Synthesis of Interface Specifications for Android Classes

Arjun Radhakrishna[1], Nicholas Lewchenko[2], Shawn Meier[2], Sergio Mover[2], Krishna Chaitanya Sripada[2], Damien Zufferey[3], Bor-Yuh Evan Chang[2], and Pavol Černý[2]

[1]University of Pennsylvania
[2]University of Colorado Boulder [3]MIT

Abstract. Event-driven programming frameworks interact with client code using callins (framework methods that the client invokes) and callbacks (client methods that the framework invokes). The protocols for interacting with such frameworks can often be described by finite-state machines we dub *asynchronous typestates*. Asynchronous typestates are akin to classical typestates, with the key difference that asynchronous typestates have both inputs (corresponding to callins) and outputs (corresponding to callbacks). Typically, callbacks are produced asynchronously. Our key contribution is an algorithm that infers asynchronous typestates for Android framework classes. It is based on the L^* algorithm. We show how to implement the oracles for membership and equivalence queries that L^* needs. The membership oracle is implemented using testing. Our main technical result is that under realistic assumptions, the equivalence oracle can be implemented using the membership oracle.

We implemented our approach and evaluated it empirically. We used our tool, Starling, to synthesize asynchronous typestates for Android framework classes both for cases where Android documentation provides the asynchronous typestate clearly, and for cases where the documentation is unclear. The results show that Starling synthesizes asynchronous typestates accurately and efficiently. Additionally, in several cases, the synthesized asynchronous typestates uncovered surprising and undocumented behavior of Android framework classes.

1 Introduction

Event-driven programming frameworks interact with client code using callins and callbacks. Callins are framework methods that the client invokes, and callbacks 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.

For example, consider the interaction of a client application and the framework when the client wants to use a particular framework service. The client asks for the service to be started by invoking a `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 $\overline{\texttt{onServiceStarted()}}$ 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.

The protocol for the interaction in the example can be described by a finite state machine (with a small number of states) we call *asynchronous typestate.* An asynchronous typestate can be seen as a behavioral interface the framework provides to the client. Typestates [25] were introduced to describe synchronous interfaces that specifies order of method calls. As they are synchronous, typestates involve only callins (which can return output values). In contrast, asynchronous typestates have both inputs (corresponding to callins) and outputs (corresponding to callbacks). In automata theory, asynchronous typestates can be seen as interface automata. An interface automaton [11] is an automaton with inputs and outputs. Importantly, interface automata allow outputs to be received asynchronously with respect to the inputs. Therefore, interface automata are suitable for modeling the protocols for client-framework interaction in event-driven programming.

Problem. The central problem we consider is how to infer asynchronous typestates from framework code. Asynchronous typestates are useful in a number of ways. First, asynchronous typestates are a useful form of documentation. They tell client application programmers in what order to invoke callins and when it is necessary to wait for a callback. Some Android framework documentation already uses pictures very similar to asynchronous typestates[1]. Second, asynchronous typestates are very useful in verification of client code. They enable checking that a client uses the framework correctly. Third, even though we infer the asynchronous typestates from framework code, they can be used for certain forms of framework verification. Specifically, one can infer an asynchronous typestate for different versions of the framework, and compare them to see if the interface the framework provides has changed.

Algorithm. We present an algorithm for inferring typestates for framework classes. Concretely, we focus on the Android framework, but our methods are equally applicable in other contexts. Our algorithm is based on Angluin's L^* algorithm [5]. In L^*, there is a learner whose goal is to learn a finite-state machine — in our case we are trying to learn the asynchronous typestate of a class C in the Android framework. The learner accomplishes this by asking a teacher a number of queries. These queries are of two types: membership queries and equivalence queries. We note that the teacher does not need to know the solution, she only needs to know how to answer the queries from a learner.

Membership and equivalence oracles The key question we answer is how to implement oracles that can answer the membership and equivalence queries. The membership query asks whether whether a given sequence of callins is legal (that is, it does not raise an exception or reach an error state when invoked on C), and if so, what sequence of callbacks C generates in response. We implement the membership oracle by testing. The equivalence query asks whether the current hypothesis H is a correct typestate for the class C. If this is not the case, the teacher is required to provide a counterexample: a sequence of callins and callbacks allowed by H which does not arise when a client interacts with C, or vice

[1] `https://developer.android.com/reference/android/media/MediaPlayer.html`

versa. Answering the equivalence query in general requires checking language inclusion for sets of traces of two programs (C and H). This is an undecidable problem in general. Our *main technical result* is that under realistic assumptions, the equivalence oracle can be implemented using a number of calls to the membership oracle. The result is based on an automata-theoretic result that bounds the length of the longest possible minimal counterexample for equivalence of two automata. The equivalence oracle crucially uses the following boundedness assumption: if C has an asynchronous typestate, then its number of states is less than a constant k. The assumption holds in practice: the largest asynchronous typestates of an Android class we encountered had fewer than 10 states.

Implementing membership oracle on Android. Our membership oracle is implemented using testing. More concretely, we run the sequence of callins, and log exceptions and errors that occur and the callbacks that are invoked. The oracle might return incorrect results. For instance, the framework might take too long to invoke a callback, and our test might miss it due to a timeout. The test would work if in practice, the time the framework takes to invoke a callback is bounded from above and from below. We explain in detail the potential sources of unsoundness and our mitigation strategies in Section 5. An alternative would be to use static analysis. However, static analysis for very large framework (such as Android) is well-known to be a hard problem [8], precisely because of the callin-callback infrastructure. However, if sufficiently powerful static analyses will be developed, we could use it to implement the membership oracle.

Empirical evaluation. We implemented our approach and evaluated it empirically. We used our tool, Starling, to synthesize asynchronous typestates for Android framework classes. Our test set included both classes where Android documentation provides the asynchronous typestate, and classes where the documentation is unclear. Furthermore, for each class from our test set, we report the overall time taken to infer the typestate, and the number of membership and equivalence queries taken (and the number of membership queries each equivalence query required). The results show that Starling can synthesize asynchronous typestates accurately (confirmed by documentation, code inspection, and manual comparison to simple Android applications) and efficiently. As an added bonus, by inspecting our typestates, we uncovered corner cases with surprising behavior that could be considered either buggy or undocumented. This was the case for Android framework classes AsyncTask, SpeechRecognizer, and SpellCheckerSession. Moreover, the asynchronous typestates uncovered an undocumented difference between versions of Android in the SpeechRecognizer class (the older version, 5.1.1, seems to have a bug).

Contributions. The main contributions of this paper are: (a) We introduce the notion of asynchronous typestates of classes and developed an approach for their automated inference based on the L^* algorithm. (b) We show how to implement membership and equivalence oracles required by the L^* algorithm. (c) We evaluate our approach on a number of examples, and show that it produces asynchronous typestates accurately and efficiently.

2 Illustrative example

Let us consider the part of an asynchronous typestate in Figure 1. The image is taken from the documentation for the Android class called MediaPlayer. The protocol that governs the client-framework interaction works as follows. The client first invokes the callin `setDataSource()`, and the protocol transitions to the `Initialized` state (top right). In this state, the client can invoke the callin `prepareAsync()`, and the protocol transitions to the `Preparing` state (top left). In the `Preparing` state, the client cannot invoked any callins. The framework can invoke the $\overline{\texttt{onPrepared()}}$ callback (asynchronously), and then, the protocol transitions to the `Prepared` state. The callback $\overline{\texttt{onPrepared()}}$ is pictured with a double arrow. At this point, the client can invoke the `start()` callin, and the media starts playing. We emphasize that the client cannot invoke the `start()` callin in the `Preparing` state, before it receives the $\overline{\texttt{onPrepared()}}$ callback.

