the most common runtime defects listed by three blog posts [81, 37, 128].

In order to obtain representative samples of each defect while minimizing our selection bias, we ask ChatGPT [95] to generate examples of each defect. ChatGPT generates responses based on patterns it has learned from training data. The exact training data used for ChatGPT is not published, but it is likely to include examples, conversations, and questions about Android development. Therefore, we believe these examples to be reasonably representative of how such defects would occur when faced by typical developers. For each defect, we prompt ChatGPT with “Give me an example of ...” where ... is a short description of the defect. We had to avoid two kinds of ChatGPT failures: (1) In some cases, ChatGPT would give us unrealistically simple examples of defects such as setting a variable to **`null`** and then immediately dereferencing the variable. To avoid this, we sent the prompt again with “non-trivial” added. (2) Several benchmarks were not actually defective, in which case we would regenerate the response. The number of times we had to regenerate each benchmark is listed in the “GPT Generations” column of Table 5.1. In most cases, we found that the first response from ChatGPT was sufficient, and if it was not, then we did not need to regenerate much before obtaining a good benchmark. Further details on the prompts given to ChatGPT can be found in Appendix B.

We then turned the code examples into compilable applications manually. This consists of taking the application code and adding the necessary additional files such as XML resource files and build files, producing an Android application which can be compiled and run. In total, we generate 11 benchmarks, which we describe briefly below.

Bitmap Mishandling: is when the application loads multiple large bitmaps into UI objects simultaneously. This is a problem because storing and manipulating these images slows the app, creating a poor user experience. We precisely define bitmap mishandling as loading two large images simultaneously into the user interface. **Device Configuration:** As a part of its lifecycle, an Android `Activity` may be destroyed and then later recreated. Before its destruction, a callback is invoked which allows the `Activity` to save its state, which is passed as an argument to its `onCreate` method when/if it is created later. If a developer does not leverage this, data may be lost if an application is suspended and later resumed. Formally, we can encode this as an `Activity` having its `onDestroy` method invoked before any state is saved using `putString`. **Device Compatibility:** issues arise as Android is a constantly evolving framework, where APIs may be deprecated over time. Such changes may be in response to in hardware changes, security improvements, and more. **Dialog Origin:** Applications may show dialog boxes to the user for a variety of reasons, such as to draw attention to something important. When an `Activity` or `Fragment` is destroyed, it must first dismiss any dialog windows it has created, or an exception is thrown as the window is leaked. Formally, this can be stated as an `Activity` a having `onDestroy` invoked, when some dialog d exists such that `d.show(a)` was invoked, but no call to `dismiss` was encountered before the `Activity` was destroyed. **Execute Twice** errors occur through the misuse of the `ASyncTask` interface. A task may be executed once and only once, and subsequent calls to `execute` will cause an `IllegalStateException` because the task is already executing or has already executed. Formally, if a task has had `execute` invoked, it is an error if the same task has already invoked `execute`. **Fragment Lifecycle:** Fragments are UI components which are attached to other UI components such as Activities. Fragments have their own lifecycle independent of an `Activity`, which can lead to issues if applications manipulate them without considering their lifecycle state. In particular, memory leaks can arise if a `Fragment` stores a reference to a `View`, and does not properly dispose of it when the `View` is destroyed. **Inefficient Network:** calls can lead to significant lag, as well as resource usage in Android applications. For instance, if an application makes a network call every time the `Activity` resumes, the application will make several calls if a user opens and closes the app frequently, even if there is no need to refresh the data, slowing the application and consuming more data than needed. If a library such as Volley is used, we can express this as a new `RequestQueue` being made when one has already been made previously, instead of reusing it. **Long Running Task:** The main thread of an application is responsible for the rendering of the UI. If a long-running process begins on this thread (such as processing bulk data), the UI can freeze from the perspective of the user. This can be encoded as witnessing an observable being attached to the thread when an `Activity` has not finished its `onCreate` call. **Memory Leak:** On top of standard memory leak scenarios, Android applications can exhibit memory leaks due to the lifecycles of `Activity` and `Fragment` components. When these are destroyed, any other components that may be able to reference them need to be handled, otherwise they cannot be garbage collected. In general, the Android framework denotes when an object should be disposed of through callbacks such as `onDestroy` or `onDestroyView`, and not properly handling these callbacks, can lead to leaks. Similar to the Fragment Lifecycle case, an example of this is a field pointing to an object whose `onDestroy` method was previously invoked. **Null Pointer Exceptions:** are common in many programming models, but in event-driven systems these errors can additionally arise from interactions between callbacks and callins, such as a callback setting a field to null, and another callback expecting the field to be non-null. **View Hierarchy:** Android application UIs are specified via XML. Deeply nesting `View` objects in an application's UI can lead to performance loss.

For each of these defects, we list statistics and the suitability of WiStoria in Table 5.1. First, the number of callbacks and callins (CB and CI) are listed in the table to give a sense of how much of the Android API each benchmark depends on. Next, we list the number of lines in each benchmark. After that, we list whether reasoning about the defect or fix relies on understanding multiple callbacks. For 9 out of the 11 benchmarks we found that callbacks were central to the defect in question. Finally, we list whether we can represent the error condition using the execution history and our fragment of separation logic. We found

Problem	GPT Generations	CB	CI	tot msg	lines	Multi-CB	WiStoria
Bitmap Mishandling	1	6	8	14	73	yes	yes
Device Configuration	1	2	3	5	47	yes	yes
Device Compatibility	1	5	7	12	64	no	no
Dialog Origin	1	3	4	7	68	yes	yes
Execute Twice	2	2	2	4	25	yes	yes
Fragment Lifecycle	2	4	7	11	80	yes	yes
Inefficient Network	1	4	3	7	95	yes	yes
Long Running Task	2	2	6	8	44	yes	yes
Memory Leak	2	3	1	4	22	yes	yes
NullPointerException	3	2	7	9	28	yes	yes
View Hierarchy	1	-	-	-	-	no	no

Table 5.1: A list of common runtime problems in Android applications acquired from online blog posts. The first column contains a name for the defect (described further in the evaluation text). We then list the number of callbacks, callins, total messages, and lines in each benchmark. The ”Multi CB” column lists whether reaching the error depends on state from multiple different callbacks. The WiStoria column captures whether we can write down the property for WiStoria using CBCFTL and our fragment of separation logic.

that only 2 of the benchmarks could not be expressed by our tool. In the first case, Device Compatibility was not applicable to our tool since the problem was caused by the set of objects that were imported. In the second case of the View Hierarchy, the benchmark used an XML configuration file to set up the views and analyzing configuration files is outside of the scope of our tool.

In Table 5.2, we list the conditions under which each benchmark will produce an error. Some benchmarks, such as Execute Twice, can be succinctly expressed purely in CBCFTL, whereas others, such as Fragment Lifecycle, require both temporal and state constraints to occur. The CBCFTL components describe the sequences of callins and callbacks necessary to witness the error, and state constraints are necessary when particular variables need certain values, such as being `null`. The bottom line result is that for 9 out of 11 of these benchmarks we can express the error conditions.

5.4.2 Producing Realistic Execution Histories

Next, we evaluate our ability to find realistic execution histories with WiStoria given an adequate CBCFTL definition of realizable message histories. We define a **realistic** execution history as a sequence of events that could happen at runtime, given the correct scheduling and timing of external events. In order

Benchmark	Encoding
Bitmap Mishandling	ciexn v.setImageResource(_) $\boxminus\!\!\rightarrow$ 0 ci v.setImageResource(_)
Device Configuration	cbexn a.onDestroy() $\boxminus\!\!\rightarrow$ cb a.onCreate() NS ci putString(s)
Dialog Origin	cbexn a.onDestroy() $\boxminus\!\!\rightarrow$ ci d.dismiss() NS ci d.show(a)
Execute Twice	ciexn t.execute() $\boxminus\!\!\rightarrow$ 0 ci t.execute()
Fragment Lifecycle	f.root $\mapsto v \wedge$ 0 cb f.onDestroyView() $\wedge$ 0 cb f.onCreateView(v) $\wedge$ 0 $v =$ ci inflate()
Inefficient Network	ciexn newRequestQueue(a) $\boxminus\!\!\rightarrow$ 0 ci newRequestQueue(a)
Long Running Task	ciexn o.putString(_) $\boxminus\!\!\rightarrow$ cbret a.onCreate() NS cb a.onCreate()
Memory Leak	f.outer $\mapsto v \wedge$ 0 cb v.onDestroy()
Null Pointer Exception	$s \mapsto$ **null**

Table 5.2: Error conditions for each benchmark. For error conditions where a message is called at the wrong time, we give a specification of an error result similar to the overview example. The exn is used to generally indicate a message occurring in an order that results in a defect (note that many of these benchmarks do not explicitly throw exceptions). For error conditions about the memory state, we specify an error condition with separation logic and temporal formula.

to objectively determine if the execution histories are realistic, we record runtime traces of the defective benchmarks in the Android emulator and determine if removing unrelated events and renaming addresses is sufficient to match the execution histories generated with WiStoria. To compare our results to the current state-of-the-art in bug finding, we run both Infer [40, 22] and Pulse [74] on our benchmarks and record whether an error was reported. The bottom line results are that WiStoria was able to produce realistic execution histories for all 9 of the benchmarks discussed earlier. In comparison, Infer was only able to detect one defect and Pulse did not find any.

We first generate the execution histories for each benchmark using WiStoria. Since our tool needs a specification of realizable message histories, we added CBCFTL history implications as needed to avoid un-realizable message histories. To give an idea of the size of the CBCFTL history implications, we list the number of messages specified in the "specified messages" column. Next, we report whether we were able to reach a realistic execution history with the "realistic" column. Finally, we report the length of the execution history first in number of callbacks (CB), then in callins (CI), and finally total messages (tot).

In order to generate the runtime traces used for comparison, we added message logging statements to each app at the beginning and end of each callback in the application as well as after each callin. Each of these message logging statements records the following to the terminal: the method name, type of message (callback, callback return, or callin), as well as the values of each argument. We additionally log each call to **new** along with the return value. In the case of defects involving the memory state, we also log the relevant memory state at the defect location so that we may look at the log and determine if the failure was reached. We then ran each benchmark app in the Android emulator executing the events corresponding to the execution history generated by WiStoria. Statistics for these runtime traces are listed in Table 5.3. We first report how many attempts it took us to witness the defect under the column "attempts". Next, we report the average length of the recorded trace (again listing the number of callbacks (CB), then in callins (CI), and finally total messages (tot)).

Problem	WiStoria					Runtime Instrumentation			
	specified msg(n)	realistic (y/n)	length CB(n)	length CI(n)	length tot(n)	attempts (n)	avg len CB(n)	avg len CI(n)	avg len tot(n)
Bitmap Mishandling	7	yes	5	9	17	1	15	32	54
Device Configuration	4	yes	2	0	2	1	2	3	5
Dialog Origin	3	yes	2	4	8	2	4	6	14.5
Execute Twice	5	yes	3	2	6	2	9.5	15.5	26
Fragment Lifecycle	4	yes	3	5	9	1	6	7	14
Inefficient Network	4	yes	2	5	7	1	5	18	32
Long Running Task	3	yes	1	2	3	1	2	6	8
Memory Leak	3	yes	3	1	6	2	5.5	2.0	11.5
NullPointerException	6	yes	2	7	13	1	2.0	7.0	13.0
total	39	9	23	35	71	12	51	96.5	178

Table 5.3: Execution histories generated by WiStoria and runtime instrumentation. For WiStoria we first list the size of the CBCFTL specification (specified msg), then if the history is realistic (i.e. can we remove unrelated events and match the runtime history), and finally the length of the history. We then list the number of attempts to reproduce each defect on the Android emulator as well as the lengths of the recorded histories.

The bottom line result is that WiStoria was able to produce realistic execution histories for each benchmark. In many cases, we were able to take history implications already written for the evaluation of Meier 2023 [89] reducing the burden of manually specifying the framework. When history implications did not previously exist, we developed them based on the documented behavior of the Android framework. Overall, the number of messages we had to specify was small compared to the number of messages that the application could emit (comparing the specified msg column with total msg from Table 5.1). In addition to being realistic, we found that in all but one case, WiStoria was able to produce shorter execution histories than the runtime method. We speculate that such shorter histories would be more useful to a developer trying to understand and fix a defect.

In comparison to Pulse and Infer, we note that WiStoria was able to find many more of the defects.

Problem	Infer	Pulse
Bitmap Mishandling	no	no
Device Configuration	no	no
Dialog Origin	no	no
Execute Twice	no	no
Fragment Lifecycle	yes	no
Inefficient Network	no	no
Long Running Task	no	no
Handling Failures	no	no
Memory Leak	no	no
NullPointerException	no	no

Table 5.4: Here, we summarize the results of Infer and Pulse on the benchmarks. Infer successfully detected the defect in Fragment Lifecycle and Pulse did not find any of the defects.

However, it is worth pointing out that our tool requires a much higher burden of manual specification of framework behavior. While this specification work can be applied across multiple applications running on the same framework, both Infer and Pulse are fully automatic. Finally, we note that neither Infer nor Pulse produce an explanation of how the defect occurs. It is also worth pointing out at this point in the discussion of the results that, ChatGPT was unable to produce specific explanations for the order of callbacks for these examples. Responses to multiple queries like “what order of callbacks produce the null pointer exception in this example” resulted in responses that were not specific to the benchmark itself but rather discussed general aspects like the Fragment Lifecycle. We believe that the explanation in the form of an execution history is a distinct advantage of WiStoria.

5.4.3 Threats to Validity

Our threats to validity consist of the application selection process, the precision of the framework model used to find witnesses, and the fragment of the Java language we support. To select applications for testing, several blog posts were consulted. These blog posts may not necessarily fully capture the experiences across a broad range of developers and event-driven frameworks. The problems discussed in the blog posts were then given to ChatGPT to generate representative examples of the bug, which were then compiled into the benchmark applications used. While this in theory gives benchmarks that are representative of a large amount of training data, we recognize that there is a chance that the data may be distorted by the large language model [14]. However, based on manual inspection, we believe the benchmarks to be reasonable. Another potential evaluation issue is the framework model used in finding traces. Since our tool takes handwritten specifications, there is a chance that we may not sufficiently restrict framework behavior. To the best of our knowledge, our specifications capture the behavior in the documentation but undocumented difficulties such as the `finish` behavior in the overview exist. Finally, we implemented our tool over a subset of the Java language meaning that further development may be needed to fully under-approximate the behavior of the framework. However, we found the subset of the Java language that we support to be sufficient for these benchmarks.

5.5 Related Work

Incorrectness Logic is a formalization of proving reachability of locations or defects in applications using hoare logic [93]. As an extension, incorrectness separation logic extends these ideas to local reasoning about heap properties [108]. Finding real bugs in programs with incorrectness logic [74] introduces the notion of manifest errors to define that a method must have a bug under any calling context. However, none of these techniques are able to reason about the state between callback invocations or reason about the history of execution. Finally, work has been done on incorrectness logic over object-oriented languages and frameworks [78] which have a close relation to such event driven frameworks but does not handle the missing framework code.

Goal-directed analysis [17, 16] presents an efficient and precise way to reason about states that **may** reach such a location of interest in able to prove safety. Specifically, this technique analyzes the program with respect to “what may lead to a crash at this location” avoiding the need to compute an invariant over the entire program.

5.6 Conclusion

This paper has introduced a novel approach for the discovery and explanation of defects in event-driven applications using history-aware incorrectness logic. By focusing on the specific execution histories that lead to defects, we have developed a method that not only identifies the sequences of callbacks and event interactions responsible for failures but also highlights the minimal set of memory interactions necessary for reaching these defects. Our approach provides a clearer insight into the causal relationships within the complex behaviors of event-driven systems, making the debugging process both more efficient and effective.

The implementation of our method in the WiStoria tool has demonstrated its practical value in commonly encountered runtime defects, successfully detecting and providing explanations for common defects encountered in event-driven programming. The results from our evaluations indicate that focusing on relevant memory and understanding event sequences can significantly improve the detection and resolution of defects.

Chapter 6

Abducing Targeted Models of Event-driven Frameworks For Verification

In this chapter, we present a technique for synthesizing framework models that are precise enough to prove a safety property such as the fixed code from the overview (Figure 2.1) but sound with respect to known reachable locations such as the crash in the buggy version of our example Figure 1.1 or the cancel button's click listener. This technique allows a developer to start with a very loose framework model and automatically refine how each callback (or other back message) may be restricted. The key challenge is that the space of framework models is huge, first because of the number of possible callbacks and callins (e.g. Antennapod implements nearly 4000 callbacks), and second because of the quantified arguments that connect messages together within the history implications making a CBCFTL framework model.

To address the challenge of synthesizing framework models for event-driven frameworks, we start with a technique for searching CBCFTL models based on syntax guided synthesis in section 6.1. Next, we identify a key property of the handwritten models in the previous chapters: these models are **connected**. By only exploring connected models, we can start to trim down the space of possible framework models. After that, we present message graphs, a technique to quickly filter expansions of the framework model that will not progress the synthesis towards a sufficiently sound model that proves the safety property. Finally, we show how this technique can be used to synthesize the framework models needed to prove the benchmarks in chapter 4.

6.1 Exploring the Space of Framework Models with Syntax Guided Synthesis

In this section, we present a method to effectively enumerate candidate framework models starting with the syntactically smallest models and growing in complexity. At a high level, we have completed the search for a framework model S when we can prove the target issue safe (i.e. $\vdash_p^S \ell\colon \top$ unreach where ℓ and p are the target issue) and each reachable location can be witnessed (i.e. $\vdash_p^S \ell$ reach for each p and ℓ in the set of known reachable issues).

