DroidStar: Callback Typestates for Android Classes

Arjun Radhakrishna*
Microsoft

Nicholas V. Lewchenko
University of Colorado Boulder

Shawn Meier
University of Colorado Boulder

Sergio Mover
University of Colorado Boulder

Krishna Chaitanya Sripada
University of Colorado Boulder

Damien Zufferey
Max Planck Institute for Software Systems

Bor-Yuh Evan Chang
University of Colorado Boulder

Pavol Černý
University of Colorado Boulder

ABSTRACT

Event-driven programming frameworks, such as Android, are based on components with asynchronous interfaces. The protocols for interacting with these components can often be described by finite-state machines we dub *callback typestates*. Callback typestates are akin to classical typestates, with the difference that their outputs (callbacks) are produced asynchronously. While useful, these specifications are not commonly available, because writing them is difficult and error-prone.

Our goal is to make the task of producing callback typestates significantly easier. We present a callback typestate assistant tool, DroidStar, that requires only limited user interaction to produce a callback typestate. Our approach is based on an active learning algorithm, L*. We improved the scalability of equivalence queries (a key component of L*), thus making active learning tractable on the Android system.

We use DroidStar to learn callback typestates for Android classes both for cases where one is already provided by the documentation, and for cases where the documentation is unclear. The results show that DroidStar learns callback typestates accurately and efficiently. Moreover, in several cases, the synthesized callback typestates uncovered surprising and undocumented behaviors.

KEYWORDS

typestate, specification inference, Android, active learning

ACM Reference Format:
Arjun Radhakrishna, Nicholas V. Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. 2018. DroidStar: Callback Typestates for Android Classes. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180232

*This work was done while Arjun Radhakrishna was employed at the University of Pennsylvania.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180232

1 INTRODUCTION

Event-driven programming frameworks interact with client code using callins and callbacks. Callins are framework methods that the client invokes and callbacks are client methods that the framework invokes. The client-framework interaction is often governed by a protocol that can be described by a finite-state machine we call *callback typestate*. Callback typestates are akin to classical typestates [36], with the key difference that their outputs (callbacks) are produced asynchronously. Our goal is to make the task of producing callback typestates significantly easier for developers.

As an example of a callback typestate, consider a typical interaction between a client application and the framework when the client wants to use a particular service. The client asks for the service to be started by invoking an `startService()` callin. After the framework receives the callin, it asynchronously starts initializing the service. When the service is started and ready to be used, the framework notifies the client by invoking a `onServiceStarted()` callback. The client can then use the service. After the client finishes using the service, it invokes a `shutdownService()` callin to ask the framework to stop the service.

Callback typestates. Callback typestates are useful in a number of ways, but they are notoriously hard to produce. First, callback typestates are a form of documentation. They tell client application programmers in what order to invoke callins and which callback to expect. Android framework documentation for some classes already uses pictures very similar to callback typestates (Figure 1). Second, callback typestates are useful in verification of client code. They enable checking that a client uses the framework correctly. Third, even though we infer the callback typestates from framework code, they can be used for certain forms of framework verification. For instance, one can infer typestates for different versions of the framework, and check if the interface has changed.

Callback typestates are very hard to produce manually. On one hand, inspecting code to see in what situation a callback arrives, and what callins are enabled after that is error-prone. Even developers familiar with the framework often miss corner-case behaviors. On the other hand, obtaining the callback typestate with manual testing is hard. One would need to run all sequences of callins, mixed in sequence with the callbacks they produce. We systematize this testing approach using an active learning algorithm.

Callback typestate assistant DroidStar**.** We present a tool that makes producing callback typestates significantly easier. Our target user is a developer who wrote an Android class that interacts

asynchronously using callbacks with client code. DroidStar is a comprehensive framework for semi-automatically inferring callback typestates. The required user interaction happens in multiple steps. In the first step, the user provides code snippets to perform local tasks, such as code for class initialization and code for invoking each callin (similarly as in unit tests). This is sufficient as long as certain widely applicable assumptions hold. First, we assume that each sequence of callins produces a sequence of callbacks deterministically (this assumption fails when for instance a callback has a parameter that is ignored at first by DroidStar but that influences the typestate). Second, we assume that the resulting typestate is finite. If these assumptions fail, in the following steps, DroidStar asks the user for a solution to the problem. For instance, one way to remove non-determinism is to refine one callback into two separate logical callbacks, based on the parameter values. This design allows DroidStar to offer the user control over the final result while requiring only limited, local, insight from the user. DroidStar is available for download at https://github.com/cuplv/droidstar

Approach. We present a method for inferring typestates for Android classes. However, our method is equally applicable in other contexts. The core algorithm is based on Angluin's L* algorithm [5] adapted to Mealy machines [32]. In this algorithm, a learner tries to learn a finite-state machine — in our case a callback typestate — by asking a teacher membership and equivalence queries. Intuitively, a membership query asks for outputs corresponding to a sequence of input callins, and the equivalence query asks if the learned typestate is correct. We note that the teacher does not need to know the solution, but only needs to know how to answer the queries.

The key question we answer is how to implement oracles for the membership and equivalence queries. We show how to implement membership queries on Android classes using black-box testing. Our main contribution here is an efficient algorithm for implementing the equivalence query using membership query. The insight here is that the number of membership queries can be bounded by a function of a new bound we call the *distinguisher bound*. We empirically confirmed that for Android classes, the *distinguisher bound* is significantly smaller than the state bound used in previous work [12, 16]. Given that the number of required membership queries depends exponentially on the distinguisher bound, the novel bound is what enables our tool to scale to Android classes.

Results. We use DroidStar to synthesize callback typestates for 16 Android framework classes and classes from Android libraries. The results show that DroidStar learns callback typestates accurately and efficiently. This is confirmed by documentation, code inspection, and manual comparison to simple Android applications. The running time of DroidStar on these benchmarks ranged between 43 seconds and 72 minutes, with only 3 benchmarks taking more than 10 minutes. The usefulness of the distinguisher bound was also confirmed. Concretely, using previously known bounds, learning the callback typestate for one of our examples (MediaPlayer) would take more than a year, whereas with the distinguisher bound, this example takes around 72 minutes. Furthermore, by inspecting our typestates, we uncovered corner cases with surprising behavior that are undocumented and might even be considered as bugs in some cases. For instance, for the commonly used AsyncTask class, if `execute()` is called after `cancel()` but before the `onCancelled()` callback is received, it will not throw an exception but will never cause the asynchronous task to be run. Section 6 presents our results in more detail.

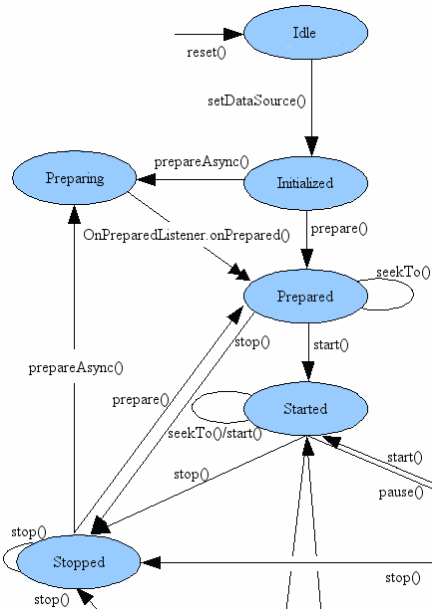

Figure 1: Part of MediaPlayer's callback typestate from https://developer.android.com/reference/android/media/MediaPlayer.html

Contributions. The contributions of this paper are: (a) We introduce the notion of callback typestates and develop an approach, based on the L* algorithm, to infer them. (b) We show how to implement efficiently membership and equivalence oracles required by the L* algorithm. (c) We evaluate our approach on examples from the Android framework, and show its accuracy and effectiveness.