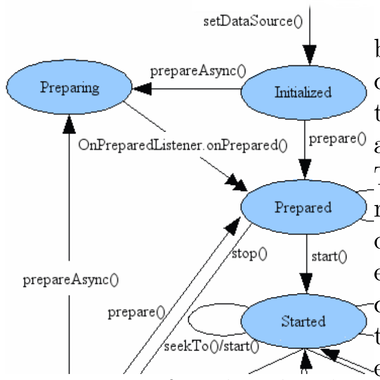


Fig. 1: Part of Media Player's asynchronous typestate.

The basic structure of our approach is given by the L* algorithm. A learner asks membership queries, until she can form a hypothesis automaton and ask an equivalence query. If a counterexample is returned, the algorithm learns from it. The process repeats until the equivalence query returns that the hypothesis is correct. For MediaPlayer, the first set of membership queries each invoke a different callin. Of these, only the query containing `setDataSource()` succeeds – the testing-based membership oracle raises an exception on all the other callins. The algorithm tries longer membership queries while building the hypothesis automaton. For instance, it learns that `prepareAsync()` and `prepare()` cannot lead to the same state: it is possible to invoke the `start()` after `prepare()`, but not after `prepareAsync()`. However, once the client receives the callback $\overline{\texttt{onPrepared()}}$, `start()` may be called. The learner thus hypothesizes the transition from the `Preparing` state to the `Prepared` state on the $\overline{\texttt{onPrepared()}}$ callback. Once the hypothesis automaton is complete, the learner asks the equivalence query. For MediaPlayer, the correct automaton is found after 5 equivalence queries. We describe the algorithm in detail in Section 4, and the threats to validity of our approach (due to the fact that we implement the membership oracle by testing) in Section 5.

3 The Asynchronous Typestate Learning Problem

We introduce models of interfaces, define the asynchronous typestate learning problem, and present an impossibility result about learning typestates.

3.1 Modelling Interfaces

Asynchronous Interfaces. Let Σ_i and Σ_o be the set of all callins and callbacks to a given asynchronous interface. We use a simple model of interface behaviours, abstracting away the details of parameter and return values of callins and callbacks. A behaviour is modelled as a *trace* $\tau_i = \sigma_0 \ldots \sigma_n \in (\Sigma_i \cup \Sigma_o)^*$. Formally, an *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 and Typestates. Asynchronous interfaces are commonly represented using interface automata [11]. An *interface automaton* A is given by a tuple $\langle Q, q_\iota, \Sigma_i, \Sigma_o, \Delta_\mathsf{A} \rangle$ where: (a) Q are a finite set of *states*, (b) $q_\iota \in Q$ is the *initial state*, (c) Σ_i and Σ_o are finite set of *input and output symbols*, and (d) $\Delta_\mathsf{A} \subseteq Q \times \{\Sigma_i \cup \Sigma_o\} \times Q$ are a set of *transitions.* A *trace* τ_a of A is given by $\sigma_0 \ldots \sigma_n$ if $\exists q_0 \ldots q_{n+1} : q_0 = q_\iota \wedge \forall i.(q_i, \sigma_i, q_{i+1}) \in \Delta_\mathsf{A}$. We say that A *models the typestate of* $\mathsf{I} = \langle \Sigma_i, \Sigma_o, \Pi_i \rangle$ if each trace of A is a trace of I, and vice versa.

3.2 Problem Statement

Given an interface $\mathsf{I} = \langle \Sigma_i, \Sigma_o, \Pi_i \rangle$, the *asynchronous typestate learning problem* is to learn an interface automaton A that models I. In our setting, we allow the learner to ask a *membership oracle* $\mathsf{MOracle[A]}$ membership queries. For a *membership query*, the learner picks $\mathsf{mQuery} = i_0 i_1 \ldots i_n \in \Sigma_i^*$ and the membership oracle $\mathsf{MOracle[A]}$ 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.

3.3 The Theory and Practice of Learning Typestates

An interface automaton A is *compatible* with a membership query mQuery and its result $\mathsf{MOracle[I](mQuery)}$ if: (a) $\mathsf{MOracle[I](mQuery)}$ is a trace of A, or (b) $\mathsf{MOracle[I](mQuery)} = \perp$ and there is no trace of A whose sequence of inputs is to mQuery. The following impossibility theorem shows that membership queries are not sufficient to effectively learn typestates.

Theorem 1. *There exists an interface* I *and an* $\mathsf{MOracle[A]}$ *such that:*

- *There exists an interface automaton* A *that models the typestate of* I*, and*
- *For every finite set of pairs* $(\mathsf{mQuery}, \mathsf{MOracle[A](mQuery)})$ *of membership queries and corresponding results, there exist interface automata* A' *and* A''*, with different sets of traces, compatible with each pair.*

Despite the above result, we claim that asynchronous typestates can be effectively learned given additional assumptions. We analyze the causes behind the impossibility and for each, highlight the assumption necessary to overcome it.

Unbounded asynchrony. Using membership queries alone, it is not possible to determine if the interface will produce more outputs (callbacks) at any point in time. Hence, we assume that:

Assumption 1: Quiescence is observable.

This assumption is commonly used in ioco-testing frameworks. Specifically, we add an input wait and an output quiet, where quiet is returned after a wait input 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 1. Using wait and quiet, in the MediaPlayer example, we have that `setDataSource()` $\cdot$ `prepareAsync()` $\cdot\, \overline{\texttt{onPrepared()}}$ $\cdot$ `wait` $\cdot$ `quiet` is a valid trace, but `setDataSource()` $\cdot$ `prepareAsync()` $\cdot$ `wait` $\cdot$ `quiet` is not.

Behaviour 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 significantly different behaviour for input sequences much longer than k. While this is a theoretical limitation, it is not a problem in practice as most asynchronous typestates are rather small ($\leq$ 10 states).

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

Non-determinism. Of the three causes behind the impossibility of learning asynchronous typestate with membership queries, this is the hardest to deal with.

Example 2. Consider an interface with $\Sigma_i = \{\texttt{input}\}$ and $\Sigma_o = \{\texttt{out1}, \texttt{out2}\}$, and traces given by $(\texttt{input} \cdot (\texttt{out1} \mid \texttt{out2}))^*$. Informally, the interface waits for an `input` and non-deterministically calls back either $\overline{\texttt{out1}}$ or $\overline{\texttt{out2}}$. All membership queries are a sequence of `input`'s; however, it is possible that the membership oracle may never return a trace where $\overline{\texttt{out2}}$ occurs. In that case, no learner will be able to learn the interface exactly. More severely, the oracle may return traces where $\overline{\texttt{out1}}$ and $\overline{\texttt{out2}}$ alternate in no discernable pattern, misleading the learner.

In practice, the non-determinism problem is somewhat alleviated due to the nature of asynchronous typestates (see Section 5). See [1] for a detailed theoretical discussion of how non-determinism affects learnability. For now, we have:

Assumption 3: The interface is deterministic

Formally, we assume that for every trace τ_a of the interface, there is at most one output $o \in \Sigma_o$ such that $\tau_a \cdot o \in \Pi_i$.

4 Learning Asynchronous Typestates using L*

Given **Assumption 1** and **Assumption 3**, we first build a "synchronous closure" of an asynchronous interface (Section 4.1). We show that the synchronous closure can effectively learned 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. For synchronous interfaces, there exists a rich body of work from which we can draw learning techniques [5,1,18].

