Automated Testing of Event-Driven Applications
Casper Svenning Jensen
PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Automated Testing of
Event-Driven Applications

A Dissertation
Presented to the Faculty of Science and Technology of Aarhus University
in Partial Fulfillment of the Requirements for the PhD Degree

by
Casper Svenning Jensen
March 26, 2015
Abstract

Event-driven applications, such as, web applications and Android mobile applications, may be tested by selecting an interesting input (i.e. a sequence of events), and deciding if a failure occurs when the selected input is applied to the event-driven application under test. Automated testing promises to reduce the workload for developers by automatically selecting interesting inputs and detect failures. However, it is non-trivial to conduct automated testing of event-driven applications because of, for example, infinite input spaces and the absence of specifications of correct application behavior.

In this PhD dissertation, we identify a number of specific challenges when conducting automated testing of event-driven applications, and we present novel techniques for solving these challenges.

First, we present an algorithm for stateless model-checking of event-driven applications with partial-order reduction, and we show how this algorithm may be used to systematically test web applications for timing related failures. Next, we present an algorithm for generating inputs to event-driven applications in a targeted manner, combining existing techniques using UI models and concolic testing in a novel way. Finally, we show how server interface descriptions can be used to simplify the process of automated testing of web applications that depend on client-server communication, and we present a learning algorithm for inferring such server interface descriptions from concrete observations.

We implement tools for web applications and Android mobile applications using the above algorithms and techniques, and we experimentally evaluate the effectiveness of the proposed solutions on real-world applications. Based on our experiments, we conclude that our proposed solutions are useful when automatically testing event-driven applications, and that our proposed solutions pushes the state-of-the-art within this area.
Resumé

Det er muligt at teste en event-dreven applikation ved at vælge et input til applikationen, såsom en event sekvens, og observere om det valgte input fører til en fejl i den event-dreven applikation. Automatiseret testning lover at reducere arbejdsbyrden for udviklere ved automatisk at vælge interessante input og identificere hvis et input fører til en fejl. Automatiseret testning for event-drevene systemer er specielt besværligt på grund af for eksempel uendelige event sekvenser og manglende specifikationer for en applikations korrektethed.

Denne Ph.d. afhandling identificerer en række praktiske udfordringer forbundet med automatiseret testning af event-drevene applikationer, samt præsenterer en række nye teknikker som løser nogle af disse udfordringer.

Vi præsenterer en algoritme for stateless model-checking af event-drevene applikationer med partial-order reduction, og vi viser hvordan denne algoritme kan bruges til systematisk testning af web applikationer. Ydermere, så præsenterer vi en algoritme til at generere input ud fra et bestemt mål i en applikation, ved at kombinere eksisterende teknikker der benytter sig af UI modeller og concolic testing. Til slut, så viser vi hvordan beskrivelser af klient-server kommunikation kan bruges til at simplificere automatiseret testning af web applikation som er afhængige af klient-server kommunikation, og vi præsentere en algoritme som kan udlede sådanne beskrivelser baseret på konkrete observationer.

Vi implementere værktøjer for web applikationer og Android applikationer som basere sig på de overstående algoritmer og teknikker, og vi evaluere vores løsninger på en række applikationer. Baseret på vores evalueringer kan vi konkludere at vores løsninger kan benyttes til at forbedre og sætte nye standarder for automatiseret test af event-drevene systemer.
Acknowledgments

Thanks to the members of the Programming Languages group at Aarhus University for contributing to an interesting and enjoyable work environment. Special thanks to Esben Andreasen for countless of interesting discussions in our office, and for providing feedback to this PhD dissertation.

I thank my advisor Anders Møller for giving me the chance of following a PhD degree, and for support and encouragement throughout my studies.

I also thank Mukul R. Prasad and Martin Vechev for hosting me at Fujitsu Laboratories America and ETH Zurich, respectively. Both very enriching stays professionally and personally.

Finally, I would like to thank my family for always being supportive; my many friends in the Skype chat for a place to go and relax in the evenings; and my girlfriend, Yu Qian, for her support, patience, and love.

Casper Svenning Jensen,
Aarhus, March 26, 2015.
# Contents

Abstract i

Resumé iii

Acknowledgments v

Contents vii

## 1 Overview

### 1.1 Introduction

## 2 Event-Driven Applications

### 2.1 The Event Model

### 2.2 JavaScript Client-Side Web Applications

### 2.3 Android Mobile Applications

## 3 Automated Testing

### 3.1 Related Work for Event-Driven Applications

### 3.2 Background

### 3.3 Summary

## 4 Nondeterministic Scheduling

### 4.1 Motivation

### 4.2 Solution

### 4.3 Related Work

### 4.4 Evaluation

### 4.5 Summary

## 5 Targeted Input Generation

### 5.1 Motivation

### 5.2 Solution

### 5.3 Related Work

### 5.4 Evaluation
Part I

Overview
Chapter 1

Introduction

It is widely accepted in software engineering that *software testing* improves overall software quality. As evidence, we can observe that many development methodologies explicitly mention testing as part of their core tasks, such as, Extreme Programming\(^1\) Test Driven Development (TDD)\(^2\) Context-Driven Testing\(^3\) the Rational Unified Process, the Waterfall Model, and the V-Model.

We can classify the act of testing by the composition of the tested components, such as, unit testing, integration testing, and system testing; or by the tested properties, such as, security testing, smoke testing, acceptance testing, regression testing, functional testing, and non-functional testing. The focus of this PhD dissertation is not on why and what we test, but *how* we test. Specifically, how to automatically test software as oppose to manual testing. Here, we define manual testing as either manually running software or manually writing test cases to be run later (the latter is not to be confused with automated testing). In contrast, automated testing refers to a fully automated system for end-to-end testing of software, with no or minimal developer interaction.

As an example, automated testing is used in practice by the QuickCheck tools for Haskell\(^4\) and Erlang\(^5\). QuickCheck allows a developer to specify the input domain of a program and an acceptance function for checking the result of a program given an input. QuickCheck then automatically selects inputs from the input domain and checks the result of running a program with those inputs using the acceptance function, freeing the developer from the responsibility of manually selecting inputs and writing tests for each selected input.

However, it is still an open problem if practical and efficient automated testing is feasible for certain kinds of software, such as, *event-driven applications* (i.e. client-side JavaScript web applications and Android mobile applications). Intuitively, an event-driven application takes as input a sequence of

\(^1\)http://www.extremeprogramming.org/
\(^2\)http://c2.com/cgi/wiki?TestDrivenDevelopment
\(^3\)http://context-driven-testing.com/
\(^4\)https://www.haskell.org/haskellwiki/Introduction_to_QuickCheck1/
\(^5\)http://www.quviq.com/
events that are processed by the event-driven application. Each event may result in any number of event handlers being invoked. It is not possible to try all possible event sequences, since the space of possible event sequences may be infinite or infeasible large. Thus, automated testing must be able to select interesting inputs, which is a non-trivial task. Furthermore, automated testing must also be able to decide if a given input results in a test failure or success, a task that requires, for example, guidance from a developer (i.e. the acceptance function in QuickCheck), a formal specification, or heuristics to detect.

**Thesis Statement** In this PhD dissertation, we focus on the problem of automated testing of event-driven applications. Specifically, we identify a number of interesting challenges, propose a series of new techniques, and evaluate those techniques, all to improve the state-of-the-art of automated testing of event-driven applications.

**Contributions** The general problem of automated testing of event-driven applications is very broad and contains numerous challenges. In this dissertation we present the following contributions:

**The R$^4$ Project** We propose an algorithm for systematically exploring event sequences using stateless model checking with partial-order reduction. The algorithm is implemented in our tool R$^4$ for automated testing of timing related bugs in web applications. Furthermore, we propose an oracle that compares execution traces in order to automatically identify timing related failures.

**The Collider Project** We present a technique for generating input for event-driven applications that reaches a specific target (i.e. a program branch or statement) in a targeted manner. We do this by combining concolic testing and UI models. We implement this technique in our tool Collider for automated testing of Android mobile applications.

**The AIL Project** We propose server interface descriptions for describing the communication between the client-side and server-side of web applications. We extend an existing tool for automated testing of web applications with support for automated generation of server-side responses and we present a learning algorithm to help developers write the server interface descriptions.

Each of the above projects correspond to a dedicated chapter (Chapters 4–6, respectively) and a publication/manuscript (Chapters 8–10, respectively).
Reading Guide  This dissertation is divided into two parts. Part I presents the context and essential contributions of the published papers and manuscripts on which the dissertation is based, while Part II provides the published papers and manuscripts themselves. Part I is written to be stand-alone and can be read in sequential order without referring to Part II. However, certain technical details are omitted from Part I to increase readability.

It is assumed that the reader has a background equivalent to a BSc in computer science. Prior knowledge of event-driven applications, automated testing, and other techniques used in this dissertation is not assumed.

Part I is further divided as follows. Chapter 2 introduces event-driven applications and the challenges of testing such applications, Chapter 3 provides an overview of existing automated testing techniques, Chapter 4–6 detail the individual contributions presented as part of this dissertation, and Chapter 7 concludes on the presented work.

Published Papers & Manuscripts  The following published papers and manuscripts are included as part of this dissertation:


These papers and manuscripts are included in Part II using their original published versions with minor layout changes to better fit into this dissertation.
Chapter 2

Event-Driven Applications

In the following chapter, we present our model of a general event-driven application, denoted as the *event model*. The event model allows us to reason about central concepts of event-driven applications and will be reused throughout Part I. Furthermore, this chapter also encodes two concrete event-driven applications, JavaScript client-side web applications and Android mobile applications, using the presented event model to (1) show the expressiveness of the event model and (2) provide a primer on web applications and Android mobile applications.

2.1 The Event Model

Event-driven applications (also denoted event-based systems, asynchronous programs, event-driven programming\(^1\) or Event Collaboration\(^2\)) are applications or systems of applications that react to the world surrounding them by receiving and processing *events*. For example, JavaScript client-side web applications, Android mobile applications, and desktop GUI applications are usually event-driven.

The event model, illustrated in Figure 2.1 consists of (1) a set of enabled events, (2) a map of event handler registrations, (3) shared state, and (4) an event loop. The event loop selects an enabled event and *processes* it by removing the selected event from the set and all related event handlers. Each event handler is allowed to read and write from the shared state, add new events to the set of enabled events, and register or unregister event handlers. Events are processed in a non-deterministic order and one at a time, that is, in one thread of execution. Finally, external actors are able to add new events to the set of enabled events, which we denote *triggering* and event. For example, a click event is triggered when a user clicks a button in a GUI application and an load event is triggered when an image is loaded in a web application.

\(^1\)[http://c2.com/cgi/wiki?EventDrivenProgramming]
\(^2\)[http://www.martinfowler.com/eaaDev/EventCollaboration.html]
Some applications combine single-threaded event-driven applications with multi-threaded concurrent programs. For example, a traditional desktop GUI application may consist of an event-driven application handling the GUI using a single thread (the UI thread), while concurrency is used to run demanding tasks concurrently. In this dissertation, we chose to focus only on automated testing of event-driven applications, or subsets of applications that are event-driven. We leave the problem of automatically testing systems that combine both paradigms as an open problem.

Formally, such a model can be expressed using the transition system $M = \langle c_0, C, \delta \rangle$, where $c_0 \in C$ is the initial configuration, $C$ the set of possible configurations, and the transition function $\delta$ is a partial function $C \rightarrow C$. We define the set of possible configurations to be $C = S \times 2^E \times H$, where:

- $S$ is the set of possible shared states. A shared state is, for example, files or memory accessible by the event handlers.
- $2^E$ is the possible sets of enabled events, where $E$ is the set of events.
- $H$ is the set of event handler registration maps, relating events with event handlers that are invoked when an event is processed.

The transition function $\delta$ represents the combined effect on the configuration of the system when processing an event, depending on the current shared state, registered event handlers, and enabled events.

For simplicity, the event model assumes that the set of enabled events in a configuration $c$ contains all events that may be added by any external actor. In practice, an event-driven application may need to wait for an external actor to add an event, e.g. for I/O related events, or force the actor to add the event.
2.1. THE EVENT MODEL

2.1.1 Events

We further refine the set of events $E$ in our transition system. We say that $E \subseteq N \times P$, where, $N$ is the set of event names and $P$ the set of event parameters. An event name identifies the kind of an event while event parameters supply additional information associated with the concrete instance of an event. We denote events that have a non-empty list of parameters as parameterized events. As an example, an event caused by a mouse click in a GUI could have the event name \textit{click} with two parameters representing the $x$- and $y$-coordinates of the click.

The model distinguishes between event names and event parameters because some concrete instances of event-driven applications associate event handlers with event names, while allowing any event parameters to be used when the event is triggered and added to the set of enabled events. Later, we will show that selecting proper event names and event parameters are both important for automated testing but require different solutions. Recall the contributions of this PhD dissertation listed in Section 1. All of the listed collections of contributions are related to generating sequences of event names or selecting event parameters to fit an event sequence.

2.1.2 Event Handlers

An event handler registration map $h \in H$ is a partial function from event names to ordered lists of event handlers, $N \rightarrow \sigma^*$, where $\sigma$ is the set of event handlers. An event handler is an executable program that can mutate the shared state, set of enabled events, and registered event handlers of the event-driven application. When an event $\langle n, p \rangle$ is processed, where $n \in N$ and $p \in P$, the event-driven application will invoke each event handler, $\sigma \in h[n]$, with $p$ as an argument to $\sigma$. The transition function $\delta$ represents the combined effect of processing the registered event handlers.

2.1.3 Summary

The event model can be used to model a number of real-world event-driven applications, as shown in the remainder of this chapter. In general, any concrete instance of the model needs to provide an initial configuration (initial state, default event handlers, initial events), and reason about the domain of possible states, events, and assignments of event handlers. The configuration of an event-driven application depends on the sequence of events processed, and specifically the event names and event parameters of those events.
2.2 JavaScript Client-Side Web Applications

JavaScript client-side web applications, henceforth denoted as web applications, are generally made up of fragments written in HTML, CSS, and JavaScript that are fetched and interpreted by a web browser. HTML is a markup language for declaring the contents of a web page, with support for embedding other kinds of fragments, or declaring external resources; CSS is dedicated to styling of web pages; and JavaScript is used for scripting. A web application is fetched and executed by a web browser in a highly responsive manner, such that, for example, rendering, script execution, and handling of user interaction are interleaved. In the following, we describe how web applications and web browsers are both instances of event-driven applications.

2.2.1 Web Applications and the DOM Event Model

The core of a web application is the DOM (Document Object Model\footnote{https://dom.spec.whatwg.org/}), which is a representation of the current web page visible in a browser with an API for interacting with this web page and the web browser through JavaScript. A web page consists of a number of nodes structured in a tree structure, denoted the DOM tree. Figure 2.2 shows a subset of a DOM tree with an html node (the root of the document), body node (root of the visible document), div node (a container), and two p nodes (text paragraphs).

The DOM includes an event-model that allows event handlers written in JavaScript to listen for events occurring on individual nodes in the DOM. As an example, it is possible to register an event handler listening for click events (caused by user interaction) on the div node in Figure 2.2. A number of built-in events are supported, such as mouse click, mouse over, key press, focus events, and load events.

DOM events can be encoded into events in the event model by concatenating the kind of event and target DOM node together to form an event name,

![Figure 2.2: A subset of the DOM tree of a small web page with two paragraph nodes. The two paragraph nodes are contained within a parent div node, which is contained within the body and html nodes.](https://dom.spec.whatwg.org/)
N. This way, event handlers can be associated with pairs of event names and nodes. For example, an event handler may be registered to handle any click events occurring on the div node in Figure 2.2.

The DOM event model also supports event propagation such that events triggered on a target node in a DOM tree may be handled by event handlers registered on nodes further up the tree. As an example, if a user clicks one of the two p nodes in Figure 2.2 it would result in any event handlers registered to handle click events on either the clicked p node, div node, body node, or html node to be invoked while processing the event. This allows an event handler registered to, for example, the div node to handle all events occurring on child p nodes. Events propagate in two phases, first they propagate down the tree towards the event target (event capturing) and then up the tree from the target (event bubbling). Event handlers are registered to be invoked either during capturing or bubbling. Furthermore, an event handler may stop further propagation of an event.

Event propagation can be encoded in the event model by carefully updating the event handler registration map, mapping multiple event names to the same event handler. For example, if an event handler is registered to the div node, then it must also be registered on the two p nodes (in the correct position in the ordered list of event handlers) to account for event propagation. Furthermore, the possibility to stop event propagation can be handled by adding appropriate guards in all event handlers.

Web applications may request resources from or send requests to external servers, while listening for responses to those requests, through a number of mechanisms. Collectively, we will refer to these mechanisms as Ajax requests, and model such communication as either load events or readyStateChange events that occur when a response arrives back from a server. Both kinds of events include a parameter with the actual response from the server. As will be discussed in further detail in Chapter 6, the external servers retain their own state and may change their responses depending on earlier requests sent to the servers. However, to simplify our presentation, we focus on the client-side of web applications only in this dissertation, that is, we assume that the servers may respond with anything, if nothing else is stated.

Finally, web applications support DOM timers that allow event handlers to be invoked at a specific time in the future. This is encoded in the event model by creating a new unique event for the DOM timer and associating the event handler with this new event. Notice, that this encoding does not enforce the order of timers based on their timeout value.

### 2.2.2 Web Browsers

To accurately model the exact execution of a web application, it is necessary to model not only the web application itself but also the web browser as one complete event-driven application.
CHAPTER 2. EVENT-DRIVEN APPLICATIONS

<table>
<thead>
<tr>
<th>Group</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing and loading</td>
<td>DocumentLoad, ScriptRunnerAsync, ScriptRunnerInOrder, HTMLDocumentParser</td>
</tr>
<tr>
<td>Network</td>
<td>Network, XMLHttpRequestProgressEventThrottle, XMLHttpRequestProgressEventThrottleDefer, CachedResourceDelay</td>
</tr>
<tr>
<td>DOM</td>
<td>DOMTimer</td>
</tr>
<tr>
<td>User interaction</td>
<td>UserEvent, Auto, FakeMouseMoveEvent, Resize, ScrollEvent, PasteEvent</td>
</tr>
<tr>
<td>UI interaction</td>
<td>Navigation, BrowserOpenUIEvent, BrowserCloseUIEvent, BrowserLoadUrl</td>
</tr>
<tr>
<td>Other</td>
<td>EventSender, PostMessage, DocumentEventQueue, PendingTasks, CSSFontSelector</td>
</tr>
</tbody>
</table>

Table 2.1: Overview of browser-level events identified in the WebKit web browser as part of the R4 project presented in Chapter 4.

We use the open source WebKit browser (the basis of the Safari and Google Chrome browsers) as an example. WebKit maintains a priority queue of events and a queue of external events (i.e., I/O, mouse events, and keyboard events), using a single UI thread to process events from both queues. Processing a browser-level event may invoke internal event handlers in the browser, JavaScript event handlers registered in the web application, or a mixture of the two. Note, some parts of WebKit may conduct certain computations concurrently, but only events processed by the UI thread have any effect on the web application and what we identify as the configuration of the complete event-driven application.

By manually inspecting the implementation of WebKit and these browser-level events, we conclude that it is possible to encode a web browser in the event model. Besides the necessary encoding for encoding events in web applications, as explained earlier, the remaining browser-level events within WebKit are much simpler and often consist of simple one to one associations between unique events and event handlers.