2 WORKFLOW AND ILLUSTRATIVE EXAMPLE

We use the Android Framework's `MediaPlayer` class to explain the standard workflow for inferring callback typestate using DroidStar. This class is highly stateful—its interface includes many methods that are only meaningful or enabled in one or two particular player states—and makes extensive use of callins and callbacks to handle the delays of loading and manipulating large media files. These properties make callback typestate a perfect fit; in fact, `MediaPlayer` has one of the very few examples where we found a complete callback typestate specification in the Android libraries documentation. This callback typestate is shown in Figure 1.

In Figure 1, callins are represented by single arrows and callbacks by double arrows. Let us look at one part of the protocol that governs the client-framework interaction. The client first invokes the callin `setDataSource()`, and the protocol transitions to the `Initialized` state. In this state, the client can invoke the callin `prepareAsync()`, and the protocol transitions to the `Preparing` state. In the `Preparing` state, the client cannot invoke any callins, but the framework can invoke the `onPrepared()` callback, and then the protocol transitions to the `Prepared` state. At this point, the client can invoke the `start()` callin, and the media starts playing.

Our goal is to semi-automatically infer the callback typestate from the figure using the tool DroidStar. The developer interacts with DroidStar in several steps, which we describe now.

2.1 Developer-Provided Snippets

To apply DroidStar to the `MediaPlayer` class, the developer provides a number of code snippets detailed below that act as an interface through which the tool can examine `MediaPlayer` instances.

Test object and environment instantiation. The main callback typestate inference algorithm of DroidStar works roughly by repeatedly performing tests in the form of sequences of method calls on an object of the given class, i.e., the `MediaPlayer`. Each test must begin with an identical, isolated, class object, and if necessary, a standard environment. In the first step, the developer provides a snippet to initialize such an object and environment. In the case of `MediaPlayer`, this snippet is as simple as discarding the previous instance, creating a new one with `new MediaPlayer()`, and registering the necessary callback listeners (explained in the *Callback instrumentation* paragraph below). In some cases this snippet is more complex. As an example, we cannot create new instances of the `BluetoothAdapter` class, so for that class this snippet would need to bring the existing instance back to a uniform initial state.

Callin declaration. The next step is to declare the alphabet of "input symbols" that represent the callins in the interface of our class—the final callback typestate will be written using these symbols—and map each symbol to the concrete code snippet it represents. In most cases, there is a one-to-one correspondence between input symbols and callin methods. For example, the code snippets associated with the input symbols **prepare**, **prepareAsync**, and **start** are `prepare();`, `prepareAsync();`, and `start();`, respectively.

In some cases, such as when a callin takes a parameter, the developer may instead map a symbol to a *set* of code snippets representing alternative forms of the input which are suspected to have different behavior. In the `MediaPlayer` class, the `setDataSource()` callin method takes a URL argument. The developer might (rightly) believe that depending on the validity and reachability of the given URL, the behavior of the callin in the typestate may differ. In this case, the developer may provide the two snippets `setDataSource(goodURL);` and `setDataSource(badURL);` for the same callin. DroidStar will consider both snippets for generating tests, and further, it will indicate if they behave differently with respect to the typestate. In case a difference is detected, the "non-determinism" is handled as explained later in this section.

The complete set of input symbols which would be declared and mapped for the `MediaPlayer` class are setDataSource, prepare, prepareAsync, start, stop, reset, release, and pause.

Callback instrumentation. As for the callin methods, which act as the input symbols in the callback typestate, the callback methods act as the output symbols in the callback typestate. The developer specifies the set of output callback symbols and associated snippets to detect when callbacks occur. In most cases, this involved adding the listeners for the callbacks in the initialization snippet as mentioned above. In the `MediaPlayer` class, the output symbols are **onCompleted** and **onPrepared**.

2.2 Automated Callback-Typestate Inference

Once the developer provides the input and output symbols and the associated snippets, DroidStar attempts to automatically learn the callback typestate following the framework of the L* algorithm.

L* inference. In L*, the learner tests sequences of inputs until she can form a consistent hypothesis automaton. Each such test (or sequence of inputs) is called a membership query. Once a hypothesis automaton is produced, an equivalence query is performed; i.e., the hypothesis automaton is checked for equivalence with the true callback typestate. If the two are equivalent, we are done; otherwise, a counter-example test is returned from which the tool learns. This process repeats until the produced hypothesis automaton is correct.

For `MediaPlayer`, the first set of membership queries each consist of a single different callin. Of these, only the query containing `setDataSource()` succeeds. The learner continues with longer membership queries while building the hypothesis automaton. For instance, it learns that `prepareAsync()` and `prepare()` do not lead to the same state: it is possible to invoke the `start()` after `prepare()`, but not after `prepareAsync()`. Once the client receives the callback `onPrepared()`, `start()` may be called. The learner thus hypothesizes a transition from the `Preparing` to the `Prepared` on `onPrepared()`. Once the hypothesis is complete, the learner asks the equivalence query. Initially, a counter-example to equivalence is returned using which the learner refines its hypothesis. The final solution is found after 5 equivalence queries.

Answering Equivalence Queries. The equivalence query, i.e., checking if a learned callback typestate is in fact the true callback typestate is undecidable in general. However, assuming a bound on the size of the typestate, the equivalence query can be implemented using further testing. However, equivalence queries are still expensive and to make them practical we present an new optimization based on a *distinguisher bound*. We can observe in Figure 1 that for any pair of states there is a transition in one state which leads to an error in the other. This corresponds to a distinguisher bound of 1. Small distinguisher bounds arise because typestates are not random automata but part of an API designed for ease of use and robustness. Such APIs are coded defensively and are fail-fast [35], i.e., errors are not buffered but reported immediately. Each state in the typestate has a specific function and an associated set of callins and callbacks. In automata terms, the alphabet is roughly the same size as the number of states and each state has only a few transitions, making any two states easy to distinguish. In Section 4.3, we explain how to use the distinguisher bound to implement equivalence queries and discuss why distinguisher bounds are small in practice.

2.3 Obstacles to Inference and Solutions

The L* based callback typestate inference algorithm makes several assumptions about the behavior of the class that do not always hold. DroidStar is designed to detect these violations of assumptions and notify the developer. Here, we discuss two such assumptions, the exceptional situations that arise when the assumptions are violated, and the additional developer intervention needed to handle such cases.

Non-determinism. In input-output automata learning theory, non-determinism makes learning impossible. Non-determinism is

the possibility of the same sequence of input callins producing different sequences of output callbacks across tests. Non-determinism may be due to various *controllable* and *non-controllable* factors. Controllable factors include cases where behavior depends on if a file exists, if a URL is reachable, etc. On the other hand, non-controllable factors include random number generators, device sensors, etc. In practice, most of the non-determinism was controllable.

The main technique for handling non-determinism is via *refinement of input or output alphabets*. Here, a single callin or callback is split into multiple "logical" inputs or outputs.

(a) Controllable non-determinism can be eliminated by incorporating the controlling factor into the inputs. For example, in the `SQLiteOpenHelper` class, the behavior of the `constructor` callin changes depending on if a file exists. However, after splitting the callin into two separate callins `constructor/fileExists` and `constructor/noFileExists`, the behavior of each of each callin becomes deterministic with respect to these callins.

(b) Another source of non-determinism is when the same callback is used to notify logically different events. For example, a class may use a generic `onComplete` callback which is passed a status parameter that can have the values "Success" and "Failure". Based on this value, different further callins are enabled, leading to non-determinism. Here, the developer may manually refine the callback into two output symbols `onEvent/Success` and `onEvent/Failure`, and the behavior is deterministic with respect to these.