In the following discussion, define $\tilde{\Sigma_i} = \Sigma_i \cup \{\mathsf{wait}\}$ and $\tilde{\Sigma_o} = \Sigma_o \cup \{\mathsf{quiet}, \epsilon, \mathsf{err}\}$. The purpose of the additional inputs and outputs are discussed below. For any $\tau_s \in (\tilde{\Sigma_i} \cdot \tilde{\Sigma_o})^*$, we define $\mathsf{async}(\tau_s) = \tau_a \in (\Sigma_i \cup \Sigma_o)^*$ where τ_a is obtained from τ_s by erasing all occurrences of wait, quiet, ϵ, 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 recursively as follows:

$$(\epsilon \in \Pi_s) \wedge \bigwedge \tau_s \in \Pi_s \left\{ \begin{array}{rcl} \mathsf{async}(\tau_s) \cdot i \in \Pi_i & \Longrightarrow & \tau_s \cdot i \cdot \epsilon \in \Pi_s \\ \mathsf{async}(\tau_s) \cdot o \in \Pi_i & \Longrightarrow & \tau_s \cdot \mathsf{wait} \cdot o \in \Pi_s \\ \mathsf{async}(\tau_s) \cdot i \notin \Pi_i & \Longrightarrow & \tau_s \cdot i \cdot \mathsf{err} \in \Pi_s \\ \forall o \in \Sigma_o : \mathsf{async}(\tau_s) \cdot o \notin \Pi_i & \Longrightarrow & \tau_s \cdot \mathsf{wait} \cdot \mathsf{quiet} \in \Pi_s \\ \mathsf{err} \text{ occurs in } \tau_s & \Longrightarrow & \tau_s \cdot i \in \Pi_s \end{array} \right\}$$

Informally, in I_s: (a) Each input is immediately followed by a dummy output ϵ; (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.

We use a slightly different formalism for membership oracles $\mathsf{SMOracle}[\mathsf{I}_s]$ for synchronous closures. Given a sequence of inputs mQuery, instead of returning a trace, $\mathsf{SMOracle}[\mathsf{I}_s]$ only returns the output sequence of a trace with input sequence mQuery. Given $\mathsf{MOracle}[\mathsf{I}]$ and **Assumption 1**, it is easy to construct the synchronous membership $\mathsf{SMOracle}[\mathsf{I}_s]$. Note that due to **Assumption 3**, there is exactly one possible output sequence for every input sequence.

Mealy machines. Instead of using interface automata, we represent synchronous interfaces with a simpler formalism. A *Mealy machine* M is given by a tuple $\langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \Lambda\rangle$ where: (a) Q, q_ι, $\tilde{\Sigma}_i$, and $\tilde{\Sigma}_o$ are states, initial states, inputs and outputs, respectively, (b) $\delta : Q \times \tilde{\Sigma}_i \to Q$ is a *transition function*, and (c) $\Lambda : Q \times \tilde{\Sigma}_i \to \tilde{\Sigma}_o$ is an *output function*. A *run segment* π of M is given by $q_0 \to^{i_0}_{o_0} \ldots \to^{i_n}_{o_n} q_{n+1}$ where $\forall 0 \leq i \leq n : \delta_{\mathsf{M}}(q_i, i_i) = q_{i+1} \wedge \Lambda(q_i, i_i) = o_i$. We often abuse notation and write: (a) $\delta(q_0, i_0 \ldots i_n)$ for q_{n+1}, and (b) $\Lambda(q_0, i_0 \ldots i_n)$ for $o_0 \ldots o_n$. If $q_0 = q_\iota$, then we say that $i_0 o_0 \ldots i_n o_n$ is a *trace* of M. Further, if $q_0 = q_\iota$, we write $\mathsf{M}(i_0 \ldots i_n) = o_0 \ldots o_n$. A Mealy machine M is the *asynchronous typestate* of I_s if every trace of M is a trace of I_s, and vice versa.

4.2 L*: Learning Mealy Machines

We describe the classical L* learning algorithm by Angluin [5] adapted to Mealy machines. Additionally, the algorithm described below also incorporates the significant counterexample suffix analysis from [22].

In the following discussion, fix an asynchronous interfaces I and its synchronous closure I_s. Assume that there exists a *target Mealy machine* $\mathsf{M}^* = \langle Q^*, q^*_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta^*, \Lambda^*\rangle$ that is the typestate of I_s. Further, assume that M^* is minimal, i.e., has the fewest possible states. Now, for a membership oracle $\mathsf{SMOracle}[\mathsf{I}_s]$ and a membership query mQuery, we have that $\mathsf{SMOracle}[\mathsf{I}_s](\mathsf{mQuery}) = \mathsf{M}^*(\mathsf{mQuery})$.

In the L* algorithm, in addition to a membership oracle, the learner has access to an *equivalence oracle* $\mathsf{EOracle}$. For an equivalence query, the learner passes a Mealy machine M to $\mathsf{EOracle}$, and is in turn returned: (a) A *counterexample input* cex such that $\mathsf{M}(\mathsf{cex}) \neq \mathsf{M}^*(\mathsf{cex})$, or (b) $\mathsf{Correct}$ if no such cex exists.

The full L* algorithm is shown in Algorithm 1. In Algorithm 1, the learner maintains the following: (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 $\mathsf{SMOracle}(w_i \cdot e)$ of length $|e|$. The entries are computed by the sub-procedure $\mathsf{FillTable}$.

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,

Algorithm 1 L* for Mealy machines

```
Input: Membership oracle SMOracle, 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:     $\forall w_i, i : \Lambda([w_i], i) \leftarrow o$   if $T(w_i)(i) = o$
12:     return $\langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \Lambda \rangle$
13: function AnalyzeCex(M,cex)
14:     for all $0 \leq i \leq |\mathsf{cex}|$ and $w_i^p, w_i^s$ such that $w_i^p \cdot w_i^s = \mathsf{cex} \wedge |w_i^p| = 1$ do
15:         $w_o^p \leftarrow \mathsf{M}(w_i^p)$; $[w_i^{p'}] \leftarrow \mathsf{M}_{\mathsf{state}}(w_o^p)$
16:         $w_o^s \leftarrow$ last $|w_i^s|$ symbols of SMOracle($w_i^{p'} \cdot w_i^s$)
17:         if $w_o^p \cdot w_o^s \neq$ SMOracle(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 SMOracle($w_i \cdot e$) of length $|e|$
```

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 [22].

Theorem 2 ([22]). *Algorithm 1 learns* $\mathsf{M}^* = \langle Q^*, q_\iota^*, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta^*, \Lambda^* \rangle$ *using at most* $O(|\tilde{\Sigma}_i| \cdot |Q^*|^2 + |Q^*| \log m)$ *membership queries and* $|Q^*|$ *equivalence queries where* m *is the length of the largest counterexample returned by the equivalence oracle. If the equivalence oracle returns minimal counterexamples,* $m \leq |Q^*|$.

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**).

State bounds. A *state bound* of $B_{\mathsf{State}} \in \mathbb{N}$ 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.

Theorem 3. *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 [23, 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 $k_{\mathsf{M}}+B_{\mathsf{State}}-1$.

Theorem 4. *Given a membership oracle* SMOracle *for a target Mealy machine* M^* *with at most* B_{State} *states, and a Mealy machine* M *with* k *states, the equivalence query can be answered using at most* $|\tilde{\Sigma}_i|^{B_{\mathsf{State}}+k-1}$ *membership queries.*

While this simple algorithm is correct and easy to implement, the performance is lacklustre. The algorithm performs many more membership queries than necessary, and does not take advantage of the structure of M. The following discussion and algorithm rectifies this short-coming.

Distinguisher bounds. A *distinguisher bound* of $B_{\mathsf{Dist}} \in \mathbb{N}$ 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., $\Lambda^*(q_1^*, w_i) \neq \Lambda(q_2^*, w_i)$. Intuitively, a small distinguisher bound implies that each state is "locally" different, i.e., can be distinguished from others using small inputs.