Figure 6.1 shows the modified CBCFTL syntax with holes. We use $\overset{\blacksquare}{S}$ to distinguish a framework model that is not yet completed from a framework model that can be used for the previous two sections. Each hole $\blacksquare_{\widetilde{\omega}}$ is a location where some sub-formula may be substituted in order to search for a framework model satisfying the reachability of the issues. The other kind of hole is a message hole $\blacksquare_{\widetilde{m}}$ which allows one message to be substituted. This restriction on where holes may be in the temporal formula means that obtaining an upper bound for a formula is straightforward. Each $\blacksquare_{\widetilde{\omega}}$ can be replaced with true and each $\blacksquare_{\widetilde{m}}$ $\mathtt{NS}$ $\widehat{m} becomes \mathsf{O}\ \widehat{m}$. Similarly, a lower bound to the framework model may be obtained by replacing each $\blacksquare_{\widetilde{\omega}}$ or $\blacksquare_{\widetilde{m}}$ $\mathtt{NS}$ $\widehat{m}$ with false.

issues $\quad i_r ::= \mathsf{p} \cdot \ell$framework model $\quad \overset{\blacksquare}{S} ::= \mathsf{true}\ |\overset{\circ}{s} \wedge \overset{\blacksquare}{S}$ $\qquad$ history implication $\quad \overset{\circ}{s} ::= \widehat{m} \mathrel{\square\!\!\rightarrow} \overset{\circ}{\widetilde{\omega}}$

temporal formula $\quad \overset{\circ}{\widetilde{\omega}} ::= \blacksquare_{\widetilde{\omega}} \mid \mathsf{O}\ \widehat{m} \mid \mathtt{HN}\ \widetilde{m} \mid \blacksquare_{\widetilde{m}}\ \mathtt{NS}\ \widehat{m} \mid \widehat{m}_2\ \mathtt{NS}\ \widehat{m}_1\ |\overset{\circ}{\widetilde{\omega}}_1 \wedge \overset{\circ}{\widetilde{\omega}}_2 | \overset{\circ}{\widetilde{\omega}}_1 \vee \overset{\circ}{\widetilde{\omega}}_2 |\ \mathsf{true} \mid \mathsf{false}$

Figure 6.1: Syntax of the target and reachable issues as well as the CBCFTL language with holes. Each issue consists of an application and a target location (the crash shown in our introduction may be compiled into such a location). CBCFTL is designed to be in negation normal form to make searching for candidate framework models easier allowing the upper and lower bound of each candidate model to be computed easily by substituting true or false for holes.

In order to focus the algorithm on the portions of the framework model that are the hardest to write, we start with a framework model containing holes such as ci l.onClick() $\mathrel{\square\!\!\rightarrow}$ O b = ci a.makeButton(l) $\wedge \blacksquare$

from subsection 2.5.1. This framework model corresponds to the user knowing how to make a button and register the `onClick` listener but not knowing how `setEnabled` or `finish` affect the clickability of a button. Such a starting point saves time with the synthesis and reduces the total number of reachable locations needed.

Figure 6.2 shows the high level steps for synthesizing a framework model. This synthesis is implemented as a worklist algorithm over the graph of candidate framework models. A framework model graph is defined by an initial node consisting of the initial location and subsequent edges to new nodes are captured by the STEP method which takes a candidate framework model and returns the set of models where one hole is replaced with one AST production (e.g. $\widetilde{m} \;\square\!\!\rightarrow \blacksquare_{\widetilde{\omega}}$ could be expanded to $\widetilde{m} \;\square\!\!\rightarrow \blacksquare_{\widetilde{\omega}} \wedge \blacksquare_{\widetilde{\omega}}$). This algorithm is initialized with a set of known reachable issues R and a target issue t. Each issue consist of an application p and a locations ℓ. The queue of the worklist is initialized with the starting framework model with holes.

R is the set of reachable issues.
t is the target issue.
Initialize the priority queue W with the initial CBCFTL model
while W is not empty **do**
    $\overset{\blacksquare}{S} \leftarrow$ extract next model from W
    if $\overset{\blacksquare}{S}$ has no holes **then**
        $S \leftarrow \overset{\blacksquare}{S}$ ▷ With no holes we have a normal CBCFTL model
        if $\textsc{Historia}(S, t)$ **and** $\textsc{Wistoria}(S, R)$ **then**
            return S ▷ Done, display the CBCFTL model
        end if
    else $\overset{\blacksquare}{S}$ has holes
        $underApproxModel \leftarrow \textsc{LowerBound}(\overset{\blacksquare}{S})$
        $overApproxModel \leftarrow \textsc{UpperBound}(\overset{\blacksquare}{S})$
        if $\textsc{Historia}(underApproxModel, t)$ **and** $\textsc{Wistoria}(overApproxModel, R)$ **then**
            $NewModels \leftarrow \textsc{Step}(m)$
            $W \leftarrow \textsc{Filter}(NewModels, t, R)$
        end if
    end if
end while

Figure 6.2: Pseudo code for the work-list algorithm to synthesize the framework model. W is a work list in the form of a priority queue that returns the syntactically smallest models that does not have holes first. If all models contain holes, then it chooses the syntactically smallest model. $\textsc{Historia}(m, t)$ takes a model and the target issue and returns whether it is safe. $\textsc{Wistoria}(m, R)$ takes a model and the reachable issue set and returns true if each are reachable (this method may time out and return false).

The worklist algorithm runs as long as it has not found a suitable framework model and there are candidate framework models in the queue W. As a result, attempting to prove an unsafe issue can either lead to a failure to synthesize a model (the queue is empty) or a timeout. The first step of the while loop is to extract a model $\overset{\blacksquare}{S}$ from the queue as shown by Line 5.

This queue must be a priority queue. There is an infinite number of possible framework models and the smallest ones tend to be better for performance and the number of reachable locations required for a sound result. We use a priority queue that prioritizes framework models that have no holes first. If all the framework models have holes, then the queue prioritizes the candidate framework model that is syntactically smallest. The syntactic size is measured by counting the AST productions in each history implication.

For a framework model that does not have any holes (i.e. it is syntactically a normal CBCFTL formual S), we may call the Historia tool on the target issue and Wistoria on each reachable issue. If Historia is able to prove the target issue and Wistoria can find reachable paths to each known reachable issue, then we are done. Line 9 returns the final framework model to the user for inspection. If this model is sound then the application is proven. If the model is not sound, then there are not enough reachable locations. However, as we will see in section 6.4 the number of reachable locations needed tends to be small.

Alternatively, if the framework model extracted from the queue on Line 5 has holes, then the search proceeds. First, the framework model is checked to see whether any expansion of the holes is sufficient to prove the target issue. This is done by first computing the framework model representing the lower bound of message histories that can result from expanding holes in the current model. We compute the lower bound operation with LowerBound method and then call Historia on the resulting model with the target issue. Similarly, we may check whether any expansions of the framework are consistent with the known reachable issues. This is done by computing the UpperBound of the framework model and then running Wistoria on the result with each reachable location. If either of these operations fails, the candidate framework model is discarded.

The Step function enumerates each syntactic expansion of the framework models as well as the possible aliasing relationships with existing messages. For example, stepping the candidate framework model ci l.`onClick()` $\square\!\!\rightarrow$ O b = ci a.`makeButton(`l`)` $\wedge$ ($\blacksquare_{\tilde{\omega}} \vee \blacksquare_{\tilde{\omega}}$) results in a set containing ci l.`onClick()` $\square\!\!\rightarrow$ O b = ci a.`makeButton(`l`)` $\wedge$ (HN ci a.`setEnabled(false)` $\vee$ $\blacksquare_{\tilde{\omega}}$). For now, assume that this method enumerates everything including `setEnabled(a)`, in the implementation it is combined with the Filter function described next.

At this point, it is clear that the search space is huge. There are numerous callbacks and callins that need to be considered and this is only made worse by the choices of aliases between arguments of the model. As a minor optimization, we first merge models that have already been searched. For example $\mathsf{ci}\ l.\mathtt{onClick}() \;\square\!\!\rightarrow\; \mathtt{0}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l) \wedge (\mathtt{HN}\ \mathsf{ci}\ a.\mathtt{setEnabled}(\mathbf{false}) \vee \blacksquare_{\widetilde{\omega}})$ is equivalent to $\mathsf{ci}\ l.\mathtt{onClick}() \;\square\!\!\rightarrow\; \mathtt{0}\ b = \mathsf{ci}\ a.\mathtt{makeButton}(l) \wedge (\blacksquare_{\widetilde{\omega}} \vee \mathtt{HN}\ \mathsf{ci}\ a.\mathtt{setEnabled}(\mathbf{false}))$. Therefore, we represent all framework models in a normal form allowing many equivalent models to be syntactically merged. Next, we describe the more interesting optimizations that make this approach work in practice.

6.2 Connected Framework Models

From the experimental evaluations in all previous chapters, we have observed an interesting pattern in framework models: they are all connected. A **connected** framework model has a path connecting shared arguments through each message in each history implication. In our motivating example, this connectedness is how we know which button is associated with which `onClick` listener. In theory, it is possible for a non-connected model to be useful (e.g. if there was an `exit()` method that immediately stops everything). But, we have not observed such a model to conform to the actual framework behavior (e.g. most `exit` like methods are more of a suggestion to eventually exit and actions can still occur after invocation).

The first job of the Filter method is to reject any framework model that is not connected. This is done using a graph search algorithm visiting each method in a history implication ensuring that at least one argument connects. For example, History Implication 1 starts with $\mathsf{cb}\ l.\mathtt{onClick}()$, finds the first connection to $b = \mathsf{cb}\ a.\mathtt{makeButton}(\mathtt{l})$ through l, the next connection is with the three occurrences of `setEnabled` through b. If such a path cannot be found, the Filter method removes the framework model.

6.3 Targeting the Framework Models Using Message Graphs

The final and most important optimization ensures that each clause added to a history implication in a CBCFTL formula could be true under some issue. This is done using a pointer analysis over the application-only transition system that we call a **message graph**. Specifically, we determine when arguments in a pair of messages could alias one another along an abstract message history reaching a target location. If arguments cannot alias one another, then adding a temporal operator to a history implication is not useful towards the goal of proving the target issue or admitting known reachable issues. In subsection 2.5.1 the history implication $\mathsf{ci}\ l.\texttt{onClick}() \mathrel{\square\!\!\rightarrow} \mathsf{O}\ b = \mathsf{ci}\ a.\texttt{makeButton}(l) \land \mathsf{O}\ \mathsf{ci}\ l.\texttt{setEnabled}(\textbf{false})$ is connected but un-targeted because the temporal operator $\mathsf{O}\ \mathsf{ci}\ l.\texttt{setEnabled}(\textbf{false})$ can never be true because the listener object l used for `onClick` cannot alias the button object used for `setEnabled`. The lack of aliasing between l and b can be determined using a pointer analysis and looking at the locations that may emit a `setEnabled` or `onClick` message in each application.

Constructing the Message Graph

The message graph is generated by first building a pointer analysis for the Android application. This pointer analysis is a flow-insensitive whole program analysis, therefore we need a "main" method sufficient for an entry point. We construct this main method using a combination of techniques related to the application only call graph [1, 2] and Jphantom [12]. Since we are using the flow insensitive Spark pointer analysis, this main method does not need to preserve callback or instantiaton order. We build this main method by creating a call to **`new`** for each object type that may be created by the framework (e.g. the `RemoverActivity` which is instantiated via reflective object reference in the real framework). Each of these objects is written to a single static "framework" field. After identifying all possible callbacks in the application, the main method is appended with a call to each callback. Similarly, each callin on the framework is replaced with code that writes each argument to the framework field and then returns a value from the framework field. Since many of the framework classes are missing due to dynamic code loading, linking, native code, and more, we search for any interfaces called by the application that do not have concrete method targets. For each of these missing callins, we generate an artifical sub-class in the style of JPhantom but utilizing application only instrumentation for the method bodies. In this way, we can create a pointer analysis where the points to regions of each application variable over-approximate the runtime behavior of the framework. We will use the pseudocode method MakeMessageGraph to refer to this process of generating the pointer analysis.

The result is that we may now search the application for all methods that may be a callback and match them with the space of abstract messages. Similarly, the callins may be matched to call sites of possible framework methods. We use the pseudocode method CanAlias to represent comparing two program variables to determine if they may be aliased. In the previous analyses, these aliasing constraints are carried along with the symbolic variables as abstract states are generated (similar to Thresher [16]). Therefore, we can also use CanAlias on symbolic variables in abstract states.

Filtering the Candidate Messages

As the synthesis proceeds, we want to only consider models that may affect the reachability of an issue. Models that consider aliasing not possible under any application within the set of issues or messages that aren't used do not help with the verification task. For example, consider a candidate model consisting of a `onClick` happening if an unknown is true: cb l.`onClick()` $\boxminus\!\!\rightarrow$ $\blacksquare_{\widetilde{\omega}}$. We can immediately remove messages where l appears in an argument that has no possible way of aliasing the receiver of `onClick`. For example ci l.`setEnabled(false)` will always be false in this history implication because the listener object l cannot alias the button objects obtained through `makeButton` based on the class hierarchy.

The real benefit of the message graph comes from combining the aliasing possibilities with the restrictions of message histories produced from issues. From the example message graph shown in Figure 2.11 it is possible for a `onClick` to alias a wide variety of message arguments. However, if we consider an abstract message history reaching the "Cancel" button's `onClick` listener, we reduce the search space significantly. Specifically, the abstract message history cb l_{dis}.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist retains alias constraints for the l_{dis} logic variable. If we search for the simplest history implication that describes the reachability of both the "OK" and "Cancel" button `onClick` callbacks, then `makeButton` is the clear expansion for the history implication cb l_{dis}.`onClick()` $\twoheadrightarrow$ ciexn t.`execute()` $\twoheadrightarrow$ okhist.

Specifically, we can see the different behaviors of a message with a possible aliasing relation versus a message with no possible aliasing relations in the Quotient judgments form section 4.3. A message from the CBCFTL formula may only match a message from the abstract message history, Match$(\widetilde{m}, \widehat{m})$ if the name and arguments are equivalent. However, if the pointer analysis shows that two arguments can never be aliased, we know that the message may never match. For Temporal operators requiring positive messages such as $\widetilde{m}$ in O $\widetilde{m}$ or $\widetilde{m}_2$ in $\widetilde{m}_1$ NS $\widetilde{m}_2$ an argument that can never alias means the clause will always be false.

Such clauses are not useful in this search.

Similarly, it is not useful to consider negative temporal operators that can never be false through the NotMatch($\widetilde{m}, \widehat{m}$) operator used in the QUOTIENT judgments. Therefore, the message graph filtering is also useful for the negative messages in temporal operators such as $\widetilde{m}$ in $\texttt{HN}\ \widetilde{m}$ and $\widetilde{m}_z$ in $\widetilde{m}_1\ \texttt{NS}\ \widetilde{m}_2$.

Since the judgment to determine if a history implication matches a message history uses an assignment from CBCFTL formula variables to concrete values, θ from section 4.2, a Match($\widetilde{m}, \widehat{m}$) or NotMatch($\widetilde{m}, \widehat{m}$) may evaluate to true due to an arbitrary chain of temporal formula. For example, the $\texttt{HN}$ ci b.`setEnabled(`**`false`**`)` clause may change meaning depending on the specific values associated with $\texttt{O}\ b$ = ci a.`makeButton(l)`.

$$\textsf{cb}\ l.\texttt{onClick()}\ \square\!\!\rightarrow \texttt{O}\ b = \textsf{ci}\ a.\texttt{makeButton}(l)\ \wedge$$
$$(\texttt{HN}\ \textsf{ci}\ b.\texttt{setEnable}(\textbf{false})\ \vee\ \textsf{ci}\ b.\texttt{setEnable}(\textbf{false})\ \texttt{NS}\ \textsf{ci}\ b.\texttt{setEnable}(\textbf{true}))$$

Therefore, we inductively define relevance as an argument in the current message that is shared with the target message or a shared argument with another message that is connected. This property is captured using the RELEVANT method in Figure 6.3. Specifically, RELEVANT takes a history implication s, an issue set I, and the current candidate framework model $\blacksquare{S}$ and returns true if each message can be connected using an abstract message history for some issue i_r, i.e. CONNECTEDTHROUGHISSUE holds for the two messages and the issue.

function RELEVANT($s, I, \overset{\blacksquare}{S}$)

    $\widehat{m}_{\text{target}} \mathrel{\Box\!\!\rightarrow} \widetilde{\omega} = s$

    for all $\widetilde{m} \in \widetilde{\omega}$. $\widetilde{m}$ is transitively connected to $\widehat{m}_{\text{target}}$ using CONNECTEDTHROUGHISSUE for some $i_r \in I$

end function

function CONNECTEDTHROUGHISSUE($\widetilde{m}_1, \widetilde{m}_2, i_r, \overset{\blacksquare}{S}$)

    $p \cdot \ell \leftarrow i_r$ ▹ get the application and location from the issue

    $G \leftarrow$ MAKEMESSAGEGRAPH(p)

    find a $\widehat{\Sigma}$ **such that** $\widehat{\Sigma} \vdash_p^{\uparrow \overset{\blacksquare}{S}} \ell : \top$ ▹ invariant map from the over approx of the current model

    find a $\ell : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \in \widehat{\Sigma}$ **such that** the initial state is included

    find a $\widehat{m}_{\text{src}}, \vartheta$ **such that** $\widehat{m}_{\text{src}} \in \widehat{\omega}$ and $\widehat{m}_{\text{src}} \simeq_\vartheta \widetilde{m}_1$

    find a $[\hat{x}_{\text{model}} \mapsto \hat{x}_{\text{abst}}] \in \vartheta$ **such that** MAYALIAS($G, \widetilde{m}_2, [\hat{x}_{\text{model}} \mapsto \hat{x}_{\text{abst}}]$)

    if any of the above cannot be found **then**

        return false

    else

        return true

    end if

end function

Figure 6.3: The RELEVANT method for determines if a history implication is relevant to the target issue or reachable issues. As inputs, it takes a single history implication s, a set of issues I (both reachable and target), and the current candidate framework model $\overset{\blacksquare}{S}$.