In summary, for controllable non-determinism, the onus is on the developer to identify the source of the detected non-determinism and provide a refinement of the input or output alphabet and corresponding code snippets to control the source. No general technique exists to handle non-controllable non-determinism, but specific cases can be handled using techniques shown in Section 5.

Non-regularity. Another basic assumption that L* based inference algorithm makes is that the callback typestate under consideration is regular. This assumption is commonly violated in request-response style behavior of classes where the number of responses (output callbacks) invoked is exactly equal to the number of requests (input callins). Our solution to this problem is to restrict the learning to a subset of the class behavior, such as inputs with at most one pending request callin using a learning purpose [1]. These restrictions makes the behavior regular and amenable to learning.

3 THE CALLBACK TYPESTATE LEARNING PROBLEM

We introduce formal models of interfaces, define the callback typestate learning problem, and present an impossibility result about learning typestates. Callback typestates have both inputs (corresponding to callins) and outputs (corresponding to callbacks). In automata theory, callback typestates can be seen as interface automata. Interface automata [14] are a well-studied model of automata that can produce outputs asynchronously w.r.t. inputs. We use the name callback typestates to emphasize that they are a generalization of typestates as used in the programming languages literature.

3.1 Definitions and Problem Statement

Asynchronous interfaces. Let Σ_i and Σ_o be the set of callins and callbacks of an asynchronous interface. We abstract away parameter and return values of callins and callbacks, and model a behavior of the interface as a *trace* $\tau_i = \sigma_0 \dots \sigma_n \in (\Sigma_i \cup \Sigma_o)^*$. The *interface* I is given by $\langle \Sigma_i, \Sigma_o, \Pi_i \rangle$ where $\Pi_i \subseteq \{\Sigma_i \cup \Sigma_o\}^*$ is the prefix-closed set of all feasible traces of the interface.

Interface automata. We use interface automata [14] to represent asynchronous interfaces. An *interface automaton* A is given by $\langle Q, q_\iota, \Sigma_i, \Sigma_o, \Delta_A \rangle$ where: (a) Q is a finite set of *states*, (b) $q_\iota \in Q$ is the *initial state*, (c) Σ_i and Σ_o are finite sets of *input and output symbols*, and (d) $\Delta_A \subseteq Q \times \{\Sigma_i \cup \Sigma_o\} \times Q$ are a set of *transitions*. A *trace* τ_a of A is given by $\sigma_0 \dots \sigma_n$ if $\exists q_0 \dots q_{n+1} : q_0 = q_\iota \wedge \forall i.(q_i, \sigma_i, q_{i+1}) \in \Delta_A$. Traces(A) is the set of all traces of A.

Problem statement. Given an interface $I = \langle \Sigma_i, \Sigma_o, \Pi_i \rangle$, the *callback typestate learning problem* is to learn an interface automaton A such that Π_i = Traces(A). We allow the learner to ask a *membership oracle* MOracle[I] membership queries. For a *membership query*, the learner picks mQuery = $i_0 i_1 \dots i_n \in \Sigma_i^*$ and the membership oracle MOracle[I] returns either: (a) a trace $\tau_a \in \Pi_i$ whose sequence of callins is exactly mQuery, or (b) $\perp$ if no such trace exists.

3.2 The Theory and Practice of Learning Typestates

In general, it is impossible to learn callback typestates using only membership queries; no finite set of membership queries fixes a unique interface automaton. However, callback typestates can be effectively learned given extra assumptions. We now analyze the causes behind the impossibility and highlight the assumptions necessary to overcome it.

Unbounded asynchrony. Membership queries alone do not tell us if the interface will emit more outputs (callbacks) at any point in time. Hence, we assume:

Assumption 1: Quiescence is observable.

This assumption is commonly used in ioco-testing frameworks [37]. In our setting, we add an input wait and an output quiet, where quiet is returned after a wait only if there are no other pending callbacks. In practice, quiet can be implemented using timeouts, i.e., pending callbacks are assumed to arrive within a fixed amount of time. If no callbacks are seen within the timeout, quiet is output.

Example 3.1. Using wait and quiet, in the MediaPlayer example, we have that `setDataSource() ·prepareAsync() ·onPrepared() ·wait ·quiet` is a valid trace, but `setDataSource() ·prepareAsync() ·wait ·quiet` is not.

Behavior unboundedness. For any set of membership queries, let k be the length of the longest query. It is not possible to find out if the interface exhibits different behavior for queries much longer than k. This is a theoretical limitation, but is not a problem in practice [7]; most callback typestates are rather small ($\leq$ 10 states).

Assumption 2: An upper bound on the size of the typestate being learned is known.

Non-determinism. We need to be able to observe the systems' behaviors to learn them and non-determinism can prevent that. Therefore, we assume:

Assumption 3: The interface is deterministic.

We assume that for every trace τ_a of the interface, there is at most one output $o \in \Sigma_o$ such that $\tau_a \cdot o \in \Pi_i$. In practice, the non-determinism problem is somewhat alleviated due to the nature of

callback typestates (see Section 5). See [1] for a detailed theoretical discussion of how non-determinism affects learnability.

Example 3.2. Consider an interface with traces given by $(\texttt{input} \cdot (\texttt{out1} \mid \texttt{out2}))^*$. All membership queries are a sequence of `input`'s; however, it is possible that the membership oracle never returns any trace containing $\overline{\texttt{out2}}$. In that case, no learner will be able to learn the interface exactly.

4 LEARNING CALLBACK TYPESTATES USING L*

Given **Assumption 1** and **Assumption 3**, we first build a "synchronous closure" of an asynchronous interface (Section 4.1). Then, we show how to learn the synchronous closure effectively given **Assumption 2** (Section 4.2 and 4.3).

4.1 From Asynchronous to Synchronous Interfaces

Using **Assumption 1** and **3**, we build a synchronous version of an interface in which inputs and outputs strictly alternate following [1]. For synchronous interfaces, we can draw learning techniques from existing work [1, 5, 25, 32].

Define $\tilde{\Sigma}_i = \Sigma_i \cup \{\text{wait}\}$ and $\tilde{\Sigma}_o = \Sigma_o \cup \{\text{quiet}, \lambda, \text{err}\}$. The purpose of the extra inputs and outputs is discussed below. For any $\tau_s \in (\tilde{\Sigma}_i \cdot \tilde{\Sigma}_o)^*$, we define $\text{async}(\tau_s) = \tau_a \in (\Sigma_i \cup \Sigma_o)^*$ where τ_a is had from τ_s by erasing all occurrences of wait, quiet, λ, and err.

Synchronous closures. The *synchronous closure* I_s of an asynchronous interface $\mathsf{I} = \langle \Sigma_i, \Sigma_o, \Pi_i \rangle$ is given by $\langle \tilde{\Sigma}_i, \tilde{\Sigma}_o, \Pi_s \rangle$ where $\tilde{\Sigma}_i$ and $\tilde{\Sigma}_o$ are as above, and $\Pi_s \subseteq (\tilde{\Sigma}_i \cdot \tilde{\Sigma}_o)^*$ is defined as the smallest set satisfying the following:

$$
\begin{aligned}
& \epsilon \in \Pi_s \\
\tau_s \in \Pi_s \wedge \text{async}(\tau_s) \cdot i \in \Pi_i \quad &\implies \quad \tau_s \cdot i \cdot \lambda \in \Pi_s \\
\tau_s \in \Pi_s \wedge \text{async}(\tau_s) \cdot o \in \Pi_i \quad &\implies \quad \tau_s \cdot \text{wait} \cdot o \in \Pi_s \\
\tau_s \in \Pi_s \wedge \text{async}(\tau_s) \cdot i \notin \Pi_i \quad &\implies \quad \tau_s \cdot i \cdot \text{err} \in \Pi_s \\
\tau_s \in \Pi_s \wedge o \in \Sigma_o \wedge \text{async}(\tau_s) \cdot o \notin \Pi_i \quad &\implies \quad \tau_s \cdot \text{wait} \cdot \text{quiet} \in \Pi_s \\
\tau_s \in \Pi_s \wedge \tau_s \text{ ends in err} \quad &\implies \quad \tau_s \cdot i \cdot \text{err} \in \Pi_s
\end{aligned}
$$

Informally, in I_s: (a) Each input is immediately followed by a dummy output λ; (b) Each output is immediately preceded by a wait input wait; (c) Any call to an input disabled in I is immediately followed by an err. Further, all outputs after an err are err's. (d) Any call to wait in a quiescent state is followed by quiet.