The following theorem shows that a state bound implies a comparable distinguisher bound. In practice, distinguishers are much smaller than the bound implied by the state bound. For example, for the media-player, the number of states is large, but only distinguishers of length 1 are required.

Theorem 5. *A state bound of* k *implies a distinguisher bound of* $k-1$.

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 4). 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.

Remark 1. 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 8 and 9; and i in line 4).

Theorem 6. *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* $\mathsf{M}(\mathsf{cex}) \neq \mathsf{M}^*(\mathsf{cex})$*. Further, it performs at most* $|Q| \cdot |\tilde{\Sigma}_i|^{B_{\mathsf{Dist}}+1}$ *membership queries.*

Proof. The complexity bound and (b) are easy to show. For part (a), assume towards a contradiction that Algorithm 2 returns Correct, but there exists an

Algorithm 2 Equivalence oracle with distinguisher bound

```
Input: Mealy machine $\mathsf{M} = \langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \Lambda \rangle$
Input: Distinguisher bound $B_{\mathsf{Dist}}$
Input: Membership oracle $\mathsf{SMOracle}$
Output: $\mathsf{Correct}$ if $\mathsf{M} = \mathsf{M}^*$, and $\mathsf{cex} \in \tilde{\Sigma}_i^*$ such that $\mathsf{M}(\mathsf{cex}) \neq \mathsf{M}^*(\mathsf{cex})$ otherwise
 1: for all $q \in Q$ do $R(q) \leftarrow w_i \mid \delta(q_\iota, w_i) = q$ such that length of $w_i$ is minimal
 2: for all $q \in Q, i \in \tilde{\Sigma}_i$ do
 3:     $w_i \leftarrow R(q) \cdot i$
 4:     if $\Lambda(q,i)$ and last symbol of $\mathsf{SMOracle}(w_i \cdot i)$ differ then return $R(q) \cdot i$
 5:     $q' \leftarrow \delta(q,i)$; $w_i' \leftarrow R(q')$
 6:     $\mathsf{suffix} \leftarrow \mathsf{check}(w_i, w_i')$
 7:     if $\mathsf{suffix} \neq \mathsf{Correct}$ then
 8:         if $\mathsf{M}(R(q) \cdot i \cdot \mathsf{suffix}) \neq \mathsf{SMOracle}(R(q) \cdot i \cdot \mathsf{suffix})$ return $R(q) \cdot i \cdot \mathsf{suffix}$
 9:         else return $R(q') \cdot \mathsf{suffix}$
10: return $\mathsf{Correct}$
11: function $\mathsf{check}(w_i,\ w_i')$
12:     for all $\mathsf{suffix} \in \tilde{\Sigma}_i^{\leq B_{\mathsf{Dist}}}$ do
13:         $w_o \leftarrow \mathsf{SMOracle}(w_i \cdot \mathsf{suffix})$; $w_o' \leftarrow \mathsf{SMOracle}(w_i' \cdot \mathsf{suffix})$
14:         if the last $|\mathsf{suffix}|$ symbols of $w_o$ and $w_o'$ differ then return $\mathsf{suffix}$
15:     return $\mathsf{Correct}$
```

input word $\mathsf{cex} = i_0 i_1 \ldots i_n$ such that $\mathsf{M}(\mathsf{cex}) \neq \mathsf{M}^*(\mathsf{cex})$. Let $\mathsf{pre}(k)$ and $\mathsf{suf}(k)$ be the prefix of length k and suffix of length $n+1-k$ of cex.

Let $q_0 \rightarrow^{i_0}_{o_0} q_1 \ldots q_n \rightarrow^{i_n}_{o_n} q_{n+1}$ be the run of M on cex. Define the function $P(k)$ for $0 \leq k \leq n+1$ as follows: $P(k) = o_0 \ldots o_{k-1} \cdot \Lambda^*(\delta^*(q_\iota^*, R(q_k))), \mathsf{suf}(k))$. The value of $P(k)$ is given by the first k symbols of $o_0 \ldots o_n$ and concatenated with the last $n+1-k$ symbols of $\mathsf{M}^*(R(q_k) \cdot \mathsf{suf}(k))$. Informally, we run $\mathsf{pre}(k)$ on M and $\mathsf{suf}(k)$ of M^*, with $R(q_k)$ acting as the link between the two machines.

As $R(q_\iota) = \epsilon$, we can derive that $P(0) = \mathsf{M}^*(\mathsf{cex})$ and $P(n+1) = \mathsf{M}(\mathsf{cex})$. Since, by assumption, $P(0) \neq P(n+1)$, we have $\exists k : P(k) \neq P(k+1)$. Now, denoting $\delta^*(q_\iota^*, R(q_k))$ as q_k^* and $\delta^*(q_\iota^*, R(q_{k+1}))$ as q_{k+1}^*, we have

$$o_1 \ldots o_{k-1} \cdot \Lambda^*(q_k^*, \mathsf{suf}(k)) \neq o_1 \ldots o_{k-1} o_k \cdot \Lambda^*(q_{k+1}^*, \mathsf{suf}(k+1))$$
$$\Lambda^*(q_k^*, \mathsf{suf}(k)) \neq o_k \cdot \Lambda^*(q_{k+1}^*, \mathsf{suf}(k+1))$$
$$\Lambda^*(q_k^*, i_k \cdot \mathsf{suf}(k+1)) \neq \Lambda(q_k, i_k) \cdot \Lambda^*(q_{k+1}^*, \mathsf{suf}(k+1))$$

The first symbol on the left is $\Lambda^*(\delta^*(q_\iota^*, R(q_k)), i_k)$ and on the right is $\Lambda(q_k, i_k)$. If these were not equal, on line 4, the algorithm would have returned a counterexample—giving us a contradiction. On the other hand, if they were equal, removing the first symbol on both sides leaves us with:

$$\Lambda^*(\delta^*(q_k^*, i_k), \mathsf{suf}(k+1)) \neq \Lambda^*(q_{k+1}^*, \mathsf{suf}(k+1))$$

Hence, $\delta^*(q_k^*, i_k)$ and q_{k+1}^* are distinguishable in M^*. Therefore, there exists a suffix of length at most B_{Dist} such that $\Lambda^*(\delta^*(q_k^*, i_k), \mathsf{suffix}) \neq \Lambda^*(q_{k+1}^*, \mathsf{suffix})$. However, by definition of q_k^* and q_{k+1}^*, the left and right hand sides are equal to $\mathsf{SMOracle}(R(q_k) \cdot i_k \cdot \mathsf{suffix})$ and $\mathsf{SMOracle}(R(\delta(q_k, i_k)) \cdot \mathsf{suffix})$, respectively. Hence, when check was called at line 6 with the state q_k and input i_k, a counterexample would have been returned—this leads to a contradiction again. $\square$

Algorithm 2 and Theorem 5 give us improved complexity for state bounds.

Corollary 1. *Assuming a state bound B_{State}, equivalence checking for a Mealy machine can be implemented using at most $|Q| \cdot |\tilde{\Sigma_i}|^{B_{\mathsf{State}}}$ membership queries.*

Several optimizations can reduce the number of membership queries:

(a) *Quiescence transitions.* Transitions with input wait and output quiet need not be checked at line 6; 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.

4.4 Putting it all together

Starting with an asynchronous interface I and a membership oracle MOracle[I], using **Assumption 1** and **Assumption 3** we defined the synchronous closure I_s and SMOracle[I_s]. Given a state or a distinguisher bound on the typestate of I_s (using **Assumption 2**), we constructed an equivalence oracle EOracle for I_s. Oracles SMOracle[I_s] and EOracle can then be used to learn Mealy machine M that models the typestate of I_s. This Mealy machine gives us the asynchronous 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.