Notice, different web browsers have slightly different implementations, bugs, and interpretations of standards, resulting in different initial configurations and different transition functions when encoding to the event model.
2.2. JAVASCRIPT CLIENT-SIDE WEB APPLICATIONS

```html
<!DOCTYPE html >
<html lang="en">
<head >
... 
<script >
var i = 0;
function goto_page(id) {
    jQuery.ajax(GET_PAGE_URL + '?page=' + id, {
        'dataType': 'json',
        'success': function(response) {
            populate_table(response);
        }
    });
}
function populate_table(attendees) {
...
</script>
</head>
<body >
table id="table_of_attendees">...
<button onclick="goto_page(++i);">Next </button >
<button onclick="goto_page(--i);">Previous </button >
</body>
</html>

Figure 2.3: Example of a web application inspired by the AIL paper [58].

2.2.3 Example

Figure 2.3 shows a small web application extended from an example given in the AIL project presented in Chapter 6. Parts of the web application, marked using ellipses (...), are omitted to simplify the presentation. This web application contains a list of people in a table and two navigational buttons for showing the next and previous page of people, respectively.

The navigational buttons are implemented through two event handlers registered using the `onclick` attributes on each button (lines 22–23). A newer style of registering event handlers also exist using the `addEventListener` method provided by the DOM. In the event model, these two event handlers are now mapped to `click` events on the two `button` nodes.

The user can interact with the page by clicking on the two buttons, which in turn triggers `click` events on the respective buttons (including `mouseover`, `mousedown`, and `mouseup` prior to the `click` event). Triggering the `click` events will add the events to the set of enabled events in our event model, for later processing by the event loop. As a result, a user may trigger a series of events to move between the different pages, and possibly uncover a bug when browsing negative page numbers.

When processing one of these `click` events, the JavaScript event handler
CHAPTER 2. EVENT-DRIVEN APPLICATIONS

will send an Ajax request to a server (lines 8–13) using the jQuery library. The response from this request results in a `readyStateChange` event that will subsequently invoke the callback given as an argument to the jQuery library (lines 10–12). This callback populates the table with the response from the server.

This raises some interesting questions. First, what happens if the user clicks on the next button multiple times? In which order will the responses from the server arrive and their respective events be processed? Furthermore, what would the contents of the server response be? These two questions are the subject of $R^4$ project and the AIL project and will be further explored in Chapters 4 and 6, respectively.

2.3 Android Mobile Applications

Android mobile applications are event-driven, with support for concurrency through, for example, asynchronous tasks. It is possible to encode the event-driven parts of Android using our event model, however while ignoring concurrency. As previously mentioned, we chose to focus only on the event-driven parts of an application in this dissertation, and leave the handling of both paradigms as an open problem.

In general, an Android mobile application consists of different activities and UI widgets in those activities. An activity represents screens visible by a user, and an Android mobile application is made up of multiple activities that the user can transition between. Widgets are UI elements within an activity, usually structured in a hierarchy. A user can interact with an application by, for example, touching widgets or rotating the device, and the system may interact with an activity by, for example, stopping and starting an application (part of the activity life-cycle).

Events are used to convey, and react to, user interaction, changes in the activity life-cycle, handle information from external sensors, etc. Usually, event handlers are registered either as callback functions or specifically designated methods on activity or widget objects. Compared to web applications, the encoding of these events and event handlers is simpler since there is no event propagation. However, events occurring on specific objects (e.g. touch events occurring on a specific widget in an activity) must still be encoded by combining the event kind and object into a unique event name.

In some cases, multiple listeners can be associated with the same event, such as for the mentioned touch event. If a touch event occurs on a widget, first an `onTouchListener` callback is called if registered, and then second the default `onTouchEvent` method is invoked on the widget object. However, the `onTouchListener` callback can prevent the default `onTouchEvent` method from being invoked. Similar to how web applications are encoded in the event
model, this can be encoded by adding additional guards in event handlers checking if event handling has been stopped.

One major difference between web applications and Android mobile applications is the activity life-cycle. An activity goes through a complicated life-cycle, in which it can be paused, resumed, started, stopped, created, and destroyed. Each of these correspond to an event, and each event can only occur under certain conditions, such as the stop event that can only occur after a pause event. The activity life-cycle events may be encoded in the event model, taking into account the constraints on valid life-cycle events when determining the set of enabled events. In Chapter 5 we present a UI model for Android mobile applications that models the possible sequences of user interaction events and certain activity life-cycle events.

2.3.1 Example

Figure 2.4 shows an example Android mobile application, which is also used as part of our COLLIDER project presented in Chapter 5. The example application is a personal tax calculator used to compute the income tax liability for a given income amount. In other words, the application conducts the calculation \( \max(income - deductions, 0) \times taxrate \) based on an income amount entered by the user in activity (a) and shows the result in activity (b) when the user clicks the Calculate button in activity (a).

Event handlers are registered for touch events on a number of widgets in the shown activities. For example, an event handler is registered to the button 0, in activity (a), which will multiply the current income value by 10 and update the display when invoked. Some event handlers may invoke transitions between activities, such as, the event handler registered to the settings menu item shown in (c), which will transition the application to the settings activity (d) when pressed.

The size of the deduction is configurable and is by default 0. The deduction may be configured in the settings activity (d) by clicking on the Amount of Tax Deduction button, which will open a system dialog for entering a value, shown in (e). Furthermore, it is possible to enable or disable the usage of tax deductions completely by toggling the Enable Tax Deduction button.

As a result, in order to run the tax liability calculation, with tax deductions enabled, a user must (1) navigate to the settings screen, (2) change the tax deduction configuration, (3) navigate back to the main screen, and (4) click the calculate button. In fact, this requires a minimum of 8 events to accomplish. Automatically generating such an event sequence using automated testing is non-trivial, and is the subject of the COLLIDER project presented in Chapter 5.

\[ \text{http://developer.android.com/reference/android/app/Activity.html} \]
CHAPTER 2. EVENT-DRIVEN APPLICATIONS

(a) Income activity.  (b) Result activity.

(c) Open menu.  (d) Settings activity.  (e) Deduction dialog.

Figure 2.4: Example Android mobile application used in the COLLIDER project presented in Chapter 5. The application is a simple tax calculator including (a) an income entry activity, (b) a result activity displaying the income, deductions, and resulting tax, (c) a menu, (d) a settings activity, and (e) a dialog for entering a new deduction amount.
Chapter 3

Automated Testing

In this chapter we discuss the challenges encountered when conducting automated testing of event-driven applications, and we present a survey of related work on the topic (Section 3.1). We also provide a short crash-course on feedback-directed testing and concolic testing (Section 3.2), which are specific testing techniques we use in later chapters.

Test, verb
“Take measures to check the quality, performance, or reliability of (something), especially before putting it into widespread use or practice.” — Oxford Dictionary

Fundamentally, software is tested by (1) executing the software under test using an input and (2) deciding if the execution was as expected or uncovered a bug in the software. We denote these two steps collectively as a test case.

Automated testing is a method for automatically generating test cases, which entails automatically choosing interesting inputs (denoted the input generation problem) and deciding if the execution of the software using the selected inputs uncovers any bugs (denoted the oracle problem). The former is challenging because of the large space of possible inputs to choose from, while the latter is a challenge because software often lacks any kind of formal specification of the expected behavior.

As an alternative to testing we could, in principle, consider software verification, which is a collection of many techniques for verifying properties of software for all inputs. However, verification may be infeasible exactly because verification reasons about all possible executions of a program and a program may represent a very large number of possible executions. This problem can be mitigated by, for example, sound reductions in the search space, program annotations, or by approximating program behavior using models. However, these solutions are not always feasible, acceptable, or precise enough, respectively. Furthermore, verification may result in false positives, while testing
presents concrete inputs that are shown to uncover bugs. For this reason, we chose to focus on testing as our method of choice.

3.1 Related Work for Event-Driven Applications

This section will discuss related work on automated testing of event-driven applications. Specifically, we highlight the input generation problem and the oracle problem. Related work relevant to the contributions in this PhD dissertation but outside of the domain of event-driven applications is deferred to Chapters 4–6.

3.1.1 The Input Generation Problem

Recall the event model in Chapter 2. We can use this model to explain the possible space of inputs to an event-driven application. In fact, we identify three main areas of the model that are of interest when generating inputs for testing:

1. Changing the execution environment (i.e. a web browser), effectively changing the initial configuration and transition relation.
2. Trying to execute different sequences of events.
3. Changing the event parameters for events in an event sequence.

We assume that the program executed by the event-driven application is not mutated. In the following we will describe how each of these points have been handled in related work on automated testing, and point out where the contributions of this PhD dissertation belong.

Execution Environment

The same event-driven application may be executed within different execution environments. Unfortunately, we often find inconsistencies between different implementations of such execution environments. As an example, different web browsers differ slightly in initial state and how they execute web applications, sometimes intentionally or sometimes because of bugs in the web browsers themselves.

For the domain of web applications, Choudhary et al. [22] proposes to use DOM and visual comparisons to identify inconsistencies between different browsers, while Mesbah and Prasad [78] proposes to crawl web applications in different web browsers and comparing the resulting models for inconsistencies. These two lines of work have been combined in the CrossCheck tool by Choudhary et al. [23], which in turn has been improved upon in the X-PERT tool by Choudhary et al. [24, 25], adding support for comparing layouts and reducing false positives.
Furthermore, a number of commercial tools exist for testing both Android mobile applications and web applications using different device configurations and web browsers.

In our work we assume a fixed execution environment and do not take into account different behaviors in different execution environments or different software versions. We make this choice to simplify our work.

Event Sequence Generation

In general, the domain of possible event sequences is too large to fully explore. Thus, it is key that any technique for automated testing is able to quickly choose interesting event sequences. The problem of quickly choosing interesting inputs is highlighted by Böhme and Paul [15], who present formulas for calculating if an automated testing tool outperforms simple random testing.

It is possible to test individual event handlers in isolation, assuming that the system could be in any state when invoking the event handler. This would reduce to unit testing of event handlers. However, unit testing may uncover potential bugs in event handlers that would never occur in a real system because of restrictions on the set of possible event sequences, restricting the possible states in which an event handler may be invoked. Gross et al. [48] highlights this concern as their prime argument for conducting system testing, that is, generating complete event sequences to uncover bugs. In this work, we conduct automated testing using concrete event sequences, thus avoiding the risk of generating false positives.

Model-Based Testing

One popular technique for event sequence generation is model-based testing (also denoted GUI-based testing). In general, a model is used to represent all possible event sequences, such that a traversal of the model would generate candidate event sequences (e.g. by using a depth-first and random traversal). Such a model may be built iteratively as event sequences are executed.

Memon et al. [75, 77], Nguyen et al. [91] present the GUITAR framework and the GUI ripper tool for automated testing of GUI applications. Their model consists of a GUI tree, representing all possible GUI windows and events triggering the creation of new windows, and an Event Flow Graph representing the set of enabled events following the execution of an earlier event [76]. The GUITAR framework has also been used as a basis for exploring GUIs for Android mobile applications by Amalfitano et al. [5, 6].

In the domain of web applications, Benedikt et al. [12] explore multi-page web applications and point out central challenges such as identifying relevant events on a page. However, they do not clearly state their definition of a state and how they differentiate states, so it is unclear how well they handle modern web applications. This is improved upon by Mesbah et al.
who presents the Crawljax framework. Crawljax uses a state-flow graph as its model, which represents application states as vertices and events as transitions. Events are identified either based on their tag name (e.g. link or button elements), annotations, or using a domain specific language. Application states (the DOM) are compared using an edit distance algorithm with a configurable threshold for when states differ. Later work by Roest et al. [99] extends Crawljax with improved comparison of application states.

Crawljax’s state-flow graph may be seen as a simplified version of GUI-TAR’s model, with removed support for multiple concurrently open windows which is irrelevant for web applications. Furthermore, state-flow graphs are also related to models using finite state machines (FSMs), proposed by, for example, Amalfitano et al. [3, 4] who conduct state matching based on enabled events and Marchetto et al. [74] who conduct state matching of DOM state while abstracting away concrete values in the DOM. In general, model based techniques mainly differ in their choice of model and how they represent states. This choice impacts the precision of the model, size of the model, and guarantees provided by the model when traversing it.

In our Collider project (Chapter 5) we use a UI model of an Android mobile application to reason about possible event sequences. This model differentiates states by the set of enabled events and transitions between states as individual events (similar to [3, 4]). The resulting model represents an over-approximation of possible event sequences without taking into account shared state or registered event handlers.

Guided Exploration A number of techniques have been suggested for guiding exploration of event sequences, either as an extension to model-based testing or for generating a set of new event sequences based on an old generation of event sequences.

Feedback-directed testing (described in Section 3.2) relies on feedback from executing earlier inputs to generate new inputs. Artzi et al. [11] present Artemis, an automated testing tool for web applications, that uses feedback-directed testing to generate new event sequences based on observing the execution of an old event sequence, and to select the most interesting unexplored event sequence to try next. This is done by using, for example, read- write-sets and coverage metrics. Likewise, Fard and Mesbah [35] and Dincturk et al. [28] use feedback-directed testing to guide the exploration of a model in model-based testing, prioritizing which unexplored transition to follow next. A similar technique is search-based testing that is used by Gross et al. [48] and Marchetto and Tonella [71] to decide which transition to follow next while exploring a model.

Our AIL project (Chapter 6) builds on top of Artemis [11], and improves the way Artemis selects event parameters.
3.1. RELATED WORK FOR EVENT-DRIVEN APPLICATIONS

Dependency Analysis Dependancy analysis [16, 36] may be used to prune away event sequences from an exploration that are guaranteed to be subsumed, or to be equivalent to, already explored event sequences. Intuitively, dependency analysis identifies dependencies between pairs of events, such that two events are dependent if their execution does not commute. It follows, that if two events are independent then it is not necessary to explore any event sequence that only changes the order of the two independent events, and if they are dependent reordering the two events would be interesting.

Arlt et al. [9] use a static analysis of event handlers written in Java to build a graph of dependencies between events. This graph is then combined with a model of possible event sequences to generate sequences of events that depend on each other. Artzi et al. [11] dynamically construct read-write-sets for all executed events and uses these sets to guide further exploration, as part of its feedback-directed testing, by extending an old event sequence with events that depend on events in the event sequence.

Ganov et al. [39] introduce symbolic widgets that are symbolic counterparts to concrete GUI widgets used for data entry by a user. The purpose of symbolic widgets is to omit any events otherwise related to data entry through widgets when generating event sequences, by implicitly triggering all data entry events on symbolic widgets in the beginning of any event sequence, and in turn reducing the search space. Symbolic widgets are identified by analyzing the memory locations accessed by different widgets.

Anand et al. [7] present a tool for concolic testing of Android applications that explores event sequences of increasing length incrementally up to a bound $k$. To reduce the search space, they introduce the concept of read-only events, which are events that do not mutate any state. If a newly explored event sequence ends in a read-only event, then the event sequence is subsumed by the event sequence excluding the read-only event, and it is sound to omit further exploration from the subsumed event sequence.

Our COLLIDER tool (Chapter 5) uses dependency analysis when building event sequences in a backwards manner. Specifically, given a sub-sequence of events $\tau$, COLLIDER evaluates the set of events that depend on $\tau$ as possible candidates for prefixing $\tau$. This is similar to Arlt et al. [9] and Artzi et al. [11] except that COLLIDER builds event sequences in a backwards manner and does not conduct a search in a forwards manner. Li et al. [66] build on top of COLLIDER by combining concolic testing and dependency analysis to generate event sequences, however, while generating possible event sequences in a forward manner like Arlt et al. [9] and Artzi et al. [11]. Furthermore, Li et al. [66] optimize the dependency analysis by only considering dependencies between shared state that is used in unexplored branch conditions.

Finally, in our $R^4$ project (Chapter 4) we propose to use model checking to systematically explore all possible event sequences within some bound. To reduce the search space, we use partial-order reduction which uses dependency analysis to identify equivalent event sequences and reduce the search space.
Event Parameters

Closely related to the event sequence problem is the problem of deciding proper event parameters for each event. This problem is related to the problem of automatically selecting proper inputs to traditional programs, for example, unit tests. A number of techniques for traditional programs have thus been reused for event-driven applications, such as random testing [101], feedback-directed testing [11], concolic testing [7, 39, 69, 62, 66, 84, 100], profiles/expert systems [12], and static analysis [13].

Our AIL project (Chapter 6) uses specifications of client-server communication to generate realistic service responses as part of the Artemis tool [11] to test web applications without a server side. Furthermore, such an approach allows for white-box fuzz testing [45] of possible server responses. Furthermore, our Collider project (Chapter 5) uses concolic testing, explained in Section 3.2.2, to choose proper event parameters in a similar fashion as existing work using concolic testing, however, differing in the approach used for generating event sequences.

3.1.2 The Oracle Problem

The oracle problem is not specific to event-driven applications but is relevant when automatically testing any kind of software. However, solutions to the oracle problem are often application specific, since they may, for example, assume common properties of web applications to automatically decide whether an execution uncovers a bug or not.

Mesbah and Van Deursen [79]. Mesbah et al. [82] propose a technique for using invariants, pre-defined or user-defined, as test oracles for web applications. The pre-defined invariants suggested are: checking the validity of the DOM, DOM conformance to accessibility standards, and the existence of error messages embedded in the DOM. Automatically identifying invariants has been explored by Pattabiraman and Zorn [95] for DOM invariants, Mirshokraie and Mesbah [84] for JavaScript running in web applications, and Ernst et al. [34] for general programs.

Hong et al. [54] compares execution traces of web applications in order to identify changes caused by non-deterministic behavior. They detect changes by (1) comparing uncaught exceptions, (2) the final state of the application, and (3) the ability to successfully execute the same sequence of events in both traces.

Finally, Roest et al. [99] proposes a technique for coping with natural dynamism in web applications, such as handling dates and dynamic content from the server-side, which could help oracles avoid raising errors caused by expected nondeterministic behavior. Likewise, Sprenkle et al. [106] suggests a technique using multiple comparators comparing subsets of the DOM or adds additional semantic meaning to the DOM, such as handling changes to the
order of \textbf{option} elements in a form, in order to guard against insignificant changes in the DOM.

Our \textit{R$^4$} project (Chapter 4) uses a number of these techniques to decide whether changing the order of two events in an event sequence results in a failure. Specifically, it compares event sequences similar to Hong et al. [54]. However, as we show in Chapter 4 we improve upon Hong et al. [54] by comparing single changes to event sequences, and by expecting that changes to an event sequence may result in harmless changes to the execution of the event sequence, using a series of heuristics to identify harmless and harmful patterns. Other related work, such as, DOM invariants, is not used but may be incorporated.

3.2 Background

In this section, we give a short introduction to central testing techniques used in the following chapters. Thus, this section can be safely skipped if the reader has prior knowledge of feedback-directed testing and concolic testing.

3.2.1 Feedback-Directed Testing

Pacheco et al. [94] propose to use feedback-directed testing for constructing sequences of method invocations which they use for unit testing. They continuously select a known sequence of method invocations, and randomly select a new method invocation to extend the known sequence with. They then evaluate the new sequence and either (i) discard the sequence if an equivalent sequence has already been explored, (ii) save the sequence for later exploration if it leads to new state relevant for further testing, or (iii) create a new unit test based on the sequence. Thus, this method uses feedback from previous executions to select initial sequences to extend, and to decide whether a new sequence of method invocations should be discarded.

Artzi et al. [11] present the Artemis tool that uses feedback-directed testing to automatically test web applications, with the purpose of generating tests with high code coverage. Artemis is built on top of the WebKit web browser, using instrumentations within WebKit to gather feedback.

Artemis iteratively constructs longer and longer event sequences by extending previously tested event sequences with new events. Each new event sequence is executed concretely, and any violations of the HTML specification or uncaught exceptions in JavaScript are reported as errors. Feedback is used to restrict the set of events used for extending event sequences, prioritize the event sequence to explore next, and select event parameters. We will use Artemis in Chapter 6 as a platform for our AIL project.

\textbf{Event Sequence Generation} Each time Artemis executes an event sequence, $\tau$, it generates a new set of event sequences by extending $\tau$ with any
enabled events, \( \{ \tau \cdot e \mid e \in E_\tau \} \), where \( E_\tau \) is the set of events enabled after executing \( \tau \). The set of enabled events is identified using feedback from executing \( \tau \), by identifying any events with registered event handlers. Furthermore, feedback is used to prioritize which newly extended event sequences should be explored next, using, for example, read-write-sets and coverage metrics.

As an example, in the example web application shown in Section 2.2.3 we observe that event handlers are registered to click events on each of the two button elements. Initially, Artemis would load the web application using and execute an empty event sequence. Two new event sequences are then created, one with a click event on the first button, and one with a click event on the second button. Notice, that the empty sequence could have been extended with, for example, a click event on the table element. However, using feedback about event handler registrations such event sequences are avoided. The two new event sequences are given the same priority, since they both trigger events that have not been explored yet and have no coverage.

**Event Parameters**  Artemis uses random values for most event parameters. However, when executing event sequences, it also gathers feedback about any string literals used for comparisons within in the JavaScript interpreter. These string literals are then used for event parameters in newly generated event sequences, for example, for event parameters related to filling out forms inputs.

Other event parameters are selected based on external inputs. Specifically, event parameters that reflect the response from external servers, after sending an Ajax request, are set to the concrete response given by the external server. However, this results in tight coupling between the client-side web application and its server-side, and requires a fully functioning server to be configured and subsequently reset for each event sequence executed by Artemis. The AIL project proposed in Chapter 6 proposes a solution to this issue.

### 3.2.2 Symbolic Execution and Concolic Testing

Symbolic execution and concolic testing are techniques for generating inputs for exploring different paths within a program. For this reason, these are usually used for selecting parameters for events given a fixed event sequence. In addition, a side effect of these techniques is detailed knowledge about the flow of certain values and dependencies between events. For this reason, these techniques may also be used for event sequence generation, as we will show in our Collider project in Chapter 5. In the following, we will introduce classical symbolic execution and concolic testing which builds upon symbolic execution.
3.2. BACKGROUND

![Image](image.png)

Figure 3.1: Example of a small C function suitable for symbolic execution.

**Symbolic Execution**

Symbolic execution [65] is an alternative to concrete execution of programs where symbolic values, representing entire classes of values, are used in place of concrete values. In symbolic execution all inputs are given distinct symbols, all values calculated using these symbols are represented using symbolic expressions, and every branch is associated with an invariant, denoted a *path condition*. A branch can only be traversed if its associated path condition is satisfiable.

As an example, Listing 3.1 shows a simple function with two inputs, \(a\) and \(b\). Let us handle these two inputs as symbols, such that the symbolic value of \(c\) at line 2 is equal to the symbolic expression \(a + (b \times 2)\). Furthermore, the if statement at line 3 branches the program, such that execution can either reach line 4 or line 6. The path conditions at line 4 and 6 represent possible values for symbols for execution to reach the respective lines. For example, the path condition at line 4 is \(a + (b \times 2) > 0\) and the path condition at line 6 is \(a + (b \times 2) \leq 0\).

Using a constraint solver it is possible to decide whether a path condition is feasible and if the branch associated with the path condition can be traversed. This allows symbolic execution to explore an application and decide which parts of an application are reachable. Thus, by encoding interesting properties as a reachability problem, such as null pointer dereferences, it is possible to show the presence or absence of such faults. However, symbolic execution depends on the capabilities of the underlying constraint solver and the ability of the symbolic execution engine to translate the entire program semantics into symbolic expressions and path constraints. However, this is not always feasible. Listing 3.2 illustrates a classic example using a *hash* function. In this example, the hash function is too complex for the symbolic execution engine to construct a symbolic expression for \(\text{hash}(0)\). This prevents symbolic execution from reasoning about the program and constructing path conditions for the two branches.
Concolic Testing

This problem is solved in a variant of symbolic execution denoted concolic testing [103, 104] (also known as dynamic symbolic execution). Concolic testing mixes symbolic execution with concrete execution by executing an application using symbolic and concrete execution at the same time. If the symbolic execution engine is unable to construct a symbolic expression it can fall back to concrete values from the concrete execution. For example, the concrete value of \texttt{hash(0)} in Listing 3.2 can be used when solving the symbolic constraints in the same example. This allows concolic testing to analyze a larger set of applications than symbolic execution.

Concolic testing can be used to systematically explore all possible paths in a program. The basic approach is to initially execute the program using a random input. This results in a path condition representing all branches taken in the initial execution. The last branch in the path condition is then negated, solved and the new input is tested. This is repeated until the entire program has been explored, essentially resulting in a depth first exploration of all feasible paths in a program.

Intuitively, both symbolic execution and concolic testing can be used to handle event parameters for event sequences, by executing the same event sequence multiple times while treating event parameters as symbols.

3.3 Summary