Specifically, CONNECTEDTHROUGHISSUE first finds an alarming invariant map using Historia $\widehat{\Sigma} \vdash_p^{\uparrow \overset{\blacksquare}{S}} \ell : \top$ and the upper bound of the candidate framework model $\uparrow \overset{\blacksquare}{S}$. Within this invariant map there is an abstract state $\ell : \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \in \widehat{\Sigma}$ corresponding to the alarm because it includes the initial concrete state (i.e. the includes initial procedure from section 4.3). Next, a message is found in the alarming history $\widehat{m}_{\text{src}} \in \widehat{\omega}$ that matches the first message $\widehat{m}_{\text{src}} \simeq_\vartheta \widetilde{m}_1$ with some substitution ϑ. Finally, search the message graph for a message that may alias the formula variable and constrained by the pointer analysis in the abstract message history using MayAlias. We leave the aliasing information implicit in the pure variable from the abstraction $\hat{x}_{\text{abst}}$. However, it can be thought of as a set of reaching allocation sites. This is also constrained by the variable shared between $\widetilde{m}_1$ and $\widetilde{m}_2$ named $\hat{x}_{\text{model}}$ in the figure.

As illustrated by the abstract message history discussed earlier, $\mathsf{cb}\ l_{\text{dis}}.\mathtt{onClick()} \twoheadrightarrow \mathsf{ciexn}\ t.\mathtt{execute()} \twoheadrightarrow \mathsf{okhist}$, the message that needs to be added to the framework model may not be captured by the abstract message history from the alarm. Two messages need to be added in order to expand the history implication $\mathsf{cb}\ l.\mathtt{onClick()} \boxminus\!\!\rightarrow \blacksquare_{\widetilde{\omega}}$ into what is needed to prove the fixed version of the application and reach the bug and the `onClick` listner for the "Cancel" button. These messages are the $b = \mathsf{ci}\ a.\mathtt{makeButton()}$, $\mathsf{cb}\ b.\mathtt{setEnabled}(\mathbf{true})$, and $\mathsf{cb}\ b.\mathtt{setEnabled}(\mathbf{true})$. However, the alarm does not show any of these messages. For this reason, the message graph is needed to find the messages that may be emitted for refining the history implication.

The advantage of this technique is that the number of messages that need to be considered when expanding a history implication may be significantly reduced. Additionally, the message graph technique eliminates the majority of aliasing possibilities between arguments further reducing the space of candidate framework models. However, there are a few other filters that we have found to be effective in making framework model synthesis performant.

Additional Model Filtering

In addition to filtering framework models that rely on aliasing not possible in the issue set, there are a few other classes of framework models that have been found to be not useful in the search. Our implementation detects these classes of framework model and filters them out in the Filter method.

First, there are framework models that can be trivially shown to be equivalent to false. One class of models is when a back message requires that the same back message has happened in the past. For example, $\mathsf{cb}\ a.onCreate() \boxminus\!\!\rightarrow \mathsf{O}\ \mathsf{cb}\ a.onCreate() \wedge \ldots$ effectively prevents the `onCreate` callback since each new invocation of `onCreate` requires a previous one.

Another class of framework models that we avoid are models where an existentially quantified variable only shows up in a negative temporal operator. For example, a framework model of the form $\forall l.\mathsf{cb}\ l.\mathtt{onClick()} \boxminus\!\!\rightarrow \exists b.\mathtt{HN}\ \mathsf{ci}\ b.\mathtt{hideButtonWithListener(l)}$ problematic since the temporal operator

$\texttt{HN}$ ci $b.\texttt{hideButtonWithListener(l)}$ is always false as an arbitrary b instance can be chosen. Instead, the framework model should have a "do not care" variable represented by _ in the history implication: $\forall l.\mathsf{cb}\ l.\texttt{onClick()} \boxminus\!\!\rightarrow \texttt{HN}\ \mathsf{ci}\ \_.\texttt{hideButtonWithListener(l)}$. This is equivalent to universal quantification over the b $\forall l, b.\mathsf{cb}\ l.\texttt{onClick()} \boxminus\!\!\rightarrow \texttt{HN}\ \mathsf{ci}\ b.\texttt{hideButtonWithListener(l)}$ (the underscore and universal quantification is discussed further in subsection 4.2.1 with symbolic messages $\widetilde{m}$).

Finally, we avoid usages of the Not Since operator where the negative message has variables not in the positive message. For example $\mathsf{ci}\ b_1.\texttt{setEnabled}(true)\ \texttt{NS}\ \mathsf{ci}\ b_2.\texttt{setEnabled}(false)$ as opposed to both messages having the b receiver in the original history implication. Avoiding this last class of framework models is simply a heuristic based on the observation that "toggling" behavior typically does not span multiple objects.

6.4 Empirical Evaluation

To evaluate our contributions, we implemented our techniques in a tool named Synthoria. Here, we evaluate the possibility of verification of callback order sensitive assertions when the user of Synthoriais not able to write a full model of the framework. In particular, we test the ability of our technique to (1) reject unsound framework models given an example of a reachable location and (2) efficiently search for a consistent model of callback control flow that is sufficient to prove the fixed version of the application.

Implementation The abduction via program synthesis described in subsection 2.5.1 is implmented via a work-list algorithm over candidate framework models. Message graphs from section 6.3 are constructed using the result of the instrumented application-only control flow graph and SPARK from the Soot framework [76] described in subsection 2.5.1.

In this section, we ask the following research question:

Feasibility: Is it possible to abduce a sound framework model for real-world bug-fixes? Can we avoid unsound framework models using a **reasonable** number of reachable locations?

To answer this question, we performed an experiment to test whether temporal constraints could be synthesized to prove fixes of callback-order sensitive bugs. We use the benchmarks identified for evaluating Historia (research question 2 in subsection 4.4.1). These benchmarks were selected by watching the issue trackers of popular open-source Android applications and selecting stack traces where the fix alters the possible order of callbacks from the framework. We list these benchmarks in the first column of Table 6.1 by the app name (AP for Antennapod and CB for ConnectBot) and the pull request fixing the issue.

As mentioned by the overview, we can start with a framework model and search space based on partial specifications. The header ”input” on Table 6.1 gives statistics on this template as well as the reachable locations. This partial framework comes as a CBCFTL specification with holes where the user of our tool does not know what to write. We refer to the CBCFTL specification with holes as a template. The table captures the size and complexity of this template by counting the number of history implications in the column ”specs”, counting the messages in the right-hand side of the history implications with ”msg”, and finally, counting the number of variables that bind to objects in the ”obj” column. Next we show the set of messages that were identified by the user of our tool as being relevant to the the holes in the history implications and count these messages in the ”search msg” column.

As discussed in the overview, it is reasonable to assume that reachable locations could eventually be drawn from logging systems such as Firebase that capture stack traces or other logging data. Due to the current scalability of the Historia analysis, we provide programs and locations as unsound framework models are proposed by Synthoria. In addition to buggy versions of the applications as shown in the overview, we add simple applications and reachable locations as needed. These are simple applications that contain callbacks in the unsound proposed history implication and a finite application state that is updated as callbacks are invoked. Intuitively, such locations could come from normal patterns in Android applications such as initializing and using an object in separate callbacks. To measure how much runtime information was needed to synthesize the model, we tracked the number of times we needed to add a reachable issue in the first column.

Issue	learned model										performance			
	template			search		added			probability		Historia		Wistoria	
	specs (n)	msg (n)	obj (n)	msg (n)	models (n)	iss (n)	msg (n)	obj (n)	optim (1/n)	naive (1/n)	calls (n)	time (s)	calls (n)	time (s)
`getAct`[8]	3	3	4	5	8	9	3	1	3.17e-3	1.13e-4	11	13	63	25
`execute`[10]	3	3	4	2	22	5	3	0	1.98e-3	2.27e-4	29	2214	60	4239
`dismiss`[12]	2	4	3	4	3	10	1	0	2.50e-1	1.25e-1	5	735	27	9
`finish`[null]	2	0	3	5	346	7	5	1	6.13e-6	8.87e-10	436	1675	1820	858
`subs`[null]	6	10	10	3	9	2	2	1	1.05e-2	3.90e-4	11	569	14	8
total	16	20	24	19	388	33	14	3			492	5206	1984	5139

Table 6.1: Benchmarks demonstrating fixes for issues in widely-used open-source Android applications. We were able to synthesize framework models sufficient to prove each fix and document the size of the starting specification as well as the size of the learned model and the performance of the synthesis.

Discussion of the results The bottom line result is that we were able to use our synthesis approach to complete holes in all 5 benchmarks. In general, we find that the number of issues is fairly small maintaining the tractability of the problem. We never found a case where an unsound framework model was proposed that we could not eliminate by writing an app with a known reachable location.

The next header ”learned model” describes the complexity of the specifications synthesized in comparison to the template. The ”added msg” column is the number of times the syntactic expansion of a history implication added a single message. Additionally, we count the number of times a variable was added during synthesis with the ”added obj” column.

In general, we found that the synthesis algorithm would never terminate if any single optimization we described was turned off. This was primarily due to the extremely large search space of the problem. However, another challenging aspect is that many candidate framework models can result in extremely long runtimes from Historia. Fortunately, we found that almost every case of a proposed model that caused long run times in Historia also did not correspond to reasonable behavior that could be implemented in an event-driven framework. In order to evaluate how much of the search space is being cut down by the optimizations, we use the probability that random choices would result in the target framework model as a proxy for search space size and compare this probability between the optimized version and a naive expansion of the holes in the framework model. These probabilities are listed under the ”probability” header which is sub-divided into the ”optimized” and ”naive” probabilities. We find that the optimizations typically cut the number of bad choices down by an order of magnitude. In the extreme case of `dismiss`[12] , we find that these choices are cut down by 4 orders of magnitude.

Finally, we discuss the performance of our technique. First, we count the number of intermediate template framework models needed to be evaluated to get to the final model in the ”models” column. Since the bottleneck is running the Historia analysis and corresponding reachability analysis, the fewer intermediate templates, the more efficient the technique will be. In general, we find the synthesis to finish with at most a few hundred intermediate template models as compared to the models required if we did naive enumeration up to the size of the model we found. The total time taken by Synthoria is within a second of the sum of the time to run Historia and the reachability analysis (recorded in the ”historia time” and ”reach time” columns). We also record the total number of calls to Historia and the reachability analysis in the corresponding ”calls” columns. The average time per call is significantly lower than one would expect if each call was made from scratch. In order to further improve the runtime, we aggressively cached arguments and results for intermediate results within both the reachability analysis and Historia.

6.4.1 Threats to Validity

The biggest threat to validity is the assumption that suitable reachable locations may be found to contradict each unsound intermediate model. However, we believe that this threat to validity is somewhat mitigated by the fact that such reachability data may be recorded with existing instrumentation quite easily through issue trackers, logging systems, and sampling profilers. Ideally, the extremely large amount of code being run by many users is enough to provide the reachability data needed to avoid such unsound models. The next threat to validity is the scalability of the technique. Future work is needed to both select the reachable locations from a large corpus before running the goal-directed analysis and to improve the runtime of the analysis itself. However, such work is beyond the scope of the current paper.

6.5 Related Work

There are only a couple of techniques that attempt to model event-driven frameworks automatically from observing the running system. Synthesizing framework models for symbolic execution is the closest related work [68] that can passively synthesize framework modeling code. In this paper, runtime instrumentation is used to record events triggered at the event loop and the corresponding callbacks. From there, the sketch program synthesis technique [118] is used to fill in design patterns to trigger the callbacks. However, the limitation to this work is that the synthesized framework model cannot capture properties about the code that enqueues events such as the `onClick` described in the overview Figure 2.2. To overcome this limitation, Droidstar [109] investigates an active learning approach using the L* algorithm [8] to learn a finite state input/output automata known as a Mealy machine [116]. There are three limitations to the Droidstar work: (1) Mealy machines are limited to a finite alphabet meaning that the relationship between multiple interacting components cannot be effectively expressed. For example, the unbounded number of interacting buttons and listener objects illustrated in Figure 1.2. (2) Since it is an active learning technique, runtime instrumentation needs to be built that sends and receieves each kind of message that needs to be modeled. Such instrumentation is not only difficult to write, but also may not capture the variety of values and other contexts associated with such messages. (3) Active learning must be performed on each kind of device and version of the framework to soundly represent the behavior across application users. Due to the large number of devices, such repetition would be practically difficult. In contrast, our technique only needs crash reports from these devices, a kind of data that is already logged in many applications. Similarly, there are techniques that use recorded traces to mine type state like properties [48, 106, 132, 135]. However, these techniques cannot handle the presence of callbacks.

Another class of framework modeling techniques directly summarize or augment the framework source code to be directly analyzable [18, 25]. Such techniques showed promise when reasonable portions of the Android framework could be obtained and analyzed. However, since these works were published, the use of techniques that effectively "hide" framework code have seen increasing utilization in the Android framework. These techniques include dynamically loaded code, native code, interaction with the operating system, and interprocess communication. To make matters worse, the specific implementation of the framework that these techniques utilize tends to change between devices, manufacturers, and age of the device.

Similar to how simpler CBCFTL formula tend to be the best explanation to prove safety, there are techniques that leverage techniques to minimize the complexity of assumptions. For example, the paper "Specification Inference Using Context-Free Language Reachability" [13] uses a specification inference algorithm to find a minimal set of assumptions proving an application to be free of data flow related security issues.

Programming by Examples [56, 20] is a related idea where a program is generated by input output pairs. Our technique builds on this as input output pairs are a program location and then a "proof of unreachability" a "proof of reachability" or a "I don't know". The main contribution of this paper over previous programming by example techniques is the ability to correctly deal with under and over-approximation of program analysis while additionally using domain specific information to prune the search space of candidate models. Our work can be thought of as a kind of angelic verification specific to event ordering [31]. However, we add the important constraint that reachable locations must remain reachable under any appropriate framework model.

Chapter 7

Conclusion and Future Directions

Over the last few decades, event-driven frameworks have evolved enormously. Considering early computers such as the Xerox Alto with simple graphical user interfaces, events were limited to actions like clicking on items, entering text, or serial communication. However, over the following 50 years, event driven frameworks have taken over our lives. A modern cell phone has utilities to provide events for an incredible variety of functionality. There are events that report biometrics for monitoring health metrics such as SPO2 and heart rate. Other events can activate routines when a user enters a geographic area. Further events report physical properties of the environment that were well beyond what anyone expected even a decade ago. These capabilities will continue to improve with technology and the corresponding event-driven frameworks will be the backbone of how applications interact with the world.

Alongside this rapid improvement in functionality, we have also seen an incredible need for event-driven code to be secure and reliable. At the time of writing this dissertation, there is an explosion in safety critical uses for applications. Companies are starting to experiment with phone linked smartwatches that monitor for atrial fibrillation, low SPO2 and more. Pilots are switching out charts and manuals for tablets. Even cars on the road are increasingly operated using a touch screen connected to an event-driven framework.

At the same time, all of these systems run on a wide variety of different event-driven frameworks and utilize varying hardware. This explosion in complexity and capability means that verifying correct operation of these devices is going to continue to increase in difficulty. In particular, the added complexity from new features, different event-driven frameworks, and varying behavior between devices will push the limits of our capability to verify safety in real-world applications.

In this dissertation, we have presented a basis for soundly modeling event-driven framework behavior and using those models for program analysis in real-world Android applications. Then, to support the ever-increasing complexity and variety of event-driven frameworks, we presented a method of automatically synthesizing framework models. We have shown how this technique can target the properties that applications rely on for safety and how the resulting models can meet a high bar of soundness. However, there is still much work to be done to make program verification a tool that developers can rely on. Next, we will discuss the future directions that could develop this technology into a useful tool for real-world application developers.

7.0.1 Future Directions

This dissertation lays a foundation for message-history based modeling and analysis in event-driven software that enables the possibility of a wide variety of future work. These are a few of the research directions that are particularly interesting.

Scaling Framework Model Synthesis

The first research direction is to scale our synthesis approach to handle observed runtime behavior from crash reporting and logging utilities. Whereas we were able to synthesize framework models with 10 or fewer reachable locations, these locations were hand selected to demonstrate the unsound behavior. If the algorithms behind this synthesis were more efficient, we speculate that hand-selection of reachable issues would no longer be needed.

With over 3 billion active users of Android phones alone and logging utilities that report crashes and other critical application events, there is an enormous amount of information about how event-driven applications behave in the real world. The next major challenge is to utilize this information in a privacy-preserving way to improve our models of how event-driven frameworks operate. Our motivating example demonstrates why such data could be useful: rare behaviors of event-driven frameworks can cause software to behave badly. We believe it is unreasonable for an application developer to keep track of such rare behaviors. This is especially true for behaviors that may be specific to a particular piece of hardware or version of the framework.

Improving the Expressiveness of Framework Models

Up until now, our framework models have been fairly targeted at properties that applications rely on to avoid crashes such as buttons that may be enabled or disabled. Throughout the evaluation section, there are many other examples of such toggling behavior such as canceling background tasks or the framework indicating that an object is created or destroyed. Our fragment of temporal logic is well suited to modeling and reasoning about such properties. However, There are many other frameworks that deal with specific kinds of data that aren't precisely expressible in CBCFTL or efficient when interpreting abstract states with SMT.

The primary example of a property to express would be numeric properties where arithmetic operations matter. As the current theory and implementation are, it is possible to capture constant values such as **`false`** in the message `cb` $b.setEnabled(false)$. More specifically, constants are converted to uninterpreted sorts and axioms (e.g. there exists **`true`** and **`false`** in the set of booleans and they are not equal). However, it would be more difficult to express the behavior when numbers are involved. One such example would be proving timing dependent properties of event-driven systems. In such a system, the time of arrival of the event could be considered a parameter and safe operation may require that a subsequent callin happens quickly or that a callback does not block the user interface thread for too long. Work has been done to model timing dependent systems with approaches such as timed automata [3]. Extensions of these techniques may prove useful for event-driven frameworks. Numeric properties may also be useful when considering concurrency or distributed protocols that may rely on numeric relationships. As an example, distributed leader election algorithms [49] often require operations such as "if my identifier is larger than the previous node then perform a specific action". It is also reasonable to imagine applications where floating point numbers matter such as input from sensors. For example, a program monitoring someone's heart rate or geographic position may need to prove a safety property relating to the values reported in events.