Theorem 7. *Given a deterministic interface* I *with observable quiescence, suppose there exists an interface automaton* A *modeling the typestate of* I *with fewer than B_{State} states with distinguishers of length at most B_{Dist}. The above procedure learns* A *with at most $O(|\Sigma_i| \cdot B_{\mathsf{State}}^2 + B_{\mathsf{State}} \cdot |\Sigma_i|^{2B_{\mathsf{Dist}}-1})$ membership queries.*

5 Learning Interfaces in the Android Framework

We discuss the implementation of the membership oracle via testing for the Android framework. The main challenges we face are the following: (a) The *environment*, i.e., the state of the device and external events that may influence an application, is inherently non-deterministic; however, any non-determinism is a violation of **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 implementation of the protocol we are learning may not be a regular language. Note that this is a violation of **Assumption 2**.

We detail these aspects and the solutions we adopted to overcome them. Due to Theorem 1, we cannot effectively learn an asynchronous interface without making assumptions. If the framework does not respect these assumptions, the method is neither sound nor complete. However, experimental evaluation showed that our approach works well in practice, despite the potential unsoundness. The resulting typestates were manually confirmed to be correct, and are useful: they uncovered surprising behavior in several Android framework classes.

Environment non-determinism for callins. While our algorithm is deterministic, the execution of both callins and callback in Android can be influenced by the external environment. For example, in the class `SQLiteOpenHelper`, the `getReadableDatabase()` may either trigger a $\overline{\texttt{onCreate()}}$ 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.

We resolve this non-determinism by manually modeling how the environment may affect the callbacks. In the `SQLiteOpenHelper` example, we manually 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).

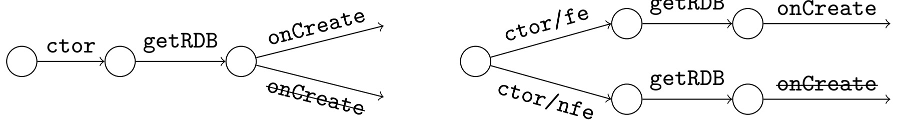


Fig. 2: Eliminating non-determinism in `SQLiteOpenHelper`

Timing non-determinism for callbacks. In our approach we assume *bounded asynchrony* 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, assuming that quiescence is reached when this time is elapsed. Hence, bounded asynchrony imposes an upper bound $\mathsf{t}_{\mathsf{max}}$ on the time we wait for an output. However, Android does not provide any worst case execution time on the execution of asynchronous operations. The membership query algorithm also assumes the existence of a minimum time $\mathsf{t}_{\mathsf{min}}$ before a callback occurs. This ensures that if we issue a membership query with two consecutive callins (so, without a wait input in between), we can execute the second callin assuming that no output was still generated from the first callin.

Consider the example of MediaPlayer described in Section 2. The membership query `setDataSource(URL)` $\cdot$ wait $\cdot$ `prepareAsync()` $\cdot$ wait may not return the final $\overline{\texttt{onPrepared}}$ callback as expected if the upper bound $\mathsf{t}_{\mathsf{max}}$ is violated, i.e., if the callback does not arrive even after timeout. On the other hand, while testing, it is possible that the `prepareAsync()` $\cdot$ `start()`, which is expected to return an error, might not return an error if the lower bound $\mathsf{t}_{\mathsf{min}}$ is violated. To avoid such issues we try to reliably control the execution environment and parameters to ensure that callbacks occurred between $\mathsf{t}_{\mathsf{min}}$ and $\mathsf{t}_{\mathsf{max}}$. For the MediaPlayer case, we needed to pick the right media source file.

Choice of callins parameters. The testing-based membership query algorithm must also provide the concrete parameters to execute a callin. We cannot explore all the possible parameters' values. However, in the practical cases we usually do not need to exercise all the parameters of callin, since they do not change with the typestate automaton. In the cases where parameters have an effect on the typestate automaton, invoking the same callin on two different parameters end up in a different state of the automaton, the same strategy described for the environment non-determinism for callins (splitting input actions) is used.

Protocol is not a regular language. An intrinsic limitation of L* is that it can only learn regular languages. However, some classes expose protocols that are not regular. Common cases include situations where a "request" callin may be called an unbounded number of times, and a "response" callback is called exactly the same number of times. For example, in the SpellCheckerSession class, callin `getSuggestion()` and callback $\overline{\texttt{onGetSuggestions}}$ follow this pattern.

However, even in such cases, it is useful to build a regular approximation of the typestate. For example, restricting the typestate to behaviours 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 approximations of the infinite typestate. Using learning purposes with L*, we can restrict the learning to a regular subset of behaviours.

6 Empirical Evaluation

Implementation. We implemented the learning algorithm described in the paper, along with the membership oracle for the Android framework, in a tool called Starling. Starling is implemented in Java and can be run on both emulators, as well as physical devices (our experiments were run on real hardware).

Goal. We perform experiments to empirically evaluate the following question:

Does our technique learn typestates accurately and efficiently?

We need to evaluate accuracy of our technique due to two potential sources of unsoundness: our implementation of membership oracle might be unsound, and our distinguisher bound may be too low.

Methodology. For the evaluation, we sampled 10 classes from the Android framework and ran Starling on it. For each class, the relevant callins and callbacks were identified manually, and a test harness was written connecting inputs and outputs from the L* algorithm to code to be executed on the class. Each of these test harnesses are $100 - 200$ lines of Java consisting mostly of boiler-plate code. In addition, for some examples, we need to provide manually a learning purpose, as explained in Section 5.

To evaluate accuracy, we use two approaches. First, for classes whose documentation contains a picture or a description of what effectively is an asynchronous typestate, we compare our result to the documentation. Second, for all the other classes we perform manual code inspection and run test apps to evaluate correctness of the produced asynchronous typestates.

To evaluate efficiency, we measure the overall time taken to produce the asynchronous interface, as well as measures that characterize the performance of L* and of our algorithm for implementing the equivalence oracle. These include the number of membership queries, the number of equivalence queries, and the number of membership queries for implementing each equivalence query. 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 the length of the media file chosen during testing.

Class name	# states	Time (s)	# MQ	# EQ	# MQ per EQ
AsyncTask	5	49	372 (94)	1	356 (0)
CountDownTimer	3	134	232 (61)	1	224 (0)
FileObserver	6	104	743 (189)	2	351 (8)
MediaCodec	8	371	1354 (871)	1	973 (482)
MediaPlayer	10 (19)	4262	13553 (2372)	5	2545 (384)
MediaRecorder	8	248	1512 (721)	1	1280 (545)
MediaScannerConnection	4	200	403 (161)	2	163 (57)
SpeechRecognizer	7	3460	1968 (293)	3	646 (35)
SpellCheckerSession	6	133	798 (213)	4	374 (8)
SQLiteOpenHelper	8	43	1364 (228)	2	665 (6)

Table 1: Experimental results. We report the number of states of the typestate, the time to learn, number of membership (# MQ) and equivalence (# EQ) queries, and average membership (# MQ per EQ) queries for implementing each equivalence queries

Results and Discussion. Table 1 presents results of our empirical evaluation. The table shows that the learning algorithm runs reasonably fast: several of the benchmarks run within a few minutes. The largest one takes 71 minutes, still being within the realm of nightly testing. The numbers for membership queries are reported as $X(Y)$—X is the number of membership queries called by the L* 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 automaton matched the actual behaviour of the class. Surprisingly, in 2 cases, we found discrepancies between the learned behaviour and documented behaviour in certain corner cases. In such cases, we carefully examined the discrepancy, by framework source examination and manually writing test applications, and found that the learned typestate was correct, and the documentation was faulty. In 3 other cases, we believe the implemented behaviour is not the intended behaviour, i.e., these are bugs in the Android implementation. Further, in 1 additional case, we found that the typestate learned on different versions of the Android framework were different. These differences were again confirmed manually. These results suggest that typestate learning can serve a valuable role, serving both as documentation generation and validation of changes to framework code.