In this chapter we have detailed how the input generation problem and oracle problem is handled by existing work for event-driven applications. Furthermore, we identify three main areas within the event model that can be focused on when handling the input generation problem: (1) the execution environment, (2) event sequence generation, and (3) event parameter selection. The event sequence generation sub-problem, event parameter selection sub-problem, and the oracle problem are further elaborated upon in the following chapters.

Finally, we have provided sections on feedback-directed testing and symbolic execution/concolic testing that will be used in later chapters.
Chapter 4

Nondeterministic Scheduling

In this chapter, we present our work on automatic testing of nondeterministic scheduling, documented in the manuscript “Stateless Model Checking of Event-Driven Applications” included in Chapter 8. The purpose of this chapter is to give an intuitive overview of our work and expand upon topics that did not fit into the manuscript format. We refer to this work collectively as the R4 project.

In the following, we motivate our work (Section 4.1), provide an informal explanation of our solution (Section 4.2), provide an overview of related work (Section 4.3), and list the most significant results from the evaluation of R4 (Section 4.4).

4.1 Motivation

Recall, in our event model (Chapter 2), the event loop nondeterministically selects an enabled event for processing. This nondeterminism represents the fact that external actors may emit events to the event-driven application in any order. For example, in Section 2.2 we observed that web applications handle HTML parsing, timers, network communication, and user events in a responsive manner. In practice, the internal scheduler within WebKit (a specific web browser implementation) handles events in a first-come first-served manner. However, events are added by external actors in nondeterministic order, for example, the order of network events rely on network latencies, and machine load impacts the number of HTML parsing events (the HTML parser yields after a fixed amount of time, such that the number of tokens parsed depends on system resources allocated to the HTML parser).

As a result, failures can occur if events are processed in an unexpected order. Previous studies have shown such nondeterminism to cause many subtle failures that are difficult to detect by ordinary testing [8, 18, 54, 83, 88, 96, 98].
We divide events into user and system events. A user event is added to the set of enabled events as a direct action from a user of the system, for example, mouse click events, while all other events, for example, network events and parsing events, are denoted system events. From the point of view of a user, the order of user events is deterministic, while the relative order of system events is nondeterministic. We say that nondeterministic scheduling exists if the order of system events is nondeterministic but not the order of user events.

Our Contributions We want to automatically test web applications for failures caused by nondeterministic scheduling. We do this by recording an initial execution of a web application and systematically exploring alternative orderings of system events (the input generation problem). Furthermore, we implement an oracle to check whether any explored event sequence causes a failure (the oracle problem).

We propose $R^4$ and its four phases: record, reorder, replay, and report. The record phase captures an initial recording of a web application, for example, by observing a user session or using another automated testing tool to construct an initial event sequence. The initial recording is explored by continuously applying the reorder and replay phases, mutating an event sequence and replaying the mutated sequence, respectively. Finally, the report phase compares the executions of an original and mutated event sequence, and decides whether a bug was uncovered or not. In summary, we present the following contributions:

- We present an algorithm for stateless model checking with partial-order reduction of event sequences for event-driven applications (reorder phase). The algorithm is based on the dynamic partial-order reduction [38] algorithm for concurrent programs.

- We extend our algorithm with conflict-reversal bounding, which bounds the number of changes in any explored event sequence, approximate replay, which reduces divergence from the initial execution (replay phase), and filters that classify explored event sequences as either harmful or harmless (report phase).

- We implement $R^4$ for web applications and evaluate our implementation on a set of real-world web applications. We show experimentally that $R^4$ can systematically explore nondeterministic schedules and detect failures with increased precision compared with prior work.
4.2. SOLUTION

1. $\tau_{\text{init}} := \text{record}(\ldots)$
2. add $\tau_{\text{init}}$ to explored
3. while $\sigma, \tau = \text{reorder}(\text{explored})$ do
   4. $\tau' := \text{replay}(\sigma)$
   5. report$(\tau, \tau')$
   6. add $\tau'$ to explored
   7. end while

Figure 4.1: High-level view of the $R^4$ phases. The while loop iterates over mutated event sequences identified by the reorder phase until no such mutated event sequence exist.

4.2 Solution

In the following, we present a running example (Section 4.2.1) and show how each of the four phases of $R^4$, record, reorder, replay, and report, apply to the running example.

Figure 4.1 shows how the four phases interact, intuitively. Initially, a recording is made of an event sequence, $\tau_{\text{init}}$, and added to the set of explored event sequences (Section 4.2.2). The reorder phase (Section 4.2.3) then continuously selects an old execution, $\tau$, from the set of explored sequences and proposes an alternative event sequence, $\sigma$, based on $\tau$ but with reordered system events. The replay phase (Section 4.2.4) then executes $\sigma$ and the report phase (Section 4.2.5) compares the old execution $\tau$ and the newly executed $\tau'$.

4.2.1 Running Example

Figure 4.2 shows an example web application affected by nondeterministic scheduling. The web application implements a slide-show widget with a single image (line 28) and a next button (line 27) for transitioning the current image to the next image. The widget uses deferred loading of images to minimize the initial load time and thereby maximize responsiveness. The widget requests images from the server when the page is done loading (lines 20–24). Each time an image is loaded, it is added to a queue (line 8).

Whenever the user clicks the next button, the current image is replaced by the next one from the queue (line 17), provided that one is available. If the queue is empty because the next image has not yet arrived, a loading indicator is shown and a timer is set to retry after 100ms (lines 14–15). The timer ensures that the application will eventually transition to the next image when available. We denote the usage of such a timer, and similar solutions that implement synchronization without any synchronization primitives, as ad-hoc synchronization.
Finally, a statistics script is included (line 29) in a way that is common for modern client-side statistics solutions. The script is deferred, such that it is fetched and executed after all HTML has been parsed. When executed, the script will send an Ajax request containing information about the user and the visited page.

Intuitively, the running example is affected by nondeterministic scheduling since the order of image loads is nondeterministic. This nondeterminism affects the execution, for example, click events on the next button may either, if the queue is non-empty, show a new image, or if the queue is empty, enable a DOM timer event.
4.2. SOLUTION

4.2.2 Record Phase

First, an initial execution of the tested event-driven application is recorded. This initial recording can be conducted either manually, by recording user sessions, or generating a single random event sequence. Our implementation supports manual and random exploration of web applications.

Figure 4.3a shows an initial recording of the running example where:

1. The user loads the page (for simplicity, let us in this example consider that as a single event $e_1$ although it technically consists of multiple HTML parse and network events).
2. Image image2.png loads (event $e_2$).
3. The user clicks the next button twice (events $e_3$ and $e_4$).
4. The statistics script is executed and it pings an external server (events $e_5$ and $e_6$).
5. Image image3.png loads (event $e_7$).
6. The timer spawned by $e_4$ is triggered (event $e_8$).

The events $e_3$ and $e_4$ are controlled by the user and are thus user events, while all other events are system events. Thus, in the reorder phase we are interested in systematically exploring alternative event sequences where system events are reordered but $e_3$ and $e_4$ are kept in their current order.

4.2.3 Reorder Phase

The reorder phase takes an initial recording as input and systematically explores alternative event sequences by reordering system events. We present an algorithm for stateless model checking of nondeterministic schedules with partial-order reduction. This algorithm is used to systematically explore the space of possible event sequences.

A model checker systematically explores a model to verify some property. In our case, we use the model checker for testing by exploring a subset of a model representing the space of possible event sequences, using a bound to limit the search. Figure 4.3a shows part of such a model, where each vertex represents a sequence of events (the sequence of events from the root to the vertex) and each transition represents an event. The algorithm is stateless, that is, the complete program state is not stored as part of the exploration.

Our algorithm is based on an existing algorithm for concurrent programs [1][38] that we adapted for event-driven applications as part of this work. In this section, we apply our algorithm on our running example. A comparison with related work is provided in Section 4.3. We refer the reader to Chapter 8 for a formal presentation of our algorithm.
Exploration

At any point in time, the algorithm maintains a representation of the current event sequence under consideration (a subset of the model). Figure 4.3a shows the initial representation after the record phase. The algorithm updates this representation with transitions to explore later, in this case 3 such transitions have been added at $e_1$, $e_3$, and $e_6$.

A naive approach would be to add a transition for each enabled event at each vertex in the model (i.e. exploring all possible event sequences), which would explore $O(n^l)$ event sequences, where $n$ is the number of enabled events in any state and $l$ is the length of the event sequence. However, we know from previous work [98] that event-driven applications, such as web applications, may operate with sequences with thousands of events. Thus, the naive approach is infeasible.

To reduce the search space, this algorithm uses partial-order reduction, such that event sequences that are equivalent to prior explored event sequences are not explored. Intuitively, if events $x$ and $y$ commute, such that $x \cdot y$ and $y \cdot x$ always lead to the same shared state, then it is sufficient to explore event sequences in which $x$ precedes $y$, or vice versa, to reach all final states. This observation is used to only explore a subset of the model by only selecting a subset of transitions, to be explored later, at each vertex in the model. We select this subset of transitions by identifying conflicts between events.
Conflict Detection  Given an event sequence, the executed event sequence is analyzed for any conflicts. Intuitively, a conflict exists between two system events if (1) it is possible to change the order of the two events, and (2) the two events do not commute. A conflict implies that reversing the order of two system events may yield an alternative execution that may in turn uncover a bug. Furthermore, we filter away any conflicts which require the reversal of another conflict in order to be reversed. Conflicts are identified using a state-of-the-art race detector for web applications, EventRacer [98], capable of answering questions about commutativity and happens-before relations between events in an executed event sequence. By default, EventRacer dynamically observes any memory locations accessed during an execution, and if two events access the same memory and at least one access is a write, then we say that the two events do not commute. In addition, as part of this work, we extend EventRacer to soundly identify other access patterns to be commutative, for example, it is safe to ignore a memory location if the memory location is only written and never read.

For each conflict we generate a new event sequence with the two conflicting events reversed. We denote such an event sequence as a conflict-reversal. Notice, since reversing the two conflicting events may affect the execution, it is not possible to guarantee (and we do not try to predict) that all events in the conflict-reversal can be executed. We handle this problem using approximate replay explained in Section 4.2.4.

Figure 4.3a shows the initial recording annotated with three conflict-reversals to be explored later (see events \( e_1, e_3, \) and \( e_6 \)).

Backtracking  After updating the representation with any conflict-reversal, the algorithm proceeds to backtrack. Backtracking iteratively removes the last event of the event sequence representation until an unexplored conflict-reversal is associated with the last event in the representation. If such a conflict-reversal is found, then it is explored using the replay phase explained in Section 4.2.4, otherwise if the algorithm is unable to backtrack further then the exploration has completed and the algorithm terminates. Exploring a conflict-reversal will update the representation, and the conflict detection and backtracking steps are repeated on the new representation.

As an example, in Figure 4.3a events \( e_8 \) and \( e_7 \) are removed while backtracking until \( e_6 \) is reached. The conflict-reversal \( e_8 \cdot e_7 \) stored at \( e_6 \) is then executed, resulting in the event sequence shown in Figure 4.3b. Notice that after exploring the conflict-reversal, a new timer event, \( e_9 \), is now in the set of enabled events. This event is added by the timing event \( e_8 \), because \( e_8 \) is now processed prior to the image \( e_7 \) is loaded. Deciding if this behavior is harmful or not is non-trivial, and is the topic of our filters described in Section 4.2.5.
Conflict-Reversal Bounding

When identifying conflicts in the event sequence shown in Figure 4.3c (the next iteration after Figure 4.3b), we uncover a new conflict between \( e_7 \) and \( e_3 \), leading to the conflict-reversal \( e_7 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \) stored at \( e_2 \). We make two important observations here: (1) exploring new event sequences may uncover new conflicts and especially uncover new conflicts involving events earlier in the event sequence, and (2) complete exploration would require exploring all possible combinations of conflicts in an event sequence. However, as observed by Raychev et al. [98], a normal web application may contain hundreds of conflicts. Complete exploration is thus infeasible for many web applications since the search space is exponential in the number of conflicts. As a result, we introduce conflict-reversal bounding. Intuitively, conflict-reversal bounding counts the number of conflict-reversals explored in order to identify any new conflict-reversal, denoted the conflict-reversal depth \( k \), and restricts exploration to some bound \( d \).

As an example, the conflict-reversal depth for the exploration shown in Figure 4.3a is 0, while the depths of the explorations shown in Figures 4.3b and 4.3c are 1. Thus, if the bound is 1, then we will never try to explore the newly identified conflict-reversal in \( e_3 \) in Figure 4.3c.

4.2.4 Replay Phase

As noted earlier, there is no guarantee that the events in a conflict-reversal can be executed in the given order. As an example, observe the conflict-reversal \( e_7 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_8 \) stored at event \( e_3 \) in Figure 4.3b. The result of executing this conflict-reversal is shown in Figure 4.3c. It turns out that the event \( e_8 \) is never enabled when executing the conflict-reversal, since both images are loaded prior to the last user click event such that the DOM timer event \( e_8 \) is never enabled. Similar to the case where a new event appears, it is non-trivial to decide if missing \( e_8 \) is harmful or not.

We introduce the term approximate replay for executing a conflict-reversal expecting that not all events in the conflict-reversal may be enabled. Approximate replay may skip events and takes into account that event parameters may vary slightly between executions. For example, network responses contain a URL as part of their event parameters (representing the original requested URL). The URLs may differ because of, for example, session IDs. We implement heuristics for handling a number of common patterns for matching event parameters between executions.
4.2.5 Report Phase

In what follows, we describe the report phase that decides whether an explored event sequence uncovers a failure or not.

The reporting phase consists of a series of filters that analyze individual executions. Some of the following filters compare a newly explored conflict-reversal and the parent event sequence from which the conflict-reversal was discovered. For example, the conflict-reversal shown in Figure 4.3b is compared to the event sequence shown in Figure 4.3a. The report phase classifies a conflict-reversal as harmless if at least one of the following filters applies:

**Commuting operations:** If the final state of the JavaScript runtime, including the DOM, is equivalent between the explored event sequence and its parent event sequence, then the two operations commute. Two states are trivially equivalent if they are identical, but we also allow for certain data to differ between the two states such as the order of class names in class name strings (a set of class names is encoded as a space separated list in the DOM).

**Timer ad-hoc synchronization:** We observe a common pattern of recurring DOM timers used for ad-hoc synchronization. For example, a DOM timer event checks a condition $c$, registering a new DOM timer until $c$ is true. When $c$ is true, a task (the critical section) is executed.

This pattern manifests itself in a number of ways when exploring alternative event sequences, for example, by delaying when the condition is satisfied such that a new DOM timer is registered, or satisfying the condition earlier such that a series of DOM timer events are never enabled. We discover this pattern by identifying if (1) one or more new DOM timer events are registered, or (2) any number of DOM timer events are never enabled without changing the final DOM state.

**Late attachment of event handler:** If the explored conflict between two events is caused by the registration of an event handler and the triggering of the same event handler, then we flag such a conflict as harmless. This pattern is often caused by, for example, event handlers which are not registered before the document has loaded (representing a common pattern) and it is expected and accepted that such event handlers cannot be triggered before being registered.

If none of the above filters apply, then we classify the explored conflict-reversal as harmful and report an error to the user, provided that any of the following filters are satisfied:

**Page differences:** If the rendered pages differ between the explored event sequence and its parent event sequence, then we mark the explored event sequence as harmful.
**Uncaught exceptions**: If the set of raised JavaScript exceptions differ between the explored event sequence and its parent event sequence, then we mark the explored event sequence as harmful. It is common for web applications to raise exceptions, thus we only react to changes in the observed exceptions and not the existence of exceptions.

**Different network communication**: If the set of network requests sent to a server (i.e. Ajax requests) differ between the explored event sequence and its parent event sequence, then we mark the explored event sequence as harmful.

The filters suggested above are not unique to \( R^4 \). Similar solutions to the commuting operations, late attachment of event handler, page differences, and uncaught exceptions filters have been suggested earlier [54, 98]. However, the timer ad-hoc synchronization and different network communication filters are, to our knowledge, new in this domain.

### 4.2.6 Summary of Solution

In summary, we present an algorithm for stateless model checking of event-driven applications with partial-order reduction, conflict-reversal bounding, and approximate replay, in addition to a set of filters for classifying the explored event sequences. We combine these into \( R^4 \) and its four phases: record, reorder, replay, and report.

### 4.3 Related Work

This section provides an overview of the background that \( R^4 \) builds upon and explains the state-of-the-art related work that we compare \( R^4 \) with.

#### 4.3.1 Record and Replay

The replay phase is related to work on deterministic record and replay that observes executions of programs and repeats the executions deterministically.

Narayanasamy et al. [89] propose a method for deterministic record and replay of application executions on uni-processor architectures at the system level by (1) creating an image of the machine state before executing an application, (2) keeping a log of all context switches made during an execution, and (3) logging any effects of black-box behavior, such as, interrupts and system calls. It is possible to implement an event-driven application on top of a uni-processor system with non-preemptive scheduling, executing one event at a time without interrupting the execution. Thus, the usage of images and logs to describe an execution fits well with event driven applications. We use a similar approach when recording and representing an execution in \( R^4 \).
4.3. RELATED WORK

Deterministic record and replay can be implemented at both the application, the system level, and inbetween. While Narayanasamy et al. [89] is an example of record and replay at the system level, we find many examples of work targeting the application level, especially for event-driven applications.

Gomez et al. [46] suggest to record all triggered user events (i.e. touch events) for Android mobile applications and replay them back with the same timing, ignoring system events, such as, activity life-cycle events. Likewise, Mickens et al. [83] and Sen et al. [105] instrument JavaScript and use external proxies to record the timing of user events and a subset of system events for web applications. By instrumenting JavaScript, record and replay can be implemented in a manner agnostic to the underlying web browser, however, at the loss of any deep introspection and control of the web browser’s execution, such as, parsing of the HTML and execution of deferred scripts in web applications. In contrast, Burg et al. [18] present the Timelapse tool that instruments a web browser to log event sequences and accessed nondeterministic input, allowing for full introspection and control of the web browser’s execution.

We take a similar approach as Burg et al. [18], instrumenting the WebKit web browser to introspect and control execution, using logs (similar to Narayanasamy et al. [89]) to record the sequence of executed events, including event names and parameters. Our instrumentation differs slightly from Timelapse. For example, Timelapse disables yielding inside the HTML parser, such that all pages are parsed completely before any other task can run, while we maximize yielding inside the HTML parser, to emulate the fact that the HTML parser yields in a nondeterministic manner.

4.3.2 Fuzzing Schedules

Fuzzing schedules to uncover timing related bugs is a known technique in the domain of concurrent programs. Intuitively, an initial execution is observed (i.e. an event sequence or concurrent execution with context switches) and subsequently mutated to uncover bugs.

Sen [102] uses a race detector to identify all possible races in a Java program, possibly with many false positives compared to dynamic race detectors that only reason about a single concrete execution, and concretely executes event sequences to check if the identified races are real races. A random scheduler is used while executing the program in a manner that ensures the exploration of the races. In $R^2$, we systematically explore possible event sequences using a dynamic race detector to guide our exploration.

A similar approach is taken by Narayanasamy et al. [90] who explore two schedules for each dynamically identified data race in a binary program, classifying alternative schedules as harmless if the final state of executing an alternative schedule is identical with the final state of the original schedule. However, Narayanasamy et al. [90] assume that schedules are always exe-
cutable after reordering conflicts, and if not they interpret the root cause to be a timing related failure, while we use approximate replay and filters to anticipate changes when reordering conflicts.

In the domain of web applications, race fuzzing has been employed by Hong et al. [54], who present the WAVE tool, and by Andrica and Candea [8]. The WAVE tool observes an initial event sequence and constructs a number of alternative event sequences with as many changes to the order of events as possible, using happens-before relations to identify pairs of events that are not possible to reorder (i.e. two parse events following each other). Similar to Narayanasamy et al. [90], WAVE assumes that if a reordered event sequence is not executable then it is caused by a timing related failure. In $R^4$, we only reorder a single conflict at a time, which allows for easier identification of the root cause of any failure, and we anticipate that reordering events may have a harmless effect. Andrica and Candea [8] record an initial event sequence of user interactions (i.e. mouse click events) and replays the same sequence of events back to the application, reducing the time interval between user interactions to zero when replaying, effectively changing the order of user events relative to other system events. Compared to $R^4$, Andrica and Candea [8] only supports reordering user events and other events by changing the timeout between user events, while $R^4$ presents a much more general solution.

In summary, $R^4$ also conducts fuzzing of event sequences by identifying pairs of events (conflicts) that may affect each other and by exploring alternative event sequences with the order of the conflicting events reversed. However, we differ from existing work on fuzzing schedules by (1) systematically exploring event sequences using a model checking algorithm, (2) using approximate replay in order to anticipate changes in the execution, and (3) only changing a single conflict at a time to better detect the cause of a timing related bug.

4.3.3 Model Checking

Model checking has been used to systematically explore possible schedules of concurrent programs, with the purpose of uncovering timing related bugs. Compared to simply fuzzing of schedules, model checking differs by systematically exploring all possible schedules (possibly up to some bound).

One classic model checker is the CHESS tool [86, 87]. CHESS conducts an iterative preemption-bounded search of concurrent programs by instrumenting a program to automatically yield at specific program points, for example, when interacting with locks. At each program point, CHESS systematically determines if the current thread should proceed or if the thread should be preempted by another thread. This results in a very large search space. CHESS introduces preemption-bounding to limit the number of preemptive context switches allowed in any explored execution, while allowing any number of non-preemptive context switches (context switches caused by a thread yielding by
4.3. RELATED WORK

The evaluation by Musuvathi et al. [87] finds that a low number of preemptive context switches, such as 2, is sufficient to find most bugs.

Another approach to reduce the search space when using model checking is to use partial-order reduction [11]. Flanagan and Godefroid [38] propose a technique for stateless model checking with dynamic partial-order reduction (DPOR) while exploring a concurrent program. DPOR uses dependency analysis to identify instructions within processes that are dependent on each other, and systematically explores alternative schedules where dependent instructions are reordered if possible. Compared to CHESS, DPOR prunes the search space by not exploring equivalent event sequences only differing in the order of independent instructions. CHESS does propose an extension which prunes such event sequences after exploring them, but the extension requires storing all visited states, and is thus not stateless like DPOR. A number of papers extend DPOR: Coons et al. [26] combine DPOR with preemption bounding in a sound manner, Yang et al. [112] propose a sound way to add state checking to DPOR and Abdulla et al. [1] suggest a number of improvements to further reduce the search space, including source sets and wakeup trees.

The algorithm we propose builds on top of the DPOR algorithm, however, with two major differences: (1) we compare events that have been executed, while DPOR compares executed instructions with yet-to-be executed instructions in all known processes in a program when visiting a new state, and (2) we reformulate the algorithm without the notion of processes, replacing processes with a more general happens-before relation. Both changes are paramount for event-driven applications.

Furthermore, we store complete event sequences (conflict-reversals) for later exploration by approximate replay. This differs from DPOR, which only stores single events for later exploration and explores any event sequence after the stored event. Our Approach is related to wakeup trees proposed by Abdulla et al. [1]. Wakeup trees are sub-sequences of events stored for later exploration that guarantee the reversal of a conflict if explored. However, the sub-sequences only contain events leading to the first of the two conflicting instructions (or events in our case), and afterwards any event sequence may be followed. This makes it difficult to determine if a failure is caused by the conflict or not.

Finally, we bound our exploration in the same spirit as delay bounding proposed by Emmi et al. [32], which bounds the number of delays introduced in a scheduler for a concurrent program while exploring. We adapt this to event-driven applications by bounding the number of conflicts reordered. In practice, we differ in that Emmi et al. [32] bounds the number of fine grained changes to a schedule while we bound groups of changes.
The Oracle Problem

A number of papers explore the oracle problem of automatically identifying if a timing related bug is uncovered by an execution. Sen [102] checks for raised exceptions. Andrica and Candea [8] use manual validation. The WAVE tool by Hong et al. [54] explores both an original and alternative event sequence, and classifies the alternative event sequence as a failure if the DOM states does not match, an uncaught exception occurs, or the event sequence cannot be executed exactly. A similar solution is used by Narayanasamy et al. [90].

We differ from the above work by (1) relaxing the execution of alternative event sequences, allowing the execution to continue even if it deviates from the expected path and (2) suggesting heuristics for identifying ad-hoc synchronization in web applications. In practice, this reduced false positives, for example, in connection with animations that are canceled earlier but have no effect on the final rendering of the page. Petrov et al. [96] and Raychev et al. [98] observe that not all races in web applications are harmful, and observe that many benign races are caused by ad-hoc synchronization (implementations of locks without any locking primitives). As an example of ad-hoc synchronization, see the running example in Section 4.1 and the timer spawned when clicking the next button with an empty queue. In this work, we present a method for automatically exploring races and classify if they are harmful or not. Likewise, Mutlu et al. [88] also observe that not all races in the domain of web applications are harmful. As a result, Mutlu et al. [88] define the term observable races as races that can be observed by comparing two renderings of the same web page, and discuss the possibility of systematically exploring observable races as a possible research direction. R^4 is one such example approach of systematic exploration.