In a different direction, it is also possible to imagine programs that depend on a more expressive language of events. CBCFTL focused on a few temporal operators capturing messages that happened Once, Not Since, or Have Not happened. However, there are event-driven systems that rely on events evolving in a more nuanced way. There is a large body of work on register automata, LTL with freeze quantifiers, and more [34, 32, 35] One example would be terminal applications and serial connections often send data in chunks of text. In such a system, each character could be considered an event making expressing more specific properties important. An extreme example would be the G-Code communication protocol which is used to tell a 3d printer or other computer controlled tool how to move. G-Code usually sends commands like “move to a given location” or “set the temperature to 60 degrees”. However, G-Code can behave like a programming language with conditional “goto” statements and variables. In such a case, the language of events may need to be Turing complete.

Supporting Dynamic Languages

Another avenue of future work would be to support dynamic languages such as Javascript or Python. Such languages are extremely popular in a wide variety of software systems including web pages, servers, and applications running on phones. However, the software that truly motivates such extension of these techniques is the Javascript based user interface on the flight control system of the SpaceX Dragon capsule [120]. Work has been done to use the goal-directed analysis for Javascript applications [126].

One of the biggest challenges with extending the work in this dissertation to dynamic languages will be the construction of call graphs and pointer analysis. Due to the nature of dynamic languages call graphs and pointer analysis are significantly more challenging to construct [66, 121, 70]. Since our tool relies on such pointer analysis and call graphs to scale, the performance and soundness will be a challenge when analyzing applications in dynamic languages.

The Most Crucial Aspect: Software Engineering

Finally, the most crucial future work is the software engineering to integrate these techniques into the workflows of real developers. It is well known that program analysis tools can be extremely helpful to developers if used well. However, the process of integrating these tools into the workflows of developers is extremely challenging [38]. False alarms pointing to defects that don’t exist can waste the limited and expensive time of developers. On the other hand, false negatives can mean that critical problems go unnoticed. With the ever-increasing criticality of software in our daily lives, it is important that we explore every option to improve the quality and safety of the computer systems that run our world.

Bibliography

[1] Karim Ali and Ondrej Lhoták. Application-only call graph construction. In European Conference on Object-Oriented Programming (ECOOP), volume 7313, 2012.

[2] Karim Ali and Ondrej Lhoták. Averroes: Whole-program analysis without the whole program. In European Conference on Object-Oriented Programming (ECOOP), volume 7920, 2013.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. Theor. Comput. Sci., 126(2):183–235, 1994.

[4] Android Developers. The Activity lifecycle. `https://developer.android.com/guide/components/activities/activity-lifecycle.html`, 2018.

[5] Android Developers. UI/Application exerciser monkey. `https://developer.android.com/studio/test/monkey.html`, 2018.

[6] Android Developers. Class index (api level 32). `https://developer.android.com/reference/classes`, 2024.

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

[8] Dana Angluin. Learning regular sets from queries and counterexamples. Inf. Comput., 75(2):87–106, 1987.

[9] Yauhen Leanidavich Arnatovich, Minh Ngoc Ngo, Hee Beng Kuan Tan, and Charlie Soh. Achieving high code coverage in android UI testing via automated widget exercising. 2016.

[10] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the Android framework. In International Conference on Software Engineering (ICSE), 2016.

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

[12] George Balatsouras and Yannis Smaragdakis. Class hierarchy complementation: soundly completing a partial type graph. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pages 515–532. ACM, 2013.

[13] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, 2015.

[14] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021, 2021.

[15] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for Android applications. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2015.

[16] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: precise refutations for heap reachability. In Programming Language Design and Implementation (PLDI), 2013.

[17] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2015.

[18] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to Android framework modeling. In State of the Art in Program Analysis (SOAP), 2015.

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

[20] Ruven E. Brooks. "watch what I do: Programming by demonstration, " edited by allen cypher (book review). Int. J. Man Mach. Stud., 39(6), 1993.

[21] Business of Apps. Android statistics 2024, 2024. Accessed: 2024-05-29.

[22] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In NASA Formal Methods (NFM), 2011.

[23] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In NASA Formal Methods (NFM), 2015.

[24] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. HornDroid: Practical and sound static analysis of Android applications by SMT solving. In European Symposium on Security and Privacy (EuroS&P), 2016.

[25] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In Network and Distributed System Security (NDSS), 2015.

[26] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In Computer-Aided Verification (CAV), 2014.

[27] Adrian Chifor. Swiftnotes. `https://f-droid.org/en/packages/com.moonpi.swiftnotes/`, 2015.

[28] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981, volume 131 of Lecture Notes in Computer Science, pages 52–71. Springer, 1981.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Principles of Programming Languages (POPL), pages 238–252, 1977.

[30] D120. Kistenstapeln. `https://github.com/d120/Kistenstapeln-Android`, 2015.

[31] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, volume 9206, 2015.

[32] Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, and Sasha Rubin. Pure-past linear temporal and dynamic logic on finite traces. In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence, pages 4959–4965, 2021.

[33] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 4963, 2008.

[34] Stéphane Demri and Ranko Lazić. Ltl with the freeze quantifier and register automata. ACM Trans. Comput. Logic, 10(3), apr 2009.

[35] Stéphane Demri and Ranko Lazić. Ltl with the freeze quantifier and register automata. ACM Transactions on Computational Logic (TOCL), 10(3):1–30, 2009.

[36] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In Programming Language Design and Implementation (PLDI), 2013.

[37] Ivan Dimoski. Top 10 most common android development mistakes. `https://www.toptal.com/android/top-10-most-common-android-development-mistakes`. Accessed: 2024-04-09.

[38] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. Commun. ACM, 62(8), 2019.

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

[40] Facebook. Infer: A static analysis tool. `https://fbinfer.com/`, 2024. Accessed: April 17, 2024.

[41] Martin Fietz. 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`, 2015.

[42] Martin Fietz. FeedRemover: already running - issue #1304 - AntennaPod/AntennaPod. `https://github.com/AntennaPod/AntennaPod/issues/1304`, 2015.

[43] Martin Fietz. Fix dismiss illegalargumentexception. `https://github.com/AntennaPod/AntennaPod/issues/2148`, 2016.

[44] Martin Fietz. 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, 2018.

[45] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol., 17(2), 2008.

[46] Jeffrey Fischer, Rupak Majumdar, and Todd D. Millstein. Tasks: language support for event-driven programming. 2007.

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

[48] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In Mary Jean Harrold and Gail C. Murphy, editors, Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008, pages 339–349. ACM, 2008.

[49] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst., 5(1):66–77, 1983.

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

[51] Marko Gargenta and Masumi Nakamura. Learning Android. 2014.

[52] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In International Joint Conference on Artificial Intelligence (IJCAI), 2013.

[53] Google. Android is for everyone, 2024. Accessed: 2024-05-29.

[54] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. In Network and Distributed System Security (NDSS), 2015.

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

[56] Sumit Gulwani. Programming by examples - and its applications in data wrangling. In Dependable Software Systems Engineering, volume 45. 2016.

[57] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. Found. Trends Program. Lang., 4(1-2), 2017.

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

[59] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, 1969.

[60] Gerard J. Holzmann. The model checker SPIN. IEEE Trans. Software Eng., 23(5):279–295, 1997.

[61] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. Race detection for event-driven mobile applications. In Programming Language Design and Implementation (PLDI), 2014.

[62] Yongjian Hu and Iulian Neamtiu. Static detection of event-based races in android apps. In Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018.

[63] Huaxun Huang, Lili Wei, Yepang Liu, and Shing-Chi Cheung. Understanding and detecting callback compatibility issues for android applications. In Automated Software Engineering (ASE), 2018.

[64] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Principles of Programming Languages (POPL), 2001.

[65] Casper Svenning Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin T. Vechev. Stateless model checking of event-driven applications. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2015.

[66] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings, volume 5673 of Lecture Notes in Computer Science, pages 238–255. Springer, 2009.

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

[68] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, 2016.

[69] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded Java programs. In Automated Software Engineering (ASE), 2008.

[70] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for javascript. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, pages 121–132. ACM, 2014.

[71] 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. Lessons learned from the chameleon testbed. In USENIX. 2020.

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

[73] Konstantin Korovin. Non-cyclic sorts for first-order satisfiability. In Frontiers of Combining Systems (FroCoS), volume 8152, 2013.

[74] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. Proc. ACM Program. Lang., 6(OOPSLA1), 2022.

[75] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. Higher-Order and Symbolic Computation, 19(4), 2006.

[76] Ondrej Lhoták and Laurie J. Hendren. Scaling Java points-to analysis using SPARK. In Compiler Construction (CC), 2003.

[77] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, 2002.

[78] Wenhua Li, Quang Loc Le, Yahui Song, and Wei-Ngan Chin. Incorrectness proofs for object-oriented programs via subclass reflection. In Programming Languages and Systems - 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26-29, 2023, Proceedings, volume 14405, 2023.

[79] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 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), 2013.

[80] Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. The glory of the past. In Logics of Programs, 1985.

[81] Codeverse Weenggs Solutions LLP. Common errors in android app development. https://www.codementor.io/android/tutorial/most-common-android-errors-how-to-fix-them, 2023. Accessed: 2024-04-09.

[82] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven Node.Js javascript applications. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2015.

[83] Pallavi Maiya, Rahul Gupta, Aditya Kanade, and Rupak Majumdar. Partial order reduction for event-driven multi-threaded programs. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2016.

[84] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In Programming Language Design and Implementation (PLDI), 2014.

[85] Zohar Manna and Amir Pnueli. The temporal logic of reactive and concurrent systems - specification. 1992.

[86] Mariana Trench. Mariana trench. https://mariana-tren.ch/, 2022.

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

[88] Shawn Meier, Sergio Mover, and Bor-Yuh Evan Chang. Lifestate: Event-driven protocols and callback control flow. In European Conference on Object-Oriented Programming (ECOOP), volume 134, 2019.

[89] Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. Historia: Refuting callback reachability with message-history logics. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2023.

[90] Atif M. Memon. An event-flow model of GUI-based applications for testing. Softw. Test., Verif. Reliab., 17(3), 2007.

[91] Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2008.

[92] NextGis. NextGisLogger. `https://github.com/nextgis/nextgislogger`, 2017.

[93] Peter W. O'Hearn. Incorrectness logic. Proc. ACM Program. Lang., 4(Principles of Programming Languages (POPL)), 2020.

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

[95] OpenAI. Chatgpt, 2023. Available from OpenAI: `https://www.openai.com/chatgpt`.

[96] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang., 1(OOPSLA), 2017.

[97] Linjie Pan, Baoquan Cui, Hao Liu, Jiwei Yan, Siqi Wang, Jun Yan, and Jian Zhang. Static asynchronous component misuse detection for android applications. In European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE), 2020.

[98] Linjie Pan, Baoquan Cui, Jiwei Yan, Xutong Ma, Jun Yan, and Jian Zhang. Androlic: an extensible flow, context, object, field, and path-sensitive static analysis framework for android. In Software Testing and Analysis (ISSTA), 2019.

[99] Étienne Payet and Fausto Spoto. An operational semantics for Android activities. In Partial evaluation and program manipulation, (PEPM), 2014.

[100] Danilo Dominguez Perez and Wei Le. Predicate callback summaries. In International Conference on Software Engineering (ICSE), 2017.

[101] Danilo Dominguez Perez and Wei Le. Specifying callback control flow of mobile apps using finite automata. IEEE Trans. Software Eng., 47(2), 2021.

[102] PingPlusPlus. Ping Plus Plus. `https://github.com/PingPlusPlus/pingpp-android`, 2017.

[103] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In Computer-Aided Verification (CAV), 2013.

[104] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015, 2015.

[105] Steve Pomeroy. The complete Android Activity/Fragment lifecycle v0.9.0. `https://github.com/xxv/android-lifecycle`, 2014.

[106] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009, pages 371–382. IEEE Computer Society, 2009.

[107] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings, volume 137 of Lecture Notes in Computer Science, pages 337–351. Springer, 1982.

[108] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II, volume 12225, 2020.

[109] Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Cerný. Droidstar: callback typestates for android classes. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, 2018.

[110] Red Reader. Crash during commenting #467 RedReader. `https://github.com/QuantumBadger/RedReader/issues/467`, 2017.

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

[112] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In Logic in Computer Science (LICS), 2002.

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

[114] Atanas Rountev and Dacong Yan. Static reference analysis for GUI objects in Android software. In Code Generation and Optimization (CGO), 2014.

[115] sh1ro. NovelDroid. `https://github.com/sh1r0/NovelDroid/blob/f3245055d7a8bcc69a9bca278fbe890081dac58a/app/src/main/java/com/sh1r0/noveldroid/SettingsFragment.java`, 2016.

[116] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In Ana Cavalcanti and Dennis Dams, editors, FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings, volume 5850 of Lecture Notes in Computer Science, pages 207–222. Springer, 2009.

[117] Eric Smith and Alessandro Coglio. Android platform modeling and Android app verification in the ACL2 theorem prover. 2015.

[118] Armando Solar-Lezama. The sketching approach to program synthesis. In Zhenjiang Hu, editor, Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings, volume 5904 of Lecture Notes in Computer Science, pages 4–13. Springer, 2009.

[119] Fu Song and Zhilin Wu. On temporal logics with data variable quantifications: Decidability and complexity. Inf. Comput., 251, 2016.

[120] SpaceX Software Team. We are the spacex software team, ask us anything! Reddit AMA, June 2020. Accessed: 2024-06-08.

[121] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble, editor, ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings, volume 7313 of Lecture Notes in Computer Science, pages 435–458. Springer, 2012.

[122] StackOverflow Post. Got exception: fragment already active. `https://stackoverflow.com/questions/10364478/got-exception-fragment-already-active`, 2012.

[123] StackOverflow Post. Alertdialog creating exception in android. `https://stackoverflow.com/questions/15104677/alertdialog-creating-exception-in-android`, 2013.

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

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

[126] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static analysis with demand-driven value refinement. Proc. ACM Program. Lang., 3(OOPSLA):140:1–140:29, 2019.

[127] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng., 12(1), 1986.

[128] Jessica Thornsby. Infographic: 11 most common android errors and how to fix them. `https://www.codementor.io/android/tutorial/most-common-android-errors-how-to-fix-them`. Accessed: 2024-04-09.

[129] Matthias Urhahn. AudioBug. `https://github.com/d4rken/audiobug`, 2017.

[130] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In Proceedings of CASCON 1999, 1999.

[131] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. Autom. Softw. Eng., 10(2):203–232, 2003.

[132] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy, pages 248–257. IEEE Computer Society, 2008.

[133] Yan Wang, Hailong Zhang, and Atanas Rountev. On the unsoundness of static analysis for Android GUIs. In State of the Art in Program Analysis (SOAP), 2016.

[134] Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. Precise static happens-before analysis for detecting UAF order violations in android. In Conference on Software Testing, Validation and Verification, ICST, 2019.

[135] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006, pages 282–291. ACM, 2006.

[136] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. Autom. Softw. Eng., 25(4), 2018.

[137] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In International Conference on Software Engineering (ICSE), 2015.

Appendix A

Appendix: Refuting Callback Reachability with Message History Logics

A.1 Assumptions of the Application-Only Transition System

In practice, the translation from a compiled application to the application-only transition system makes some assumptions about the execution environment that we list in this section. These assumptions simplify reasoning about concurrency, reflection, and library behavior. First, we assume that all callbacks occur on the same thread (i.e., individual statements inside two separate callbacks cannot be interleaved during execution). In practice, some callbacks do occur concurrently. For example, the callback passed to `Single.create(...)` in the place of the `...` is typically executed on a background thread. Specific kinds of program analysis exist to address the interleaving caused by threaded concurrency, but that is out of scope for our paper. Additionally, we do not consider exceptional control flow involving try/catch blocks.

For the application only transition system, we assume that the framework cannot directly modify the app store except through the invocation of a callback. In theory, the framework may directly modify fields such as `this.act` generating an execution not represented by the application only transition system. However, such direct access is rare. Since the framework must be compiled without the application, any method or field that the framework accesses must typically extend a type declared by the framework (e.g., `Fragment` which declares an abstract method `onCreate`). The only exception is reflection, which may be used to access the app store such as the `this.act` field directly, but we assume such uses of reflection are rare, as it breaks encapsulation. More commonly, the framework uses reflection to create instances of objects such as `PlayerFragment`. Ali 2012 [1] coined the phrase **separate-compilation assumption** to describe how a reasonable set of assumptions on framework behavior may be used to generate an application-only call graph. Our application-only transition system follows the same principle and is based on the same assumptions.

As mentioned in the Overview (subsection 2.3.3), we assume callin invocations do not **synchronously** call back to the app (i.e., a callin stays in framework code until it returns). This assumption is frequently mirrored by other static analysis, such as Flowdroid [10], by framework stub implementations that do not invoke callbacks. In general, we find that not many Android methods have synchronous callbacks. In fact, none of our examples use a method exhibiting this behavior.

This assumption is reflected by the single $\ell \dashv\!\![\mathsf{ci}\; x'\; md(\overline{x})]\!\!\mapsto \ell'$ boundary transition. Our boundary stacks are degenerate here in that they will only ever have at most one activation k (for the active callback). The boundary stack κ is conceptually the subsequence of the run-time call stack corresponding to boundary transitions (i.e., the pending calls alternating between callbacks and callins), which ensures that the message history consists of matching calls and returns.

One can extend the language to support synchronous callbacks by splitting the callin invocation boundary transition in two: one for the callin invocation into the framework (i.e., from ℓ to Fwk) and one for a call return back from the framework (i.e., from Fwk to ℓ'), analogous to callback invocations and callback returns. Note that if synchronous callbacks can be soundly modeled as asynchronous ones, then it is also unnecessary to add the complexity of synchronous callbacks (which is generally the case for Android).

A.2 Proofs for Message History Program Logic

In this section, we prove the theorems for the message history program logic (MHPL) that we describe in section 4.1. We make the following assumptions:

Assumption 1 (Application Step Soundness)**.** If $\widehat{\Sigma} \vdash t$ and $\langle \sigma', t \rangle \Downarrow \sigma$ and $\sigma \models_S \widehat{\Sigma}(\text{post}(t))$ then $\sigma' \models_S \widehat{\Sigma}(\text{pre}(t))$.

Providing a sound specification is the responsibility of the user of Historia.

Property 1 (All Prefixes are Realizable)**.** If $\omega; m \in \Omega$ then $\omega \in \Omega$.