Detailed results. Of 10 benchmarks, we pick 2 and explain them in more detail. The remaining benchmarks are in the appendix.

SpellCheckerSession This class provides an interface to request spelling suggestions. An app can request suggestions for a particular sentence (via `getSentenceSuggestions()`) which will be delivered via callback (`onGetSentenceSuggestions()`) unless a `cancel()` or `close()` is called. The full interface is non-regular; the number of callbacks with results is equal to the number of request callins. We therefore used the learning purpose approach to restrict learning to the fragment with at most one call to `getSentenceSuggestions()`. The resulting asynchronous typestate showed that calling `cancel()` will not prevent receiving callback with results. This might hurt applications that rely on the callback not arriving after `cancel()`.

MediaPlayer. This is the class from the introductory example in Section 2. There are two interesting aspects about the run of Starling which differ from the existing documentation:

- The learned typestate has the `pause()` callin is enabled in the "paused" state. Though undocumented, we do not believe this behaviour is a bug.
- The class calls back $\overline{\texttt{onPrepared()}}$ even after the synchronous `prepare()` callin — this callback is unneeded as it is not necessary to wait for it after calling prepare. However, this unnecessary callback leads to `Starling` generating many more states than necessary; every state after `prepare()` is called has an "equivalent" duplicate: one where the unnecessary $\overline{\texttt{onPrepared()}}$ is pending, and one where it is not. In the table, the number of states reported in brackets is the number including the duplicate states. These states can be eliminated automatically; however, this is not implemented currently.

7 Related Work and Future Work

Works which automatically synthesize specifications of the valid sequences of method calls (e.g. [3,24,4,14]) typically ignore the asynchronous callbacks.

Static analysis has been successfully used to infer typestates specifications (importantly, without callbacks) [3,16,24]. The work in [3] infers interfaces 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.

Inferring the interface using execution traces of client programs using framework is another common approach [4,28,2,10,13,26,29,21]. 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 framework has recently gained a lot of attention (e.g. [6,8,9,19]). 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 [6,8,15,27,12] or dynamically detecting race conditions [20,17,7]. These approaches manually hard-code the behavior of the framework to increase the precision of the analysis. The asynchronous 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]. The optimizations we use include the counterexample suffix analysis from [22] and the optimizations for prefix-closed languages from [18].

Future Work. This paper enables several new research directions. First, we plan to work on inferring pushdown typestates for classes that can submit multiple requests and get back the corresponding number of results. Second, we will investigate mining parameters of callins from instrumented trace from real user interactions. Third, we will develop methods for using asynchronous typestates in Android verification and program repair.

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. Pavol Bielik, Veselin Raychev, and Martin T. Vechev. Scalable race detection for android applications. In *OOPSLA*, pages 332–348, 2015.
8. Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Selective control-flow abstraction via jumping. pages 163–182, 2015.
9. 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.
10. 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.
11. L. de Alfaro and T. Henzinger. Interface automata. In *FSE*, pages 109–120, 2001.
12. Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. pages 576–587, 2014.
13. 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.
14. 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.
15. Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow analysis of Android applications in DroidSafe. 2015.
16. 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.
17. 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.
18. H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In *CAV*, pages 315–327, 2003.

19. 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.
20. Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for android applications. In *PLDI 2014*, page 34, 2014.
21. Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.
22. Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.
23. J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.
24. 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.
25. R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.
26. 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.
27. 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.
28. 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.
29. 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.

A Detailed Descriptions of Benchmarks

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

In order to test this class, the `get()` input had to be dismissed (i.e., we did not include `get()` in our learning purpose, see Section 5) because its results were non-deterministic when called directly after the $\overline{\texttt{onPostExecute()}}$ callback. Our experiment revealed an unexpected edge-case: if `execute()` is after `cancel()` but before the $\overline{\texttt{onCancelled()}}$ callback is received, it will not throw an exception but will never cause the asynchronous task to be run.

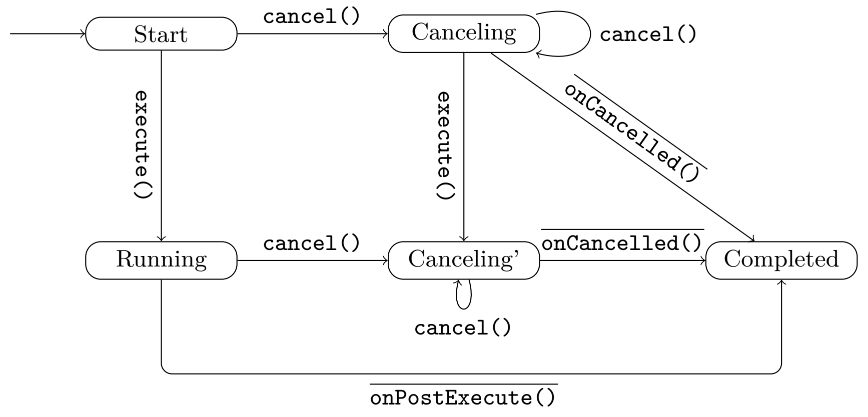


Fig. 3: Learned typestate of the AsyncTask class

SpeechRecognizer The SpeechRecognizer is a simple interface to the speech-to-text services provided in the Android framework, and is necessarily asynchronous due to the time spent waiting for speech and communicating with remote speech-to-text services over the network. An application constructs it with a listener for various error and result callbacks, and then calls `startListening()` to set off the process. The SpeechRecognizer calls $\overline{\texttt{onReadyForSpeech()}}$ and then either returns the recognized speech results with $\overline{\texttt{onResults()}}$ or one of several error cases with $\overline{\texttt{onError()}}$.

Like the SQLiteOpenHelper, this class provides another case of "Environment non-determinism" because the particular callback that signals the end of the speech session—either an $\overline{\texttt{onResults()}}$ or an $\overline{\texttt{onError()}}$—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 $\overline{\texttt{onResults()}}$ or $\overline{\texttt{onError()}}$

would be the same and merged the two callbacks into a single $\overline{\texttt{onFinished()}}$ symbol. The determinism observed in the resulting typestate confirmed this assumption.

Our results revealed two interesting corner cases for the ordering of inputs. First, if an app calls `cancel()` between calling `startListening()` and receiving the $\overline{\texttt{onReadyForSpeech()}}$ 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.

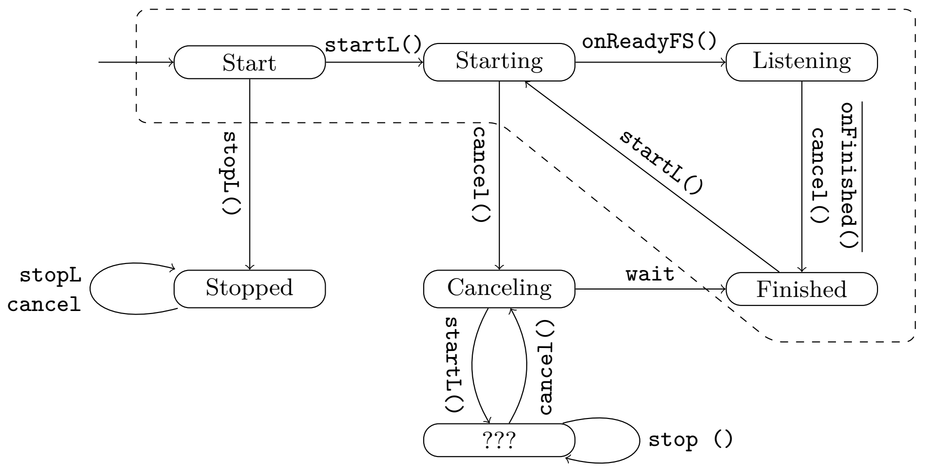


Fig. 4: Learned typestate of the SpeechRecognizer class. The behaviour inside the dotted line is "normal" behaviour, while everything outside is unusual or "exceptional" behaviour.