Given MOracle[I] and **Assumption 1**, it is easy to construct the membership MOracle[I_s]. Note that due to **Assumption 3**, there is exactly one possible reply MOracle[I_s](mQuery) for each query mQuery. Further, by the construction of the synchronous closure, the inputs and outputs in MOracle[I_s](mQuery) alternate.

Mealy machines. We model synchronous interfaces using the simpler formalism of Mealy machines rather than interface automata. A *Mealy machine* M is a tuple $\langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \text{Out} \rangle$ where: (a) Q, q_ι, $\tilde{\Sigma}_i$, and $\tilde{\Sigma}_o$ are states, initial state, inputs and outputs, respectively, (b) $\delta : Q \times \tilde{\Sigma}_i \to Q$ is a *transition function*, and (c) $\text{Out} : Q \times \tilde{\Sigma}_i \to \tilde{\Sigma}_o$ is an *output function*. We abuse notation and write $\text{Out}(q, i_0 \ldots i_n) = o_1 \ldots o_n$ and $\delta(q, i_0 \ldots i_n) = q'$ if $\exists q_0, \ldots, q_{n+1} : q_0 = q \wedge q_{n+1} = q' \wedge \forall 0 \le i \le n : \delta(q_i, i_i) = q_{i+1} \wedge \text{Out}(q_i, i_i) = o_i$. A sequence $i_0 o_0 \ldots i_n o_n \in (\tilde{\Sigma}_i \cdot \tilde{\Sigma}_o)^*$ is a *trace* of M if $\text{Out}(q_\iota, i_0 \ldots i_n) = o_0 \ldots o_n$. We often abuse notation and write $\text{M}(i_0 \ldots i_n)$ instead of $\text{Out}(q_\iota, i_0 \ldots i_n)$. We denote by Traces(M) the set of all traces of M.

4.2 L*: Learning Mealy Machines

For the sake of completeness, we describe the classical L* learning algorithm by Angluin [5] as adapted to Mealy machines in [32]. A reader familiar with the literature on inference of finite-state machines may safely skip this subsection.

Fix an asynchronous interface I and its synchronous closure I_s. In L*, in addition to a membership oracle MOracle[I_s], the learner has access to an *equivalence oracle* EOracle[I_s]. For an equivalence query, the learner passes a Mealy machine M to EOracle[I_s], and is in turn returned: (a) A *counterexample input* $\text{cex} = i_0 \ldots i_n$ such that $\text{M}(\text{cex}) = o_0 \ldots o_n$ and $\text{MOracle}[\mathsf{I}_s](\text{cex}) \neq i_0 o_0 \ldots i_n o_n$, or (b) Correct if no such cex exists.

The full L* algorithm is in Algorithm 1. In Algorithm 1, the learner maintains: (a) a set $S_Q \subseteq \tilde{\Sigma}_i^*$ of *state-representatives* (initially set to $\{\epsilon\}$), (b) a set $E \subseteq \tilde{\Sigma}_i^*$ of *experiments* (initially set to $\tilde{\Sigma}_i$), and (c) an *observation table* $T : (S_Q \cup S_Q \cdot \tilde{\Sigma}_i) \to (E \to \tilde{\Sigma}_o^*)$. The observation table maps each prefix w_i and suffix e to $T(w_i)(e)$, where $T(w_i)(e)$ is the suffix of the output sequence of $\text{MOracle}(w_i \cdot e)$ of length $|e|$. The entries are computed by the sub-procedure FillTable.

Intuitively, S_Q represent Myhill-Nerode equivalence classes of the Mealy machine the learner is constructing, and E distinguish between the different classes. For S_Q to form valid set of Myhill-Nerode classes, each state representative extended with an input, should be equivalent to some state representative. Hence, the algorithm checks if each $w_i \cdot i \in S_Q \cdot \tilde{\Sigma}_i$ is equivalent to some $w_i' \in S_Q$ (line 3) under E, and if not, adds $w_i \cdot i$ to S_Q. If no such $w_i \cdot i$ exists, the learner constructs a Mealy machine M using the Myhill-Nerode equivalence classes, and queries the equivalence oracle (line 5). If the equivalence oracle returns a counterexample, the learner adds a suffix of the counterexample to E; otherwise, it returns M. For the full description of the choice of suffix, see [30, 32].

Theorem 4.1 ([32]). *Let there exist a Mealy machine* M *with* n *states such that* Traces(M) *is the set of traces of* I_s. *Then, given* MOracle[I_s] *and* EOracle[I_s], *Algorithm 1 returns* M *making at most* $|\tilde{\Sigma}_i|^2 n + |\tilde{\Sigma}_i| n^2 m$ *membership and* n *equivalence queries, where* m *is the maximum length of counterexamples returned by* EOracle[I_s]. *If* EOracle[I_s] *returns minimal counterexamples,* $m \le O(n)$.

4.3 An Equivalence Oracle Using Membership Queries

Given a black-box interface in practice, it is not feasible to directly implement the equivalence oracle required for the L* algorithm. Here, we demonstrate a method of implementing an equivalence oracle using the membership oracle using the boundedness assumption (**Assumption 2**). As before fix an asynchronous interface I and its synchronous closure I_s. Further, fix a target minimal Mealy machine M* such that Traces(M*) is the set of traces of I_s.

State bounds. A *state bound* of B_{State} implies that the target Mealy machine M* has at most B_{State} states. Given a state bound, we can replace an equivalence check with a number of membership queries using the following theorem.

Algorithm 1 L* for Mealy machines

Input: Membership oracle MOracle, Equivalence oracle EOracle
Output: Mealy machine M

1: $S_Q \leftarrow \{\epsilon\}; E \leftarrow \tilde{\Sigma}_i; T \leftarrow$ FillTable$(S_Q, \tilde{\Sigma}_i, E, T)$
2: **while** True **do**
3: **while** $\exists w_i \in S_Q, i \in \tilde{\Sigma}_i : \nexists w'_i \in S_Q : T(w_i \cdot i) = T(w'_i)$ **do**
4: $S_Q \leftarrow S_Q \cup \{w_i \cdot i\}$; FillTable$(S_Q, \tilde{\Sigma}_i, E, T)$
5: M $\leftarrow$ BuildMM$(S_Q, \tilde{\Sigma}_i, T)$; cex $\leftarrow$ EOracle(M)
6: **if** cex = Correct **then return** M
7: $E \leftarrow E \cup$ AnalyzeCex(cex, M); FillTable$(S_Q, \tilde{\Sigma}_i, E, T)$
8: **function** BuildMM(S_Q,$\tilde{\Sigma}_i$,$\tilde{\Sigma}_o$,T)
9: $Q \leftarrow \{[w_i] \mid w_i \in S_Q\}$; $q_\iota \leftarrow [\epsilon]$
10: $\forall w_i, i : \delta([w_i], i) \leftarrow [w'_i]$ if $T(w_i \cdot i) = T(w'_i)$
11: $\forall w_i, i :$ Out$([w_i], i) \leftarrow o$ if $T(w_i)(i) = o$
12: **return** $\langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta,$ Out$\rangle$
13: **function** AnalyzeCex(M,cex)
14: **for all** $0 \le i \le |\text{cex}|$ and w_i^p, w_i^s such that $w_i^p \cdot w_i^s =$ cex $\wedge\ |w_i^p| = 1$ **do**
15: $w_o^p \leftarrow$ M(w_i^p); $[w_i^{p'}] \leftarrow \delta([\epsilon], w_o^p)$
16: $w_o^s \leftarrow$ last $|w_i^s|$ of Out(MOracle$(w_i^{p'} \cdot w_i^s)$)
17: **if** $w_o^p \cdot w_o^s \neq$ Out(MOracle(cex)) **then return** w_i^s
18: **procedure** FillTable(S_Q,$\tilde{\Sigma}_i$,E,T)
19: **for all** $w_i \in S_Q \cup S_Q \cdot \tilde{\Sigma}_i$, $e \in E$ **do**
20: $T(w_i)(e) \leftarrow$ Suffix of Out(MOracle$(w_i \cdot e)$) of length $|e|$