Property 2 (Sound Excludes Init)**.** If $\vdash_S \widehat{\sigma}$ $\mathsf{excludesinit}$ then $\sigma_{\text{init}} \not\models_S \widehat{\sigma}$.

Property 3 (Entailment Soundness)**.** If $\widehat{\sigma} \vdash_S \widehat{\sigma}'$ and $\sigma \models_S \widehat{\sigma}'$ then $\sigma \models_S \widehat{\sigma}$.

The sound excludes init Property 2 is a consequence of the sound encoding of message history program logics section 4.3, a sound first order logic encoding, and a sound SMT solver.

$$\boxed{\sigma \models_S \widehat{\sigma}} \qquad loc_1 : \mu \models_S loc_2 : \widehat{\mu} \quad \text{iff} \quad loc_1 = loc_2 \text{ and } \mu \models_S \widehat{\mu}$$

$$\boxed{\mu \models_S \widehat{\mu}}$$

$$\mu \models_S \widehat{\mu}_1 \vee \widehat{\mu}_2 \quad \text{iff} \quad \mu \models_S \widehat{\mu}_1 \text{ or } \mu \models_S \widehat{\mu}_2$$

$$\omega \cdot \kappa \cdot \rho \models_S \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} \quad \text{iff} \quad \text{there exists } \theta \text{ such that } \omega \cdot \theta \models_S \widehat{\omega} \text{ and } \rho \cdot \theta \models \widehat{\rho} \text{ and } \kappa \cdot \theta \models \widehat{\kappa}$$

$$\boxed{\rho \cdot \theta \models \widehat{\rho}}$$

$$\rho \cdot \theta \models \top \qquad \rho \cdot \theta \models \widehat{\rho}_1 * \widehat{\rho}_2 \quad \text{iff} \quad \mathrm{dom}(\widehat{\rho}_1) \cap \mathrm{dom}(\widehat{\rho}_2) = \emptyset \text{ and } \rho \cdot \theta \models \widehat{\rho}_1 \text{ and } \rho \cdot \theta \models \widehat{\rho}_2$$

$$\rho \cdot \theta \models x \mapsto \hat{x} \quad \text{iff} \quad \rho(x) = v \text{ and } \theta(\hat{x}) = v \qquad \rho \cdot \theta \models \widehat{\rho}_1 \vee \widehat{\rho}_2 \quad \text{iff} \quad \rho \cdot \theta \models \widehat{\rho}_1 \text{ or } \rho \cdot \theta \models \widehat{\rho}_2$$

$$\boxed{\kappa \cdot \theta \models \widehat{\kappa}}$$

$$\kappa \cdot \theta \models \top \qquad \kappa; \mathsf{cb}\, md(\overline{v}) \cdot \theta\overline{[\hat{x} \mapsto v]} \models \widehat{\kappa} \bullet \mathsf{cb}\, md(\overline{\hat{x}})$$

$$\boxed{\sigma_{\text{init}} \rightarrow^{*\Omega}_{p} \sigma} \qquad \text{C-TRANSITIVE-STEP}\ \frac{\sigma = \sigma'}{\sigma \rightarrow^{*\Omega}_{p} \sigma'} \qquad \text{C-TRNASITIVE-STEP-INDUCTIVE}\ \frac{\sigma \rightarrow^{*\Omega}_{p} \sigma'' \qquad \sigma'' \rightarrow^{\Omega}_{p} \sigma'}{\sigma \rightarrow^{*\Omega}_{p} \sigma'}$$

Figure A.1: Extended concretization and transitive step for application only transition system.

Lemma 3.1 (HOARE TRIPLE SOUNDNESS) If $\vdash \{\widehat{\sigma}_{\text{pre}}\}\ b\ \{\widehat{\sigma}_{\text{post}}\}$ and $\langle \sigma_{\text{pre}}, b\rangle \Downarrow^{\Omega} \sigma_{\text{post}}$ such that $\sigma_{\text{post}} \models_S \widehat{\sigma}_{\text{post}}$ and $\Omega \subseteq \Omega_S$, then $\sigma_{\text{pre}} \models_S \widehat{\sigma}_{\text{pre}}$.

Proof of Lemma 1. By cases on $\mathcal{D} = \langle \sigma, b_{\text{pre}}\rangle \Downarrow^{\Omega} \sigma_{\text{post}}$

$$\text{Case } \mathcal{D} = \text{C-CALLBACK-INVOKE}\ \frac{\omega; \mathsf{cb}\, md(\overline{v}) \in \Omega}{\langle \mathsf{Fwk}\colon \omega \cdot \kappa \cdot \rho, \mathsf{Fwk} \dashv\!\![\mathsf{cb}\, md(\overline{x})]\!\!\mapsto \ell\rangle \Downarrow^{\Omega} \ell\colon \omega; \mathsf{cb}\, md(\overline{v}) \cdot \kappa; \mathsf{cb}\, md(\overline{v}) \cdot \rho\overline{[x \mapsto v]}}$$

The only judgment that applies is:

$$\frac{}{\vdash \{\mathsf{Fwk}\colon \twoheadrightarrow \widehat{\omega}\mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\kappa}\cdot\widehat{\rho}\}\ \mathsf{Fwk} \dashv\![\mathsf{cb}\,md(\overline{x})]\!\mapsto \ell\ \{\ell\colon \widehat{\omega}\cdot\widehat{\kappa}\bullet \mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\rho} * \circledast\,\overline{x\mapsto\hat{x}}\}}\ \textsc{a-callback-invoke}$$

Let $\widehat{\sigma}_{\text{post}} = \ell\colon \widehat{\omega}\cdot\widehat{\kappa}\bullet \mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\rho} * \circledast\,\overline{x\mapsto\hat{x}}$.

Let $\widehat{\sigma}_{\text{pre}} = \mathsf{Fwk}\colon \twoheadrightarrow \widehat{\omega}\mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\kappa}\cdot\widehat{\rho}$.

Let $\sigma_{\text{post}} = \ell\colon \omega;\mathsf{cb}\,md(\overline{v})\cdot\kappa;\mathsf{cb}\,md(\overline{v})\cdot\rho\overline{[x\mapsto v]}$

Let $\sigma_{\text{pre}} = \mathsf{Fwk}\colon \omega\cdot\kappa\cdot\rho$

We have that $\sigma_{\text{post}}\cdot\theta \models_S \widehat{\sigma}_{\text{post}}$ holds for $\theta = \theta'\overline{[\hat{x}\mapsto v]}$ by the definition of concretization.

We prove that same assignment, θ ensures that $\sigma_{\text{pre}}\cdot\theta \models_S \widehat{\sigma}_{\text{pre}}$. We break this down into sub-cases for each part of the abstract state:

Sub Case: If $\omega;\mathsf{cb}\,md(\overline{v})\cdot\theta \models_S \widehat{\omega}$ and $\omega;\mathsf{cb}\,md(\overline{v}) \in \Omega$ then $\omega\cdot\theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{cb}\,md(\overline{\hat{x}})$.

$\mathsf{cb}\,md(\overline{v})\cdot\theta \models \mathsf{cb}\,md(\overline{\hat{x}})$ since $\theta = \theta'\overline{[\hat{x}\mapsto v]}$.

Since the premise of the lemma states $\Omega \subseteq \Omega_S$ then for all ω, $\omega\in\Omega$ implies $\omega \models S$.

Then from $\omega;\mathsf{cb}\,md(\overline{v}) \in \Omega$ we have that $\omega;\mathsf{cb}\,md(\overline{v}) \models S$.

Therefore, $\omega\cdot\theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{cb}\,md(\overline{\hat{x}})$ holds by the definition of concretization.

Sub Case: If $\kappa;\mathsf{cb}\,md(\overline{v})\cdot\theta \models \widehat{\kappa}\bullet\mathsf{cb}\,md(\overline{\hat{x}})$ then $\kappa\cdot\theta\models\widehat{\kappa}$.

Since $\theta = \theta'\overline{[\hat{x}\mapsto v]}$ this case holds by the definition of concretization.

Sub Case: If $\rho\overline{[x\mapsto v]}\cdot\theta \models \widehat{\rho} * \circledast\,\overline{x\mapsto\hat{x}}$ then $\rho\cdot\theta\models\widehat{\rho}$.

Since $\theta = \theta'\overline{[\hat{x}\mapsto v]}$ this case holds by the definition of concretization.

$$\text{Case } \mathcal{D} = \frac{m = \mathsf{cbret}\,\rho(x')\,md(\overline{v}) \qquad \overline{v} = \overline{\rho(x)} \qquad \omega;m\in\Omega}{\langle \ell\colon \omega\cdot\kappa;\mathsf{cb}\,md(\overline{v})\cdot\rho,\ \ell \dashv\![\mathsf{cbret}\,x'\,md(\overline{x})]\!\mapsto \mathsf{Fwk}\rangle \Downarrow^{\Omega} \mathsf{Fwk}\colon \omega;m\cdot\kappa\cdot\rho}\ \textsc{c-callback-return}$$

The only judgment that applies is:

$$\frac{\widehat{\rho} = \widehat{\rho'} * x' \mapsto \hat{x}' * \circledast\,\overline{x\mapsto\hat{x}}}{\vdash \{\ell\colon \twoheadrightarrow \widehat{\omega}\mathsf{cbret}\,\hat{x}'\,md(\overline{\hat{x}})\cdot\widehat{\kappa}\bullet\mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\rho}\}\ \ell \dashv\![\mathsf{cbret}\,x'\,md(\overline{x})]\!\mapsto \mathsf{Fwk}\ \{\mathsf{Fwk}\colon \widehat{\omega}\cdot\widehat{\kappa}\cdot\widehat{\rho}\}}\ \textsc{a-callback-return}$$

Let $\widehat{\sigma}_{\text{post}} = \mathsf{Fwk}\colon \widehat{\omega}\cdot\widehat{\kappa}\cdot\widehat{\rho}$.

Let $\widehat{\sigma}_{\text{pre}} = \ell\colon \twoheadrightarrow \widehat{\omega}\mathsf{cbret}\,\hat{x}'\,md(\overline{\hat{x}})\cdot\widehat{\kappa}\bullet\mathsf{cb}\,md(\overline{\hat{x}})\cdot\widehat{\rho} * x' \mapsto \hat{x}'$.

Let $\sigma_{\text{post}} = \mathsf{Fwk}\colon \omega;\mathsf{cbret}\,v'\,md(\overline{v})\cdot\kappa\cdot\rho$

Let $\sigma_{\text{pre}} = \ell\colon \omega\cdot\kappa;\mathsf{cb}\,md(\overline{v})\cdot\rho$

We have that $\sigma_{\text{post}} \cdot \theta \models_S \widehat{\sigma}_{\text{post}}$ holds for $\theta = \theta'[\hat{x}' \mapsto v']\overline{[\hat{x} \mapsto v]}$ by the definition of concretization.

We prove that $\sigma_{\text{pre}} \cdot \theta \models_S \widehat{\sigma}_{\text{pre}}$.

We break this down into sub-cases for each part of the abstract state:

Sub Case: If $\omega; \mathsf{cbret}\, v'\, md(\overline{v}) \cdot \theta \models_S \widehat{\omega}$ then $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{cbret}\, \hat{x}'\, md(\overline{\hat{x}})$

$\mathsf{cbret}\, v'\, md(\overline{v}) \cdot \theta \models \mathsf{cbret}\, \hat{x}'\, md(\overline{\hat{x}})$ since $\theta = \theta'[\hat{x}' \mapsto v']\overline{[\hat{x} \mapsto v]}$.

Since the premise of the lemma states $\Omega \subseteq \Omega_S$ then for all ω, $\omega \in \Omega$ implies $\omega \models S$.

Then from $\omega; \mathsf{cbret}\, v'\, md(\overline{v}) \in \Omega$ we have that $\omega; \mathsf{cbret}\, v'\, md(\overline{v}) \models S$.

Therefore, $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{cbret}\, \hat{x}'\, md(\overline{\hat{x}})$ holds by the definition of concretization.

Sub Case: If $\kappa \cdot \theta \models \widehat{\kappa}$ then $\kappa; \mathsf{cb}\, md(\overline{v}) \cdot \theta \models \kappa; \mathsf{cb}\, md(\overline{\hat{x}})$.

This case holds by the definition of concretization of the stack.

Sub Case: If $\rho[x' \mapsto v']\overline{[x \mapsto v]} \cdot \theta \models \widehat{\rho}' * x' \mapsto \hat{x}' ** \overline{x \mapsto \hat{x}}$

then $\rho[x' \mapsto v']\overline{[x \mapsto v]} \cdot \theta \models \widehat{\rho}' * x' \mapsto \hat{x}' ** \overline{x \mapsto \hat{x}}$.

This case holds trivially.

$$\text{Case } \mathcal{D} = \cfrac{\overline{v} = \overline{\rho(x)} \qquad m = \mathsf{ci}\, v'\, md(\overline{v}) \qquad \omega; m \in \Omega}{\langle \ell \colon \omega \cdot \kappa \cdot \rho, \ell \dashv [\mathsf{ci}\, x'\, md(\overline{x})] \mapsto \ell' \rangle \Downarrow^{\Omega} \ell' \colon \omega; m \cdot \kappa \cdot \rho[x' \mapsto v']}\ \textsc{c-callin-invoke}$$

The only judgment that applies is:

$$\cfrac{\widehat{\rho} = \widehat{\rho}' * * \overline{x \mapsto \hat{x}}}{\vdash \{\ell \colon \twoheadrightarrow \widehat{\omega}\mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}}) \cdot \widehat{\kappa} \cdot \widehat{\rho}\}\ \ell \dashv [\mathsf{ci}\, x'\, md(\overline{x})] \mapsto \ell'\ \{\ell' \colon \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} * x' \mapsto \hat{x}'\}}\ \textsc{a-callin-invoke}$$

Let $\widehat{\sigma}_{\text{post}} = \ell' \colon \widehat{\omega} \cdot \widehat{\kappa} \cdot \widehat{\rho} * x' \mapsto \hat{x}'$.

Let $\widehat{\sigma}_{\text{pre}} = \ell \colon \twoheadrightarrow \widehat{\omega}\mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}}) \cdot \widehat{\kappa} \cdot \widehat{\rho}$.

Let $\sigma_{\text{post}} = \ell' \colon \omega; \mathsf{ci}\, v'\, md(\overline{v}) \cdot \kappa \cdot \rho[x' \mapsto v']$

Let $\sigma_{\text{pre}} = \ell \colon \omega \cdot \kappa \cdot \rho$

We have that $\sigma_{\text{post}} \cdot \theta \models_S \widehat{\sigma}_{\text{post}}$ holds for any θ such that $\theta = \theta'\overline{[\hat{x} \mapsto v]}[\hat{x}' \mapsto v']$ by the definition of concretization.

We prove that $\sigma_{\text{pre}} \cdot \theta \models_S \widehat{\sigma}_{\text{pre}}$.

We break this down into sub-cases for each part of the abstract state:

Sub Case: If $\omega; \mathsf{ci}\, v'\, md(\overline{v}) \cdot \theta \models_S \widehat{\omega}$ then $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}})$

$\mathsf{ci}\, v'\, md(\overline{v}) \cdot \theta \models_S \mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}})$ since $\theta = \theta'\overline{[\hat{x} \mapsto v]}[\hat{x}' \mapsto v']$

Since the premise of the lemma states $\Omega \subseteq \Omega_S$ then for all ω, $\omega \in \Omega$ implies $\omega \models S$.

Then from $\omega; \mathsf{ci}\, v'\, md(\overline{v}) \in \Omega$ we have that $\omega; \mathsf{ci}\, v'\, md(\overline{v}) \models S$.

Therefore, $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{\omega}\mathsf{ci}\, \hat{x}'\, md(\overline{\hat{x}})$ holds by the definition of concretization.

Sub Case: If $\kappa \cdot \theta \models \widehat{\kappa}$ then $\kappa \cdot \theta \models \widehat{\kappa}$ holds trivially.

Sub Case: If $\rho[x' \mapsto v'] \cdot \theta \models \widehat{\rho} * x' \mapsto \hat{x}'$ then $\rho \cdot \theta \models \widehat{\rho}$.

This case holds by the definition of concretization of app stores.

□

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

Proof of Lemma 2. By inversion on $\widehat{\Sigma} \vdash_S b$ we have $\widehat{\Sigma}(\text{post}(b)) \vdash_S \widehat{\sigma}$ and $\vdash \{\widehat{\sigma}'\}\ b\ \{\widehat{\sigma}\}$ and $\widehat{\sigma}' \vdash_S \widehat{\Sigma}(\text{pre}(b))$.

By applying Property 3, we have $\sigma \models_S \widehat{\sigma}$.

By applying Lemma 1, we have $\sigma' \models_S \widehat{\sigma}'$.

By applying Property 3, we have $\sigma' \models_S \widehat{\Sigma}(\text{pre}(b))$.

□

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 $\sigma' \models_S \widehat{\Sigma}(\text{loc}(\sigma'))$.

Proof of Lemma 3. Proof by induction on the derivation $\mathcal{D} = \sigma_{\text{init}} \rightarrow_p^{*\Omega} \sigma$.

$$\text{Case } \mathcal{D} = \cfrac{\sigma = \sigma'}{\sigma' \rightarrow_p^{*\Omega} \sigma}\ \text{C-TRANSITIVE-STEP}$$

By inversion on C-TRANSITIVE-STEP, we have $\sigma' = \sigma$.

From $\widehat{\Sigma} \vdash_p^S \widehat{\sigma}$ and by inversion on A-INDUCTIVE, we have $\widehat{\sigma} \vdash_S \widehat{\Sigma}(\text{loc}(\widehat{\sigma}))$. From $\sigma \models_S \widehat{\sigma}$ and $\sigma' = \sigma$ and Property 3 we have $\sigma' \models_S \widehat{\Sigma}(\text{loc}(\sigma'))$ satisfying the consequent of Lemma 3.3.

c-trnasitive-step-inductive

$$\text{Case } \mathcal{D} = \frac{\sigma' \rightarrow^{*\Omega}_{p} \sigma'' \qquad \sigma'' \rightarrow^{\Omega}_{p} \sigma}{\sigma' \rightarrow^{*\Omega}_{p} \sigma}$$

The inductive hypothesis states: If $\widehat{\Sigma}'' \vdash^{S}_{p} \widehat{\sigma}''$ and $\sigma'' \models_S \widehat{\sigma}''$ and $\sigma' \rightarrow^{*\Omega}_{p} \sigma''$ then $\sigma' \models_S \widehat{\Sigma}''(\text{loc}(\sigma'))$.