While we performed all other tests on an Android 5.1.1 system, the SpeechRecognizer's callback behavior was so buggy—callbacks would repeat themselves several times, sometimes interleaving with later callbacks—that we could not learn a consistent automaton. We therefore performed this experiment on an Android 6.0.1 system, where this behavior was observed to be fixed.

CountDownTimer The CountDownTimer is an asynchronous timer that is initialized with a period of time and then performs the $\overline{\texttt{onFinish()}}$ callback when that time is up, unless a `cancel()` method is called in the meantime. This class is not a regular language; calling `start()` multiple times will result in that same number of $\overline{\texttt{onFinish()}}$ callbacks. In order to learn a regular language subset, we used the learning purpose approach to limit the number of `start()` inputs in a valid query to one.

FileObserver The FileObserver is a listener class that can track changes to a file or directory, making callbacks as they occur. The class itself has only two input methods—`startWatching()` and `stopWatching()`—so for our experiment we considered the environmental actions of modifying (and possibly creating) and deleting the watched file as additional inputs. An $\overline{\texttt{onEvent()}}$ callback returns with various event types; the ones we track are `MODIFY` and `DELETE_SELF`. One interesting bit of behavior we correctly observe (which is in the documentation for the class) is that if the file is deleted while being watched, no further callbacks will be made even if it is recreated.

MediaPlayer This is the class from the introductory example in Section 2. There are two interesting aspects about the run of Starling on MediaPlayer and the learned typestate:

- The learned typestate has the `pause()` callin is enabled in the "paused" state. Though undocumented, we do not believe this behaviour is a bug.
- The class returns the $\overline{\texttt{onPrepared()}}$ callback even after the synchronous `prepare()` callin — this callback is unnecessary as it is not necessary to wait for it after calling prepare. However, this unnecessary callback leads to `Starling` generating many more states than necessary; every state after `prepare()` is called has an "equivalent" duplicate there: one where the unnecessary $\overline{\texttt{onPrepared()}}$ is pending, and one where it is not. In the table, the number of states reported in brackets is the number including the duplicate states. We believe that these states can be eliminated automatically; however, this is not implemented currently.

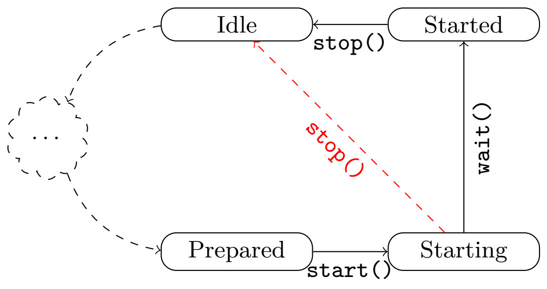
MediaCodec The MediaCodec class provides an interface for decoding encoded media files (audio and video) in a continuous manner, i.e., audio and video data is continuously passed into the interface, and the interface passes the results back as soon as they are ready. The interaction with this interface is done fully asynchronously: the interface calls the $\overline{\texttt{onInputBufferAvailable()}}$ callback whenever it is ready to accept input, and $\overline{\texttt{onOutputBufferAvailable()()}}$ callback when results are ready. The interaction is started and stopped using the `start()` and `stop()` callins; and further, the `flush()` callin invalidates all available buffers. The typestate of MediaCodec is described with a high-level diagram in the documentation: `https://developer.android.com/reference/android/media/MediaCodec.html`. Starling is able to learn the typestate of the interface exactly in this case, and the learned typestate matches the description in the documentation.

MediaRecorder The MediaRecorder class provides a single interface to recording audio and video various sources such as the microphone, camera, and voice calls. The typestate for this class is present in the documentation at `https://developer.android.com/reference/android/media/MediaRecorder.html`. Of our benchmarks, the MediaRecorder is particularly unique, as the interface has no callbacks related to starting or stopping, i.e., at looks like a standard (not event-driven asynchronous) interface. The callbacks which do exist are for receiving updates or exceptions during recording. However, internally, the implementation of the class is asynchronous. This can be seen as follows: calling `start()` followed immediately by `stop()` in the "prepared" state throws an exception. However, calling `start()`, waiting for a short time (wait), followed by `stop()` is not an error. Note that the interface does not perform any callback on wait (i.e., we get quiet). This behaviour is not documented, and also, completely unexpected from the client programmer point of view. Other than this behaviour, the typestate is equivalent to the one shown in the documentation.

The unexpected behaviour can be fixed in one of two ways:

(a) Expose the asynchronous nature of the class, i.e., add a callback `onStart()`, and make calling `stop()` legal only after `onStart()` is called back.

(b) Modify the implementation of the `start()` callin to wait and only return after the task is truly completed, i.e., do not let the asynchronous nature of the implementation leak to the client.



Part of the typestate for the MediaRecorder. If the class was truly synchronous, the states "Starting" and "Started" would be the same, i.e., `stop()` would be enabled immediately after `start()`. Starling was able to discover this inconsistency automatically.

Fig. 5: MediaRecorder

MediaScannerConnection The MediaScannerConnection class implements an interface that applications may use to inform the standard android media services about new media files that the application has created. For example, an application may download an image, and then scan it using MediaScannerConnection: after the scan, the new image will be available through the standard image handling services (for example, the media gallery).

The MediaScannerConnection interface itself is simple with 3 relevant callins: `connect()`, `disconnect()`, and `scanFile()`. Further, it has 1 callback important to the typestate: `onMediaScannerConnected()`. The protocol for using the interface is as follows: (a) `connect()` is called to establish connection with the media scanner service; (b) the `onMediaScannerConnected()` callback is called by the interface to inform the client that the connection has been established; (c) the client may scan any number of files through the `scanFile()` callin; and (d) the client finally calls `disconnect()` to disconnect from the media scanner service. Starling was able to learn this typestate exactly. The only point of note is that the $\overline{\texttt{onMediaScannerConnected()}}$ callback will be called even if `disconnect()` is called immediately after `connect()` before the callback; however, the interface is in the disconnected state.

SpellCheckerSession This class provides a high-level interface for apps to request spelling suggestions on text fragments. An app can request suggestions for a particular sentence (via `getSentenceSuggestions()`) which will be delivered via callback ($\overline{\texttt{onGetSentenceSuggestions()}}$) unless a `cancel()` or `close()` is called. The interface is asynchronous due to the computational intensity of the text-processing involved.

The full interface is non-regular; the number of callbacks with results is equal to the number of request callins. We therefore used the learning purpose approach to restrict learning to the fragment with at most one call to `getSentenceSuggestions()`. For this fragment, the resulting asynchronous typestate showed that calling `cancel()` will not prevent receiving callback with results. This might hurt applications that rely on a guarantee that the callback does not arrive after `cancel()`.

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 $\overline{\texttt{onConfigure()}}$ is called, followed by an $\overline{\texttt{onCreate()}}$ if the database didn't exist yet or an $\overline{\texttt{onUpgrade()}}$ if the database had a lower version number than was passed to the SQLiteOpenHelper constructor, all followed finally by an $\overline{\texttt{onOpen()}}$ when the database is ready for reading. The database can then be closed with a `close()`.