Kasikci et al. [64] explore different orderings of races in concurrent programs, comparing the outcome of the execution traces in order to classify the races. They correctly point out that differences in outcomes do not necessarily constitute harmful behavior. Thus, they classify races as either spec violations if a program specification is violated, output differs if the output of the program may differ, or k-witness harmless indicating that k witness traces support that the race is harmless. Additional witnesses are gathered by exploring additional suffixes to traces after a race or symbolically exploring traces (using concolic testing on program input) in order to observe additional paths within a program. These ideas could be applied to R^4, for example, by applying concolic testing on all explored event sequences, or using specifications to check for correct behavior. However, we argue that for, for example, web applications, it is uncommon to have a specification of correct behavior.
4.4 Evaluation

In this section we provide a summary of the evaluation of $R^4$ as presented in Chapter 8. We evaluate our implementation of $R^4$ (available online\footnote{https://github.com/eth-srl/WebERA}) experimentally by (1) applying it to a number of real-world websites and (2) evaluating the limitations of conflict-reversal bounding. A detailed comparison with the WAVE tool is omitted here but can be found in Chapter 8.

4.4.1 Bug Finding on Real-World Websites

In the first experiment we compare the $R^4$ tool’s ability to classify explored event sequences compared to the EVENTRACER tool by Raychev et al. \cite{Raychev2011}. Recall, EVENTRACER only conducts an analysis of a single execution, while $R^4$ explores multiple conflict-reversals and thus provides witnesses for any uncovered bugs, i.e., screenshots of the two outcomes. The purpose of this experiment is to evaluate whether the filters suggested for $R^4$ improves upon the automatic classification conducted by EVENTRACER.

A total of 32 real-world websites (randomly sampled from the top 100 fortune websites) are tested. An initial recording is conducted automatically for each website by triggering events by random, until either 250 events have been processed or until 15 seconds have elapsed. The conflict-reversal depth is set to 1, such that all uncovered races reported by EVENTRACER are explored as conflicts by $R^4$ (uncovered races in EVENTRACER are equivalent with conflicts in $R^4$). We then compare the analysis of the two tools.

Table 4.1 summarizes the results from running $R^4$ and EVENTRACER on the 32 websites. The first two rows show the number of uncovered races

<table>
<thead>
<tr>
<th>Metric</th>
<th>Number per website</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENTRACER uncovered races</td>
<td>68.8</td>
<td>20</td>
<td>1032</td>
<td></td>
</tr>
<tr>
<td>EVENTRACER marked as harmless</td>
<td>8.0</td>
<td>3</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>$R^4$ explored sequences</td>
<td>68.8</td>
<td>20</td>
<td>1032</td>
<td></td>
</tr>
<tr>
<td>Marked as harmless:</td>
<td>60.2</td>
<td>17</td>
<td>995</td>
<td></td>
</tr>
<tr>
<td>- Commuting operations</td>
<td>52.2</td>
<td>7</td>
<td>987</td>
<td></td>
</tr>
<tr>
<td>- Timer ad-hoc sync.</td>
<td>7.1</td>
<td>6</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>- Late attachment of event handler</td>
<td>2.7</td>
<td>0</td>
<td>53</td>
<td></td>
</tr>
</tbody>
</table>

Harmful Filters

<table>
<thead>
<tr>
<th>Filter</th>
<th>Mean</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different DOM</td>
<td>0.2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Different uncaught exceptions</td>
<td>0.3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Different network communication</td>
<td>0.5</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1: Explored event sequences for 32 tested websites.
found and filtered as harmless by EVENTRACER, respectively. The next two rows show the same for $R^4$, followed by details on how many sequences each harmless filter applied to. Results are given as mean, median, and maximum values observed for the websites.

On average, EVENTRACER only filters 8.0 out of 68.8 uncovered races as harmless, while $R^4$ is able to filter away 60.2 on average. This improvement is mainly due to the commuting operations filter, further improved by the ad-hoc timer filter. A manual study is also conducted on a subset of the filtered conflicts to ensure that any conflict filtered as harmless is truly harmless.

Of the remaining 8.6, on average, conflicts not marked as harmless, we further classify the conflicts using the different DOM, different uncaught exceptions and different network communication filters. The share of conflicts on which these filters apply are given in the last three rows in the table. Remaining conflicts, neither marked as harmless or harmful, are still reported but with lower confidence that they contain errors.

The uncovered bugs include: a case where a script registers an onload event handler but do so after the onload event occurs, resulting in a missing banner ad; and functions that are invoked before they are defined during parsing, resulting in missing widgets.

### 4.4.2 Evaluation of Conflict-Reversal Bound

The next experiment evaluates the effect of conflict-reversal bounds and consists of two sub-experiments.

First, we run our $R^4$ tool on each of the 32 websites used in the former experiment, incrementally increasing the conflict-reversal bound until the exploration is unable to terminate within one hour. The purpose of this experiment is to uncover the depth $R^4$ can explore on real-world websites.

Figure 4.4 shows the reached depths and number of websites that could be explored at those depths. We conclude that while some websites we are only able to cover depth 1, we also see a number of websites that allow for even deeper exploration. The current implement still allows for further efficiency improvements, and additional user guidance, for example, annotations
4.5. **SUMMARY**

<table>
<thead>
<tr>
<th>Site</th>
<th># Events</th>
<th>Explored sequences</th>
<th>Depth</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gallery3</td>
<td>516</td>
<td>3</td>
<td>1</td>
<td>&lt; 1m</td>
</tr>
<tr>
<td>TYPO3</td>
<td>1556</td>
<td>24</td>
<td>1</td>
<td>5m</td>
</tr>
<tr>
<td>WordPress</td>
<td>2043</td>
<td>22</td>
<td>3</td>
<td>&lt; 1m</td>
</tr>
<tr>
<td>AjaxPlorer</td>
<td>1528</td>
<td>38</td>
<td>1</td>
<td>28m</td>
</tr>
<tr>
<td>Feng Office</td>
<td>1451</td>
<td>24</td>
<td>1</td>
<td>9m</td>
</tr>
</tbody>
</table>

Table 4.2: List of web applications where an initial recording exposes a single confirmed bug, as used in the WAVE paper. The length of the initially recorded event sequence is reported, together with the number of event sequences explored, required conflict-reversal depth to expose the bug, and the running time for the analysis.

or configuration avoiding exploration of trusted libraries, would improve the results.

Second, we run our \( R^4 \) tool on a set of benchmarks used in the evaluation of the WAVE tool [54]. Each of these benchmarks contain a single known bug caused by nondeterministic scheduling. The purpose of this experiment is to find the conflict-reversal depth required to uncover the known bug in each benchmark.

Table 4.2 shows the depth required for \( R^4 \) to uncover the bug in each benchmark. We observe that all but one bug are uncovered within depth 1 and a single bug requires depth 3. The limited number of benchmarks does not allow for drawing general conclusions, however, we see this result as a promising indicator that many interesting bugs can be found at a low depth.

4.5 **Summary**

In summary, we propose \( R^4 \), a technique for exploring nondeterministic schedules in event-driven applications. We present a stateless model checking algorithm with partial-order reduction, conflict-reversal bounding, and approximate replay for event-driven applications, and propose a number of filters for automatically classifying executed event sequences to identify timing related failures. We implement our contributions in a tool for automated testing of web applications, and evaluate our tool on a number of real-world websites. We show that \( R^4 \) improves the state-of-the-art in automatic classification of timing related failures, and we argue that \( R^4 \) provides better insight in the root cause of failures compared to the state-of-the-art.

As discussed, and shown in the evaluation in Chapter 8, reordering events often affect the execution of an event-driven application, but not necessarily in a harmful manner, for example, because of ad-hoc synchronization in web applications. Thus, reordering event sequences and raising errors if the reordered event sequences cannot be executed would result in a high number of false positives. Therefore, we propose that any practical technique that reorders events to explore nondeterministic schedules should take such
harmless nondeterminism into account, for example, as we do using filters and approximate replay.

Furthermore, we find that some common JavaScript libraries access values in ways that are identified as racing by our race detector EventRacer, even though they do not race in practice. We extend EventRacer with support for identifying a number of memory access patterns, which reduced the number of reported races and in turn improved our results. Thus, we conclude that such improvements are important for web applications.

Finally, we conclude that real-world websites do contain timing related bugs, and our proposed method is sufficient to uncover such bugs. However, the real challenge is to identify the harmful failures among the harmless effects of, for example, ad-hoc synchronization. We find that concretely executing alternative event sequences allow us to filter away such harmless effects compared to, for example, race detectors such as EventRacer. Furthermore, we see a large design space in improving and developing additional filters, and possible library and application specific filters, to further reduce noise.
Chapter 5

Targeted Input Generation

In this chapter, we propose a technique for targeted event sequence generation, documented in the paper “Automated testing with targeted event sequence generation” Jensen et al. \[50\] included in Chapter 9. We refer to this work collectively as the Collider project.

The purpose of this chapter is to give an intuitive overview of our work and present an extended example illustrating our algorithm, compared to the more formal presentation in Chapter 9. In the following, we motivate our work and outline our contributions (Section 5.1), explain our algorithm using an example (Section 5.2), give an overview of related work (Section 5.3), evaluate our work (Section 5.4), and finally we summarize (Section 5.5).

5.1 Motivation

Existing work on automated testing of event-driven applications, described in Chapter 3, focus on generating as many interesting test inputs as possible. However, it is well known that it is infeasible to generate all interesting test inputs for non-trivial applications.

This highlights the interesting subject of trade-offs between different testing techniques and their computational and memory cost. For example, a technique that selects random event parameters may be able to try many more test inputs compared to a technique that selects event parameters using concolic testing, but may not select as interesting event parameters.

We suggest to use a targeted approach to test input generation in order to reach challenging targets in an application under test. A challenging target is a specific program point (i.e. a program statement) in an application that has not been reached by other testing techniques. We envision that our targeted approach may be used to test parts of an application that conventional testing techniques did not cover, for example, by running a random testing tool on an application and then using our approach to target any uncovered program points.
Our Contributions Our algorithm combines work by Ma et al. [67] on directed symbolic execution for traditional programs using a backwards search of a call graph and Arlt et al. [9] on using UI models and dependency analysis to generate event sequences. Specifically, we use concolic testing to select event parameters and calculate dependencies between event handlers, and we conduct a backwards search of a GUI model to iteratively build an event sequence from a challenging target back to the initial state of a program.

In summary, we present the following contributions:

• We propose an algorithm for targeted event sequence generation, designed for handling long event sequences and event parameters.

• We propose heuristics for prioritizing the backwards search, reducing search time.

• We implement our algorithm and heuristics in a tool, COLLIDER, for Android mobile applications, and evaluate COLLIDER on a collection of benchmarks. The evaluation shows that COLLIDER is able to reach targets otherwise unreachable by random testing and systematic exploration of a UI model.

5.2 Solution

In the following we show an example Android mobile application, and explain step by step how our algorithm generates an event sequence and selects event parameters that reach a specific target in the application.

5.2.1 Example

Recall the example Android mobile application, which is used for calculating tax payments, presented in Section 2.3. Figure 5.1 shows a UI model of this application, representing the UI and possible actions available in the UI, related to the work on model-based testing listed in Section 3.1.1. In our UI model, a state is identified by a set of enabled events, and a transition represents the state change that may occur by processing an event. The UI model excludes any events that do not have any registered event handlers and assumes that there is only one event handler registered to any event (multiple event handlers can be modeled by adding an additional intermediate state).

Since there is a one-to-one correspondence between events and event handlers in our UI model, we use the term event and event handler interchangeably. Furthermore, the UI model also includes a program entry, representing the initialization of the program when started. In our UI model, the program entry is marked using the start transition, or $s$ for short.

As an example, the transition $e_2$, in Figure 5.1, is the event handler triggered when a user clicks the calculate button on the entry screen, resulting
in a calculation of the user’s tax liability and the application transitioning to
the \textit{result} activity. Figure 5.2 shows a snippet of the code executed when pro-
cessing $e_2$. For the purpose of this example, we select the statement at line 8
in the code snippet as our target. Reaching this target requires \texttt{taxable} to
be negative. This is only possible if \texttt{deduction} is larger than \texttt{income} (line 6),
and this is only possible if tax deductions are enabled and set to a non-zero
value in the application settings (line 4). The application settings are changed
using the settings menu accessible through $e_4$.

Thus, in order to reach our target, an automated tool must generate the
event sequence $\text{start} \cdot e_4 \cdot e_5 \cdot e_7 \cdot e_8 \cdot e_{10} \cdot e_9 \cdot e_6 \cdot e_2$, navigating to the settings
activity ($e_4 \cdot e_5$), enabling tax deductions ($e_7$), entering the deductible amount
in a text entry widget ($e_8 \cdot e_{10}$), and finally navigating to the result page
($e_6 \cdot e_2$).

Multiple transitions from a state may be labeled with the same event, if
the exact transition taken depends on the shared state of the event-driven
application. In that case, the UI model over-approximates the possible event
sequences in an application.
5.2.2 Key Observations

Based on the above example, we make two key observations:

**Observation 1** A challenging target may be reached by processing a *target event* while the application is in a specific state, for example, processing the $e_2$ event and executing the code in Figure 5.2 will reach our target if deductions are enabled and the deduction amount is larger than zero. We denote this subset of the application state as the *dependency set* (i.e. a set of memory locations).

We want to identify other events that must be processed prior to processing the target event, to mutate the program state such that the target is reached. We denote such events as *anchor events*. The challenge is then to identify a sequence of anchor events from the program entry to the target event. As an example, there exist two anchor events for the target event ($e_2$) in our running example: $e_7$ that enables tax deductions, and $e_{10}$ that sets the tax deduction amount.

**Observation 2** We also observe that a number of events exist purely for navigating between UI states, for example, the events $e_4, e_5, e_8$ in Figure 5.1 that only exist to navigate to and within the settings menu. We notice that such events often do not mutate any state. We use such events, denoted as *connector events*, to navigate between anchors events.

A sequence of anchor and connector events is denoted as a *partial sequence*. A partial sequence that stretches from the program entry to a target is denoted a *complete sequence*. A partial sequence also has a dependency set that is the application state that the entire sequence depends on in order to reach the target. Notice, an event is either a connector or an anchor event depending on the partial sequence it precedes.

5.2.3 The Algorithm

Based on our two observations, we propose an algorithm that given a UI model (1) analyzes every event to determine its effect on the application state and (2) conducts a backwards search from a target back to the application entry point by iteratively prepending anchors and connectors to partial sequences. We do this in two phases: in the first phase we conduct symbolic summarization of all event handlers, and in the second phase we construct sequences.

**Symbolic Summarization**

First, we apply concolic testing to each event handler (i.e. each transition) in the provided UI model. As described in Section 3.2.2 concolic testing explores paths through a program that are reachable by changing the values
### Table 5.1: Overview of all path summaries for our running example. The first column is the path summary number, the second is the transition the path summary belongs to in the UI model, and the third and fourth columns are the path condition and symbolic state, respectively. Path summaries $W_3$ and $W_5$ for the transition $e_2$ reach our target. We omit the symbolic state for $W_3$, $W_4$, $W_5$, and $W_6$ for clarity.

<table>
<thead>
<tr>
<th>$W$</th>
<th>T.</th>
<th>Path Condition</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_1$</td>
<td>s</td>
<td>true</td>
<td>[income$_1 = 0$; deduction$_1 = 0$; deductionEnabled$_1 = false]$</td>
</tr>
<tr>
<td>$W_2$</td>
<td>$e_1^0$</td>
<td>true</td>
<td>[income$_1 = $income$_0 * 10$]</td>
</tr>
<tr>
<td>$W_3$</td>
<td>$e_2$</td>
<td>deductionEnabled$_0$ &amp; ...</td>
<td>income$_0 −$deduction$_0 &lt; 0$</td>
</tr>
<tr>
<td>$W_4$</td>
<td>$e_2$</td>
<td>deductionEnabled$_0$ &amp; ...</td>
<td>income$_0 −$deduction$_0 \leq 0$</td>
</tr>
<tr>
<td>$W_5$</td>
<td>$e_2$</td>
<td>$¬$deductionEnabled$_0$ &amp; ...</td>
<td>income$_0 −$deduction$_0 &lt; 0$</td>
</tr>
<tr>
<td>$W_6$</td>
<td>$e_2$</td>
<td>$¬$deductionEnabled$_0$ &amp; ...</td>
<td>income$_0 −$deduction$_0 \leq 0$</td>
</tr>
<tr>
<td>$W_7$</td>
<td>$e_3$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_8$</td>
<td>$e_4$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_9$</td>
<td>$e_5$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_{10}$</td>
<td>$e_6$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_{11}$</td>
<td>$e_7$</td>
<td>$¬$deductionEnabled$_0$</td>
<td>[deductionEnabled$_1 = true$]</td>
</tr>
<tr>
<td>$W_{12}$</td>
<td>$e_7$</td>
<td>deductionEnabled$_0$</td>
<td>[deductionEnabled$_1 = false$]</td>
</tr>
<tr>
<td>$W_{13}$</td>
<td>$e_8$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_{14}$</td>
<td>$e_9$</td>
<td>true</td>
<td>[]</td>
</tr>
<tr>
<td>$W_{15}$</td>
<td>$e_{10}$</td>
<td>true</td>
<td>[deduction$_1 = \alpha$]</td>
</tr>
</tbody>
</table>
of predefined symbolic inputs. Our symbolic summarization phase marks the entire application state as symbolic before exploring possible paths through an event handler. Each explored path in an event handler is represented using the triple $W = \langle pc, \sigma, \tau \rangle$, where $pc$ is the path condition of that path, $\sigma$ is the symbolic state mutated by executing the path, represented using a sparse map from memory locations to symbolic expressions, and $\tau$ is a log of bytecodes executed in the path. We denote such a triple as a path summary, and the set of all paths summaries for an event handler is denoted as an event handler summary.

Table 5.1 lists all path summaries for our running example, excluding the bytecode log. We represent memory locations using the variable names deductionEnabled, for the global setting enabling/disabling the deduction calculation set by $e_7$, deduction, for the deduction amount set by $e_{10}$, and income, for the income entered by the user by the many $e_1$ transitions. In practice, an alias analysis is needed to determine how memory locations are shared between event handlers.

The program entry $s$ has a single path summary, $W_1$, which initializes the shared state. Pressing the zero button on the entry screen appends a zero to the current number (i.e. multiplies by 10), which is represented by the $W_2$ path summary. Notice, we differentiate between shared state prior to executing a path summary (i.e. income$_0$ and after executing a path summary (i.e. income$_1$). The $e_2$ event handler has four path summaries, differentiated by their path conditions. The $e_3$, $e_4$, $e_5$, $e_6$, and $e_8$ events have no effect on the application state, as can be seen in their respective path summaries. The $e_7$ event handler has two path summaries, setting deductionEnabled to true if the value is false, and vice versa. Finally, the $e_{10}$ event handler has a single path summary setting the value of deduction to be equal to the symbolic event parameter $\alpha$.

Sequence Generation

Next, we use the event handler summaries and the UI model to generate event sequences using the search algorithm presented in Figure 5.3. Intuitively, the search algorithm maintains a prioritized worklist of partial sequences, and continuously prepends an anchor and a sequence of connectors to the partial sequence with the highest priority. This is done until a complete sequence is constructed. The algorithm operates on path summaries, such that anchors and connectors are path summaries, and partial sequences are sequences of path summaries.

Initially, the search algorithm initializes the worklist with path summaries that reach the target (lines 2–7), for example, $W_3$ and $W_5$ in our example. The algorithm then continuously selects the partial sequence from the worklist with the highest priority (line 9), and generates a list of extended partial sequences by prepending the selected partial sequence with anchors and con-
function sequenceSearch(target, summaries, model)
  worklist = ∅
  for W = (pc, σ, τ) ∈ summaries do
    if target ∈ τ then
      ENQUEUE(worklist, W)
  end if
  end for
  while worklist is not empty do
    partialSequence = DEQUEUE(worklist)
    extendedPartialSequences = empty list of sequences
    for anchor in ANCHORS(partialSequence, summaries, model) do
      for path in PATHS(anchro, partialSequence, summaries, model) do
        newPartialSequence = COMBINE(anchro, path, partialSequence)
        if isCOMPLETE(newPartialSequence) then
          potentialTestCase = EXTRACTTestCase(newPartialSequence)
          if REACHESTARGET(potentialTestCase, target) then
            return potentialTestCase
          end if
        end if
        end if
      end for
    end for
    ENQUEUE(worklist, extendedPartialSequences)
    PRIORITIZE(worklist, extendedPartialSequences)
  end while
  return no test case found
end function

Figure 5.3: The event sequence generation algorithm. The input target denotes the target of interest, summaries is the set of all event handler summaries, and model is the UI model of the application.

nectors (lines 11–22). The extended sequences are then inserted back into the worklist (line 23) and their priorities are updated (line 24). This process continues until the worklist is either empty (line 26), indicating that no solution exists, or a complete sequence is found (lines 14–19). The partial sequences are extended using the ANCHORS (line 11) and the PATHS (line 12) functions that identify anchors and sequences of connectors, respectively.

Anchors The ANCHORS function identifies the set of path summaries that affect the dependency set of the selected partial sequence. The path summaries are found through a breadth-first search of the UI model, starting from the first path summary in the partial sequence. The function also returns the program entry if it is reachable.

For example, the worklist initially contains the path summaries $W_3$ and $W_5$ for transition $e_2$ in Table 5.1. Let us assume the selected partial sequence contains $W_3$. The dependency set of the selected partial sequence is then \{deductionEnabled, income, deduction\} and possible anchors are the path summaries $W_1$, $W_2$, $W_{11}$, $W_{12}$, and $W_{15}$, since each of those
path summaries write to symbolic state in the dependency set, and are reachable in the UI model without passing through another anchor.

The Anchors function then filters away any of the identified path summaries that would yield an unsatisfiable execution if combined with the partial sequence, for example, if shared state written by an identified path summary makes one or more path conditions in the partial sequence unsatisfiable. In practice, we stitch together the path conditions and symbolic state of an identified path summary and partial sequence, creating one unified path condition. This path condition is then checked whether it is satisfiable using a constraint solver.

For example, if we combine the path summary $\mathcal{W}_{12}$ with the selected partial sequence $\mathcal{W}_3$, we get the combined path condition $\text{deductionEnabled}_{\mathcal{W}_{12}} \land \text{income} - \text{deduction} < 0$, where $\text{deductionEnabled}_{\mathcal{W}_{12}}$ is set to false by $\mathcal{W}_{12}$ (according to its symbolic state). A constraint solver will decide that this path condition is not satisfiable, and $\mathcal{W}_{12}$ would be discarded as an anchor.

In our example, the path summaries $\mathcal{W}_2$, $\mathcal{W}_{11}$, and $\mathcal{W}_{15}$ are thus returned by the Anchors function.

**Paths** The Paths function enumerates the possible acyclic sequences of connectors between an anchor and a partial sequence according to the UI model.

For example, between the anchor $\mathcal{W}_{11}$ and the selected partial sequence $\mathcal{W}_3$ two sequences of path summaries are found connecting the two: $\mathcal{W}_6$, and $\mathcal{W}_8 \cdot \mathcal{W}_9 \cdot \mathcal{W}_6$. Notice, other anchors may not be included in connector sequences, for example, $\mathcal{W}_{10}$, and paths should be acyclic (i.e. never follow the same transition twice).

Similar to the Anchors function, the Paths function checks whether the combined path from anchor to connectors to partial sequence is satisfiable for each identified sequence of connectors using a constraint solver. Only satisfiable paths are returned.

Partial sequences are extended until a complete path is found from the program entry to the target, and the combined path condition for the complete sequence is satisfiable. Such complete sequences are then converted into a concrete test case (i.e. an event sequence and event parameters) and applied to the tested application (lines 14–19). This is done to ensure that false positives are not returned because of over-approximation in the UI model. In our running example, the complete sequence would be $\mathcal{W}_1 \cdot \mathcal{W}_8 \cdot \mathcal{W}_9 \cdot \mathcal{W}_{11} \cdot \mathcal{W}_{13} \cdot \mathcal{W}_{15} \cdot \mathcal{W}_{14} \cdot \mathcal{W}_{10} \cdot \mathcal{W}_3$. 
5.3 Related Work

Concolic testing and symbolic execution have previously been applied to event-driven applications [7, 39, 62, 66, 85] for handling the event parameter problem. The state-of-the-art techniques either systematically explore all possible event sequences [7, 39], manually select event sequences [62], operate on unspecified event sequences [85], or operate on randomly generated sequences [100].

We are closely related to the work by Anand et al. [7] and Ganov et al. [39], who systematically generate longer and longer event sequences, one event at a time, and apply concolic testing on the event parameters of the generated sequences. However, we (1) use dependency analysis and UI models to extend event sequences with multiple events at each iteration, and (2) present an algorithm that is directed by a specific target, building event sequences in an backwards manner. Our combination of concolic testing, UI models, and dependency analysis has later been used by Li et al. [66] to conduct testing of web applications in an undirected (and forwards) manner.

Ma et al. [67] propose a technique for targeting specific branches in functions. They use symbolic execution to explore a function and find possible paths leading to a target. They then conduct an iterative backwards search using a call graph, to identify a possible chain of callers that leads from a program entry point to the target. Symbolic execution is continuously used to verify that any prepended callers allow the target to be reached. This approach is similar to our algorithm, with the difference that we use UI mod-
CHAPTER 5. TARGETED INPUT GENERATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Targets</th>
<th>Reached</th>
<th>Average size</th>
<th>Pruning of anchors</th>
</tr>
</thead>
<tbody>
<tr>
<td>TippyTipper</td>
<td>16</td>
<td>7</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>ConnectBot</td>
<td>42</td>
<td>16</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Munchlife</td>
<td>10</td>
<td>6</td>
<td>29</td>
<td>7</td>
</tr>
<tr>
<td>OpenManager</td>
<td>18</td>
<td>9</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>DieDroid</td>
<td>13</td>
<td>8</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 5.2: The table lists the benchmarks, the number of targets, number of targets reached by Collider, the average sequence length of complete sequences, average number of connector events in each sequence, and the percentage of anchors pruned from the search during sequence generation using the constraint solver. The numbers in the rightmost three columns are for the reached targets only.

els and dependency analysis, inspired by Arlt et al. [9], to prepend a partial sequence with multiple events in each iteration. Our approach allows us to quickly direct the search to interesting events (i.e. anchors), and use heuristics, such as the path connector heuristic, to prioritize our search.

5.4 Evaluation

We implement the proposed algorithm for Android mobile applications in our tool Collider. We reuse solver infrastructure from Symbolic Java PathFinder and use the debugger interface for the Android VM running on the Android emulator to observe and control concrete execution. Using Collider, we evaluate our proposed algorithm on a set of benchmarks listed in Table 5.4. We refer to Chapter 9 for details about the benchmarks and their selection. We manually construct a UI model for each benchmark application.

We conduct an experiment where we (1) apply two different testing techniques to automatically test the selected benchmarks, and (2) use Collider to reach any remaining challenging targets, that is, program points in the benchmarks not reached by the automated testing tools.

First, we use a crawler that fully explores the UI model, and the random testing tool Monkey, provided by the Android SDK, to automatically test our benchmarks. We iteratively increase the time budget for both tools until we observe a stabilization in the achieved coverage (i.e. increasing the time budget does not increase coverage). All uncovered branches in the benchmarks, excluding library code and dead code, is then used as challenging targets.

Next, we apply Collider on the challenging targets. Table 5.4 shows the result of running Collider on our benchmarks. The Targets and the Reached columns denote the number of targets and number of targets reached by Collider within some iteration limit, respectively. In total, Collider

\[\text{http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc}\]
reaches 46 out of 99 targets (46%). Manual inspection of the remaining 53 unreached targets shows that they are unreachable by COLLIDER because of restrictions in our concolic testing infrastructure, for example, because of missing support for string solving. Further inspection of the targets show that 50% of the targets require specific event parameters to be reached, while the remaining targets are reachable by constructing the right event sequence only. This shows that a technique for selecting event parameters, such as, concolic testing, is necessary for a subset of the targets.