Theorem 4.2. *Let* M *and* M′ *be Mealy machines having k and k' states, respectively, such that $\exists w_i \in \tilde{\Sigma}_i^* : \mathsf{M}(w_i) \neq \mathsf{M}'(w'_i)$. Then, there exists an input word w'_i of length at most $k + k' - 1$ such that $\mathsf{M}(w'_i) \neq \mathsf{M}'(w'_i)$.*

The proof is similar to the proof of the bound $k + k' - 2$ for finite automata (see [33, Theorem 3.10.5]). We can check equivalence of M* and any given M by testing that they have equal outputs on all inputs of length at most $k_{\mathsf{M}} + B_{\text{State}} - 1$, i.e., using $O(|\tilde{\Sigma}_i|^{B_{\text{State}}+k-1})$ membership queries. While this simple algorithm is easy to implement, it is inefficient and the number of required membership queries make it infeasible to implement in practice. Other algorithms based on state bounds have a similar problems with efficiency (see Remark in Section 4.3). Further, the algorithm does not take advantage of the structure of M. The following discussion and algorithm rectifies these short-comings.

Distinguisher bounds. A *distinguisher bound* of $B_{\text{Dist}} \in \mathbb{N}$ implies that for each pair of states q_1^*, q_2^* in the target Mealy machine M* can be distinguished by an input word w_i of length at most B_{Dist}, i.e., Out*$(q_1^*, w_i) \neq$ Out*(q_2^*, w_i). Intuitively, a small distinguisher bound implies that each state is "locally" different, i.e., can be distinguished from others using small length input sequences. The following theorem shows that a state bound implies a comparable distinguisher bound.

Theorem 4.3. *State bound k implies distinguisher bound $k - 1$.*

Small distinguisher bound. In practice, distinguishers are much smaller than the bound implied by the state bound. For the media-player, the number of states is 10, but only distinguishers of length 1 are required. This pattern tends to hold in general due to the following principles of good interface design:

- *Clear separation of the interface functions.* Each state in the interface has a specific function and a specific set of callins and callbacks. There is little reuse of names across state. The typestate's alphabet is roughly the same size as the number of states.
- *Fail-fast.* Incorrect usage of the interface is not silently ignored but reported as soon as possible. This makes it easier to distinguish states as disabled callins lead directly to errors.
- *No buffering.* More than just fail-fast, a good interface is interactive and the effect of callins must be immediately visible rather than hidden. A good interface is not a combination lock that requires many inputs that are silently stored and only acknowledged at the very end.

This observation also is not specific to callbacks typestates and it has been already observed for libraries [11].

Equivalence algorithm. Algorithm 2 is an equivalence oracle for Mealy machines using the membership oracle, given a distinguisher bound. First, it computes state representatives $R : Q \to \tilde{\Sigma}_i^*$: for each $q \in Q$, $\delta(q_\iota, R(q)) = q$ (line 1). Then, for each transition in M, the algorithm first checks whether the output symbol is correct (line 5). Then, the algorithm checks the "fidelity" of the transition up to the distinguisher bound, i.e., whether the representative of the previous state followed by the transition input, and the representative of the next state can be distinguished using a suffix of length at most B_{Dist}. If so, the algorithm returns a counterexample. If no transition shows a different result, the algorithm returns Correct.

Two optimizations further reduce the number of membership queries: (a) *Quiescence transitions.* Transitions with input wait and output quiet need not be checked at line 7; it is a no-op at the interface level. (b) *Error transitions.* Similarly, transition with the output err need not be checked as any extension of an error trace can only have error outputs.

Remark. *Note that if Algorithm 2 is being called from Algorithm 1, the state representatives from L* can be used instead of recomputing R in line 1. Similarly, the counterexample analysis stage can be skipped in the L* algorithm, and the relevant suffix can be directly returned (*suffix *in lines 10 and 11; and i in line 5).*

Theorem 4.4. *Assuming the distinguisher bound of B_{Dist} for the target Mealy machine* M*, *either (a) Algorithm 2 returns* Correct *and $\forall w_i \in \tilde{\Sigma}_i^* : \mathsf{M}(w_i) = \mathsf{M}^*(w_i)$, or (b) Algorithm 2 returns a counterexample* cex *and* M(cex) $\neq$ M*(cex). *Further, it performs at most $|Q| \cdot |\tilde{\Sigma}_i|^{B_{\text{Dist}}+1}$ membership queries.*

Remark (Relation to conformance testing algorithms). *Note that the problem being addressed here, i.e., testing the equivalence of a given finite-state machine and a system whose behavior can be observed, is equivalent to the conformance testing problem from the model-based testing literature. However, several points make the existing conformance testing algorithms unsuitable in our setting.*

Popular conformance testing algorithms, like the W-method [12] and the W_p-method [16], are based on state bounds and have an unavoidable $O(|\tilde{\Sigma}_i|^{B_{\text{State}}})$ factor in the complexity. In our experiments, the largest typestate had 10 states and 7 inputs. The $O(|\tilde{\Sigma}_i|^{B_{\text{State}}})$ factor leads to an infeasible (i.e., $> 10^8$) number of membership queries.

Algorithm 2 Equivalence oracle with distinguisher bound

Input: Mealy machine $\mathsf{M} = \langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \text{Out}\rangle$, Distinguisher bound B_{Dist}, and Membership oracle MOracle
Output: Correct if $\mathsf{M} = \mathsf{M}^*$, or cex $\in \tilde{\Sigma}_i^*$ s. t. $\mathsf{M}(\text{cex}) \neq \mathsf{M}^*(\text{cex})$

for all $q \in Q$ **do** $R(q) \leftarrow w_i \mid \delta(q_\iota, w_i) = q$ s. t. $|w_i|$ is minimal
for all $q \in Q, i \in \tilde{\Sigma}_i$ **do**
  $w_i \leftarrow R(q) \cdot i$
  if $\text{Out}(q, i) \neq$ last symbol of $\text{Out}(\text{MOracle}(w_i \cdot i))$ **then**
    return $R(q) \cdot i$
  $q' \leftarrow \delta(q, i)$; $w_i' \leftarrow R(q')$
  suffix $\leftarrow$ check(w_i, w_i')
  if suffix $\neq$ Correct **then**
    if $\mathsf{M}(R(q) \cdot i \cdot \text{suffix}) \neq \text{Out}(\text{MOracle}(R(q) \cdot i \cdot \text{suffix}))$ **then**
      return $R(q) \cdot i \cdot$ suffix
    else return $R(q') \cdot$ suffix
return Correct
function check(w_i, w_i')
  for all suffix $\in \tilde{\Sigma}_i^{\leq B_{\text{Dist}}}$ **do**
    $w_o \leftarrow \text{Out}(\text{MOracle}(w_i \cdot \text{suffix}))$
    $w_o' \leftarrow \text{Out}(\text{MOracle}(w_i' \cdot \text{suffix}))$
    if the last |suffix| symbols of w_o and w_o' differ **then**
      return suffix
  return Correct

However, since distinguisher bounds are often much smaller than state bounds, $O(|\tilde{\Sigma}_i|^{B_{\text{Dist}}})$ membership queries are feasible (i.e., 10^3). The W- and W_p-methods cannot directly use distinguisher bounds.