From the premise of Lemma 3.3, we assume $\widehat{\Sigma} \vdash^{S}_{p} \widehat{\sigma}$ and $\sigma \models_S \widehat{\sigma}$ and $\sigma' \rightarrow^{*\Omega}_{p} \sigma$.

By inversion on c-transitive-step-inductive we have $\sigma' \rightarrow^{*\Omega}_{p} \sigma''$ and $\sigma'' \rightarrow^{\Omega}_{p} \sigma$.

By inversion on $\sigma'' \rightarrow^{\Omega}_{p} \sigma$ we have two cases for an application and boundary step.

Sub Case c-app-stepp: By inversion on c-app-step we have $\langle \sigma'', t \rangle \Downarrow^{\Omega} \sigma$ and $t \in p$.

By inversion from $\widehat{\Sigma} \vdash^{S}_{p} \widehat{\sigma}$ we have $\widehat{\sigma} \vdash_S \widehat{\Sigma}(\text{loc}(\widehat{\sigma}))$ and $\widehat{\Sigma} \vdash t$ for all $t \in p$.

Applying Assumption 1 to $\langle \sigma'', t \rangle \Downarrow \sigma$ and $\widehat{\Sigma} \vdash t$ and $t \in p$ we have $\sigma'' \models_S \widehat{\Sigma}(\text{pre}(t))$ and $\text{pre}(t) = \text{loc}(\sigma'')$.

By the inductive hypothesis, we have $\sigma' \models_S \widehat{\Sigma}(\text{loc}(\sigma'))$.

Sub Case c-boundary-step we have $\langle \sigma'', b \rangle \Downarrow^{\Omega} \sigma$ and $b \in p$.

By inversion from $\widehat{\Sigma} \vdash^{S}_{p} \widehat{\sigma}$ we have $\widehat{\sigma} \vdash_S \widehat{\Sigma}(\text{loc}(\widehat{\sigma}))$ and $\widehat{\Sigma} \vdash_S b$ for all $b \in p$.

Applying Lemma 2 to $\langle \sigma'', b \rangle \Downarrow^{\Omega} \sigma$ and $\widehat{\Sigma} \vdash b$ and $b \in p$ we have $\sigma'' \models_S \widehat{\Sigma}(\text{pre}(b))$.

From the abstract boundary step rules $\text{pre}(b) = \text{loc}(\sigma'')$.

By the inductive hypothesis, we have $\sigma' \models_S \widehat{\Sigma}(\text{loc}(\sigma'))$.

□

Theorem 3.4 (refute soundness) If $\vdash^{S}_{p} \widehat{\sigma}$ $\mathsf{unreach}$ and $\sigma' \rightarrow^{*\Omega}_{p} \sigma$ and $\sigma \models_S \widehat{\sigma}$ and $\Omega \subseteq \Omega_S$ then $\sigma' \neq \sigma_{\text{init}}$.

Proof of Theorem 4. Proof by contradiction: if we assume that $\vdash^{S}_{p} \widehat{\sigma}$ $\mathsf{unreach}$ and $\sigma_{\text{init}} \rightarrow^{*\Omega}_{p} \sigma$ and $\sigma \models_S \widehat{\sigma}$.

By inversion on $\vdash^{S}_{p} \widehat{\sigma}$ $\mathsf{unreach}$ we know $\widehat{\Sigma} \vdash^{S}_{p} \widehat{\sigma}$ and $\vdash_S \widehat{\Sigma}(\mathsf{Fwk})$ $\mathsf{excludesinit}$ then we can derive a contradiction.

From $\widehat{\Sigma} \vdash^{S}_{p} \widehat{\sigma}$ and $\sigma_{\text{init}} \rightarrow^{*\Omega}_{p} \sigma$ and Lemma 3 we have that $\sigma_{\text{init}} \models_S \widehat{\Sigma}(\mathsf{Fwk})$ (note that $\text{loc}(\sigma_{\text{init}}) = \mathsf{Fwk}$).

By Property 2 it must be the case that $\sigma_{\text{init}} \not\models \widehat{\Sigma}(\mathsf{Fwk})$.

From $\sigma_{\text{init}} \not\models_S \widehat{\Sigma}(\mathsf{Fwk})$ and $\sigma_{\text{init}} \models_S \widehat{\Sigma}(\mathsf{Fwk})$ we derive a contradiction. Therefore, any $\sigma \models_S \widehat{\sigma}$ cannot be reached from the initial state.

□

A.3 Proofs for Combining Abstract Message Histories with Callback Control-Flow

Proof of Theorem 5(1). In the forward direction (1) we proceed by induction on the derivation $\mathcal{D} = \ \vdash_S \widehat{\omega} \equiv \widetilde{\omega}$

$$\text{Case } \mathcal{D} = \frac{}{\vdash_S \mathsf{okhist} \equiv \mathrm{true}}\ \text{TEMPORAL-OKHIST}$$

If $\vdash_S \mathsf{okhist} \equiv \mathrm{true}$ and $\omega \cdot \theta \models_S \mathsf{okhist}$ then $\omega \cdot \theta \models \mathrm{true}$. This case holds trivially.

$$\text{Case } \mathcal{D} = \frac{S, \widehat{m}_1 \vdash \widetilde{\omega}'_1 \qquad \vdash_S \widehat{\omega}_2 \equiv \widetilde{\omega}_2 \qquad \vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \,⨾\, \widehat{m}_1}{\vdash_S \twoheadrightarrow \widehat{\omega}_2 \widehat{m}_1 \equiv \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2}\ \text{TEMPORAL-HYPMSG}$$

Let $\mathcal{D}_1$ be the premise $\vdash_S \widehat{\omega}_2 \equiv \widetilde{\omega}_2$.

Assume $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{m}_1 \widehat{\omega}_2$.

Pick a concrete message m_1 such that $m_1 \cdot \theta \models \widehat{m}_1$ as concrete message always exists for any $\widehat{m}$.

By the definition of concretization, $\models_S$, and since $\omega{\cdot}\theta \models_S \twoheadrightarrow \widehat{m}_1 \widehat{\omega}_2$ if $m_1{\cdot}\theta \models \widehat{m}_1$ then $\omega; m_1{\cdot}\theta \models_S \widehat{\omega}_2$.

By the inductive hypothesis on $\mathcal{D}_1$ with $\omega; m_1 \cdot \theta \models_S \widehat{\omega}_2$, we have $\omega; m_1 \cdot \theta \models \widetilde{\omega}_2$.

From $\omega; m_1 \cdot \theta \models_S \widehat{\omega}_2$, the base case of concretization, and the fact that realizable message histories are closed under prefix, we have that $\omega; m_1 \models S$.

By Lemma 6(1), and the premise $S, \widehat{m}_1 \vdash \widetilde{\omega}'_1$ of $\mathcal{D}$, and $\omega; m_1 \models S$ we have that $\omega \cdot \theta \models \widetilde{\omega}'_1$.

By Lemma 7(1), and the premise $\vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \,⨾\, \widehat{m}_1$ of $\mathcal{D}$, and $\omega; m_1 \cdot \theta \models \widetilde{\omega}_2$ we have that $\omega \cdot \theta \models \widetilde{\omega}'_2$.

Therefore $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$.

□

Proof of Theorem 5(2). In the backward direction (2) we use Proof by induction on the derivation $\mathcal{D} = \vdash_S \widehat{\omega} \equiv \widetilde{\omega}$

$$\text{Case } \mathcal{D} = \cfrac{\scriptstyle \text{TEMPORAL-OKHIST}}{\vdash_S \textsf{okhist} \equiv \text{true}}$$

If $\vdash_S$ $\textsf{okhist} \equiv$ true, $\omega \cdot \theta \models$ true and $\omega \models S$ then $\omega \cdot \theta \models_S$ $\textsf{okhist}$. This case holds trivially by the definition of $\models_S$.

$$\text{Case } \mathcal{D} = \cfrac{S, \widehat{m}_1 \vdash \widetilde{\omega}'_1 \qquad \vdash_S \widehat{\omega}_2 \equiv \widetilde{\omega}_2 \qquad \vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \fatsemi \widehat{m}_1}{\vdash_S \twoheadrightarrow \widehat{\omega}_2 \widehat{m}_1 \equiv \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2} \ \text{TEMPORAL-HYPMSG}$$

Let $\mathcal{D}_1$ be the premise $\vdash_S \widehat{\omega}_2 \equiv \widetilde{\omega}_2$.

Assume $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$.

Pick m_1 such that $m_1 \cdot \theta \models \widehat{m}_1$ as a message always exists for any $\widehat{m}$.

From $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$ we have $\omega \cdot \theta \models \widetilde{\omega}'_1$.

From Lemma 6 (2), $\omega \models S$, $m_1 \cdot \theta \models \widehat{m}_1$, the premise $S, \widehat{m}_1 \vdash \widetilde{\omega}'_1$, and $\omega \cdot \theta \models \widetilde{\omega}'_1$, we have that $\omega; m_1 \models S$.

From $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$ we have $\omega \cdot \theta \models \widetilde{\omega}'_2$.

From Lemma 7 (2), $m_1 \cdot \theta \models \widehat{m}_1$, and the last premise of $\mathcal{D}$, $\vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \fatsemi \widehat{m}_1$, we have that $\omega; m_1 \models \widetilde{\omega}_2$.

By the induction hypothesis on $\mathcal{D}_1$, $\omega; m_1 \models S$, and $\omega; m_1 \cdot \theta \models \widetilde{\omega}_2$, we have that $\omega; m_1 \cdot \theta \models_S \widehat{\omega}_2$.

From the right to left direction of the definition of concretization $\models_S$, message m_1 can be appended and model the abstract trace $\omega; m_1 \cdot \theta \models_S \widehat{\omega}_2$ and m_1 is in the abstract message $m_1 \cdot \theta \models \widehat{m}_1$, we have that $\omega \cdot \theta \models_S \twoheadrightarrow \widehat{\omega}_2 \widehat{m}_1$.

□

Note that the following two proofs are inductive due to the "and" case of the specification, S.

Proof of Lemma 6 *in the forward direction 1.* Proof by induction over the derivation $\mathcal{D} = S, \widehat{m} \vdash \widetilde{\omega}$.

$$\text{Case } \mathcal{D} = \cfrac{\widehat{m}_1 \simeq_{\vartheta} \widehat{m}_2}{(\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}), \widehat{m}_1 \vdash [\vartheta]\widetilde{\omega}} \ \text{INSTANTIATE-YES}$$

From the lemma we have a history with a message appended models the specification $\omega; m_1 \models (\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega})$, there is a message in the abstract message $m_1 \cdot \theta_1 \models \widehat{m}_1$.

From the premise $\widehat{m}_1 \simeq_{\vartheta} \widehat{m}_2$ we have the capture avoiding substitution ϑ such that $\widehat{m}_1$ is equal to $[\vartheta]\widehat{m}_2$.

From the semantics of CBCFTL in Figure 4.3, we have that $m_2 \cdot \theta_2 \models \widehat{m}_2$ such that $\omega[i] = m_2$ implies $\omega \cdot \theta_2 \cdot i - 1 \models \widetilde{\omega}$.

Since the free variables of $\widetilde{\omega}$ are a subset or equal to the free variables of $\widehat{m}_2$, the substitution, $[\vartheta]\widetilde{\omega}$, captures all free variables of $\widetilde{\omega}$. Since this substitution is capture avoiding, all the relevant bindings in θ_2 are swapped for the corresponding $\widehat{m}_1$ values and we can apply the implication from the CBCFTL semantics getting $\omega; m_1 \cdot \theta_1 \cdot \mathrm{len}(\omega) - 1 \models [\vartheta]\widetilde{\omega}$ which is equivalent to our theorem goal, $\omega \cdot \theta_1 \cdot \mathrm{len}(\omega)[\vartheta] \models \widetilde{\omega}$.

$$\text{Case } \mathcal{D} = \frac{\widehat{m}_1 \not\simeq \widehat{m}_2}{(\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}), \widehat{m}_1 \vdash \text{true}}\ \textsc{instantiate-no}$$

Trivially, $\omega \cdot \theta \cdot \mathrm{len}(\omega) \models \text{true}$.

$$\text{Case } \mathcal{D} = \frac{(S_1), \widehat{m}_1 \vdash \widetilde{\omega}_1 \qquad (S_2), \widehat{m}_1 \vdash \widetilde{\omega}_2}{(S_1 \wedge S_2), \widehat{m}_1 \vdash \widetilde{\omega}_1 \wedge \widetilde{\omega}_2}\ \textsc{instantiate-and}$$

Since the semantics of intended for $\wedge$ (elided from figure 4.3 for brevity) are that it is a conjunction of each instantiated specification, this case holds trivially.

$$\text{Case } \mathcal{D} = \frac{}{\text{true}, \widehat{m}_1 \vdash \text{true}}\ \textsc{instantiate-true}$$

Since any history, ω, is feasible under the "true" specification, this case holds trivially. □

***Proof of Lemma 6** in the backward direction 2.* Proof by induction over the derivation $\mathcal{D} = S, \widehat{m} \vdash \widetilde{\omega}$.

$$\text{Case } \mathcal{D} = \frac{\widehat{m}_1 \simeq_{\vartheta} \widehat{m}_2}{(\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega}), \widehat{m}_1 \vdash [\vartheta]\widetilde{\omega}}\ \textsc{instantiate-yes}$$

From the lemma, we have that $\omega \cdot \theta_1 \models (\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega})$, $m_1 \cdot \theta_1 \models \widehat{m}_1$, and $\omega \cdot \theta_1 \models [\vartheta]\widetilde{\omega}$.

From the premise $\widehat{m}_1 \simeq_{\vartheta} \widehat{m}_2$ we have the capture avoiding substitution ϑ such that $\widehat{m}_1$ is equal to $[\vartheta]\widehat{m}_2$.

From the semantics of CBCFTL in Figure 4.3, we have that $m_2 \cdot \theta_2 \models \widehat{m}_2$ such that $\omega[i] = m_2$ implies $\omega \cdot \theta_2 \cdot i - 1 \models \widetilde{\omega}$.

By similar reasoning as the proof for Lemma 6 (1) we know that any history and assignment such that $\omega \cdot \theta_1 \models [\vartheta]\widetilde{\omega}$ captures the variables shared between $[\vartheta]\widehat{m}_2$ and $[\vartheta]\widetilde{\omega}$ so any message such that $m \cdot \theta_1 \models [\vartheta]\widehat{m}_2$ may be appended to a message history, ω such that $\omega \models (\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega})$ and we know that $\omega; m \models (\widehat{m}_2 \mathrel{\Box\!\!\rightarrow} \widetilde{\omega})$.

The other cases are similar to the forward proof.

$\square$

For the following proofs, we give the definition of $\text{Match}(\widetilde{m}, \widehat{m})$ as the following:

(1) If the names and message types of $\widetilde{m}$ and $\widehat{m}$ are the same, then $\text{Match}(\widetilde{m}, \widehat{m})$ results in a conjunction of comparisons between each free variable of $\widetilde{m}$ and the corresponding $\widehat{m}$ variable.

(2) If the names or message types of $\widetilde{m}$ and $\widehat{m}$ are different, then $\text{Match}(\widetilde{m}, \widehat{m})$ is false.

We define $\text{NotMatch}(\widetilde{m}, \widehat{m})$ as the negation of $\text{Match}(\widetilde{m}, \widehat{m})$.

Proof of Lemma 7 *in the forward direction 1.* Proof by induction on the derivation $\mathcal{D} = \vdash \widetilde{\omega} \equiv \widetilde{\omega}' \mathbin{⨟} \widehat{m}$. We reference the CBCFTL semantics in Figure 4.3 and the judgments in Figure 4.4.

In the following, let $\text{fv}(m) = \text{fv}(\widetilde{m})$ be the conjunction comparing each variable not quantified in the expansion of $\widetilde{m}$ into its definition $\exists \overline{\hat{x}}.\widehat{m}'$ to the corresponding value of m.

$$\text{Case } \mathcal{D} = \frac{}{\vdash \mathsf{O}\,\widetilde{m} \equiv \text{Match}(\widetilde{m}, \widehat{m}) \vee (\text{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\,\widetilde{m}) \mathbin{⨟} \widehat{m}}\ \text{QUOTIENT-ONCE}$$

From the lemma we have that, $\omega; m \cdot \theta \models \mathsf{O}\,\widetilde{m}$ and $m \cdot \theta \models \widehat{m}$. There are two subcases to consider:

Subcase 1: the message names and types are equal, $m \simeq_\vartheta \widetilde{m}$.

From the semantics of Once, $\text{fv}(m)\widetilde{m}$ implies $\omega \cdot \theta \models \mathsf{O}\,\widetilde{m}$. By the definition of message equivalence (1), this is equivalent to $\text{Match}(\widetilde{m}, \widehat{m}) \vee (\text{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\,\widetilde{m})$

Subcase 2: the names or message types are not equal, $m \not\simeq \widetilde{m}$ are not equal.

From the semantics of Once, $\omega \cdot \theta \models \mathsf{O}\,\widetilde{m}$ must hold since m cannot match $\widetilde{m}$.

Under the assumption that the name or message type does not match

$\text{Match}(\widetilde{m}, \widehat{m}) \vee (\text{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\,\widetilde{m}) \equiv \mathsf{O}\,\widetilde{m}$ by the definition of message equivalence (2).

$$\text{Case } \mathcal{D} = \frac{}{\vdash \texttt{HN}\,\widetilde{m} \equiv \texttt{HN}\,\widetilde{m} \wedge \text{NotMatch}(\widetilde{m},\widehat{m}) \fatsemi \widehat{m}}\ \text{QUOTIENT-HISTORICALLY-NOT}$$

From the lemma we have that, $\omega; m \cdot \theta \models \texttt{HN}\,\widetilde{m}$ and $m \cdot \theta \models \widehat{m}$.