We also note that disabling our prioritization heuristics reduces the set of reached targets from 46 to 21 while allowing for a budget of 10 times as many iterations.

Finally, we observe that, on average, 8–29 events are used to reach the selected targets, depending on the benchmark. Compared to, for example, the work by Anand et al. [7] and Ganov et al. [39] who are only able to conduct concolic testing of event sequences of length 3 and 4, respectively. This indicates that our targeted approach will be able to reach targets that require substantially longer event sequences than a systematic undirected search.

5.5 Summary

In summary, we have presented an algorithm for targeted event sequence generation and implemented the algorithm for Android mobile applications in our tool COLLIDER.

The proposed solution shows promising results in our evaluation by reaching targets in applications that require long event sequences and proper selection of event parameters to be covered. For example, we are able to reach targets that require event sequences of up to, on average, 8–29 events, that is considerably longer than related work on systematic exploration of event sequences with concolic testing of event parameters that only reach sequences of length up to 4 events. However, we only apply this method to a small set of benchmarks, and further experimentation is needed to determine if these results are representative.

In our experiments, we are able to reach 46% of our targets. Manual inspection of the unreached targets indicate that the reason for them to be unreachable is because of lacking support for, for example, string constraints and symbolic objects in our prototype. Further work is needed to verify if an improved implementation would be able to improve results in practice. Likewise, there is a large design space for additional heuristics and modifications to the heuristics we propose, that could further improve speed and results.

We conclude that our proposed method of running one or more automated testing methods, followed by a targeted approach of automated testing to reach remaining targets, is viable. Furthermore, the general idea of combining multiple testing techniques with different characteristics is interesting and should be explored further in future work.
Chapter 6

Server Interface Descriptions

In this chapter, we present our work on server interface descriptions for automated testing of web applications, including algorithms for learning such descriptions dynamically, documented in our paper “Server interface descriptions for automated testing of javascript web applications” Jensen et al. [58] included in Chapter 10. We refer to this work collectively as the AIL project.

The purpose of this chapter is to give an intuitive overview of our work, and provide an extended example illustrating our learning algorithm, compared to Chapter 10. In the following, we motivate our work and outline our contributions (Section 6.1), explain our algorithm for learning server interface descriptions (Section 6.2), give an overview of related work (Section 6.3), and finally present our evaluation of our work (Section 6.4).

6.1 Motivation

Artzi et al. [11] presents the Artemis tool for automated testing of web applications using feedback directed testing, presented in Section 3.2.1. Artemis tests the JavaScript client-side of a web application by generating sequences of events and selecting event parameters based on feedback from executing other inputs. However, as observed by Artzi et al. [11], it is not always possible to test the client-side of a web application in isolation without taking into account its server-side. The client-side uses servers to, for example, fetch static resources, store state, validate input, and implement business logic. The server-side acts as additional application state that is shared between multiple clients.

The solution suggested by Artzi et al. [11] is to use real servers when testing the client-side, and resetting any server state each time the state of the client-side is reset (i.e. when executing a new event sequence). This assumes that only one client is using a server at any point in time. However, this approach (1) increases setup cost for automated testing since the any servers must be implemented, configured, and deployed before testing; (2) the servers must be initialized with proper data to provide representative responses; and (3) hooks
must be written for the automated tester to reset server state.

In general, the client-server communication can be handled as an instance of the event parameter selection problem. For example, the `readyStateChanged` event processed when a response is given from a server can be parameterized with response data from the server. The challenge when testing the client-side is then how to select such parameters. Here, Artzi et al. \[11]\ queries a real server to select the parameter.

We propose to use a specification of the client-server communication, henceforth denoted as a server interface description. Using such a description, we hypothesise that client-side web applications can be automatically tested in isolation, without the need of a real server by, for example, generating random responses based on the description. Furthermore, such a description is useful in a number of ways, for example, for documentation, and for implementing the server- or client-side separately.

However, writing server interface descriptions from scratch for existing applications is time consuming and is not aligned with the purpose of reducing developer interaction through automated testing. Thus, we propose to use a learning algorithm to infer server interface descriptions based on concrete requests and responses between clients and servers. However, as we will show later, it is non-trivial to infer server interface descriptions, and it may be necessary for developers to adjust inferred descriptions. Thus, we envision that developers would either (1) write a server interface description and maintain it as they are developing their web application, or (2) use our learning algorithm to infer a server interface description for an existing web application and then maintain the description going forward. In both scenarios, the server interface description can then be used for automated testing of the client-side of the web application, in addition to other uses of such descriptions.

**Our Contributions** In summary, we bring to the attention that server interface descriptions are useful in many aspects of development and testing of web applications, and show such descriptions can be used to support automated testing of the client-side of web applications. Our contributions are as follows:

- We present a language, Ajax server Interface description Language (AIL), for writing server interface descriptions in.
- We extend an existing tool for automated testing of web applications, Artemis, with support for server interface descriptions written in AIL.
- We propose an algorithm for learning server interface descriptions based on concrete HTTP request and response pairs observed from, for example, user sessions.
- We evaluate our learning algorithm and the effects of using server interface descriptions in Artemis on a set of real-world web applications.
6.2 Solution

In the following section, we present our AIL language and explain our learning algorithm for inferring server interface descriptions from communication between clients and servers.

6.2.1 The AIL Language

We present the Ajax server Interface description Language (AIL), a language for server interface descriptions. A server interface description written in AIL is denoted as an AIL description. The main purpose of using AIL instead of using an existing language for client-server communication such as WADL [49], is to simplify the explanation and presentation of our learning algorithm.

Figure 6.1 shows an example AIL description. An AIL description starts with the base URL of the service (line 1), followed by operation descriptions, one for each endpoint of the server. An operation description is declared using the format

\[ \text{request} : \text{response} \]

where request, denoted the request pattern, describes the HTTP request sent by the client, and response describes the response from the server. The full syntax for the request pattern is

\[ \text{method} \; \text{path}(\text{parameters}) \]

where method is a HTTP request method (i.e. GET, POST, UPDATE, PUT, etc.), path represents the URL path of the endpoint, and parameters represents the URL query string. As an example, the second operation description (line 3) in Figure 6.1 matches a GET request to the URL http://www.example.org/author/?name=JohnDoe, where author/ is the path, and name=JohnDoe is the query string.

Request patterns must be disjoint, such that a concrete request may not match multiple operation descriptions in the same AIL description.

AIL supports data type annotations for individual parts of a path and the values of every parameter, such as, the wildcard * in the name:* parameter in Figure 6.1 line 3. Supported data types include constant strings, wildcards,

1 URL http://www.example.org
2 GET news() : @items.json
3 GET author(name:*): @author.json
4 GET comments(newsId:*): @comments.json
5 POST users/login(user:*, pwd:*): @token.json

Figure 6.1: An example server interface description written in AIL.
6.2.2 Learning Algorithm

The learning algorithm takes as input, \( I \), a finite set of request and response pairs \( \langle r, s \rangle \). These can be collected from either manual interaction with a web application, running system tests, or by observing requests and responses in a live system. Since the learning algorithm operates on concrete observations, the quality of the resulting AIL description depends on the diversity of \( I \).

We observe that there exist multiple server interface description for any server. As an example, see the two AIL descriptions shown in Figure 6.3, where the two operation descriptions in the first AIL description are generalized by the second AIL description. Both solutions are equally valid, and the desired solution is completely subjective. In this work, we want our learning algorithm to create distinct operation descriptions for each logical endpoint, where a logical endpoint represents a distinct computation exposed by the server that can be referenced uniquely by a URL. In practice, servers are often

![Figure 6.2: An example of a JSON schema.](image)

a reference to an external JSON schema or RelaxNG schema file, or a union of the types (\|). Furthermore, cardinality annotations are also supported for parameters that are optional (?) or occur zero or more times (+).

Finally, AIL descriptions may define the response to be any datatype (e.g. a JSON schema) or void if no response is given to the user besides a status code. As an example, the operation description in Figure 6.1 line 3 returns a JSON response conforming to the JSON schema `author.json`. An example JSON schema is shown in Figure 6.2. The JSON schema declares that the response is a JSON object with two properties: a name property with a string value, and an optional email property with a string value.

![Figure 6.3: Two equivalent AIL descriptions with different levels of generalization.](image)
implemented with a separate component for each logical endpoint and a controller component routing requests to individual endpoints. We want to reflect this common pattern in our server interface descriptions. However, automatically identifying endpoints from URLs is non-trivial. As an example, the URLs `admin/?action=delete&id=14` and `admin/?action=publish&id=20` point to two different logical endpoints, but deciding that `action` is used to distinguish between endpoints while `id` simply conveys a value, is non-trivial.

Properties

In order to guide the learning algorithm we use the following four properties to characterize the AIL description, $d$, that we want to learn.

Completeness Any concrete input $(r, s) \in I$ is covered by the learned AIL description, $d$. That is, the request, $r$ matches an operation description in $d$ and the response, $s$, is consistent with the response type for the matching operation in $d$.

Disjointness The request patterns in the AIL description must be disjoint, such that any concrete request only matches a single operation description in $d$.

Intuitively, the completeness and disjointness properties ensure that (1) all inputs in $I$ are represented by $d$ and (2) $d$ is not ambiguous (i.e. there does not exist any request that matches two operation descriptions).

Precision The learned AIL description, $d$, should reflect the observed input $I$ as closely as possible. That is, we value a solution $d$ that expresses fewer concrete inputs while being complete.

Conciseness The learned AIL description, $d$, should be small, measured in the number of operation descriptions.

Intuitively, the precision property values AIL descriptions that are as close to $I$ as possible, for example the AIL description in Figure 6.3a, while the conciseness property values generalization, for example the AIL description in Figure 6.3b. We do not want to over-generalize the AIL description, nor do we want to make it too specialized. Thus, we want to find some middle ground between the two extremes.

We propose a learning algorithm that occupies the middle ground between precision and conciseness. However, we stress that the result is still subjective and dependent on the diversity of $I$. Thus, it may be necessary to use the learning algorithm as part of a semi-automated process where the learning algorithm infers an initial server interface description that is then checked and corrected through a manual process. We evaluate the amount of corrections necessary in Section 6.4.
Algorithm

Our learning algorithm consists of two phases: data clustering, clustering concrete observations in $I$ into a single cluster for each operation description, and pattern generation, translating each cluster into a syntactically correct AIL operation description, $d$, and external schema files, such as, JSON schema files. Data clustering is further divided into two steps: response data clustering and request data clustering. Notice, that we cluster based on response data prior to clustering according to request data. We do this because we find that clustering according to response data gives a result close to our desired result, using clustering on request data to refine the result. In the following, we explain these two steps using the example request and response pairs provided in Figure 6.4.

Response Data Clustering First, we make the observation that inputs $(r, s) \in I$ with identically structured responses often belong to the same logical endpoint.

Intuitively, response data clustering is implemented by identifying the type of a response, for example, JSON, HTML, XML, etc., and comparing the structure of identically typed responses. We implement and experiment with
Cluster 1 (JSON: {"id": string, 'name': string, 'stories': string})

GET author?name=alice
GET author?name=bob

Cluster 2 (JSON: {"id": string, 'name': string, 'email': string, 'stories': string})

GET author?name=charlie
GET author?name=eve

Cluster 3 (JSON: {'author': string, 'comment': string})

GET comments?newsId=1
GET comments?newsId=2

Cluster 4 (JSON: [{'id': string, 'title': string}])

POST news/read

Figure 6.5: Example clustering of request and response pairs in Figure 6.4 following the response data clustering step.

clustering based on a type abstraction of the JSON response, i.e. the JSON response {"success": true, "item": [1, 2, 3]} is abstracted into {"success": bool, "item": [int]}.

As an example, Figure 6.5 shows the request and response pairs from Figure 6.4 clustered according to the structure of their response. A total of four clusters are identified. The first two clusters both have JSON objects with an id, name, and stories property, but the first cluster is missing the email property. The last cluster represents an arbitrarily long list of objects with id and title properties.

Request Data Clustering  Next, the clusters are refined such that unambiguous request patterns can be constructed for each cluster, to satisfy the disjointness property.

First, we need to introduce some terminology. We say that the path and parameters of a concrete request, $r$, has a signature of features. As an example, the concrete request author/?name=foo has the signature \{GET, #0, #name\} where GET, #0, and #name are features. Notice, each path fragment in the path is represented by an increasing number (i.e. #0), while keys in the query are represented using their name. The request method is added as the first fragment in the signature.

It is trivial to convert signatures into request patterns by selecting an assignment of either concrete values or wildcards for each feature in a signature, excluding the request method, and then generating request patterns according to that assignment. As an example, the signature \{GET, #0, #name\} allows for the assignments and request patterns shown in Table 6.1.
CHAPTER 6. SERVER INTERFACE DESCRIPTIONS

Table 6.1: Possible assignments of constant values and wildcards for the signature \{GET, #0, #name\}, and resulting request patterns for the concrete requests shown in Figure 6.4.

<table>
<thead>
<tr>
<th>#0</th>
<th>#name</th>
<th>Request patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>GET <em>(name:</em>)</td>
</tr>
</tbody>
</table>
| *    | concrete | GET *(name:alice)  
          | GET *(name:bob)  
          | GET *(name:charlie)  
          | GET *(name:eve) |
| concrete    | *    | GET author(name:*)                                    |
| concrete    | concrete | GET author(name:alice)  
          | GET author(name:bob)  
          | GET author(name:charlie)  
          | GET author(name:eve) |

The request data clustering step refines the clusters identified in the response data clustering step (Figure 6.5) such that each new cluster can be assigned a request pattern, and the assigned request patterns satisfy the disjointness property. Here, we use the key insight that the result from response data clustering is very close to our desired result. Thus, we want to change these clusters as little as possible in order to satisfy the disjointness property. We refine the clusters by:

1. Splitting clusters such that all observed requests in a cluster have the same signature.

2. Selecting an assignment of concrete values and wildcards to each signature, such that the resulting request patterns for each cluster are disjoint. It may be necessary to further split or merge clusters for this to be possible.

Our solution is to evaluate all possible assignments of either concrete values or wildcards in signatures when generating request patterns, and comparing the solutions using a cost function, where the cost increases with the number of splits and merges required to create request patterns.

Figure 6.6 shows the clusters from Figure 6.5 after they have been refined. Clusters 1 and 2 in Figure 6.5 have the signature \{GET, #0, name\}. The two clusters will either be merged into one, if the name feature is assigned a wildcard, or merged into four clusters, if name is assigned constant values (as we have shown earlier). The cost of merging the two clusters is 1, while the cost of splitting each of the two clusters sums to 2. Thus, the two clusters are merged. If multiple solutions have the same cost, we select the solution with the most constant values. Finally, cluster 3 has the signature \{GET, #0,
6.3. RELATED WORK

Cluster A (JSON: `{"id": string, "name": string, "stories": string} | `{"id": string, "name": string, "email": string "stories": string}`)

GET author?name=*  

Cluster B (JSON: `{"author": string, "comment": string}`)

GET comments?newsId=*  

Cluster C (JSON: `[{"id": string, "title": string}]`)  

POST news/read

Figure 6.6: Example clustering of request and response pairs in Figure 6.4 following the request data clustering step.

newsId}, and cluster 4 has the signature `{POST, #0, #1}`. Cluster 3 and 4 do not need to be refined, since no other clusters share their signature (i.e. request patterns can be generated without splitting the clusters).

Generating AIL Descriptions Finally, for each cluster we generate an operation description by (1) generating a request pattern as explained above using the selected assignment of concrete values and wildcards, and (2) by generating schema based on the response. We omit further details on this process.

6.3 Related Work

There exist a number of standards for client-server specifications in the domain of web applications, such as, WADL, WSDL, REST, and SOAP. Thus, the proposed AIL language is not novel in itself, and is in fact closely related to WADL [49], but with a simpler syntax and support for JSON. We use AIL partly because it represents a simple to understand specification for server interface descriptions, and it closely relates to our learning algorithm and the properties we want any learned specification to satisfy. The AIL language is designed to express modern web-service APIs that we are familiar with, such as, the APIs described on Google’s API Explorer website[1].

The AIL language specifies the possible responses from the operation description, but it does not describe any constraints between operation descriptions. Hallé et al. [52] propose contracts to describe client-server communication and specifically temporal aspects of an API, for example, that a specific request to the server should always happen before another request. However, in our work we choose to over-approximate such constraints, assuming that any sequence of requests may be made to the server.

CHAPTER 6. SERVER INTERFACE DESCRIPTIONS

Halfond and Orso [50] and Halfond et al. [51] use symbolic execution to explore the server-side of web applications written in Java, to construct a specification for the client-server communication. Specifically, they mark any input parameters (i.e., key-value pairs in the query string) symbolic and translate the resulting path conditions into specifications. Each specification consists of a set of input parameters and constraints on the input parameters. This approach requires a substantial engineering effort for each target language and framework, to implement a symbolic execution infrastructure, while our black-box approach is agnostic to the server-side language and framework. However, this is at the cost of less precision, since we cannot guarantee that the input used for learning is sufficient.

Some related work focuses on client-server communication facilitated through web forms, such as, Benedikt et al. [13] who use static analysis of JavaScript to generate client-server communication specifications, and Fisher II et al. [37] who semi-automatically submit values to a single form and classify the result to create a specification of valid input parameters. We differ from this work by learning the interface of a set of distinct endpoints, while Fisher II et al. [37] and Benedikt et al. [13] focus on learning the accepted parameters for a single form only.

Broder et al. [17] suggest a method for clustering syntactically similar web pages using a distance measure. Initially, we used a similar approach for response data clustering of JSON. However, we found that the threshold of 0, i.e., not allowing any difference in structure, yielded the best results. Thus, we discarded any usage of distance measures.

6.4 Evaluation

We conduct two experiments, one to evaluate how server interface descriptions can be used in combination with the Artemis tool and one to evaluate our learning algorithm.

Table 6.2 lists the benchmarks we use, their implementation language and framework, and size of the benchmarks in lines of code (not including frameworks and libraries).

6.4.1 Automated Testing Using Server Interface Descriptions

We extend Artemis, an automated testing tool for web applications described in Section 3.2.1 with support for AIL descriptions and evaluate how this affects the testing capabilities of Artemis. Specifically, we evaluate how AIL descriptions allow Artemis to test web applications without accessing a server, reducing the effort required to, for example, install, configure, and initialize the server.

We modify Artemis to intercept any network requests made by WebKit and route them through a mock server. The mock server takes an AIL description as
6.4. EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of Code</th>
<th>Client Framework</th>
<th>Server Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleajax</td>
<td>79</td>
<td>jQuery</td>
<td>Python (Django)</td>
</tr>
<tr>
<td>resume</td>
<td>244</td>
<td>Flapjax</td>
<td>Python</td>
</tr>
<tr>
<td>globetrotter</td>
<td>347</td>
<td>jQuery</td>
<td>Java (JWIG)</td>
</tr>
<tr>
<td>impresspages</td>
<td>558</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
<tr>
<td>buggenie</td>
<td>3,716</td>
<td>Prototype</td>
<td>PHP</td>
</tr>
<tr>
<td>ellinder</td>
<td>6,724</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
<tr>
<td>tomatocart</td>
<td>8,817</td>
<td>Prototype</td>
<td>PHP</td>
</tr>
<tr>
<td>eyeos</td>
<td>17,629</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
</tbody>
</table>

Table 6.2: Benchmark applications used in the AIL evaluation. See Chapter 10 for an explanation of each benchmark.

input and uses it to return structurally correct responses with random values. The mock server can also be configured to serve static resources, dummy files, or pass the request through to a live server if no match was found in the AIL description. Artemis recognizes readyStateChange events (i.e. Ajax responses) as controllable events parameterized by a network response, and generates event sequences with explicit readyStateChange events and network responses from the mock server.

Unfortunately, Artemis is not able to run on all of the benchmarks used for evaluating our learning algorithm, as listed in Table 6.2. Thus, Artemis is only run on the subset of benchmarks listed in Table 6.3. For each benchmark we run Artemis in 5 different configurations: using a fresh install of the server with an empty database (EmptyDB), a database manually populated to maximize test coverage (FullDB), a server returning completely random data (Random), and our mock-server (AIL). Table 6.3 shows the code coverage measured in number of lines when running Artemis for 300 iterations on each benchmark in each configuration. The Init column shows the code coverage after loading the application prior to any testing.

We observe that populating the database yields higher code coverage compared to testing on an empty database. The globetrotter and buggenie benchmarks were automatically tested in a way such that they had no distinction between a populated and an empty database. Furthermore, we observe that code coverage from testing using a populated database and AIL descriptions is comparable, indicating that returning structurally correct responses is a promising solution. In comparison, we observe that returning completely random responses (but still valid JSON responses) yield a low code coverage.

6.4.2 Learning Server Interface Descriptions

Next, we evaluate our learning algorithm. We manually explore each benchmark while recording any client-server communication. The manual exploration is black-box, that is, we have not inspected the client or server code prior to exploring the web application.
Table 6.3: Code coverage obtained by Artemis with a budget of 300 test inputs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Init</th>
<th>EmptyDB</th>
<th>FullDB</th>
<th>Random</th>
<th>AIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleajax</td>
<td>22</td>
<td>30</td>
<td>62</td>
<td>60</td>
<td>62</td>
</tr>
<tr>
<td>resume</td>
<td>12</td>
<td>105</td>
<td>108</td>
<td>14</td>
<td>113</td>
</tr>
<tr>
<td>globetrotter</td>
<td>10</td>
<td>-</td>
<td>180</td>
<td>17</td>
<td>205</td>
</tr>
<tr>
<td>buggenie</td>
<td>662</td>
<td>-</td>
<td>1,322</td>
<td>1,138</td>
<td>1,308</td>
</tr>
<tr>
<td>elfinder</td>
<td>571</td>
<td>1,236</td>
<td>1,337</td>
<td>665</td>
<td>1,366</td>
</tr>
</tbody>
</table>

Table 6.4: Number of sample request and response pairs used for AIL learning, time used for collecting sample data and learning AIL descriptions, and results from comparing the learned AIL descriptions with the manually written ones.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Samples</th>
<th>Sampling time</th>
<th>Learning time</th>
<th>Match</th>
<th>1 → N</th>
<th>N → 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleajax</td>
<td>70</td>
<td>3m</td>
<td>74ms</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>resume</td>
<td>128</td>
<td>9m</td>
<td>111ms</td>
<td>12</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>globetrotter</td>
<td>97</td>
<td>8m</td>
<td>84ms</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>impresspages</td>
<td>179</td>
<td>6m</td>
<td>224ms</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>buggenie</td>
<td>210</td>
<td>6m</td>
<td>118ms</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>elfinder</td>
<td>181</td>
<td>6m</td>
<td>124ms</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>tomatocart</td>
<td>370</td>
<td>8m</td>
<td>153ms</td>
<td>22</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>eyeos</td>
<td>611</td>
<td>6m</td>
<td>213ms</td>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>85</strong></td>
<td><strong>17</strong></td>
<td><strong>1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4 lists the benchmarks, the number of observed samples used by the learning algorithm, time spent collecting the samples, the running time for our learning algorithm, and finally the result of manually classifying the quality of the learned AIL description.

We manually classify the quality of learned AIL descriptions by comparing the AIL descriptions with the implementation of the server, manually inspecting the server to identify logical endpoints. The three last columns in Table 6.4 count the number of operation descriptions that matched directly with manually identified endpoints (Match), the number of cases where an identified endpoint is mapped to multiple learned operation descriptions (1 → N), and the number of cases where multiple identified endpoints are mapped to the same learned operation description (N → 1).

We observe that the majority of handlers are correctly learned. Manual inspection of the 1 → N cases show that these are easy to identify manually and correct. They are often caused by request patterns with constant values for ids or session ids, where a wildcard should have been used. Additional samples would reduce such errors further. The N → 1 case is a bit more problematic, because it is not always easy to spot that a wildcard is used incorrectly where it should differentiate operation descriptions, fortunately, this case is not as common.
6.5 Summary

We highlight the usefulness of server interface descriptions for specifying the communication between the client- and server of web applications, and show how such descriptions may improve a tool for automated testing of web applications. We present a small language, AIL, for writing server interface descriptions, and we recognize the fact that server interface descriptions may be tedious to write manually, thus proposing a learning algorithm for inferring AIL descriptions.

We evaluate the effect of using AIL descriptions for testing, and find the results to be similar to testing using a real server with a populated database, but without the need of deploying a real server. Results from our experiments are encouraging. Furthermore, in our evaluation, we generated structurally correct responses with random values. Results may be improved further if more advanced techniques, such as fuzz testing, are used for generating responses based on the descriptions.

Finally, we show that our learning algorithm is able to infer AIL specifications close to a correct result, and that the necessary corrections to the learned AIL descriptions are trivial. Again, these results are encouraging, and indicate that our algorithm would be useful for supporting developers in server interface descriptions with less effort compared to writing the descriptions from scratch.

We conclude that server interface descriptions are useful in connection with automated testing and is a useful alternative to using real servers as done by Artzi et al. [11], and that learning algorithms can be successfully used for supporting developers in constructing such server interface descriptions.
Chapter 7

Conclusion

As outlined in Chapter 1, the goal of this PhD dissertation is to contribute to the state-of-the-art in automated testing of event-driven applications. Specifically, we identify a number of challenges in automated testing of event-driven applications, and present contributions within the sequence generation problem, the event parameter selection problem, and the oracle problem. These contributions are grouped into three larger projects:

- In Chapter 4, we proposed an algorithm for stateless model checking of event-driven applications with partial-order reduction, and show how this algorithm can be used for systematic exploration of event sequences. Our model checking algorithm is applicable to event-driven applications in general.