The other common algorithm, the D-method [20, 22], does not apply in our setting either. The D-method is based on building a distinguishing sequence, i.e., an input sequence which produces a different sequence of outputs from every single state in the machine. However, for callback typestates, such single distinguishing sequences do not exist in practice. For similar reasons, conformance testing algorithms such as the UIO-method [31] do not apply either.

In this light, we believe that Algorithm 2 is a novel conformance testing algorithm useful in specific settings where resets are inexpensive and systems are designed to have small distinguisher bounds.

4.4 Putting It All Together

We now present the full callback typestate learning solution.

Theorem 4.5. *Given a deterministic interface* I *with observable quiescence and the membership oracle* MOracle[I]. *Assume there exists an interface automaton* A *with n states with distinguisher bound* B_{Dist} *modeling the typestate of* I. *Interface automaton* A *can be learned with* $O(|\Sigma_i| \cdot n^3 + n \cdot |\Sigma_i|^{B_{\text{Dist}}})$ *membership queries.*

Proof sketch. Starting with an asynchronous interface I and a membership oracle MOracle[I], using **Assumption 1** and **Assumption 3** we can construct the membership oracle MOracle[I_s] for the synchronous closure I_s of I. Given the distinguisher bound (or a state bound using **Assumption 2** and Theorem 4.3), we can construct an equivalence oracle EOracle[I_s] using Algorithm 2. Oracles MOracle[I_s] and EOracle[I_s] can then be used to learn a Mealy machine M with the same set of traces as I_s. This Mealy machine can be converted into the interface automata representing the callback typestate of I by: (a) Deleting all transitions with output err and all self-loop transitions with output quiet, and (b) Replacing all transitions with input wait with the output of the transition.

5 ACTIVE LEARNING FOR ANDROID

We implemented our method in a tool called DroidStar. In this section we describe how it works, the practical challenges we faced when working with Android, and our solutions to overcome them. DroidStar is implemented as an Android application and learns callback typestates from within a live Android system.

5.1 Designing an Experiment

To learn a typestate, a DroidStar user creates a test configuration (an extension of the `LearningPurpose` class) providing necessary information about a Java class under study. If known, the distinguisher bound can be provided here directly; otherwise, it can be obtained from **Assumption 2** by Theorem 4.3. The *instrumented alphabet*, also defined here, specifies an abstract alphabet for the learning algorithm and translation between the abstract alphabet and concrete callins/callbacks of the class under study. Several other options are available for adjusting the learning, the most important being the *quiescence timeout* which determines **Assumption 1**.

5.2 Observing Asynchronous Callbacks

In our approach we assume *bounded asynchrony* (**Assumption 1**) and, therefore, we can observe when the interface does not produce any new output (quiescence). We enforce this assumption on a real system with timeouts: the membership query algorithm waits for a new output for a fixed amount of time t_{max}, assuming that quiescence is reached when this time is elapsed. However, Android does not provide any worst case execution time for the asynchronous operations and we rely on the user to choose a large enough t_{max}. The membership query also assumes the existence of a minimum time t_{min} before a callback occurs. This ensures that we can issue a membership query with two consecutive callins (so, without a wait input in between), i.e., we have the time to execute the second callin before the output of the first callin.

Consider the MediaPlayer example from Section 2. The membership query `setDataSource(URL)` · wait · `prepareAsync()` · wait may not return the `onPrepared()` if t_{max} is violated, i.e., if the callback does not arrive before the timeout, and while testing it is possible that the `prepareAsync()` · `start()` might not return an error as expected if the lower bound t_{min} is violated. To avoid such issues we try to control the execution environment and parameters to ensure that callbacks occurred between t_{min} and t_{max}. In the MediaPlayer case, we must pick the right media source file.

5.3 Checking and Enforcing Our Assumptions

The simplest experiment to learn a class's callback typestate ties a single input symbol to each of its callins and a single output symbol to each of its callbacks. However, many Android classes have behaviors which cause this simple experiment to fail and require more detailed experiments to succeed.

The main challenges when designing an experiment are (a) *Nondeterministic behaviors*, i.e., the state of the device and external

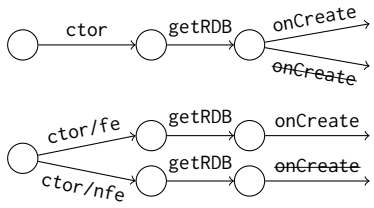

Figure 2: Eliminating non-determinism in `SQLiteOpenHelper`

events may influence an application. These elements are inherently non-deterministic; however, non-determinism violates **Assumption 3**. (b) The *parameter space* required to drive concrete test cases to witness a membership query is potentially infinite. Though we have ignored callin parameters till now, they are a crucial issue for testing. (c) The protocol we are learning may not be a regular language. Note that this is a violation of **Assumption 2**.

Non-Deterministic Behavior. Non-deterministic behavior is disallowed by our **Assumption 3**. However, to make this assumption reasonable we must make non-determinism straightforward to eliminate when it arises. We explain two primary classes of non-deterministic behaviors and strategies to eliminate these behaviors. The first class is related to controllable inputs and the second to uncontrollable ones (such as inputs from the device sensors).

Because the learning algorithm cannot learn from non-deterministic systems, DroidStar will terminate if such behavior is detected. To assist in this process, DroidStar will report a non-deterministic behavior is detected and display the disagreeing sequences to the user. It detects this by caching all membership queries as input/output sequence pairs. When a new trace is explored, DroidStar checks that the trace prefixes are compatible with the previously seen traces.

In the first case, a hidden (not modeled) controllable input influences the typestate. We resolve this non-determinism by manually adding the input value and create a finer alphabet that explicate the previously hidden state of the environment. For example, in the class `SQLiteOpenHelper`, the `getReadableDatabase()` may either trigger a `onCreate()` callback or not, depending on the parameter value to a previous callin (`constructor`)was the name of an existing database file. Hence, the behavior of the callin is non-deterministic, depending on the status of the database on disk. In the `SQLiteOpenHelper` example, we split the `constructor` callin into `constructor/fileExists` and `constructor/noFileExists` and pass the right parameter values in each case. With this extra modeling we can learn the interface automaton, since the execution `getReadableDatabase()` ends in two different states of the automaton (see Figure 2).

The second class is the effect of the uncontrollable inputs on a typestate. Such effects, by definition, cannot be controlled or made explicit prior to the call. We can sometimes to remove this non-determinism by merging different outputs, considering them to be the same. This is the dual of the previous solution.

An example is the `SpeechRecognizer`, for which calling `startListening()` produces different callbacks depending on the environment. As the environment cannot be reasonably controlled, we merge outputs to go to the same state. If outputs are erroneously merged, the non-determinism will propagate and continue to manifest. Thus there is no risk of unsound results.

Handling Callin Parameters. While parameter-less callins such as `start()` and `stop()` are common in Android classes, many parameterized callins exist. Because input symbols need to be listed in the experiment definition, the full range of parameter values cannot be explored. In practice, we found that parameters often have little effect on the typestate automaton. In cases where they do affect the automaton, multiple input symbols can be defined to represent the same method called with several different parameters. This solution is similar to splitting on environmental effects when dealing with non-determinism.