Our experiment observed the callbacks received 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.

B Proofs for theorems in Section 4

Proof of Thm 3. Let $\mathsf{M} = \langle Q, q_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta, \Lambda \rangle$ and $\mathsf{M}' = \langle Q', q'_\iota, \tilde{\Sigma}_i, \tilde{\Sigma}_o, \delta', \Lambda' \rangle$. Let $\overline{\Lambda} : Q \cup Q' \times \tilde{\Sigma}_i \to \tilde{\Sigma}_o$ (resp. $\overline{\delta} : Q \cup Q' \times \tilde{\Sigma}_i \to Q \cup Q'$) be a unified output (resp. transition) function that applies either Λ or Λ' (resp. δ or δ') depending on if the first input is in Q or Q'.

We first define a sequence of equivalence relations $\equiv_m$ on $Q \cup Q'$ for each $m > 0$. For each m, we say $q_1 \equiv_m q_2$ if for all input words w_i of length at most m, we have that $\overline{\Lambda}(q_1, w_i) = \overline{\Lambda}(q_2, w_i)$. Note that each $\equiv_{m+1}$ is a refinement of $\equiv_m$. As we have that $\mathsf{M}(w_i) \neq \mathsf{M}'(w_i)$, we get $q_\iota \not\equiv_m q'_\iota$ for some m. Further, we have that $\equiv_1$ is not universal, i.e., $\exists q_1, q_2 : q_1 \not\equiv_1 q_2$; otherwise, we will have $\overline{\Lambda}(q_1, w_i) = \overline{\Lambda}(q_2, w_i)$ for every w_i, q_1, and q_2, contradicting the premise of the theorem statement.

Below, we show that $q_1 \not\equiv_{m+1} q_2$ if and only if $q_1 \not\equiv_m q_2$ or $\exists i \in \tilde{\Sigma}_i : \overline{\delta}(q_1, i) \not\equiv_m \overline{\delta}(q_2, i)$. Hence, each $\equiv_{m+1}$ is uniquely determined by $\equiv_m$ and $\overline{\delta}$ giving us that $\equiv_m = \equiv_{m+1} \implies \equiv_{m+1} = \equiv_{m+2}$. Further, since all $\equiv_m$ are successive refinements defined on a finite set, we can only have $\equiv_{m+1} \neq \equiv_m$ for a finite number of m. Let m^* be the least m such that $\equiv_{m^*+1} \neq \equiv_{m^*}$. By the two statements above, we get that

$$\equiv_1 \neq \equiv_2 \neq \ldots \neq \equiv_{m^*} = \equiv_{m^*+1} = \equiv_{m^*+2} = \ldots$$

Hence, for every pair of states q_1 and q_2, if $\exists w_i : \overline{\Lambda}(q_1, w_i) \neq \overline{\Lambda}(q_2, w_i)$, then there exists a word w'_i of length at most m^* such that $\overline{\Lambda}(q_1, w'_i) \neq \overline{\Lambda}(q_2, w'_i)$.

Now, by the non-universality of $\equiv_1$ and the fact that each $\equiv_m$ is a strict refinement of the previous for $m \leq m^*$, we have that m^* can at most be $|Q| + |Q'| - 1$. This gives us the required result.

Now, to show that $q_1 \not\equiv_{m+1} q_2$ if and only if $q_1 \not\equiv_m q_2$ or $\exists i \in \tilde{\Sigma}_i : \overline{\delta}(q_1, i) \not\equiv_m \overline{\delta}(q_2, i)$.

- Assume $q_1 \not\equiv_m q_2$ or $\exists i \in \tilde{\Sigma}_i : \overline{\delta}(q_1, i) \not\equiv_m \overline{\delta}(q_2, i)$. If $q_1 \not\equiv_m q_2$, we have that $q_1 \not\equiv_{m+1} q_2$ as each $\equiv_{m+1}$ is a refinement of $\equiv_m$. Otherwise, assume $\exists i \in \tilde{\Sigma}_i : \overline{\delta}(q_1, i) \not\equiv_m \overline{\delta}(q_2, i)$. Now, there exists a w_i such that $\overline{\Lambda}(\overline{\delta}(q_1, i), w_i) \neq \overline{\Lambda}(\overline{\delta}(q_2, i), w_i)$. It is easy to show that $\overline{\Lambda}(q_1, i \cdot w_i)$ and $\overline{\Lambda}(q_2, i \cdot w_i)$. Further, $|i \cdot w_i|$ is at most $m + 1$ giving us $q_1 \not\equiv_{m+1} q_2$.
- Assume $q_1 \not\equiv_{m+1} q_2$. There exists a word w_i of length at most $m + 1$ such that $\overline{\Lambda}(q_1, w_i) \neq \overline{\Lambda}(q_2, w_i)$. If $|w_i| \leq m$, we get that $q_1 \not\equiv_m q_2$.
  Otherwise, let $w_i = i \cdot w'_i$. If $\overline{\Lambda}(q_1, i) \neq \overline{\Lambda}(q_2, i)$, we have that $q_1 \not\equiv_1 q_2$, and hence, $q_1 \not\equiv_m q_2$.
  In the remaining case, we have that $\overline{\Lambda}(q_1, i) = \overline{\Lambda}(q_2, i)$ and $\overline{\Lambda}(q_1, i \cdot w'_i) \neq \overline{\Lambda}(q_2, i \cdot w'_i)$. From these, it is easy to show that $\overline{\Lambda}(\overline{\delta}(q_1, i), w'_i) \neq \overline{\Lambda}(\overline{\delta}(q_2, i), w'_i)$. This gives us that $\overline{\delta}(q_1, i) \not\equiv_m \overline{\delta}(q_2, i)$.

This completes the proof. ◻

Proof of Theorem 5. Let M^* have $|Q^*| \leq B_{\mathsf{State}}$ states. Define a series of equivalence functions $\equiv_m$ on Q such that $q_1 \equiv_m q_2$ if and only if q_1 can be distinguished from q_2 by words of length $> m$. The remainder of the proof is exactly similar to the proof of Theorem 3. Again, the final bound obtained on the length of distinguishers is one less than the size of the domain of the equivalence relations. Here, the bound is $|Q^*| - 1 \leq k - 1$, giving us the theorem. $\square$

Proof of Theorem 6 (continued). We proved part (a) in the main part of the paper. For part (b), assume that Algorithm 2 return a counterexample. If the counterexample is returned at line 4, then it is easy to see that the last output symbol of $\mathsf{M}(R(q) \cdot i)$ and $\mathsf{M}^*(R(q) \cdot i)$ differ. On the other hand, if the counterexample is returned at lines 8 and 9, we have that $\mathsf{M}^*(R(q) \cdot i \cdot \mathsf{suffix})$ and $\mathsf{M}^*(R(q') \cdot \mathsf{suffix})$ differ in the last $|\mathsf{suffix}|$ positions. However, $\mathsf{M}(R(q) \cdot i \cdot \mathsf{suffix})$ and $\mathsf{M}(R(q') \cdot \mathsf{suffix})$ are equal in the last $|\mathsf{suffix}|$ positions. Hence, we cannot have both $\mathsf{M}(R(q) \cdot i \cdot \mathsf{suffix}) = \mathsf{M}^*(R(q) \cdot i \cdot \mathsf{suffix})$ and $\mathsf{M}(R(q') \cdot \mathsf{suffix}) = \mathsf{M}^*(R(q') \cdot \mathsf{suffix})$. This implies at least one of $R(q) \cdot i \cdot \mathsf{suffix}$ and $R(q') \cdot \mathsf{suffix}$ is a counterexample, which is returned. $\square$