  We instantiate our model checking algorithm for web applications, and use it to uncover timing related failures. Furthermore, we propose a number of heuristics for automatically classifying whether an event sequence uncovers a bug. Our evaluation shows that $R^4$ improves upon the state-of-the-art by reducing the number of harmless event sequences that needs to be manually inspected, and providing a concrete witness of an uncovered bug to users.

- In Chapter 5, we proposed an algorithm for targeted event sequence generation using concolic testing capable of generating event sequences and selecting event parameters that uncover a specific target branch in an event-driven application. We propose that such a technique could be used in combination with other testing techniques by targeting parts of an application that the other techniques were unable to reach, using a combination of different techniques to cover different parts of an application when testing and complement each other.

  We instantiate our algorithm for Android mobile applications, and use it to reach program points in applications that were not reached by systematic crawling of UI models and random testing. Our evaluation of
our algorithm shows encouraging results, even though we find that future work is needed in improving our implementation with, for example, support for solving string constraints to find the limits of our algorithm.

• In Chapter 6, we proposed the usage of server interface descriptions that can be used to, for example, remove the need of installing and controlling real servers when conducting automated testing of JavaScript client-side web applications. Furthermore, we proposed a learning algorithm for generating server interface descriptions from observations of concrete requests and responses, reducing the workload of developers when writing server interface descriptions for existing applications.

We implement the learning algorithm for generating server interface descriptions written in AIL, and add support for AIL descriptions to the automated testing tool for web applications, Artemis. Our evaluation shows that replacing real servers with mock servers returning structurally correct responses can maintain the same level of coverage when testing, and it is feasible to automatically infer server interface descriptions, while only requiring minor corrections by developers.

The proposed contributions do not solve the problem of automated testing for event driven applications, but they do push the state-of-the-art for certain identified challenges and contexts. One central challenge is the potentially infinite input space caused by, for example, infinite event sequences and large input domains for event parameters. We propose two techniques, our stateless model checking algorithm and directed event sequence generation, that both aim at testing within this space of possible inputs. However, the state-of-the-art, including our work, is still limited to testing event-driven applications while assuming that parts of the input domain remains unchanged, for example, assuming that a web application is executed by a specific execution environment. This problem may be mitigated by combining multiple testing techniques. It is also an open question whether any of the existing techniques scale as applications increase in size and complexity. However, recent work, such as our R² project, shows that some of these techniques are applicable to at least non-trivial real-world websites.

We conclude that automated testing is a viable approach for identifying bugs in event-driven applications. The contributions presented in this dissertation improve upon the state-of-the-art of automated testing of event-driven applications, even though there is still space for future research.
Part II

Publications & Manuscripts
Chapter 8

Stateless Model Checking of Event-Driven Applications

Abstract

Modern event-driven applications, such as, web pages and mobile applications, heavily rely on asynchrony to ensure smooth end-user experience. Unfortunately, even though these applications are executed by a single event-loop thread, they can still exhibit nondeterministic behaviors depending on the execution order of interfering asynchronous events. As in classic shared-memory concurrency, this nondeterminism makes it challenging to discover errors that manifest only in specific schedules of events.

In this work we propose the first stateless model checker for event-driven applications, called $R_4$. Our algorithm systematically explores the nondeterminism in the application and concisely exposes its overall effect, which is useful in a range of debugging scenarios, including bug discovery and repair. The algorithm builds on a combination of three key insights: i) a dynamic partial order reduction (DPOR) technique for reducing the search space, tailored to the domain of event-driven applications, ii) conflict-reversal bounding based on a hypothesis that most errors occur with a small number of event reorderings, and iii) approximate replay of entire sequences, which is critical for separating harmless from harmful nondeterminism.

We instantiate $R_4$ for the domain of client-side web applications and use it to analyze event interference in a number of real-world programs. The experimental results indicate that the precision and overall exploration capabilities of our system significantly exceed the existing techniques.

8.1 Introduction

Client-side computing platforms, as web pages and mobile applications, use an event-driven execution model to handle a diverse set of external events in a
responsive manner. These events include timers, network communication, and user-triggered actions (e.g., clicking a button on a web page or a smartphone). Even though such applications run single-threaded, their execution is sensitive to the precise timing of events, which is not fully controlled by the user. As a consequence, event handlers of these applications can run in a nondeterministic order. Further, event handlers frequently access shared memory. As a result, it is possible that two event handlers interfere and can execute in any order, leading to potentially nondeterministic results.

State-of-the-art To address this challenge in the setting of event-driven applications, recent works have proposed mechanisms for going beyond ordinary testing and detecting sources of nondeterminism where two events interfere and can execute in any order [55, 57, 70, 96, 98]. While detecting interference is a useful building block, it still lacks critical analysis capabilities. First, it does not explore new schedules of events, which is critical for finding errors that occur only in specific schedules. Second, it cannot easily classify whether the detected interference is harmful or not. Indeed, despite filtering techniques, state-of-the-art analyzers [98] report too many false positives. Approaches that do explore more than one schedule [54] suffer from serious drawbacks: they do not detect interference and thus may keep exploring equivalent schedules, they are inherently unable to report the primary cause of the nondeterminism (making it difficult to fix errors), and they provide no meaningful guarantees on the explored schedules.

In the setting of shared-memory programming, the problem of systematically exploring nondeterminism in realistic concurrent applications is addressed via an elegant technique referred to as stateless model-checking [41]. A key benefit of this approach is that it does not require storing the entire program state, which would be infeasible for realistic applications. Due to its effectiveness, the technique has been implemented in various tools targeting concurrency testing [42, 86].

This work In this work, we present the first stateless model checker for event-driven applications, called $R^4$. Our algorithm consists of four phases: (1) Record: execute a given event sequence while monitoring all nondeterministic choices. This initial event sequence may come from a user, a test suite, or an automated testing tool (e.g., [11, 69]). (2) Reorder: construct alternative event sequences by systematically reordering events. Here we adapt classic techniques, such as, dynamic partial order reduction (DPOR) [38], to the domain of event-driven applications and combine these with conflict-reversal bounding. This step also leverages advanced conflict and race detection algorithms. To help programmers find the primary cause of harmful nondeterminism, we generate sequences with few changes compared to the initial one. (3) Replay: execute alternative event sequences by introducing a notion of
approximate replay, allowing us to replay entire sequences even when events in the sequence become disabled or new events appear. (4) *Report:* analyze the consequences of the reorderings and report the ones that are most likely to indicate errors.

**Contributions** The main contributions of this paper are:

- We present the $R^4$ algorithm for stateless model-checking of event-driven applications. Our algorithm adapts DPOR to the event-driven setting and supports *conflict-reversal bounding* for controlling the number of changes in an explored sequence and *approximate replay* for reducing divergence from the initial execution.

- We instantiate $R^4$ to the domain of client-side JavaScript web applications and provide a complete end-to-end implementation including both an integration with a WebKit browser as well as state-of-the-art conflict detection techniques.

- We evaluate $R^4$ on a set of real-world JavaScript web applications and demonstrate experimentally that it can systematically explore nondeterminism to detect errors with higher precision than prior work. Not only is $R^4$ capable of producing concrete witnesses that explain the consequences of alternative event schedules; it also shows that 60% of the warnings produced by EVENTRACER are harmless. We additionally find that WAVE reports an overwhelming amount of false positives compared to $R^4$.

**Outline** In Section 8.2 we provide an informal overview of our approach illustrating the key concepts. Section 8.3 introduces the formal notation that we use in Section 8.4 where our model-checking algorithm is presented in more detail. Our implementation and experimental evaluation are described in Section 8.5. Related work is discussed in Section 8.6, and we conclude in Section 8.7.

### 8.2 Overview of $R^4$

We begin with an informal overview of $R^4$ using an illustrative example. Although our model checking algorithm works for event-driven applications in general, the implementation and examples focus on client-side web applications.

**Illustrative example** Figure 8.1 shows an example of a JavaScript web application. It contains a slide show widget that uses deferred loading of images to minimize the initial load time and thereby maximize responsiveness.
CHAPTER 8. \textit{R}^4

1 \texttt{<!DOCTYPE html>}
2 \texttt{<html lang="en">}
3 \texttt{<head>}
4 \texttt{<script>}
5 \texttt{var queue = Array();}
6 \texttt{function lazyLoad(src) {}
7 \texttt{var img = new Image();}
8 \texttt{img.onload = function() {
9 \texttt{queue.push(img);}
10 \texttt{}};
11 \texttt{img.src = src;}
12 \texttt{}};
13 \texttt{function showNextImage() {
14 \texttt{if (queue.length == 0) {
15 \texttt{... show loading image ...}
16 \texttt{setTimeout(showNextImage, 100);}
17 \texttt{}} else {
18 \texttt{... replace the current image with}
19 \texttt{the next image in the queue ...}
20 \texttt{}}
21 \texttt{window.onload = function() {
22 \texttt{lazyLoad("image2.png");}
23 \texttt{lazyLoad("image3.png");}
24 \texttt{...}
25 \texttt{}};
26 \texttt{</script>}
27 \texttt{</head>}
28 \texttt{<body>}
29 \texttt{<button onclick="showNextImage();">Next</button>}
30 \texttt{<img id="slideshow" src="image1.png"/>}
31 \texttt{</body></html>}

Figure 8.1: A JavaScript application with nondeterminism.

The widget requests the images from the server when the page loads (lines 21–25). Each time an image is received, it is added to a queue (line 8). Whenever the user clicks the \textit{Next} button (line 28), the current image is replaced by the next one from the queue (lines 17–18), provided that one is available. If the queue is empty because the next image has not yet arrived, a loading indicator is shown, and a timer is set to retry after 100ms (lines 14–15). As we will see below, this example already contains much nondeterminism making it a challenging program analysis problem.

In what follows, we will illustrate the operation of each of the four phases of \textit{R}^4 on the running example.

8.2.1 Reordering Events

Given an initial execution of the applications, a key challenge is how to construct alternative event sequences (naively trying all possible event reorderings is infeasible). To accomplish this task effectively and reduce the search
space, we adapt and extend the DPOR algorithm [38], originally designed for stateless model checking of traditional concurrent programs, to the domain of event-driven applications. In particular, to determine backtracking points, the traditional DPOR algorithm works in a forward manner by examining all transitions from a given state and comparing them against already executed transitions. However, precisely predicting how a given transition affects the current state is generally not possible in an event-driven application. The reason is that the size of the (atomic) transition can comprise thousands of statements (i.e., the instructions of an event handler). Instead, our DPOR variant only works with the past and compares transitions that have already been executed.

At any point in time, our algorithm maintains an event sequence from the initial state of the application to the current point in the exploration. An example of such a sequence (consisting of five events) obtained from running our example is shown in Figure 8.2a. In this sequence, called $Q_1$, an image is first loaded (line 8), then the user clicks a button twice (line 28), then another image is loaded (line 8) and finally a timer event is executed (line 15).

To explore the nondeterminism present in such an execution, the algorithm performs two steps: selection of relevant conflicts and creation of new event sequences based on those conflicts.

**Step 1: Selection of relevant conflicts** In the first step, we analyze the current sequence for pairs of relevant conflicting events. Intuitively, two events $x$ and $y$ are conflicting if when $x$ and $y$ are swapped: i) one of the two events disables the other one, or ii) the two events do not disable or enable
each other, but they do interfere (e.g., access the same shared memory). The precise notion of conflicting events and a procedure for identifying such events are described formally in Sections 8.3 and 8.4. We do not always select all possible conflicting events, but a subset of these where it is possible to reorder the events without changing the order of any other conflicting events.

As an example of relevant conflicts, consider again the original sequence $Q_1$ in Figure 8.2a. Here we have three conflicts involving the image load events $e_0$ and $e_3$, at points $s_0$, $s_2$, and $s_3$. The conflicts occur between the event pairs $\langle e_0, e_1 \rangle$, $\langle e_2, e_3 \rangle$, and $\langle e_3, e_4 \rangle$, respectively. These conflicts arise since all of these events access the same memory location (queue) and could have been executed in a reverse order with a different nondeterministic scheduling.

Note that $e_1$ and $e_2$ do not conflict because they are both user events, which in our work are always ordered, because we are not interested in exploring event sequences that differ from the user’s point of view. Also note that $e_0$ and $e_2$ do conflict but they are not relevant conflicts because reordering them would require reordering conflicting events $e_0$ and $e_1$.

**Comparing with traditional DPOR** Recall that a traditional DPOR algorithm works in a forward manner by examining all transitions from a given state, even disabled ones, and comparing them against already executed transitions. For example, detecting that there is a conflict between $e_0$ and $e_1$ would require the analysis to reason about effects of the unexplored event $e_1$, which is highly nontrivial since this would require precise reasoning about the code of the entire event handler triggered by $e_1$.

**Step 2: Creating reordered event sequences** In the second step, after the relevant conflicts have been identified, we create new event sequences where the order of the events participating in these conflicts are swapped. To achieve this, we may also need to reorder some non-conflicting events. We refer to the construction of these new event sequences as conflict reversals. The algorithm stores the conflict reversals, which will be explored later, in the data structure representing the event sequence.

The algorithm then selects the unexplored conflict reversal from this data structure (the selected conflict reversal is the one with the maximal prefix match with the current event sequence). The approximate replay phase (described in Section 8.2.3) then tries to execute the new event sequence, and the above two-step process will repeat until all conflict reversals are explored.

As an example, the last occurring conflict in the event sequence $Q_1$ in Figure 8.2a is found at $s_3$. This conflict leads to the sequence $Q_2$ in Figure 8.2b. Figure 8.2b also illustrates the explored sequences after six conflict reversals. Note how the three relevant conflicts identified in $Q_1$ have been expanded into $Q_2$, $Q_3$, and $Q_4$. Further, exploring $Q_3$ leads to $Q_5$, and $Q_4$ leads to $Q_6$ and $Q_7$. In $R^4$, the report phase is interleaved with the reordering phase, meaning
that each new execution $Q_i$ is immediately compared to the prior sequence which consists of one less conflict reversal than $Q_i$ and is called the parent of $Q_i$. The executions $Q_2$, $Q_3$, and $Q_4$ are thus compared to its parent $Q_1$, $Q_5$ is compared to its parent $Q_3$, and $Q_6$ and $Q_7$ are compared with their parent $Q_4$.

The maintained event sequence, e.g. $Q_1$, and the exploration algorithm are stateless. That is, the algorithm does not keep track of states or maintain a representation of the entire execution tree as shown in Figure 8.2b, only of the current event sequence (excluding intermediate states) as well as the associated information about unexplored conflicts. However, to enable detection of harmful conflicts between events by comparing executions in the report phase, we do keep some information about the final state of an explored event sequence until all of its children have been explored.

Replaying sequences vs. single events The above approach differs from traditional DPOR, which only stores a single event to be explored later and not a complete event sequence. However, we want to enable reasoning about a specific pair of conflicting events and to determine if reversing that pair of events causes different results. Thus, we store the entire event sequence, such that we can observe the effect of reversing the identified conflict without changing the remaining event sequence (and inevitably reversing other conflicts). This approach of constructing sequences rather than single events, when combined with the approximate replay (Section 8.2.3), is used later in the reporting phase (Section 8.2.4) to enable classifying conflicts as harmful or harmless.

Detecting conflicts in practice In practice, a set of conflicting events for a trace is detected (approximated) by invoking a dynamic race detector on that trace. Therefore, it is important that this race detector is scalable and precise. In our work, we use the state-of-the-art dynamic race detector EventRacer [98].

Interestingly, however, direct use of such read-write race detectors sometimes leads to benign conflicts. The reason is that these detectors report read-write conflicts between events even though the events actually commute. This has the unfortunate consequence that $R^4$ explores many unnecessary event sequences. To handle this problem, we employ advanced logical commutativity detection techniques inspired by Dimitrov et al. [27]. As we show in Section 8.5, combining these techniques with $R^4$ enables us to explore real-world applications in a clean and systematic manner.

8.2.2 Conflict-Reversal Bounding

The algorithm as described so far explores all possible conflict reversals. However, it is known that realistic event sequences in real world event-driven ap-
Applications often contain hundreds of conflicts [98], hence exploring all possible combinations is infeasible.

Similar to the hypothesis that many concurrency bugs are found with a low number of context switches [86], we conjecture that typical errors involving nondeterminism in event-driven applications can be found with a low number of conflict-reversals. We substantiate this hypothesis experimentally in Section 8.5.3. Based on this hypothesis, we introduce conflict-reversal bounding, inspired by the use of delay-bounded scheduling in traditional reasoning for concurrent programs [32].

Let $d$ denote the number of conflict reversals for a given event sequence as compared to the initial one. That is, $d = 0$ for the initial event sequence, and for each subsequent event sequence, the value is 1 higher than for its parent. For the exploration in Figure 8.2b, we have that $d(Q_1) = 0$, $d(Q_2) = d(Q_3) = d(Q_5) = 1$, and $d(Q_4) = d(Q_6) = d(Q_7) = 2$. We can now bound the systematic exploration by a parameter $k$, meaning that we will only explore event sequences with $d \leq k$. Intuitively, $k$ represents the maximum number of deviations to be explored in the nondeterministic behavior compared to the initial execution.

8.2.3 Approximate Replay

Whenever a conflict is identified between two events $x$ and $y$ in a given event sequence, a new event sequence is generated by swapping the two events. This may effect the execution of $x$ and $y$, and in turn all subsequent events. That situation is particularly likely in an event-driven setting where the event handlers are not just simple read and write transitions but are complex operations that often use ad-hoc synchronization Raychev et al. [98], similar to the timeout in our running example. Prior work, both in standard shared memory [90] and in event-driven applications [54] essentially ignores this phenomenon.

To handle this issue, we introduce the concept of approximate replay, which tries to execute a modified event sequence as close to as possible. This works as follows: For each event in a given event sequence, execute the event if it is enabled and skip the event otherwise. Nondeterministic values (e.g., random numbers, network responses) are kept consistent across executions. For example, when performing approximate replay of an Ajax response event $e$ in a conflict reversal, the server response data handled by $e$ is repeated between the parent event sequence and the one produced by the conflict reversal.

**Example**  Continuing with our running example, consider the event sequence $Q_1$ and the conflict $\langle e_2, e_3 \rangle$. In this situation, we execute the events $e_0 \cdot e_1$ exactly, followed by an approximate replay of $e_3 \cdot e_2 \cdot e_4$, resulting in $Q_3$ in Figure 8.2b. The approximate replay will detect that the DOM timer event $e_4$ is not enabled and is hence skipped in this execution.
8.2.4 Reporting Errors

So far we have described the core exploration capabilities of \( R^4 \). These capabilities can be used for a broad range of debugging and bug detection scenarios. For example, we can use them to check for common issues, such as, harmfulness of conflicts, application crashes, assertion failures, and output discrepancies (e.g., [23, 54, 64, 82, 102]).

Our report phase classifies each conflict as either harmful or harmless, helping identification of errors caused by conflicts between event handlers. A conflict is classified by comparing the conflict-reversal with the event sequence in which the conflict was discovered. If the two event sequences are fully equivalent or approximately equivalent, then the conflict is harmless. Two event sequences are fully equivalent if they lead to the same DOM and JavaScript state. Approximate equivalence uses heuristics to allow for minor differences in the two executions. In our case, we implement heuristics identifying three common patterns of harmless conflicts in web applications: (1) conflicts caused by the registration of an event handler and the execution of the same event handler, since event registration is often deferred to page load in web applications and such behavior is expected; (2) conflicts involving an unload event, which are often harmless since they occur when a page is closed; and (3) conflicts caused by ad-hoc synchronization using DOM timers waiting for some condition to be satisfied, which either results in disabled timers or new timers.

Every error report issued by \( R^4 \) about conflicting events includes two concrete event sequences that – in contrast to prior work [54] – differ only by a single conflict reversal, along with an explanation of the filtering decision. This is important for reducing the amount of false positives and making the error reports comprehensible and accessible.

Example  
Returning to our running example in Figure 8.1, recall that a timer is spawned (\( e_4 \)) because the user clicks the next button (\( e_2 \)) before the next image is available (\( e_3 \)). The conflict between \( e_2 \) and \( e_3 \) is explored in \( Q_3 \). However, when exploring this conflict, approximate replay will skip the timer event \( e_4 \) because it is disabled (as the image is already available when the user clicks the next button). A race detector such as EVENTRACER will report the conflict between \( e_2 \) and \( e_3 \) as a harmful race. Similarly, tools such as WAVE will report an error (they do so whenever they are unable to execute the given sequence exactly). As we show in Section 8.5.2, such approaches suffer from a high number of false positives. In this example, the conflict is clearly harmless, and the inability to execute the event sequence exactly is caused by ad-hoc synchronization. The report phase of \( R^4 \) will identify the execution of \( Q_3 \) as the result of ad-hoc synchronization using timers and mark it as harmless.
8.2.5 Summary of $R^4$

In summary, $R^4$ uses DPOR techniques to reduce the search space. Unlike traditional DPOR, which determines the effect of a candidate transition, the variant used by $R^4$ only deals with already executed transitions. This is necessitated by our domain of event-driven applications where it is difficult to predict the effect of executing the event handler code. $R^4$ also uses conflict-reversal bounding based on the hypothesis that most errors should be found with a small number of conflict reversals. Finally, unlike traditional DPOR techniques, which schedule single events for execution, $R^4$ supports approximate replay where entire sequences are stored and considered for execution. This capability, combined with the fact that the exploration reverses one conflict at a time, enables effective classification of conflicts.

8.3 Background

In this section we introduce the necessary formal concepts to explain our model checking algorithm in Section 8.4. Most of the definitions are standard [38], except that we need to adapt them to the domain of event-driven applications.

Transitions and traces  An event-driven application is captured by a labeled transition system, \( \langle S, s_0, E, \delta \rangle \), where \( S \) is a set of states, \( s_0 \in S \) is the initial state, \( E \) is a set of events (labels), and \( \delta \subseteq S \times E \times S \) is the transition relation. For example, a state can capture the HTML DOM and the JavaScript heap of a running web application, and events include both user and system events. We assume that the transition relation is deterministic: for a given state \( s \) and event \( e \), there exist at most one state \( s' \) such that \( \langle s, e, s' \rangle \in \delta \) (note that the overall system behavior may still be nondeterministic due to the scheduling order of different events).

A finite trace \( \tau \) of the transition system is defined as \( \tau = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} s_n \), where \( \langle s_i, e_{i+1}, s_{i+1} \rangle \in \delta \) for all \( 0 \leq i < n \) and \( s_0 \) is the initial state of the transition system. To reduce clutter, we sometimes omit the states and use the shorthand notation \( \tau = e_1 \cdot e_2 \cdot \cdots \cdot e_n \). We use \( \tau_i \) to refer to the event \( e_i \) in \( \tau \), \( \tau_{i..j} \) to refer to the sub-sequence of events \( e_i \ldots e_j \in \tau \) and \( \tau \cdot \tau' \) to denote concatenation of two traces \( \tau \) and \( \tau' \).

The function \( enabled(s) \) captures the set of events enabled in state \( s \), that is, \( enabled(s) = \{ e \mid \langle s, e, s' \rangle \in \delta \} \). We use the shortcut \( enabled(\tau) \) to denote the set of events enabled in the last state \( s_n \) of the trace \( \tau \), that is, \( enabled(\tau) = enabled(s_n) \).

Independence  We say that two events \( x, y \in E \) are independent iff they do not affect each other in any executable event sequence in the transition system, that is, \( x \) and \( y \) do not enable or disable each other and they are commutative.
Definition 1 (Independence). The relation independent is the smallest symmetric relation over \( E \) such that independent\((x, y)\) holds for \( x, y \in E \) if:

1. \( \forall s \xrightarrow{s} s' \in \delta : y \in \text{enabled}(s) \text{ iff } y \in \text{enabled}(s') \), and

2. \( \forall s \in S \text{ where } x, y \in \text{enabled}(s) \text{, there is a unique state } s' \text{ such that } s \xrightarrow{s} s_x \xrightarrow{y} s' \text{ and } s \xrightarrow{y} s_y \xrightarrow{s} s' \text{ for some } s_x, s_y. \)

Notice that the second rule implies that the set of enabled events in \( s' \) is unaffected by the order of \( x \) and \( y \). The independence relation can naturally be parameterized on a given event sequence \( \tau \), denoted as independent\(_\tau\)(\(x, y\)), such that independence is defined for the state reachable by the event sequence \( \tau \). The complementary dependence relations are denoted respectively by dependent\((x, y)\) and dependent\(_\tau\)(\(x, y\)). All of these functions are naturally lifted to sets of events.

There may be multiple event sequences that only differ in the order of independent events.

Definition 2 (Transitive Dependency). For a trace \( \tau \), we define the transitive dependency relation \( \rightarrow_\tau \) as the binary relation over the events in \( \tau \) such that:

1. if \( i < j \) and dependent\(_{\tau_{1...i-1}}(\tau_i, \tau_j) \) then \( \tau_i \rightarrow_\tau \tau_j \), and

2. \( \rightarrow_\tau \) is transitively closed.

The transitive dependency relation is an important concept as it enables partial order reduction: an exploration algorithm need only explore single linearizations of elements not ordered by the relation, thus reducing the total number of explored event sequences.

We also say that two events \( x, y \in E \) conflict iff they are both enabled and do not commute.

Definition 3 (Conflict). The relation conflict is symmetric over \( E \) such that conflict\((\tau, x, y)\) holds for \( x, y \in E \) and the trace \( \tau \) if:

1. dependent\(_\tau\)(\(x, y\)), and

2. \( x, y \in \text{enabled}(\tau) \).

Intuitively, the conflict relation is similar to the dependency relation, except the conflict relation excludes events from the relation that are always ordered, for example, if \( x \) always enables \( y \).
Partial-order reduction  Partial-order reduction can be achieved by exploring only a subset of events from any visited state. This set is typically referred to as a persistent set.

Definition 4 (Persistent Set). A set of events $E_p \subseteq \text{enabled}(s_0)$ is persistent in $s_0$ iff, for all nonempty traces $\tau = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots s_{n-1} \xrightarrow{e_n} s_n$, where $e_i \notin E_p$ for $0 < i \leq n$, independent$_{\tau_{1..i-1}}(e_i, E_p)$ holds.

The value of the persistent set concept is that we can delay the exploration of events that are independent with the events in the persistent set, until after the events in the persistent set have been explored.

As an example, suppose three events $x$, $y$ and $z$ are enabled in a state $s$ and independent$(x, y)$, independent$(x, z)$, and dependent$(y, z)$. The set $E_p = \{x\}$ is a persistent set for $s$ since all other event sequences not including the event $x$ (i.e., $y$, $z$, $y \cdot z$, and $z \cdot y$) all reach events independent with $x$. Likewise, $E_p = \{y, z\}$ is a persistent set since the only other event sequence (i.e., $x$) is independent of both $y$ and $z$.

8.4 Stateless Model Checking

We now present our stateless model checking algorithm targeting event-driven applications. Our algorithm is based on three key ideas: dynamic partial-order reduction (DPOR) extended and adapted to the domain of event-driven applications, approximate replay, and conflict-reversal bounding. In what follows, we first discuss the overall exploration algorithm and then describe each of the three concepts.