Learning from Non-Regular Languages. An intrinsic limitation of L^* is that it learns only regular languages. However, some classes expose non-regular protocols. Common cases include situations where a *request* callin invoked *n* times trigger exactly *n response* callbacks. In the SpellCheckerSession class, callin `getSuggestion()` and callback `onGetSuggestions()` follow this pattern.

However, even in such cases, it can be useful to build a regular approximation of the typestate. For example, restricting the typestate to behaviors where there is at most one pending request (a regular subset) provides all the information a programmer would need. Hence, in such cases, we use the technique of *learning purposes* [1] to learn a regular approximations of the infinite typestate.

6 EMPIRICAL EVALUATION

We evaluated our interface-learning technique, as implemented in DroidStar, by using it to generate callback typestates for 16 classes, sampled from the Android Framework and popular third-party libraries. DroidStar is available at https://github.com/cuplv/droidstar. For these experiments, DroidStar was run on an LG Nexus 5 with Android framework version 23. Our evaluation was designed to answer the following questions:

(1) Does our technique learn typestates efficiently?
(2) What size distinguisher bounds occur in practice? Do they support the small distinguisher bound hypothesis?
(3) Do the callback typestates we learn reveal interesting or unintended behavior in the interfaces?

Methodology. For each experimental class, we manually identified a reduced alphabet of relevant callins and callbacks and provided them (along with other necessary information as explained in Section 5) to DroidStar through instances of the `LearningPurpose`. Relevant callins and callbacks for these experiments were those which, according to the available documentation, appeared to trigger or depend on typestate changes (enabling or disabling of parts of the interface). Each instance consisted of $50-200$ lines of, mostly boiler-plate, Java or Scala code.

To evaluate efficiency, we measured the overall time taken for learning, as well as the number of membership (MQ) and equivalence queries (EQ). The number of queries is likely a better measure of performance than running time: the running time depends on external factors. For example, in the media player the running time depends on play-length of the media file chosen during testing.

We validated the accuracy of learned callback typestates using two approaches. First, for classes whose documentation contains a picture or a description of what effectively is an callback typestate, we compared our result to the documentation. Second, for all other

classes we performed manual code inspection and ran test apps to evaluate correctness of the produced typestates.

We used a distinguisher bound of 2 for our experiments; further, we manually examined the learned typestate and recorded the actual distinguisher bound. For our third question, i.e., does the learned callback typestate reveal interesting behaviors, we manually examined the learned typestate, compared it against the official Android documentation, and recorded discrepancies.

6.1 Results

We discuss the results (in Table 1) and our three questions.

Question 1: Efficiency. The table shows that our technique is reasonably fast: most typestates learned within a few minutes. The longest one takes 71 minutes, still applicable to nightly testing. The numbers for membership queries are reported as $X(Y)$—X is the number of membership queries asked by the algorithm, while Y is the number actually executed by the membership oracle. This number is lower as the same query may be asked multiple times, but is executed only once and the result is cached. For each benchmark, the accuracy validation showed that the produced typestate matched the actual behavior.

Question 2: Distinguisher Bounds. As mentioned before, we used a distinguisher bound of 2 for all experiments. However, a manual examination of the learned callback typestates showed that a bound of 1 would be sufficient in all cases except the `SQLiteOpenHelper` and the `OkHttpCall` where bounds of 2 are necessary. This supports our conjecture that, in practice, interfaces are designed with each state having a unique functionality (see Section 4.3).

Question 3: Interesting Learned Behavior. Of the three questions, our experiments to examine the learned callback typestate for interesting behavior turned out to be the most fruitful, uncovering several discrepancies, including corner cases, unintended behavior and likely bugs, in the Android framework. These results reaffirm the utility of our main goal of automatically learning callback typestate, and suggest that learning typestate can serve valuable roles in documentation and validation of callback interfaces.

In 2 cases, the learned typestate and documented behavior differed in certain corner cases. We carefully examined the differences, by framework source examination and manually writing test applications, and found that the learned typestate was correct and the documentation was faulty. In 5 other cases, we believe the implemented behavior is not the intended behavior, i.e., these are likely bugs in the Android implementation. These discrepancies mostly fall into two separate categories:

Incorrect documentation. In such cases, it turned out that the discrepancy is minor and unlikely to produce bugs in client programs.

Race conditions. Several likely bugs were due to a specific category of race conditions. These interfaces have (a) a callin to start an action and a corresponding callback which is invoked when the action is successfully completed; (b) a callin to cancel an already started action and a corresponding callback which is invoked if the action is successfully cancelled. When the start action and cancel action callins are called in sequence, the expectation is that exactly one of the two callbacks are called. However, when the time between the two callins is small, we were able to observe unexpected behaviors, including neither or both callbacks being invoked.

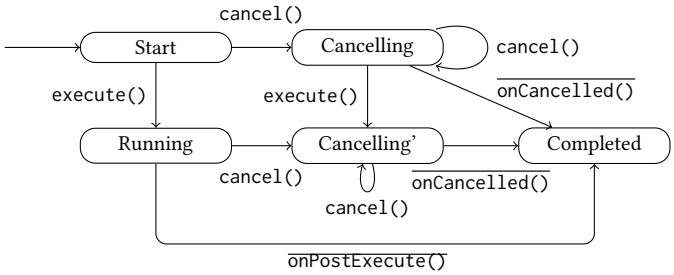

Figure 3: Learned typestate of the AsyncTask class

6.2 Selected Experiments

Of our 16 benchmarks, we briefly explain 5 here. The remaining experiments are discussed in the technical report [29][1].

MediaPlayer. This is the class from the example in Section 2. The learned typestate differs from the existing documentation. The learned typestate: (a) has the `pause()` callin enabled in the "playback completed" state, and (b) shows that `onPrepared()` is invoked even after the synchronous callin `prepare()`. Though undocumented, these behaviors are unlikely to cause any issues.

AsyncTask. The AsyncTask class turns arbitrary computations into callback operations with progress tracking and results are delivered via callbacks. For our experiment, the computation is a simple timer. A constructed AsyncTask object performs its task when it receives the `execute()` callin, and then either returns the results with the `onPostExecute()` callback, or returns an `onCancelled()` if `cancel()` is called first. The object is single-use; after it has returned a callback it will accept no further `execute()` commands.

Our experiment revealed an unexpected edge-case: if `execute()` is after `cancel()` but before the `onCancelled()` callback is received, it will not throw an exception but will never cause the callback task to be run. The learned interface is in Figure 3.

SpeechRecognizer. This class provides an example of uncontrollable environmental non-determinism. The particular callback that signals the end of the speech session—either an `onResults()` or an `onError()`—is determined by the environment (in particular, the sound around the phone during the test). In this case, to reduce the system to a deterministic one we can learn, we supposed that the state after an `onResults()` or `onError()` is the same and merged the two callbacks into a single `onFinished()` symbol.

Our results revealed two interesting corner cases for the ordering of inputs. First, if an app calls `cancel()` between calling `startListening()` and receiving the `onReadyForSpeech()` callback (represented by our "starting" output symbol), calling `startListening()` again will have no effect until after a certain amount of time, as shown by the `wait` transition from state "Cancelling" to "Finished". Delays in readiness like this can be generally considered bugs; if a system will not be ready immediately for inputs it should provide a callback to announce when the preparations are complete, so as not to invite race conditions.

Our second corner case is where the app calls `stopListening()` as the very first input on a fresh SpeechRecognizer. This will not throw an exception, but calling `startListening()` at any point after will fail, making the object effectively dead.