From the semantics of Has Not, $\omega \cdot \theta \models \texttt{HN}\,\widetilde{m}$. This holds for the formula $\texttt{HN}\,\widetilde{m} \wedge \text{NotMatch}(\widetilde{m},\widehat{m})$.

$$\text{Case } \mathcal{D} = \frac{}{\vdash \widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1 \equiv \text{Match}(\widetilde{m}_1,\widehat{m}) \vee (\text{NotMatch}(\widetilde{m}_1,\widehat{m}) \wedge \widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1 \wedge \text{NotMatch}(\widetilde{m}_2,\widehat{m})) \fatsemi \widehat{m}}\ \text{QUOTIENT-NOT-SINCE}$$

From the lemma we have that, $\omega; m \cdot \theta \models \widetilde{m}_1\ \texttt{NS}\ \widetilde{m}_2$ and $m \cdot \theta \models \widehat{m}$. There are three subcases to consider:

Subcase 1: the message names and types are equal between the positive message, $m \simeq_\vartheta \widetilde{m}_2$.

From the semantics of Not Since: not $\text{fv}(m) = \text{fv}(\widetilde{m}_2)$ implies $\omega \cdot \theta \models \widetilde{m}_1\ \texttt{NS}\ \widetilde{m}_2$. This is implied by the formula $\text{Match}(\widetilde{m}_1,\widehat{m}) \vee (\text{NotMatch}(\widetilde{m}_1,\widehat{m}) \wedge \widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1 \wedge \text{NotMatch}(\widetilde{m}_2,\widehat{m}))$ since the first disjunction is false due to the definition of message equivalence.

Subcase 2: the message names and types are equal between the negative message, $m \simeq_\vartheta \widetilde{m}_1$.

From the semantics of Not Since: not $\text{fv}(m) = \text{fv}(\widetilde{m}_1)$ implies $\omega \cdot \theta \models \widetilde{m}_1\ \texttt{NS}\ \widetilde{m}_2$. This is implied by the formula $\text{Match}(\widetilde{m}_1,\widehat{m}) \vee (\text{NotMatch}(\widetilde{m}_1,\widehat{m}) \wedge \widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1 \wedge \text{NotMatch}(\widetilde{m}_2,\widehat{m}))$ since the first disjunction is false due to the definition of message equivalence.

Subcase 3: the message names and types are not equal to either message, $m \not\simeq \widetilde{m}_1$ and $m \not\simeq \widetilde{m}_2$.

From the semantics of Not Since: $\omega \cdot \theta \models \widetilde{m}_1\ \texttt{NS}\ \widetilde{m}_2$. This is implied by the formula $\text{Match}(\widetilde{m}_1,\widehat{m}) \vee (\text{NotMatch}(\widetilde{m}_1,\widehat{m}) \wedge \widetilde{m}_2\ \texttt{NS}\ \widetilde{m}_1 \wedge \text{NotMatch}(\widetilde{m}_2,\widehat{m}))$ since $\text{Match}(\widetilde{m}_1,\widehat{m})$ and $\text{Match}(\widetilde{m}_2,\widehat{m})$ are both false by the definition of message equivalence.

$$\text{Case } \mathcal{D} = \frac{\vdash \widetilde{\omega}_1 \equiv \widetilde{\omega}'_1 \fatsemi \widehat{m} \qquad \vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \fatsemi \widehat{m}}{\vdash \widetilde{\omega}_1 \wedge \widetilde{\omega}_2 \equiv \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2 \fatsemi \widehat{m}}\ \text{QUOTIENT-AND}$$

From the lemma and the judgment, quotient-and, we have $m \cdot \theta \models \widehat{m}$, $\omega; m \cdot \theta \models \widetilde{\omega}_1 \wedge \widetilde{\omega}_2$, $\omega; m \cdot \theta \models \widetilde{\omega}_1$ implies $\omega \cdot \theta \models \widetilde{\omega}'_1$, and $\omega; m \cdot \theta \models \widetilde{\omega}_2$ implies $\omega \cdot \theta \models \widetilde{\omega}'_2$.

By the semantics of and (elided from Figure 4.3), $\omega; m \cdot \theta \models \widetilde{\omega}_1 \wedge \widetilde{\omega}_2$, and the two implications, we have that $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$.

The cases for $\vee$, $\forall \hat{x}.\widetilde{\omega}$, and $\exists \hat{x}.\widetilde{\omega}$ are similar to $\wedge$.

For all the following productions, the message history is not affected: $\hat{x}_1 = \hat{x}_2$, $\hat{x}_1 \neq \hat{x}_2$, and "true".

Therefore, they model any history $\omega \cdot \theta$.

For the case "false", it cannot model $\omega \cdot \theta$ therefore holds vacuously.

□

***Proof of Lemma 7** in the backward direction 2.* Proof by induction on the derivation $\mathcal{D} = \vdash \widetilde{\omega} \equiv \widetilde{\omega}' \fatsemi \widehat{m}$. We reference the CBCFTL semantics in Figure 4.3 and the judgments in Figure 4.4.

Let $\mathrm{fv}(m) = \mathrm{fv}(\widetilde{m})$ be the conjunction of comparing each variable not quantified in the expansion of $\widetilde{m}$ into its definition $\exists \widehat{\bar{x}}.\widehat{m}'$ to the corresponding value of m.

$$\text{Case } \mathcal{D} = \frac{}{\vdash \mathsf{O}\,\widetilde{m} \equiv \mathrm{Match}(\widetilde{m}, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\,\widetilde{m}) \fatsemi \widehat{m}} \text{ QUOTIENT-ONCE}$$

From the lemma we have that $\omega \cdot \theta \models \mathrm{Match}(\widetilde{m}, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}, \widehat{m}) \wedge \mathsf{O}\,\widetilde{m})$ and $m \cdot \theta \models \widehat{m}$. We consider two different subcases, in the case the message names are the same or not.

Assume the message names matches, $m \simeq_{\vartheta} \widetilde{m}$. It is either the case that the arguments match, $\mathrm{fv}(m) = \mathrm{fv}(\widetilde{m})$, and $\theta(\widetilde{m}) = m$ making i for Once equals $\mathrm{len}(\omega; m)$ or the arguments don't match, not $\mathrm{fv}(m) = \mathrm{fv}(\widetilde{m})$, and the i for once is in ω. In both cases $\omega; m \cdot \theta \models \mathsf{O}\,\widetilde{m}$.

Assume the message names do not match, $m \not\simeq \widetilde{m}$. In this case $\mathrm{Match}(\widetilde{m}, \widehat{m})$ is false so the i for once must be in ω, therefore $\omega; m \cdot \theta \models \mathsf{O}\,\widetilde{m}$.

$$\text{Case } \mathcal{D} = \frac{}{\vdash \mathtt{HN}\,\widetilde{m} \equiv \mathtt{HN}\,\widetilde{m} \wedge \mathrm{NotMatch}(\widetilde{m}, \widehat{m}) \fatsemi \widehat{m}} \text{ QUOTIENT-HISTORICALLY-NOT}$$

From the lemma we have that $\omega \cdot \theta \models \mathtt{HN}\,\widetilde{m} \wedge \mathrm{NotMatch}(\widetilde{m}, \widehat{m})$ and $m \cdot \theta \models \widehat{m}$.

From the definition of message not equals, we know either $m \not\simeq \widetilde{m}$ or not $\mathrm{fv}(m) = \mathrm{fv}(\widetilde{m})$ so $\theta(\widetilde{m}) \neq m$.

From the definition of Historically Not we have that $\omega \cdot \theta \models \mathtt{HN}\,\widetilde{m}$.

Since we have $\theta(\widetilde{m}) \neq m$ and $\omega \cdot \theta \models \mathtt{HN}\,\widetilde{m}$, by the definition of Historically Not, we have that $\omega; m \cdot \theta \models \mathtt{HN}\,\widetilde{m}$.

$$\text{Case } \mathcal{D} = \frac{}{\vdash \widetilde{m}_2\,\mathtt{NS}\,\widetilde{m}_1 \equiv \mathrm{Match}(\widetilde{m}_1, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}_1, \widehat{m}) \wedge \widetilde{m}_2\,\mathtt{NS}\,\widetilde{m}_1 \wedge \mathrm{NotMatch}(\widetilde{m}_2, \widehat{m})) \fatsemi \widehat{m}} \text{ QUOTIENT-NOT-SINCE}$$

From the lemma we have that $\omega \cdot \theta \models \mathrm{Match}(\widetilde{m}_1, \widehat{m}) \vee (\mathrm{NotMatch}(\widetilde{m}_1, \widehat{m}) \wedge \widetilde{m}_2\,\mathtt{NS}\,\widetilde{m}_1 \wedge \mathrm{NotMatch}(\widetilde{m}_2, \widehat{m}))$

and $m \cdot \theta \models \widehat{m}$. We have two subcases:

Subcase $m \simeq_\vartheta \widetilde{m}_2$ and $\mathrm{fv}(m) = \mathrm{fv}(\widetilde{m}_2)$: In this case the i from the definition of Not Since is at the last message, m, and $\omega; m \cdot \theta \models \widetilde{m}_1 \ \mathtt{NS} \ \widetilde{m}_2$.

Subcase $m \neq \widetilde{m}_2$ or $\mathrm{fv}(m) \neq \mathrm{fv}(\widetilde{m}_2)$: In this case, we have $\mathrm{NotMatch}(\widetilde{m}_1, \widehat{m})$ so the i in Not Since must be in ω.

by the definition of message not equals, we know that $\widetilde{m}_1 \not\simeq \widehat{m}$ or $\mathrm{fv}(\widehat{m}) \neq \mathrm{fv}(\widetilde{m}_1)$ so m must not be equal to $\widetilde{m}$ under assignment θ and therefore $\omega; m \cdot \theta \models \widetilde{m}_1 \ \mathtt{NS} \ \widetilde{m}_2$.

$$\text{Case } \mathcal{D} = \dfrac{\vdash \widetilde{\omega}_1 \equiv \widetilde{\omega}'_1 \fatsemi \widehat{m} \qquad \vdash \widetilde{\omega}_2 \equiv \widetilde{\omega}'_2 \fatsemi \widehat{m}}{\vdash \widetilde{\omega}_1 \wedge \widetilde{\omega}_2 \equiv \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2 \fatsemi \widehat{m}} \ \text{QUOTIENT-AND}$$

From the lemma and the quotient-and judgment, we have that $m \cdot \theta \models \widehat{m}$, $\omega \cdot \theta \models \widetilde{\omega}'_1 \wedge \widetilde{\omega}'_2$, $\omega \cdot \theta \models \widetilde{\omega}'_1$ implies $\omega; m \cdot \theta \models \widetilde{\omega}_1$, and $\omega \cdot \theta \models \widetilde{\omega}'_2$ implies $\omega; m \cdot \theta \models \widetilde{\omega}_2$.

By the straightforward semantics of and (elided from Figure 4.3), $\omega \cdot \theta \models \widetilde{\omega}_1 \wedge \widetilde{\omega}_2$, and the two implications, we have that $\omega; m \cdot \theta \models \widetilde{\omega}_1 \wedge \widetilde{\omega}_2$.

The cases for $\vee$, $\forall \hat{x}.\widetilde{\omega}$, and $\exists \hat{x}.\widetilde{\omega}$ are similar to $\wedge$.

For all the following productions, the message history is not affected: $\hat{x}_1 = \hat{x}_2$, $\hat{x}_1 \neq \hat{x}_2$, and "true". Therefore, they model any history $\omega; \cdot \theta$.

For the case "false", it cannot model $\omega; m \cdot \theta$ therefore holds vacuously.

□

A.4 Extended Explanations of the Benchmark Applications and Specifications

Here, we give a listing of all the specifications we wrote for section 4.4 RQ1, details on the crashes, and reasons the Flowdroid model would not be able to classify the benchmarks correctly.

The first benchmark, `getAct`[8] , is a slightly more complex version of the motivating example in the Historia paper [89]. This benchmark uses a background task with the RXJava library to load a media object. A callback `call` is invoked when the media is loaded. This defect occurs if the call happens after the application has been destroyed with `onDestroy`. Calling `getActivity` before `onCreate` or after `onDestroy` results in a null value as captured by History Implication 8. Since the `call` callback invokes `getActivity`, this results in a `NullPointerException`. History Implication 6 is used to capture the fix where `unsubscribe` prevents the crashing `call` invocation.

History Implication 6. *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)`*.*

$$\mathsf{cb}\ l.\mathtt{call}(m) \;\square\!\!\rightarrow \exists s.\ \mathsf{ci}\ s.\mathtt{unsubscribe}()\ \mathsf{NS}\ \mathsf{ci}\ s =_{.\mathtt{subscribe}(}\ l)$$

History Implication 7. *For all* `f`*, if the framework invokes* `f.onCreate()`*, the same message* cb `f.onCreate()` *is **H**istorically **N**ot possible (or, **H**as **N**ever been invoked in the past).*

$$\mathsf{cb}\ f.\mathtt{onCreate}() \;\square\!\!\rightarrow \mathsf{HN}\ \mathsf{cb}\ f.\mathtt{onCreate}()$$

History Implication 8. *The method* `getActivity` *returns null when invoked on an* `Activity` *in the paused state.*

$$\mathsf{ci}\ \mathrm{null} = f.\mathtt{getActivity}() \;\square\!\!\rightarrow \mathsf{cb}\ f.\mathtt{onActivityCreated}()\ \mathsf{NS}\ \mathsf{cbret}\ f.\mathtt{onDestroy}() \lor \mathsf{HN}\ \mathsf{cb}\ f.\mathtt{onActivityCreated}()$$

History Implication 9. *For a given* `Fragment` *instance, either* `onActivityCreated` *has not been invoked or* `onDestroy` *has not been invoked since* `onActivityCreated`*.*

$$\begin{aligned}\mathsf{cb}\ f.\mathtt{onActivityCreated}() \;\square\!\!\rightarrow & \mathsf{HN}\ \mathsf{cbret}\ f.\mathtt{onDestroy}() \land \\ & \mathsf{HN}\ \mathsf{cb}\ f.\mathtt{onActivityCreated}() \land \mathsf{HN}\ \mathsf{cb}\ f.\mathtt{onActivityCreated}()\end{aligned}$$

Flowdroid would not alarm on the buggy version of `getAct`[8] because `call` was not in the call graph making the assertion unreachable. If `call` was added to the framework model generation (e.g. by Edgeminer [25]), then the Flowdroid model would not be able to capture the effect of `unsubscribe` (i.e. History Implication 6) resulting in a false positive for the fix.

The second benchmark, `execute`[10] , has a button that starts an `AsyncTask` to perform an action in the background using the `execute` callin. `AsyncTask` is an abstract class that can be overridden by the app to encapsulate long running tasks (similar to `Single`). In order to prevent concurrency issues in the state of the overridden class, the framework enforces that `execute` crashes if called twice on the same instance of the overridden class. We capture this exceptional return with History Implication 10. If the button was clicked twice quickly, the task could be executed twice crashing the application. The fix was to disable the button using `b.setEnabled(false)` (History Implication 11). Additionally, if the listener could be registered to two different buttons via two calls to `onCreate` which calls `setOnClickListener`, then the second button could be pressed crashing the app. To rule out this case, we needed History Implication 7 (also needed by `getAct`[8]).

History Implication 10. *For any given instance of* `AsyncTask`*, the* `execute` *method throws an exception if* `execute` *has been invoked in the past.*

$$\mathsf{exn\ ci}\ t.\mathit{execute}() \boxminus\!\!\rightarrow \Diamond\ \mathsf{ciret}\ t.\mathit{execute}()$$

History Implication 11. *Every time the* `onClick` *callback occur, the associated button has not been disabled.*

$$\mathsf{cb}\ l.\mathit{onClick}() \boxminus\!\!\rightarrow \exists\, v.\ (\Diamond\ \mathsf{ci}\ v.\mathit{setOnClickListener}(l) \land$$
$$(\mathtt{HN}\ \mathsf{ci}\ v.\mathit{setEnabled}(\mathsf{false}) \lor \mathsf{ci}\ v.\mathit{setEnabled}(\mathsf{false})\ \mathtt{NS}\ \mathsf{ci}\ v.\mathit{setEnabled}(\mathsf{true})))$$

The fix for `execute`[10] would be misclassified by Flowdroid's model because the effect of the callin `b.setEnable(false)` on the button is not captured.

The third benchmark, `dismiss`[12] , uses a background task (via `AsyncTask`) and when the task finishes, the `onPostExecute` callback dismisses a `ProgressDialog`. If the `ProgressDialog` is dismissed after the parent has been paused, it throws an exception (History implication 12). The fix was to check if the parent UI was visible before dismissing the dialog using the application state. Additionally, we needed to restrict the method used to create the dialog, `show`, to return a fresh dialog instance each time (History Implication 13).

History Implication 12. *The* `dismiss` *callin throws an exception if invoked while the attached* `Activity` *is paused.*

$$\mathsf{exn\ ci}\ d.\mathtt{dismiss}() \mathrel{\square\!\!\rightarrow} \exists a.\ \mathsf{O}\ \mathsf{ci}\ d \mathrel{:=} \_.\mathtt{show}(a) \wedge \mathsf{HN}\ \mathsf{cb}\ a.\mathtt{onResume}() \vee \mathsf{cb}\ a.\mathtt{onResume}()\ \mathsf{NS}\ \mathsf{cbret}\ a.\mathtt{onPause}()$$

History Implication 13. *When the* `show` *callin returns an object, it must have been the case that the object was not returned by a different* `show` *in the past.*

$$\mathsf{ci}\ d = b.\mathtt{show}() \mathrel{\square\!\!\rightarrow} \mathsf{HN}\ \mathsf{ci}\ d = \_.\mathtt{show}()$$

The bug for `dismiss`[12] would not be detected by Flowdroid because the `onPostExecute` callback was not in the call graph. Adding `onPostExecute` to the Flowdroid would result in a false alarm on the fix because the interaction between `show`, `dismiss`, `onPause`, and `onResume` captured by History Implication 12 could not be captured due to the callins.