The procedure, shown in Figure 8.3, takes as input an event sequence and explores a set of sequences derived from it. In our setting, this input sequence is obtained from the record phase, however the exploration algorithm is general and is independent of how the input sequence is obtained.

The algorithm implements a depth first search of event sequences, using persistent sets to prune the set of explored events in a visited state (pruning is essential when exploring long sequences of events). The persistent sets are approximated dynamically as new event sequences are explored. At all times, the current sequence of events, from the root of the search tree to the current leaf, is captured by the event sequence $\tau$ (line 2). We maintain two maps $T, V : E^* \rightarrow 2^E$ where $T$ maps an event sequence to a persistent set (events which are to be explored) and $V$ maps an event sequence to a set of events that have already been explored following the event sequence. The entries of both maps are initialized to $\emptyset$.

Initially, the algorithm is given an event sequence $\tau_{\text{init}}$ and all three maps are initialized. Then, the visited sets in $V$ and the persistent sets in $T$ are updated to reflect the events that were already explored in $\tau$ (via the procedures updateVisited and updatePersistentSets). The persistent sets in $T$ are
8.4. STATELESS MODEL CHECKING

1: function EXPLORE(\(\tau_{init}\))
2: \(\tau := \tau_{init}\)
3: updateVisited(\(\tau\))
4: updatePersistentSets(\(\tau\))
5: while true do
6: if \(\exists b \in \text{unexplored}(\tau)\) then
7:     // Case I
8:     \(\tau := \text{replay}(\tau \cdot b)\)
9:     updateVisited(\(\tau\))
10:    updatePersistentSets(\(\tau\))
11: else if \(\tau \neq \epsilon\) then
12:     // Case II
13:     \(V[\tau] = \emptyset; T[\tau] = \emptyset\)
14:     \(\tau := \tau_1...|\tau|-1\) // remove last event
15: else
16:     break
17: end if
18: end while
19: end function

20: function UNEXPLORED(\(\tau\))
21: return \(\{e \mid e \in T[\tau] \land e \notin V[\tau]\}\)
22: end function

23: function UPDATEVISITED(\(\tau\))
24: for \(i := 1...|\tau|\) do
25:     \(V[\tau_{1...i-1}] \cup := \{\tau_i\}\)
26: end for
27: end function

Figure 8.3: Our stateless model checking algorithm for event-driven applications. The algorithm explores schedules starting from the initial input trace \(\tau_{init}\).

also updated to reflect the events that are about to be explored. We discuss the updatePersistentSets procedure in detail in Section 8.4.1.

In the iteration loop (lines 5–18), the algorithm considers one of two cases. In Case I (lines 6–10), an event \(b \in T[\tau]\) is explored if \(b\) is not yet visited. Here, event \(b\) is first selected via the unexplored procedure. The replay procedure executes the event sequence \(\tau \cdot b\) followed by some number of events as determined by replay (in Section 8.4.2, we will see a specific realization of replay). The resulting new sequence is then stored in \(\tau\). Finally, the invocations of updateVisited and updatePersistentSets ensure that \(T\) and \(V\) are consistent with the newly executed sequence \(\tau\). In Case II (lines 11–14), no events remain to be explored from the current \(\tau\) and hence the algorithm resets the map entries for \(\tau\) in \(T\) and \(V\) and backtracks by removing the last event in \(\tau\).
8.4.1 Approximating Persistent Sets

When selecting an instruction to be placed in the persistent set, classic DPOR algorithms fundamentally require knowing the effect of that instruction on the program state. This requires either speculatively executing the instruction or analyzing its effects statically. Both of these approaches are feasible in a traditional concurrent system where each instruction is a primitive operation (e.g., a shared read or a write). However, in an event-driven setting, such “look-ahead” is highly nontrivial as we are not dealing with a single instruction but with an entire atomic code fragment potentially containing thousands of instructions. Therefore, we instead update the persistent sets by comparing already executed events in an event sequence \( \tau \).

Figure 8.4: Procedures for updating persistent sets in \( T \) given an executed event sequence \( \tau \).

The \texttt{updatePersistentSets} procedure, shown in Figure 8.4, iterates over each event \( \tau_i \) in \( \tau \), updating the persistent set for the prefix \( \tau^p \) of \( \tau_i \).

Initially, we add \( \tau_i \) to the persistent set \( T[\tau^p] \) (line 3) using the \texttt{insertIntoT} procedure (here, \texttt{insertIntoT} does not refer to \( \tau \); in later sections, \( \tau \) will be used). The procedure then checks if \{\( \tau_i \)\} is a persistent set according to Definition 8.4 by checking for two kinds of conflicts: conflicts caused by events being disabled (lines 5–7) and conflicts caused by non-commuting events (lines 8–12).

In the first part (line 5), we check if the event \( \tau_i \) disables an event \( e \). In this case, the \texttt{conflict}(\( \tau^p, \tau_i, e \)) predicate evaluates to \texttt{true} and \( e \) is added to
8.4. STATELESS MODEL CHECKING

Figure 8.5: Updated procedures to support approximate replay.

the persistent set. Intuitively, if event $\tau_i$ disables event $e$, then we need to explore an event sequence where $e$ precedes $\tau_i$.

In the second part (lines 8–12), for all events $\tau_j$ succeeding $\tau_i$, we check if there exists a linearization of $\tau$ where $\tau_i$ and $\tau_j$ are adjacent and where $\text{conflict}(\tau^p \cdot w, \tau_i, \tau_j)$ holds. Intuitively, in this case we would like to explore the event sequence where the conflicting events $\tau_i$ and $\tau_j$ are reordered compared to the linearization of $\tau$ (ensured by invoking the function $\text{insertIntoT}$). In Section 8.4.2, we will discuss a technique for replaying an event sequence (where $\tau_i$ and $\tau_j$ are reversed) as close as possible to $\tau$.

Theorem 1. The algorithm explores at least one linearization of any reachable event sequences $\tau$, where $\text{enabled}(\tau) = \emptyset$.

A proof of Theorem 1 can be found in Appendix 8.8.1. Further, in Section 8.5, we discuss how dependencies, linearizations, and enabled events are approximated using dynamic analysis.

8.4.2 Approximate Replay

The algorithms in Figures 8.3 and 8.4 use persistent sets to represent what must be explored. The exploration of an event $b$ in a persistent set requires the replay procedure to execute an event sequence $\tau \cdot b$, followed by some sequence of events $w$ (line 8 in Figure 8.3). In principle, executing $w$ can be substantially different compared to the original event sequence that caused us to reach $\tau \cdot b$. As motivated in Section 8.2.3, we would like to minimize the difference between the original event sequence and the event sequence to be explored. We address this challenge by introducing the concept of approximate replay, a technique for guiding the exploration along an expected path.

To achieve approximate replay, we extend each persistent set to store event sequences and not single events. The map $T$ is then changed into the map
1: function insertIntoTcrb(τp, τs, τ)  
2:    Tcrs[τp] \cup := \{τs\}  
3:    d(τp \cdot τs) := d(τ) + 1  
4: end function  
5: function UNEXPLOREDcrb(τ)  
6:    return \{τ' | τ' ∈ Tcrs[τ] ∧ τ' \notin V[τ] ∧ d(τ \cdot τ') ≤ k\}  
7: end function  

Figure 8.6: Updated procedures to support conflict reversal bounding.

\(T_{crs}\) with the type \(E^* \rightarrow 2^{E^*}\). This change is an extension of persistent sets, that is, the set \(T_{crs}[τ]\) can be translated into a persistent set as follows: \(T[τ] = \{τ' | τ' ∈ T_{crs}[τ]\}\). Here, the name \(crs\) stands for conflict reversal sequences and denotes that these sequences are the result of reversing the order of two conflicting events (in Section 8.4.3 we introduce a bound on the number of conflict reversals).

The algorithms in Figures 8.3 and 8.4 stay as-is and simply use the procedures shown in Figure 8.5. In addition, the procedure updatePersistentSets_{crs} handles a tricky corner case where for a given explored sequence \(τ\) and some prefix \(τ'\) of \(τ\), there are two sequences \(a, b \in T_{crs}[τ']\) where \(a = p \cdot r\) and \(b = p \cdot q\) share the same prefix \(p\). In that case, if sequence \(a\) is explored, its prefix \(p\) will be marked as visited, precluding exploration of sequence \(b\) (even though \(b\) is not yet fully explored). To ensure that \(b\) is explored, we update the entry \(T_{crs}[τ' \cdot p]\) to contain \(q\) (the unexplored suffix of \(b\)).

Finally, the semantics of the replay procedure is adjusted as follows. It will execute any enabled event in the given sequence, and simply skip events that are not enabled. For example, the prefix \(τ\) given to the replay procedure in line 8 in Figure 8.3 will always be executed as every event in \(τ\) is enabled.

### 8.4.3 Conflict-Reversal Bounding

In general, an event-driven application (e.g., a web page) may contain thousands of conflicts and full exploration (at arbitrary depth) of all such conflicts is practically impossible. Based on the hypothesis that most errors can be found within a small number of reversals derived from a given sequence, we introduce the concept of conflict-reversal bounding which limits the number of conflict reversals.

To enforce bounding, we maintain a conflict-reversal depth map \(d: E^* \rightarrow \mathbb{N}\) from event sequences to natural numbers (the entries of this map are initialized to 0). Intuitively, we associate each event sequence \(τ^*\) inserted into \(T_{crs}[τ^p]\) with a conflict-reversal depth, \(d(τ^p \cdot τ^*)\). When exploring an event sequence \(τ\) with depth \(d\), all newly discovered event sequences are assigned depth \(d + 1\). Conflict reversal bounding prevents exploration of any event sequence with \(d > k\) where \(k\) is the conflict-reversal bound. To incorporate conflict reversal
bounding, the algorithms can simply use the procedures $\text{insertIntoT}_{\text{crb}}$ and $\text{unexplored}_{\text{crb}}$ shown in Figure 8.6.

8.5 Evaluation

We evaluate the $R^4$ algorithm in the context of client-side web applications. Our implementation is built using (1) an instrumented version of the WebKit browser to observe and control nondeterministic input and nondeterministic scheduling, similar to Burg et al. [18], Hong et al. [54], and (2) a modified version of EventRacer, which provides the infrastructure to approximate persistent sets. Additionally, we use EventRacer to i) approximate the dependency relation by computing races, ii) find linearizations of a given event sequence by using the happens-before relationship defined in EventRacer, and iii) compute the enabled relation as defined in Section 8.3.

Goals  Our experiments have the goal of evaluating the effectiveness of $R^4$ compared to EventRacer [98] when analyzing real-world web applications, its bug isolation capabilities compared to WAVE [54], and the use of conflict-reversal bounds.

8.5.1 Effectiveness Compared to EventRacer

To evaluate the effectiveness compared to EventRacer we randomly picked 32 of the Fortune 100 websites used in the main EventRacer bug detection study, and systematically explored them with a conflict-reversal depth of 1 (to fairly compare the two tools).

For each website, one initial recording was created by loading the website and triggering events (having corresponding event handlers) in an arbitrary order, for either 15 seconds or until 250 events had been triggered, whichever came first. Next, the reorder, replay, and report phases were applied, exploring and classifying event sequences. The total analysis time was on average 21 minutes for each benchmark, or approximately 18 seconds for each explored event sequence.

We summarize our results in Table 8.1. For comparison purposes, we include information about uncovered races reported by EventRacer, which operates on a single trace. For each race, EventRacer shows the racing events as well as debug information, such as, stack traces of the racing operations. In comparison, $R^4$ explores one trace for each uncovered race and displays an actual witness of potential bugs for each race, as well as debug information and a screenshot of the final state of the webpage.

For each explored trace, $R^4$ automatically identifies full and approximate equivalence between the explored trace and the initial recording (as described

\cite[18]{URL for implementation and all experimental data omitted for double-blind review.}
Table 8.1: Explored event sequences for 32 tested websites.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Number per website</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td><strong>EventRacer uncovered races</strong></td>
<td>68.8</td>
</tr>
<tr>
<td><strong>(R^4)</strong> – Reordered traces compared to original trace</td>
<td></td>
</tr>
<tr>
<td>Fully equivalent</td>
<td>52.2</td>
</tr>
<tr>
<td>Approximately equivalent</td>
<td>8.0</td>
</tr>
<tr>
<td>Different</td>
<td>8.6</td>
</tr>
<tr>
<td><strong>(R^4)</strong> – Details about differences from original trace</td>
<td></td>
</tr>
<tr>
<td>Different DOM</td>
<td>0.2</td>
</tr>
<tr>
<td>Different uncaught exceptions</td>
<td>0.3</td>
</tr>
<tr>
<td>Different network communication</td>
<td>0.5</td>
</tr>
<tr>
<td>Only JavaScript heap</td>
<td>7.7</td>
</tr>
</tbody>
</table>

From 68.8 explored traces on average per initial trace, the majority (52.2) are fully equivalent to the original exploration trace and 8.0 traces on average only contained minor differences, such as, disabled events due to later attachment of event handlers or ad-hoc synchronization using DOM timers. Thus, on average a total of 60.2 out of 68.8 traces were marked as harmless.

The remaining traces (8.6 on average) lead to a state of the JavaScript heap and the DOM that is different than the state in the original recorded trace. To further analyze the actual effect on the webpage, \(R^4\) automatically raises warnings in a number of cases summarized in the last 4 rows of Table 8.1. Multiple warnings can be raised for the same trace. While most of the differences only surface as a difference in the JavaScript heap, a number of cases lead to different DOM than in the original page, although the user actions are unchanged. \(R^4\) also warns if the JavaScript runtime exceptions differ between the explored event sequence and its parent event sequence, or when the rescheduling leads to different network communications (e.g., different number of AJAX requests) to a server component.

In total, \(R^4\) reports 275 warnings about harmful conflicts in the 32 websites. Manually inspecting a random selection of 79 of those warnings indicate 13 actual bugs.

Compared to EventRacer, \(R^4\) provides additional information about explored traces, by including concrete witnesses in the warning messages, with screenshots, DOM state, and an description of why the two traces differ. Furthermore, \(R^4\) is able to identify, on average, 60.2 out of 68.8 traces as harmless based on comparing traces. In comparison, EventRacer only marked 8.0
(not shown in Table 8.1) out of the 68.8 uncovered races as harmless based entirely on reasoning about the initial recording. Thus, we conclude that the additional information provided by exploring traces using $R^4$ does improve on the overall usefulness of the tool.

8.5.2 Bug Isolation Capabilities Compared to WAVE

We evaluate the bug isolation capabilities of $R^4$, in comparison to the WAVE tool. First, we note that an exact implementation of the WAVE algorithm is not practically feasible with precise happens-before as that algorithm requires the enumeration of all event sequences ordered by a happens-before relation. In our evaluation, we operate on substantially longer event sequences (the recordings contain 3,742 events on average, while Hong et al. [54] reports on sequences averaging only 7.2 events). To ensure a fair comparison, we therefore compare $R^4$ with an algorithm that samples traces in a manner similar to WAVE as follows: (1) generate $x$ copies of the recorded event sequence, where $x$ is the number of explored events in the original recorded sequence; (2) swap multiple random pairs of events in each event sequence if allowed by the happens-before relation; (3) execute each resulting event sequence, and apply the same detectors for erroneous event sequences as WAVE: DOM state differences, existence of uncaught exceptions, and inability to execute an event sequence.

This experiment resulted in 100% of the executed event sequences to be flagged as erroneous! Manual inspection of a subset of the executed event sequences confirmed that it is difficult to classify and identify the cause of divergence in event sequences with many changes compared to one change. The high percentage of erroneous sequences is caused by the high number of ad-hoc synchronization and harmless conflicts, which exist in all of the used benchmarks. If any of these are triggered it will result in a state flagged as erroneous. We also observe cases, such as, for the FedEx website, where the explored event sequences all trigger a user click early in the sequence, which stops parsing and directs the user to a different page, thus pruning away any bugs that could have been discovered in the first page. Thus, even though real errors may be exposed using the WAVE approach, we find that they tend to drown in harmless and ad-hoc synchronization races, which $R^4$ is designed to avoid.

8.5.3 Effects of Conflict-Reversal Bounding

To explore the cost of increasing the conflict-reversal bound, we additionally performed exploration with increasing bound until a time limit of one hour was reached for each benchmark. The results are summarized in Figure 8.7. For around one half of the websites, only conflict reversal bound of 1 was feasible within our time limit, while only for two sites we could not complete
bound 1 within the limit, and four sites were fully explored within the time limit. Thus, without conflict-reversal bounding a number of the benchmarks would not terminate within reasonable time.

To further evaluate the effects of conflict-reversal bounds, we selected five web applications each with a single known timing related bug as described in the WAVE paper [54]. For each web application and bug, one initial recording was made by manually following a series of steps given by the authors of WAVE to reproduce the target bug. The steps are reproduced under common conditions, such that the timing dependent bug is not triggered. For each benchmark, we ran $R^4$ with increasing conflict-reversal depth until the bug is exposed.

Table 8.2 shows the results. We observe how four of the five bugs are found at conflict-reversal depth 1, while a single bug, in WordPress, is found at depth 3. The WordPress bug is caused by a load event, $x$, which defines a function, and a user click event, $y$, which triggers the same function. A conflict-reversal depth of 3 is required because of (1) a conflict between $y$ and the DOM load event triggered by $x$, and (2) a conflict between $x$ and a user click event which immediately precedes $y$. However, only the conflict between $x$ and $y$ is harmful.

This experiment indicates that a low conflict-reversal depth is sufficient to expose bugs, and at least a small set of known bugs are found within this low bound. Of course, no guarantees are made that this is always the case. Specifically, we observe that the approximation of dependence and the amount of independence between events has a measurable impact on the results of the overall algorithm.

Furthermore, this experiment highlighted the need of a good approximation of conflicts and specifically the identification of events that are independent because of commutativity. As an example, we observed a common pattern of the statement $x = x ? x : \{\}$ used for lazy initialization of a variable $x$ with an object if $x$ is uninitialized. This pattern always writes $x$ even if initialized, will cause a race detector to report races between any events triggering this check. Because in our exploration algorithm, the notion of conflict is parametric, we were able to easily extend EVENTRACER with commutativ-
8.6. RELATED WORK

<table>
<thead>
<tr>
<th>Site</th>
<th># Events</th>
<th>Explored seqs</th>
<th>Depth</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gallery3</td>
<td>516</td>
<td>3</td>
<td>1</td>
<td>&lt; 1m</td>
</tr>
<tr>
<td>TYPO3</td>
<td>1556</td>
<td>24</td>
<td>1</td>
<td>5m</td>
</tr>
<tr>
<td>WordPress</td>
<td>2043</td>
<td>22</td>
<td>3</td>
<td>&lt; 1m</td>
</tr>
<tr>
<td>AjaxPlorer</td>
<td>1528</td>
<td>38</td>
<td>1</td>
<td>28m</td>
</tr>
<tr>
<td>Feng Office</td>
<td>1451</td>
<td>24</td>
<td>1</td>
<td>9m</td>
</tr>
</tbody>
</table>

Table 8.2: List of web applications where an initial recording exposes a single confirmed bug, as used in the WAVE paper. The length of the initially recorded event sequence is reported, together with the number of event sequences explored, required conflict-reversal depth to expose the bug, and the running time for the analysis.

Summary

Overall, our experimental results demonstrate that \( R^4 \) provides a promising approach for systematically exploring the nondeterminism in event-driven applications. When tested on real-world obfuscated websites, our tool proved useful in finding and reproducing concurrency errors, such as, exceptions or nondeterminism of the DOM. For other errors, such as, JavaScript heap nondeterminism, the reports should be investigated on a non-obfuscated version of the site and possibly by inserting assertions. Finally, when assertions are present, the system can model check and verify if a website is free of errors up to the specified bound of conflict reversals.

8.6 Related Work

We next survey some of the most closely related work.

Testing event-driven applications

A number of tools exist for JavaScript which can record and deterministically replay executions, taking into account nondeterministic inputs and scheduling. Mugshot and Jalangi use instrumentation of JavaScript and external proxies to record the timing of events and selected inputs. \( R^4 \) instruments the browser directly, allowing for finer control of the execution, including exact control of parsing, at the cost of less flexibility. This approach is similar to Timelapse, which also instruments the browser directly in order to implement deterministic record and replay. However, Timelapse is only concerned with program understanding,
and not exploring alternative event sequences and the effects of nondeterminism.

Fuzzing schedules to uncover timing related bugs is a known technique in the domain of concurrent programs, for example, as done by Sen [102], Narayanasamy et al. [90] for Java applications, Andrica and Candea [8], and WAVE [54] for web applications. A central challenge for each of these tools is the identification of errors if they occur while fuzzing schedules: Sen [102] detects raised exceptions; Narayanasamy et al. [90] and WAVE check if a fuzzed schedule can be executed and lead to the same final state, while Andrica and Candea [8] use manual validation only. However, as discovered by Raychev et al. [98], web applications contain many races, hence manually inspecting all of these races is infeasible. Furthermore, web applications frequently use ad-hoc synchronization, and therefore reordering events is expected to impact the enable-ness of other events as well as the states of an execution. It is also common for web applications to raise exceptions even in normal operation. Therefore, approaches, such as, Hong et al. [54], Narayanasamy et al. [90], which simply compare the state are not effective in this setting (as they will flag almost every interference as harmful). Our proposed solution uses approximate replay to continue execution even when the expected event sequence cannot be executed exactly. Moreover, we introduce a report phase that classifies explored conflicts as harmful or not.

Mutlu et al. [88] discuss the problem of benign races in the domain of web applications and coin the term observable races, i.e., races that can be observed by comparing two renderings of the same web page. In their position paper, they also discuss the possibility of systematically exploring observable races as a possible research direction. \( R^4 \) is one such example approach of systematic exploration.

Model checking and DPOR The reorder phase of \( R^4 \) is based on the DPOR [88] algorithm modified to fit the domain of event-driven applications. An overview for how \( R^4 \) compares with state-of-the-art existing work in DPOR is shown in Table 8.3. \( R^4 \) differs from classical DPOR by (1) comparing executed events to identify conflicts (denoted backwards comparison), while classical DPOR reasons about transitions that may or may not have been executed in a reached state (denoted forwarded comparison), and (2) classical DPOR uses the concept of processes when reasoning about instructions that must be executed prior to other instructions. In addition, we extend DPOR with approximate replay, which stores not only single events in persistent sets but entire event sequences in order to observe the effects of a single conflict for each iteration.

The optimal DPOR algorithm [11] also uses backwards comparison, however, their reason is different: they use backwards comparison because their proposed extension to classical DPOR involves precision of maximal event se-
Algorithm | Domain | Comparison | Granularity | Bounding
--- | --- | --- | --- | ---
Classical DPOR [38] | multi-threaded | forwards | single | -
Optimal DPOR [1] | multi-threaded | backwards | subsequence | -
Bounded DPOR [26] | multi-threaded | forwards | single | preemption
R$^4$ & | event-driven | backwards | approximate | conflict-reversal

Table 8.3: Different styles of dynamic partial-order reduction. The *Comparison* column shows the direction of comparisons; the *Granularity* column shows if the method adds a single transition to the persistent set, a subsequence of transitions, or a complete sequence of transitions that may be realizable (approximate); and finally the *Bounding* column shows the form of bounding used during the search.

Quences, while we use backwards comparison to avoid the need for predicting the effects of complex event handlers. Furthermore, optimal DPOR introduces wakeup trees, which store sub-sequences of events for backtracking, to ensure that any new iteration will explore the reversal of two conflicting events. This differs from approximate replay, which, in addition to the subsequence leading to the reversal of a conflict, also stores additional events to guide the exploration following the reversal. Finally, the DPOR algorithm does not operate on single-threaded event-driven applications, but on concurrently executing processes.

Sleep sets [41], which are often used to optimize DPOR, is trivially applicable to R$^4$, but we have omitted them to simplify the presentation and implementation. Finally, our proposal of conflict-reversal bounding is related to delay bounding [32] in terms of intent. Specifically, our bound limits the divergence from an initial execution, while delay bounding limits the amount of times a scheduler is forced to deviate from the expected schedule. Since event-driven applications execute single-threaded, other related approaches of bounding the search are not applicable (e.g., preemption bounding [26, 86], which limits the number of preemptions when switching between processes).

8.7 Conclusion

We have presented the first practical stateless model checker for event-driven applications, called R$^4$. Our algorithm can systematically and efficiently explore the scheduling nondeterminism in a given execution. It builds on three key insights: an adaptation of DPOR to the domain of event-driven applications where we only work with transitions occurring in the past and do not require determining the effects of a future transition, ii) a conflict-reversal bound based on the idea that most harmful errors occur with a small number of event reorderings, and iii) approximate replay which minimizes the divergence from the original execution.

We implemented R$^4$ for the domain of client-side web applications and showed that the technique is robust and scalable enough to analyze real-world
programs. Furthermore, the evaluation indicates that our analysis is significantly more precise in classifying harmful nondeterminism than state-of-the-art alternatives.

8.8 Appendix

8.8.1 Proof of Completeness of Exploration

**Lemma 1.** Given the state $s$ and event $x \in \text{enabled}(s)$, we say that if there exists event sequence $w$ and event $y$ such that $s \xrightarrow{w_1} \ldots \xrightarrow{w_{|w|}} s_w \xrightarrow{y} s_y$, $x \notin w$, and $\forall e \in w: \text{independent}(x, e)$, then one of the following event sequences must also exist:

- $s \xrightarrow{x} s_x \xrightarrow{w_1} \ldots \xrightarrow{w_{|w|}} s_{w'} \xrightarrow{y} s_{y'}$, or
- $s \xrightarrow{x} s_x \xrightarrow{w_1} \ldots \xrightarrow{w_{|w|}} s_{w'}$ where $y \notin \text{enabled}(x \cdot w)$.

**Proof.** Trivially, $x$ is independent with any event in $w$, so prefixing $w$ with $x$ has no effect. However, $y$ may be dependent on $x$, such that either (1) $y$ exist but may have a different behaviour, or (2) $y$ is disabled by $x$.

**Theorem 1.** The algorithm explores at least one linearization of any reachable event sequences $\tau$, where $\text{enabled}(\tau) = \emptyset$.

**Proof.** We prove Theorem 1 by induction over our algorithm’s exploration of an event sequence $\tau$. The property given in Theorem 1 must hold for all event sequences reachable through $\tau$ when the algorithm backtracks from $\tau$ to $\tau_{1, |\tau|-1}$ (Algorithm 1 line 15). If the property holds for the empty event sequence, then the theorem holds for the exploration in general.

**Base case:** Given the event sequence $\tau$ where $\text{enabled}(\tau) = \emptyset$, the theorem trivially holds for all sequences reachable from $\tau$ (there are none) when backtracking from $\tau$.

**Induction case:** We assume that for any $e \in \text{enabled}(\tau)$, we can explored at least one linearization of any reachable event sequence on the form $\tau \cdot e \cdot w$ where $\text{enabled}(\tau \cdot e \cdot w) = \emptyset$. We now show that the same applies for $\tau$.

First, by contradiction, let us assume that there exist an event sequence $\tau \cdot z$, where $\text{enabled}(\tau \cdot z) = \emptyset$, which is not a linearization of any event sequence explored on the form $\tau \cdot e \cdot w$ for $e \in T[\tau]$. It follows that:

- $z_1 \notin T[\tau]$, otherwise a linearization of $\tau \cdot z$ would already have been explored, and

- there must be some prefix $z_{1..i}$ of $z$ where $\exists e \in T[\tau] : e \notin z_{1..i} \land \text{dependent}(e, z_i)$, otherwise $\tau \cdot z$ would be a linearization of an already explored event sequence.
However, both of the two above cases are contradicted by Definition 4, which says that any event in \( z \) must be independent of any event in the persistent set \( T[\tau] \), \( \forall e \in z, \forall e' \in T[\tau] : \) \textit{independent}(e, e').

Thus, if \( T[\tau] \) is a persistent set, then \( w \) can not exist and the Theorem must hold for \( \tau \) after all events in \( T[\tau] \) have been explored.

Next, we show that \( T[\tau] \) is a persistent set. By contradiction, if \( T[\tau] \) is not a persistent set, then by Definition 4, there exists an event sequence \( w \) such that \( \exists e \in T[\tau] : \text{dependent}(e, w_0) \wedge \forall w_i \in w : w_i \not\in T[\tau] \). According to Lemma 1, if such a \( w \) exists, then there must also exist one of two possible event sequences prefixed by \( e \). However, the algorithm specifically checks for the existence of either of the two event sequences following \( e \), and updates \( T[\tau] \) accordingly. Thus, \( w \) does not exist and \( T[\tau] \) must therefore be a persistent set.
Chapter 9

Automated Testing with Targeted Event Sequence Generation

Abstract

Automated software testing aims to detect errors by producing test inputs that cover as much of the application source code as possible. Applications for mobile devices are typically event-driven, which raises the challenge of automatically producing event sequences that result in high coverage. Some existing approaches use random or model-based testing that largely treats the application as a black box. Other approaches use symbolic execution, either starting from the entry points of the applications or on specific event sequences. A common limitation of the existing approaches is that they often fail to reach the parts of the application code that require more complex event sequences.