[1] http://arxiv.org/abs/1701.07842

Class name	LP LoC	states	Time (s)	MQ	EQ	MQ per EQ	B_{Dist} (needed)
AsyncTask	79	5	49	372 (94)	1	356 (0)	2 (1)
BluetoothAdapter	161	12	1273	839 (157)	2	420 (16)	2 (1)
CountDownTimer	94	3	134	232 (61)	1	224 (0)	2 (1)
DownloadManager	84	4	136	192 (43)	1	190 (0)	2 (1)
FileObserver	134	6	104	743 (189)	2	351 (8)	2 (1)
ImageLoader (UIL)	80	5	88	663 (113)	2	650 (33)	2 (1)
MediaCodec	152	8	371	1354 (871)	1	973 (482)	2 (1)
MediaPlayer	171	10	4262	13553 (2372)	5	2545 (384)	2 (1)
MediaRecorder	131	8	248	1512 (721)	1	1280 (545)	2 (1)
MediaScannerConnection	72	4	200	403 (161)	2	163 (57)	2 (1)
OkHttpCall (OkHttp)	79	6	463	839 (166)	2	812 (13)	2 (2)
RequestQueue (Volley)	79	4	420	475 (117)	1	460 (0)	2 (1)
SpeechRecognizer	168	7	3460	1968 (293)	3	646 (35)	2 (1)
SpellCheckerSession	109	6	133	798 (213)	4	374 (8)	2 (1)
SQLiteOpenHelper	140	8	43	1364 (228)	2	665 (6)	2 (2)
VelocityTracker	63	2	98	1204 (403)	1	1156 (0)	2 (1)

Table 1: DroidStar experimental results.

SQLiteOpenHelper. This class provides a more structured interface for apps to open and set up SQLite databases. It has callbacks for different stages of database initialization, allowing apps to perform setup operations only as they are needed. When a database is opened with `getWritableDatabase()`, a callback `onConfigure()` is called, followed by an `onCreate()` if the database didn't exist yet or an `onUpgrade()` if the database had a lower version number than was passed to the SQLiteOpenHelper constructor, all followed finally by an `onOpen()` when the database is ready for reading. The database can then be closed with a `close()`.

Our experiment observed the callbacks when opening databases in different states (normal, non-existent, and out of date) and performing the `close()` operation at different points in the sequence. We found that once the `getWritableDatabase()` method is called, calling `close()` will not prevent the callbacks from being run.

VelocityTracker. This class was a special case with no asynchronous behavior; it was a test of our tool's ability to infer traditional, synchronous typestates. The class has a `recycle()` method that we expected to disable the rest of the interface, but our tool found (and manual tests confirmed) that the other methods can still be called after recycling. The documentation's warning that "You must not touch the object after calling [recycle]" is thus not enforced.

7 RELATED WORK

Works which automatically synthesize specifications of the valid sequences of method calls (e.g. [3, 4, 18, 34]) typically ignore the asynchronous callbacks.

Static analysis has been successfully used to infer typestates specifications (importantly, without callbacks) [3, 23, 34]. The work in [3] infers classical typestates for Java classes using L*. In contrast, our approach is based on testing. Therefore, we avoid the practical problem of abstracting the framework code. On the other hand, the use of testing makes our L* oracles sound only under assumptions. Similarly, [19] uses L* to infer classical typestates, including ranges of input parameters that affect behavior. However, their tool is based on symbolic execution, and thus would not scale to systems as large and complex as the Android Framework.

Inferring interfaces using execution traces of client programs using the framework is another common approach [2, 4, 13, 17, 28, 38, 40, 41]. In contrast to dynamic mining, we do not rely on the availability of client applications or a set of execution traces. The L* algorithm drives the testing.

The analysis of event-driven programming frameworks has recently gained a lot of attention (e.g. [6, 9, 10, 26]). However, none of the existing works provide an automatic approach to synthesize interface specifications. Analyses of Android applications mostly focus on either statically proving program correctness or security properties [6, 9, 15, 21, 39] or dynamically detecting race conditions [8, 24, 27]. These approaches manually hard-code the behavior of the framework to increase the precision of the analysis. The callback typestate specifications that we synthesize can be used here, avoiding the manual specification process.

Our work builds on the seminal paper of Angluin [5] and the subsequent extensions and optimizations. In particular, we build on L* for I/O automata [1, 32]. The optimizations we use include the counterexample suffix analysis from [30] and the optimizations for prefix-closed languages from [25]. The relation to conformance testing methods [12, 16, 20, 22, 31] has been discussed in Section 4.3.

8 CONCLUSION

We have shown how to use active learning to infer callback typestates. We introduce the notion of distinguisher bound which take advantage of good software engineering practices to make active learning tractable on the Android system. Our method is implemented in the freely available tool called DroidStar. This paper enables several new research directions. We plan to investigate mining parameters of callins from instrumented trace from real user interactions, as well as the inference of structured typestates (for instance, learning a typestate as a product of simpler typestates).

ACKNOWLEDGMENTS

This research was supported in part by DARPA under agreement FA8750-14-2-0263. Damien Zufferey was supported in part by the European Research Council Grant Agreement No. 610150 (ERC Synergy Grant ImPACT (http://www.impact-erc.eu/)).

REFERENCES

[1] F. Aarts and F. Vaandrager. Learning I/O automata. In *CONCUR 2010*, pages 71–85, 2010.

[2] M. Acharya, T. Xie, and J. Xu. Mining interface specifications for generating checkable robustness properties. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 311–320, Nov 2006.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, pages 98–109, 2005.

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

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

[6] 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. page 29, 2014.

[7] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 2–26, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for android applications. In *OOPSLA*, pages 332–348, 2015.

[9] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. pages 163–182, 2015.

[10] 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 *NDSS*, 2015.

[11] Guido De Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.*, 22(3):25:1–25:46, July 2013.

[12] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[13] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.

[14] L. de Alfaro and T. Henzinger. Interface automata. In *FSE*, pages 109–120, 2001.

[15] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. pages 576–587, 2014.

[16] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, Jun 1991.

[17] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[18] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *ICSE*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[19] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, pages 248–264, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Guney Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Computers*, 19(6):551–558, 1970.

[21] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. 2015.

[22] F. C. Hennie. Fault detecting experiments for sequential circuits. In *5th Annual Symposium on Switching Circuit Theory and Logical Design, Princeton, New Jersey, USA, November 11-13, 1964*, pages 95–110, 1964.

[23] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 31–40, New York, NY, USA, 2005. ACM.

[24] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *PLDI*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[25] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *CAV*, pages 315–327, 2003.

[26] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node.js javascript applications. In *OOPSLA*, OOPSLA 2015, pages 505–519, New York, NY, USA, 2015. ACM.

[27] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *PLDI 2014*, page 34, 2014.

[28] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.

[29] Arjun Radhakrishna, Nicholas Lewchenko, Shawn Meier, Sergio Mover, Krishna Chaitanya Sripada, Damien Zufferey, Bor-Yuh Evan Chang, and Pavol Černý. Learning asynchronous typestates for android classes. *CoRR*, abs/1701.07842, 2017.

[30] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[31] Krishan K. Sabnani and Anton T. Dahbura. A new technique for generating protocol test. In *SIGCOMM '85, Proceedings of the Ninth Symposium on Data Communications, British Columbia, Canada, September 10-12, 1985*, pages 36–43, 1985.

[32] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 207–222, 2009.

[33] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.

[34] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA*, ISSTA '07, pages 174–184, New York, NY, USA, 2007. ACM.

[35] J. Shore. Fail fast [software debugging]. *IEEE Software*, 21(5):21–25, Sept 2004.

[36] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[37] Frits W. Vaandrager. On the relationship between process algebra and input/output automata. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 387–398, 1991.

[38] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[39] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *SIGSAC*, pages 1329–1341, 2014.

[40] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM.

[41] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291, New York, NY, USA, 2006. ACM.