The fourth benchmark, `finish`[null] , has a button that dereferences a field in a `onClick` callback. This field is set to **`null`** when the `Activity` is paused (via `onPause`) to save memory. However, via what may be considered a bug in some versions of the Android framework itself, calling `finish` on an `Activity` can result in the `onClick` callback occurring after `onPause` (History Implication 14). Similar to other benchmarks, there is a `onCreate` that registers the button and can only happen once (History Implication 7). Since an `Activity` may have multiple simultaneous instances in an Android app, we also need a spec that says a button may only come from one `Activity` instance (History Implication 15).

History Implication 14. *Every time the* `onClick` *callback happens on the listener object* `l`*, the listener was registered on a* `Button` *object* `v`*, and the* `Activity` *object* `a` *was either in the "resumed state" (i.e., after a* `onResume` *but before than a* `onPause`*) or the message* `finish` *happened.*

$$\text{cb } l.\mathit{onClick}() \boxminus\!\!\rightarrow \exists\, a, v.\ (\text{ci } v.\mathit{setOnClickList}(\text{null})\ \text{NS ci } v.\mathit{setOnClickList}(l) \wedge$$
$$\Diamond\ \text{ci } v = a.\mathit{findViewById}(\_) \wedge$$
$$(\text{cbret } a.\mathit{onPause}()\ \text{NS cb } a.\mathit{onResume}() \vee \Diamond\ \text{ci } a.\mathit{finish}()))$$

History Implication 15. *Every time a* `findViewById` *on an* `Activity` *object* a *returns a* `View` *object* v*, any previous invocation of* `findViewById` *that returned* v *was invoked on the same* `Activity` a*.*

$$\text{ci } v = a.\mathit{findViewById}(id) \boxminus\!\!\rightarrow \forall\, a2.\ (\text{HN ci } v = a2.\mathit{findViewById}(\_) \vee a = a2)$$

The bug for `finish`$^{\text{null}}$ would be missed by Flowdroid's model because this model assumes a `onClick` may only occur after an `onCreate` and before an `onPause`. If `onClick` were allowed to occur after `onPause`, then the Flowdroid model would still need to reason about the callin `v.setOnClickListener(null)` which prevents `onClick`.

The fifth benchmark `subs`$^{\text{null}}$ is caused by a dereference of a nullable field callback that is used by RXJava for synchronizing the results of a background task with the UI thread. Similar to the first benchmark, this bug would not be detected by a whole-program analysis using the Flowdroid main method because the synchronization callback is missing from the call graph.

History Implication 16. *The* `dispose` *callin prevents calls to the subscribe callback.*

$$\text{cb } l.\mathit{subscribe}() \boxminus\!\!\rightarrow \exists\, m, s.\ \Diamond\ \text{ci } m = \mathit{create}(l) \wedge$$
$$\text{ci } s.\mathit{dispose}()\ \text{NS ci } s = m.\mathit{subscribe}()$$

History Implication 17. *The* `subscribeOn` *callin always returns its recieiver (Java builder pattern).*

$$\text{ci } t = v.\mathit{subscribeOn}() \boxminus\!\!\rightarrow t = v$$

History Implication 18. *The* `observeOn` *callin always returns its recieiver (Java builder pattern).*

$$\mathsf{ci}\ t = v.\mathtt{observeOn}() \mathrel{\square\!\!\rightarrow} t = v$$

History Implication 19. *The* `onStart` *callback may only occur first or after an* `onStop`*.*

$$\mathsf{cb}\ f.\mathtt{onStart}() \mathrel{\square\!\!\rightarrow} (\mathtt{HN}\ \mathsf{cb}\ f.\mathtt{onStart}() \land \mathtt{HN}\ \mathsf{cb}\ f.\mathtt{onStop}()) \lor$$

$$\mathsf{cb}\ f.\mathtt{onStart}()\ \mathtt{NS}\ \mathsf{cb}\ f.\mathtt{onStop}()$$

History Implication 20. *A value may only be returned from the* `create` *callin once.*

$$\mathsf{ci}\ m = \mathtt{create}() \mathrel{\square\!\!\rightarrow} \mathtt{HN}\ \mathsf{ci}\ m = \mathtt{create}()$$

Appendix B

Generation of ChatGPT Benchmarks for Discovery and Explanation of Even-Driven Defects Evaluation

In section 5.4 the evaluation was performed on a set of the most common runtime problems as reported by a set of blog posts. To avoid researcher bias in how the benchmarks are constructed, we used ChatGPT to generate examples of the runtime problems from the blog posts. This consisted of prompting the model for a non-trivial example of the issue in question, and inspecting the result. If the output is a reasonable program exhibiting the defect, we proceed with our process of adding the necessary build and resource files to produce a full application. If the output does not exhibit the issue, we proceed to reprompt the model, asking for a less trivial example. In this section, we collect the prompts used to produce our benchmarks. Only the prompt producing the final, usable output is shown for each benchmark, for brevity.

Bitmap Mishandling:

User: Give me a non-trivial example of an android application that does not handle bitmaps correctly.

Output:

```
public class MainActivity extends AppCompatActivity {

    private Integer[] imageIDs = {
        R.drawable.image1, R.drawable.image2, R.drawable.image3,
        R.drawable.image4, R.drawable.image5, // Assume these are large
            bitmap resources
    };
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    GridView gridView = findViewById(R.id.gridView);
    gridView.setAdapter(new ImageAdapter(this));
}

public class ImageAdapter extends BaseAdapter {
    private Context context;

    public ImageAdapter(Context c) {
        context = c;
    }

    public int getCount() {
        return imageIDs.length;
    }

    public Object getItem(int position) {
        return position;
    }

    public long getItemId(int position) {
        return position;
```

```
        }

        // Create a new ImageView for each item referenced by the
           Adapter
        public View getView(int position, View convertView, ViewGroup
           parent) {
            ImageView imageView;
            if (convertView == null) {
                imageView = new ImageView(context);
                imageView.setLayoutParams(new ViewGroup.LayoutParams
                   (350, 350));
                imageView.setScaleType(ImageView.ScaleType.CENTER_CROP)
                   ;
            } else {
                imageView = (ImageView) convertView;
            }

            imageView.setImageResource(imageIDs[position]); // Loading
               the full-sized image
            return imageView;
        }
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridView"
```

```
android:layout_width="match_parent"
android:layout_height="match_parent"
android:numColumns="auto_fit"
android:columnWidth="100dp"
android:stretchMode="columnWidth"
android:gravity="center"/>
```

Explanation: Loading bitmaps is memory-intensive, and this application renders several of these at the same time.

Device Configuration

User: `Give me a non-trivial example of an android application that does not handle device configuration changes correctly.`

Output:

```
public class MainActivity extends AppCompatActivity {

private EditText usernameInput;
private EditText emailInput;
// Assume more fields here

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    usernameInput = findViewById(R.id.usernameInput);
    emailInput = findViewById(R.id.emailInput);
    // Initialize other fields here
```

```
        // No handling of savedInstanceState to restore state
    }

    // Missing onSaveInstanceState method to save the activity's state
}
```

```
  <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android
   "
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

    <EditText
        android:id="@+id/usernameInput"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Username" />

    <EditText
        android:id="@+id/emailInput"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Email" />
```

```
    <!-- Additional input fields -->

</LinearLayout>
```

Explanation: When an `Activity` is created, it may have some saved state to load, which can be used for restoring text fields, for instance. This application does not check if this state is null, and so ignores any previously saved data.

Device Compatibility

User: Give me a non-trivial example of an android application where the developer overlooked device compatibility. Output:

```
public class CameraActivity extends Activity implements SurfaceHolder
    .Callback {

    private Camera camera;
    private SurfaceView surfaceView;
    private SurfaceHolder surfaceHolder;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_camera);

        surfaceView = findViewById(R.id.cameraSurfaceView);
        surfaceHolder = surfaceView.getHolder();
        surfaceHolder.addCallback(this);
    }
```

```
private boolean safeCameraOpen() {
    boolean qOpened = false;
    try {
        releaseCamera();
        camera = Camera.open();
        qOpened = (camera != null);
    } catch (Exception e) {
        Toast.makeText(this, "Failed␣to␣open␣camera", Toast.
           LENGTH_LONG).show();
    }
    return qOpened;
  }

@Override
public void surfaceCreated(SurfaceHolder holder) {
    if (camera != null) {
        try {
            camera.setPreviewDisplay(holder);
            camera.startPreview();
        } catch (Exception e) {
            Toast.makeText(this, "Failed␣to␣start␣camera␣preview",
               Toast.LENGTH_LONG).show();
        }
    }
  }

@Override
```

```
public void surfaceChanged(SurfaceHolder holder, int format, int
     width, int height) {
       // Handle changes
     }

    @Override
    public void surfaceDestroyed(SurfaceHolder holder) {
        // Release camera
      }

    private void releaseCamera() {
        if (camera != null) {
            camera.stopPreview();
            camera.release();
            camera = null;
        }
      }

    @Override
    protected void onPause() {
        super.onPause();
        releaseCamera();
    }
  }
```

Explanation: This application uses the deprecated `Camera` API, which may lead to bugs in devices which do not support this API.

Dialog Origin

User: Give me an example of an android application that throws the ”Activity has leaked window that was originally added here” exception. Output:

```
public class MainActivity extends AppCompatActivity {

  private ProgressDialog progressDialog;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      // Showing a ProgressDialog
      progressDialog = new ProgressDialog(MainActivity.this);
      progressDialog.setMessage("Loading...");
      progressDialog.show();

      // Simulate some operation that finishes after the activity has
          been destroyed
      // In a real app, this could be a network request or a long
         computation
      new Thread(new Runnable() {
          @Override
          public void run() {
              try {
                  Thread.sleep(10000); // 10 seconds
```

```
                    progressDialog.dismiss(); // Attempt to dismiss
                        after activity is possibly destroyed
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }
}
```

Explanation: This application spawns a thread which creates a dialog window, sleeps, and then dismisses the dialog. If the `Activity` which contained the dialog is destroyed before the dialog is dismissed, then an exception is thrown, as the dialog is leaked.

Execute Twice

User: Give me an example of an android application that throws the error ”java.lang.IllegalStateException: Cannot execute task: the task is already running.” Output:

```
public class MainActivity extends AppCompatActivity {

    private Button startDownloadButton;
    private DownloadTask downloadTask;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        downloadTask = new DownloadTask();
```

```
        startDownloadButton = findViewById(R.id.startDownloadButton);

        startDownloadButton.setOnClickListener(new View.OnClickListener
            () {
            @Override
            public void onClick(View v) {
                // Check if downloadTask is already running
                if (downloadTask.getStatus() != AsyncTask.Status.
                    RUNNING) {
                    // If not running, execute a new task
                    downloadTask.execute("http://example.com/file.zip")
                        ;
                } else {
                    Toast.makeText(MainActivity.this, "Download already
                        in progress", Toast.LENGTH_SHORT).show();
                }

                // Attempt to reuse the same AsyncTask instance for a
                    new download
                // This line is problematic if uncommented; it's here
                    for demonstration.
                // downloadTask.execute("http://example.com/anotherfile
                    .zip");
            }
        });
    }
```

```
private static class DownloadTask extends AsyncTask<String, Integer
    , String> {
    @Override
    protected String doInBackground(String... urls) {
        // Simulate a download operation
        try {
            Thread.sleep(5000); // Simulate time delay of download
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return "Downloaded␣from:␣" + urls[0];
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        // Show download result (simplified for demonstration)
        System.out.println(result);
    }
  }
  }
```

Explanation: In this application, on each button press, an `ASyncTask` is executed, unless it is currently running. This can lead to a crash, as once the `ASyncTask` is finished executing, the check will fail, and the task will receive a second `execute` invocation, causing a crash.

Fragment Lifecycle

User: Give me a non-trivial example of an android application that does not handle the fragment lifecycle properly.

Output:

```
    public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        if (savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction()
                    .replace(R.id.fragment_container, new
                       ArticlesFragment())
                    .commit();
        }
    }
}

public class ArticlesFragment extends Fragment {

    private View rootView;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup
       container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
```

```
    if (rootView == null) {
        rootView = inflater.inflate(R.layout.fragment_articles,
            container, false);
    }
    return rootView;
}

@Override
public void onAttach(Context context) {
    super.onAttach(context);
    // Load articles assuming the context is always available
    loadArticles();
}

private void loadArticles() {
    // Simulate loading articles from the network or database
}

@Override
public void onDestroyView() {
    super.onDestroyView();
    // Not properly nullifying the rootView can lead to memory
       leaks
    // rootView = null; // Uncommenting this line is necessary to
       avoid leaks
}
```

```
    @Override
    public void onDestroy(){
        super.onDestroy();
    }
}
```

Explanation: When `onDestroyView` is invoked, the fragment should dispose of its reference to the view in question, but does not.

Inefficient Network

User: Give me a non-trivial android application that demonstrates inefficient network calls.

Output:

```
    public class MainActivity extends AppCompatActivity {

    private ListView listView;
    private ArrayList<String> titles = new ArrayList<>();
    private ArrayAdapter<String> adapter;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        listView = findViewById(R.id.listView);
        adapter = new ArrayAdapter<>(this, android.R.layout.
           simple_list_item_1, titles);
        listView.setAdapter(adapter);
```

```
    fetchArticles();
}

@Override
protected void onResume() {
    super.onResume();
    fetchArticles(); // Inefficiently fetches articles every time
       the activity resumes
}

private void fetchArticles() {
    String url = "https://example.com/api/articles";

    RequestQueue queue = Volley.newRequestQueue(this);
    StringRequest stringRequest = new StringRequest(Request.Method.
       GET, url,
        new Response.Listener<String>() {
            @Override
            public void onResponse(String response) {
                try {
                    JSONArray jsonArray = new JSONArray(response);
                    titles.clear();
                    for (int i = 0; i < jsonArray.length(); i++) {
                        JSONObject jsonObject = jsonArray.
                           getJSONObject(i);
                        titles.add(jsonObject.getString("title"));
                    }
```

```
                    adapter.notifyDataSetChanged();
                } catch (JSONException e) {
                    e.printStackTrace();
                }
            }
        }, new Response.ErrorListener() {
            @Override
            public void onErrorResponse(VolleyError error) {
                // Handle error
            }
        });

        queue.add(stringRequest);
    }
}
```

Explanation: Every time the `onResume` callback is invoked, such as when the `Activity` is brought to the foreground, the app performs another network call, which can lead to behavior where it is performed frequently.

Long Running Task

User: Give me a non-trivial example of an Android application that blocks the UI thread with a long running task.

The output given is too trivial, but recommends using RxJava, a more modern library for these tasks, so further prompting is taken in this direction.

User: What if the data processing was done with RXJava, could there still be a long running task on the UI thread?

The output given describes the issue but shows a correct program, so we further prompt:

User: Give me the example code where RXJava blocks the UI thread.

Output:

```
public class MainActivity extends AppCompatActivity {


private TextView textView;


@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);


    textView = findViewById(R.id.textView);


    // Incorrectly starting data processing on the main thread with
        RxJava
    processDataRxIncorrectly();
}


private void processDataRxIncorrectly() {
    Observable.fromCallable(() -> processLargeDataSet(
        loadLargeDataSet()))
        .observeOn(AndroidSchedulers.mainThread()) // Observing on
            the main thread
        .subscribeOn(AndroidSchedulers.mainThread()) // Incorrect:
            Subscribing on the main thread
        .subscribe(result -> textView.setText("Data␣processed␣
```

```
            successfully!"),
                    error -> textView.setText("Error␣processing␣data
                       "));
}

// Mock method to simulate data processing
private String processLargeDataSet(List<String> largeDataSet) {
    // Simulate processing the dataset
    for (String item : largeDataSet) {
        // Simulate complex processing by sleeping
        try {
            Thread.sleep(10); // This sleep is just to simulate
               long processing
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    return "Processed";
}

// Mock method to simulate loading data
private List<String> loadLargeDataSet() {
    List<String> dataSet = new ArrayList<>();
    for (int i = 0; i < 10000; i++) {
        dataSet.add("Item␣" + i);
    }
    return dataSet;
```

```
    }
}
```

Explanation: When `onCreate` is invoked on the `Activity`, the application starts a long task on the main thread, which is also in charge of rendering the UI. This leads to the application appearing to take a long time to start, as the task must complete first.

Memory Leak

User: `Give me an example of an Activity leak in android without static fields.`

Output:

```
public class LeakActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_leak);

    // Starting a long-running thread with an inner class holding a
        reference to the Activity
    new Thread(new Runnable() {
        @Override
        public void run() {
            // Simulate long-running task
            try {
                Thread.sleep(100000); // Sleep for 100 seconds
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
            }
        }).start();
    }
}
```

Explanation: When the `Runnable` is constructed, it holds a reference to the `Activity` that is spawning it, and prevents the destruction of it until after the `Runnable` is done executing, even if the user navigates away.

Null Pointer Exception

User: Give me source code in java for an android application that exhibits a non-trivial null pointer exception.

Output:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    // Method to launch SecondActivity
    public void launchSecondActivity(View view) {
        Intent intent = new Intent(this, SecondActivity.class);
        intent.putExtra("extra_data", "Hello,␣SecondActivity!");
        startActivity(intent);
    }
}
```

```java
  public class SecondActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);

        // Attempting to retrieve the extra data and display it
        String data = getIntent().getStringExtra("wrong_key"); // This
           key is incorrect
        TextView textView = findViewById(R.id.text_view);
        textView.setText(data.toUpperCase()); // This line can throw a
           NPE if data is null
    }
}
```

Explanation: When the `SecondActivity` is created, it looks up the incorrect key in its `Intent`, which leads to a NullPointerException when it is used without verifying it as non-null.

View Hierarchy

User: Give me a non-trivial example of an android application that uses a view hierarchy that is too deep.

Output:

```xml
    <?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android
   "
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
```

```
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Personal␣Information" />
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal">
            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="First␣Name:" />
            <EditText
                android:layout_width="match_parent"
                android:layout_height="wrap_content" />
        </LinearLayout>
        <!-- Additional nested LinearLayouts for Last Name, Email, etc.
            -->
    </LinearLayout>
    <!-- Similar nested structures for Contact Details, Preferences,
       etc. -->
</LinearLayout>
```

Explanation: When rendering a layout, each nesting level requires the UI framework to perform more measurements, which can lead to performance issues.