We propose a two-phase technique for automatically finding event sequences that reach a given target line in the application code. The first phase performs concolic execution to build summaries of the individual event handlers of the application. The second phase builds event sequences backward from the target, using the summaries together with a UI model of the application. Our experiments on a collection of open source Android applications show that this technique can successfully produce event sequences that reach challenging targets.

9.1 Introduction

Mobile applications are often structured as collections of screens where user interactions and other events trigger transitions from one screen to another and cause updates of the internal application state. To test such applications, the developers face the challenge of constructing test inputs that exercise the functionality and cover all reachable application code. A test input consists
of a sequence of events, each with some parameters depending on the kind of event, for example, coordinates for click events and string values when text fields are filled in. In contrast to other kinds of software, a key challenge in mobile application testing is managing the explosion in the number of possible event sequences [7]. Since it can be difficult and tedious to construct these test inputs we wish to automate the work. We focus on Android, which has become the most widely used platform for mobile software. More specifically, our goal is to improve automated testing for Android applications that are not computationally heavy but may have complex user interaction patterns.

One popular technique is black-box random testing or crawling [6, 56, 63]. Other related approaches are based entirely on abstract models of the applications [92, 107] or only involve the application code for extracting, for example, available event handlers and basic information on how they access shared state [9, 68]. Common to these approaches is that they cannot effectively reach branches of the application code that are highly constrained by the event parameters. In contrast, symbolic execution, which analyzes the application code in more detail, has shown to be a powerful approach to find appropriate event parameters. However, most existing techniques that do apply symbolic execution are not able to effectively construct event sequences that consist of many events. For example, the experiments reported by Anand et al. [7] are limited to event sequences of length 4. Others, for example, Mirzaei et al. [85] use symbolic execution, but only for deriving the event parameters, not for the sequencing of events. As a consequence, these existing approaches are not able to effectively reach parts of the application code that require many events and with highly constrained event parameters.

We propose a targeted approach to generation of event sequences. Given a target location in the application code, for example, a branch that is not reachable with the existing automated testing techniques, we wish to find an event sequence that leads from the application entry to the target. Note that it may be easy to reach the entry of an event handler that contains the target but significantly more difficult to reach the target itself, since it may be guarded by conditionals that depend on events earlier in the event sequence.

Our approach is inspired by the work of Ma et al. who consider the line reachability problem for C programs [67]. They present an algorithm that works backward in the call graph from a given target, using traditional forward symbolic execution of each function, until it finds a feasible path from the start of the program. Call graphs of C programs resemble finite-state UI models of event-driven applications, however with the important difference that calls in a C program are controlled by the program itself whereas navigation in event-driven applications is largely controlled by the user. This means that a simple backward search toward the application entry would likely lead to an explosion of different paths to consider. To address this problem, we draw inspiration from another source: the model-based testing technique by Arlt et al. [9]. In model-based testing of event-driven applications, test inputs are
9.1. **INTRODUCTION**

constructed from a finite-state abstraction of the user interface behavior that shows how the event handlers are connected. In the technique by Arlt et al., the conventional UI model is augmented by event dependence information that for each pair of event handlers gives an indication of how much state may be written by one of them and read by the other. This information provides the basis for construction of abstract event sequences, which are subsequently extended to executable event sequences using the UI model. In our approach, when we search backward through the event handlers from the target, we use this idea of exploiting UI models and event dependence information to narrow the search space – although with some fundamental differences that we explain in Section 9.8. For the kind of UI models we use, each state represents a combination of registered event handlers, and transitions correspond to execution of event handlers. Previous work has shown that it is often possible to infer such UI models automatically [111].

Combining these ideas, our approach to targeted event sequence generation works in two phases:

1) We first preprocess the application by performing concolic execution [44] of each event handler to infer path conditions and symbolic states for its paths. The result is a summary for each event handler, reminiscent of the use of function summaries in compositional symbolic execution [43]. This phase is independent of the selected target.

2) Given a target location in the application code, the main phase uses the event handler summaries together with a UI model of the application to build a concrete event sequence that leads from the entry state of the application to the target. This is structured as a worklist algorithm where each worklist item consists of a path through one or more event handlers ending at the target. Each path is extended incrementally by searching for an event handler that may be triggered in front of the path to satisfy some of the constraints in the path condition, following the idea from Ma et al. [67]. This search uses the UI model and the symbolic states to bypass event handlers that are likely not relevant for the path condition. When candidate event sequences are found, we compose the path summaries and check satisfiability. This process continues until the entry state of the UI model is reached.

Our contributions can be summarized as follows:

• We present a framework for automated testing of event-driven applications that combines concolic execution and UI models for targeted generation of test inputs to reach application code that may require many events and with highly constrained event parameters. An important part of this approach is how concolic execution is applied to individual event handlers and the resulting summaries are composed for reasoning about event sequences. Another central idea is to extract information about data dependence between event handlers from the concolic execution and exploit this to narrow the search space.
• The framework can be tuned by the prioritization mechanism of the worklist. We suggest three example prioritization heuristics that consider different aspects of how execution of event handlers affect application state.

• We provide an experimental evaluation involving five Android applications. Our prototype implementation uses a novel approach to concolic execution that utilizes the debugging interface of the Android emulator. The experimental results show that the approach can successfully cover challenging targets that are beyond the reach of random testing and conventional model-based test sequence generation.

Targeted generation of application inputs can be useful not only for maximizing coverage in automated testing but also for reproduction of reported errors and evolution of test suites.

Although our work is motivated by practical challenges in Android application development and our experimental tool is built for this specific platform, we believe that our approach may also be applicable to other kinds of event-driven programs, such as, JavaScript web applications and desktop GUI applications. However, our approach is particularly suitable for mobile applications, where event sequences are often longer and event handlers are smaller than in web or GUI applications.

9.2 Motivating Example

In this section we introduce a simple Android application, TaxCalculator, that we use as a motivating example to illustrate different aspects of our approach.

![Figure 9.1: Two screens in TaxCalculator: (a) the INCOME entry screen and (b) the RESULT screen displaying the income, deductions, and resulting tax.](image-url)
TaxCalculator is a personal tax calculator used to compute the income tax liability for a given income amount. Figure 9.1 shows screen-shots illustrating its simplest use case.

On the entry screen, denoted INCOME, the user enters an income amount through a numeric keypad. Clicking the Calculate button takes the user to a result screen, denoted RESULT, which displays the calculated tax amount. By default this is a fixed percentage of the income amount. Figure 9.2 shows a part of the UI model that captures the relevant event sequences in this application. Intuitively, the states denote principal GUI screens and the transitions denote user actions, such as, clicks on buttons or changes to text fields. For example, $e_0, e_1, \ldots, e_9$ denote a click on the 0...9 buttons in the numeric keypad, $e_2$ is a click on Calculate, and $e_3, e_6,$ and $e_9$ are clicks on the device’s back button (in the lower left corner on the screen).

The default tax calculation can be modified by optionally specifying an income tax deduction amount that is deducted from the income before calculating the tax. To do this, the user must press the device’s menu button (in the top right corner of the screen), corresponding to $e_4$ in the UI model, to get to the MENU screen, and from there click a Settings widget, $e_5$, to access the preferences screen, PREF. That screen contains a radio button for toggling tax deduction calculation. The user needs to click this button, $e_7$, and then click another button, $e_8$, which opens a dialog box denoted by DEDUCT. Here, the user can specify the deduction amount via a text field, $e_{10}$. The value being entered here is available as an event parameter, which is abstracted away in the UI model. The back button can then be used to navigate back to PREF and further to INCOME, corresponding to $e_9$ and $e_6$, respectively. The user can now enter the income amount and click Calculate to perform the modified tax calculation.

Figure 9.3 shows a fragment of code from TaxCalculator, used for performing the tax calculation. It is executed each time the user clicks Calculate on the INCOME screen. This code, though simple, is not trivial to test. Specifically,
1 income = this.appState.enteredAmount;
2 deduction = 0;
3 if (Settings.getEnableTaxDeduction()) {
4     deduction = Settings.getTaxDeduction();
5 }
6 taxable = income - deduction;
7 if (taxable < 0) {
8     taxable = 0; // example target
9 }
10 tax = taxable * TAX_RATE;
11 result = income - tax;

Figure 9.3: Snippet from the onCreate method in the TaxResult activity in TaxCalculator.

the calculation contains an if-statement with the predicate taxable < 0. In order to reach line 8, the application must be configured to take tax deduction into account, and the tax deduction amount must be greater than the income entered. The shortest sequence of user actions that can fulfill this constraint contains 8 events: e4, e5, e7, e8, e10, e9, e6, e2. Moreover, the branch on line 7 depends on the text value entered at e10. In other words, reaching the target requires not only a long event sequence but also specific values in event parameters earlier in the sequence. Such a combination of requirements on event sequences and event parameters makes it difficult for existing automated testing techniques to reach the target line.

In our approach, we start at the given target at line 8. Concolic execution infers a path constraint that involves three variables in the application state: the income value, the deduction value, and the flag that controls whether tax deduction is enabled. It also infers event handler summaries that show which events may influence these variables. This information is then used when constructing event sequences.

9.3 UI Models for Event-Driven Applications

The literature on automated testing contains many different views on what constitutes an event-driven application. This section establishes the essential terminology that we use in the description of our proposed approach.

We view an event-driven application, in particular, an Android application, as a collection of event handler methods. During execution, event handlers can be attached to GUI widgets. An event handler registration is a triple of a GUI widget object, an event kind (click, text input, etc.), and an event handler method that has been attached to the widget. At any point during execution of the application we thus have a set of such event handler registrations. For simplicity, we assume that a single main method acts as entry point to
the application for setting up the initial event handler registrations. The application is then driven by a sequence of events, each triggering the execution of an event handler from the current set of event handler registrations. Events with no corresponding event handler registration are ignored. We focus on user events, which represent a human user’s interaction with the application, but our approach is equally applicable to system events that arise, for example, when new activities are created or paused. Some events are parameterized, for example, to indicate coordinates for click events or string values for text field alterations.

Our approach falls under the category of model-based testing. It operates on a UI model of the behavior of the graphical user interface of the Android application under test. Figure 9.2 shows a graphical view of the UI model for our motivating example. Formally, a UI model $\mathcal{M}$ is a finite-state machine denoted by a 4-tuple $\mathcal{M} = (S, s_0, E, T)$. Here, $S$ is a finite set of abstract states representing different GUI screens, where $s_0 \in S$ is the initial state that describes the opening screen after the main method has been executed. $E$ is a finite set of event handler registrations, as defined above, and $T \subseteq S \times E \times S$ is a transition relation, corresponding to the edges in the graphical view. Each abstract state $s \in S$ is uniquely characterized by its set of event handler registrations defined by $R_s = \{ e_i \in E \mid (s, e_i, _) \in T \}$. We sometimes refer to event handlers and event handler registrations simply as events when the meaning is clear from the context.

A sequence of events $\langle e_1, ..., e_n \rangle$ is consistent with a sequence of states $p_0, ..., p_n$ where each $p_i \in S$ if for each $i = 1, ..., n$, either $(p_{i-1}, e_i, p_i) \in T$ or $e_i \notin R_{p_{i-1}}$. The latter case accounts for ignored events. In this way, every given sequence of events gives rise to a non-empty set of state sequences through $\mathcal{M}$.

A UI model is sound if it represents an over-approximation of the possible behavior of the application. More precisely, for any sequence of events $e = \langle e_1, ..., e_n \rangle$, let $R_e$ denote the set of event handler registrations that exist after executing $e$ on the concrete application starting from its entry state. For $\mathcal{M}$ to be sound we now require that there exists a state sequence $p = p_0, ..., p_n$ that is consistent with $e$ and where $p_0 = s_0$ and $R_{p_n} = R_e$. Using an unsound UI model may prevent exploration of valid event sequences. Conversely, over-approximation could suggest infeasible event sequences, however, such sequences will be rejected by our algorithm, which tests candidates using concrete execution.

### 9.4 Approach Overview

Given an Android application under test, a UI model of the application, and a set of targets, the objective of our technique is to generate a test case for each target, that is, an event sequence that brings the application from its
initial state to the target. Such targets, which can be lines or branches in
the application code, arise in a number of different scenarios, as discussed
in Section 9.1. The UI model could be specified manually or generated au-
tomatically through one of the model generation techniques proposed in the
literature (see Section 9.8).

The motivating example in Section 9.2 demonstrates that reaching a tar-
et generally requires execution of a series of event handlers that mutate the
program state, sometimes based on strings or numbers provided by the user
in the form of event parameters, and navigation between these event handlers,
ultimately executing the event handler that contains the target. More gen-
erally, our study of Android applications suggests that executions exercising
specific targets often have a particular structure:

- There exists a small set of events, which we call anchor events, that are
  responsible for setting the necessary program state for a target to be
  executed.

- There is a disjoint set of events used only for connecting the initial state,
  the anchor events, and the target. These connector events do not affect
  the program state used at any anchor event or at the target.

For example, in the test case \(e_4, e_5, e_7, e_8, e_9, e_6, e_2\) for the target at line 8
in Figure 9.3, \(e_7\) and \(e_{10}\) are the anchor events, \(e_4, e_5, e_8, e_9,\) and \(e_6\) are
connector events, and \(e_2\) exercises the target.

These observations motivate the key idea of our target event sequence
generation algorithm. We identify a series of anchor events in reverse chrono-
logical order, starting at the target. The anchor events guide the search for
a feasible test case by focusing on identifying events and paths in the appli-
cation that are indispensable for reaching the target. In effect, this prunes
away many sequences that can never reach the target. Further, we need to
find suitable connector events to connect the initial state with the sequence
of anchor events to the target. Thus, our approach works backward from the
target, iteratively identifying anchor events and connector events, extending
a partial sequence, until the initial state is reached.

We use symbolic analysis of the application source code to identify anchor
events, build feasible paths exercising the target, and compute appropriate
values for user event parameters. The UI model is used as the basis for
selecting suitable connector events to connect the initial state, the anchor
events, and the targets. To build a test case exercising a target, our analysis
reasons at the level of individual execution paths within the event handlers.
We refer to an execution path in an event handler that is triggered by an
anchor event as an anchor path (or simply, an anchor). Similarly, an execution
path for a connector event handler is called a connector path (or simply, a
connector).
We note that the same event handler may be considered many times in the construction of a test case. To exploit this, we compute a symbolic summary of each event handler, once, in a target agnostic manner. This is a key ingredient of our approach. The construction of test cases now uses these summaries, without considering the actual application code.

Our overall approach is thus divided into two phases: a target agnostic symbolic summarization phase, followed by a sequence generation phase that searches for a test case for each target:

**Symbolic Summarization** This phase operates on the executable Android application. Symbolic analysis is applied to each event handler in turn to produce an event handler summary characterizing its behavior. This summary ideally includes necessary data and control-flow information about every execution path in the event handler code.

**Sequence Generation** This phase uses the event handler summaries generated in the first phase, along with the UI model to find a test case for a given target. In this search, the UI model and event handler summaries are used both to limit the search space, and as guides for the search space exploration order. The event sequence generation algorithm starts from the target and builds a sequence of events backward until it reaches the initial state, combining individual paths from the event handler summaries compositionally. In order to avoid false positives, candidate event sequences are executed concretely, using the executable application.

The algorithms for these two phases are presented in the following sections.

### 9.5 Symbolic Summarization

The symbolic summarization phase preprocesses the application code to produce a symbolic characterization, called an event handler summary, for each event handler. The event handlers can be located either using the UI model or by a simple static analysis of the Dalvik bytecode. An event handler summary is a set of path summaries, one for each execution path within the event handler code. Execution paths and their summaries encompass not only the event handler method itself but also other methods that may be called directly or indirectly from that method. A path summary $W$ for a path $P$ is a symbolic representation of the behavior of $P$, as in classical symbolic execution [65].

More formally, a path summary is denoted by a triple $W = (pc, \sigma, \tau)$, where $pc$ is the symbolic path condition of that path, $\sigma$ is the symbolic state at the end of the execution of $P$, and $\tau$ is a log of bytecodes executed in the path, which serves as a unique signature of the path itself. The symbolic state, $\sigma$, is a map from variables in the application state to symbolic expressions,
1: function sequenceSearch(target, summaries, model)
2:  worklist = initialize(target, summaries, model)
3:  while worklist is not empty do
4:    partialSequence = dequeue(worklist)
5:    extendedPartialSequences = empty list of sequences
6:    for anchor in anchors(partialSequence, summaries, model) do
7:      for path in paths(anchor, partialSequence, summaries, model) do
8:        newPartialSequence = combine(anchor, path, partialSequence)
9:        if isComplete(newPartialSequence) then
10:          potentialTestCase = extractTestCase(newPartialSequence)
11:          if reachesTarget(potentialTestCase, target) then
12:            return potentialTestCase
13:          end if
14:        end if
15:        append(extendedPartialSequences, newPartialSequence)
16:      end for
17:    end for
18:    enqueue(worklist, extendedPartialSequences)
19:    prioritize(worklist, extendedPartialSequences)
20:  end while
21:  return no test case found
22: end function

Figure 9.4: The event sequence generation algorithm. The input target denotes the
target of interest, summaries is the set of all handler summaries produced in the
symbolic summarization phase, and model is the UI model of the application. The
algorithm either returns a test case that reaches the target, returns that it is unable
to find a test case, or it diverges.

such that \( \sigma(v) \) represents the value of \( v \) at the end of the execution of \( P \). The
values of event parameters and object fields are treated symbolically.

Event handler summaries can be computed by performing concolic execution [44]. Each iteration of concolic execution symbolically explores one
path and hence computes its path summary. In this way, both the state that
is shared between event handlers and the event parameters are treated sym-
bolically at the entry of the event handler, so the event handler summary
characterizes the event handler in its most general environment, independent
of the preceding event sequence and event parameters.

In practice, concolic execution may not be able to cover all possible execution
paths within a given event handler. This means that our event handler
summaries may be incomplete, which can potentially affect the efficacy of
our overall approach. However, this possibility is mitigated by the fact that
event handlers in mobile applications are often relatively small, with much of
the complexity of the application code lying in the dependencies between the
event handlers.

Note that reachability of a given target in an event handler \( e_i \) cannot
be decided based on the summary of \( e_i \) alone. In case a path from the entry
point in \( e_i \) to the target has a nontrivial path condition \( pc \), we need to produce
an event sequence that brings the application from its entry state to a state
where \( e_i \) can be triggered, i.e. it exists as an event handler registration, and
moreover, \( pc \) is satisfied. We address this challenge in the following section.

9.6 Sequence Generation

The event sequence generation phase generates a test case for each given tar-
get, based on the event handler summaries generated in the symbolic summa-
ration phase and the UI model. For this phase, we propose an algorithm that
gradually explores sequences of events backward, from the target to the
application entry point. The algorithm, given in Figure 9.4, is organized around
a prioritized worklist of partial sequences that are gradually extended until a
complete sequence is found. The prioritization mechanism guides the selection
of worklist items to be explored next.

A partial sequence is a sequence of path summaries representing a concrete
path \( \langle \tau_1, \tau_2, \ldots, \tau_n \rangle \) through the application, combined with an abstract state
\( s \) in the UI model. Each \( \tau_i \) is a complete path in an event handler for an event
\( e_i \) of the UI model, where the segment \( \tau_n \) exercises the target. The event
sequence \( \langle e_1, e_2, \ldots, e_n \rangle \) is consistent with a state sequence starting from \( s \) in
the UI model.

The initialize function (line 2) initializes the worklist as follows. For each
path summary \( W \) that exercises the target of interest (that is, the bytecode
log of the path summary contains the target) and each abstract state \( s \) in the
UI model where \( s \) has an outgoing transition labelled \( e_i \) such that \( W \) belongs
to \( e_i \), we add the partial sequence of length 1 defined by \( W \) and \( s \) to the
worklist. For the example in Section 9.2, the target is the event handler for
\( e_2 \), which appears as an outgoing edge from INCOME in Figure 9.2. Only a
single path summary exercises the target in this example, so the worklist will
be initialized to a single partial sequence defined by that path summary and
the INCOME abstract state.

Next, the main search loop is entered (lines 3–20). A partial sequence is se-
lected from the worklist and extended into a number of new partial sequences.
This extension is conducted in two steps:

1. A set of anchors for the partial sequence is found using the ANCHORS
function (line 6) described in Section 9.6.1. This function provides a set
of event handler paths, each of which (1) write to some program state
that the partial sequence depends on according to its path condition,
and (2) has a symbolic state that is consistent with the path condition
of the partial sequence.

2. For each anchor, we extract a set of feasible sequences of connectors
that lead from the anchor to the partial sequence (line 7), using the
PATHS function described in Section 9.6.2. For each sequence, we con-
struct a new partial sequence consisting of the anchor, the sequence of
connectors, and the original partial sequence.

We say that a partial sequence is complete if it starts at the entry state
of the application and reaches the target. Such a sequence may give rise to
a concrete test case for the target. The isComplete function checks if an
extended partial sequence is complete (line 9). In that case, we extract a po-
tential test case using the extractTestCase function (line 10) and check
that it reaches the target when executed concretely by the reachesTarget
function. If the new partial sequence is not complete, it is added to a list of ex-
tended partial sequences (line 15). On lines 18–19 these partial sequences are
added to the worklist, and their priorities are computed using the prioritize
function that we describe in Section 9.6.3.

9.6.1 Construction of Anchors

The anchors function produces a set of anchors for a partial sequence. Recall
from Section 9.4 that an anchor is an execution path in an event handler that
writes to some program state that the partial sequence depends on, according
to its path condition. We define the dependency set of a partial sequence as
the set of variables that occur in its path condition. In this way, an anchor
corresponds to an execution path in an event handler that affects the values
in the dependency set and thereby potentially discharges some of the clauses
in the path condition of the partial sequence.

The anchors are identified using the UI model and the event handler sum-
maries. First, we perform a breadth-first backward traversal in the UI model,
starting from the abstract state of the partial sequence, until the nearest an-
chors are located. More precisely, at each traversed transition in the UI model,
the dependency set of the partial sequence is compared with each path sum-
mary that belongs to the event handler of the transition. A path summary is
marked as an anchor if it affects the dependency set.

In the example from Section 9.2 the target in the event handler e2 shown in Figure 9.3 depends on the symbolic constraint variable Settings.enableTaxDeduction. A partial sequence containing a path sum-
mary for e2 will include this variable in its dependency set. Since the path summaries for the event handler e7 all affect this particular variable, they will
be identified as anchors for the partial sequence.

Some of these anchors, however, can safely be pruned away. If the symbolic
state of an anchor is inconsistent with the path conditions of the current
partial sequence (i.e. their conjunction is unsatisfiable), then using this anchor
for extending the partial sequence would not lead to any feasible paths. By
removing such anchors from further consideration, we effectively reduce the
search space of the sequence generation algorithm. The resulting set of anchors
is returned by the anchors function.
As mentioned, the idea of using anchors is to guide the sequence generation from the target backward toward the entry state. For a complete sequence, that is, a partial sequence that has reached the goal, the dependency set is empty. The idea in our algorithm is that putting an anchor in front of a partial sequence will likely reduce the dependency set. However, there is no guarantee that the dependency set is in fact reduced by this step, since the anchor itself may introduce additional dependencies. Our experimental evaluation in Section 9.7 investigates how the use of anchors guides the search in practice.

9.6.2 Construction of Connector Sequences

The \texttt{paths} function generates a set of possible connector sequences between the given anchor and partial sequence. For this, we use the UI model to find all sequences of connectors between the two. Each of these sequences have the following two properties: (1) it corresponds to an acyclic path in the UI model from a transition that has the anchor as label to the abstract state at the beginning of the partial sequence, and (2) none of the connectors, where each corresponds to a single transition in the UI model, is an anchor. The first property can be ensured using a basic graph traversal algorithm. Section 9.6.1 provides the information for ensuring the second property.

Each connector sequence that has these properties is a candidate for connecting the given anchor and partial sequence. Not all of these candidates are feasible, however. If the symbolic state of the anchor is inconsistent with the composition of the symbolic summaries of the connectors, then no corresponding concrete path exists. The remaining feasible paths are then returned by the \texttt{paths} function.

Continuing the example of the partial sequence containing a path summary for $e_2$ and an anchor for $e_7$ from Section 9.6.1, a path summary for $e_6$ is a connector, since it connects the two in the UI model and it does not affect the dependency set of the partial sequence.

9.6.3 Prioritization

A key part of the algorithm is the \texttt{prioritize} function that assigns priorities to newly added partial sequences. This function initially selects the priority of a new sequence as the priority of the sequence it extends. The priority is then adjusted using a series of \textit{reprioritization functions} representing different heuristics that we describe in the following.

Equivalent-Anchors Reprioritization

An event handler summary consists of a set of path summaries. When extending a partial sequence with anchors, we look at their path summaries to
determine if they write any program state that the partial sequence depends on. Since multiple paths in an event handler can result in the same mutation of the variables that appear in the dependency set of the partial sequence, the resulting set of anchors will likewise contain multiple candidates with the same effect. Each of these anchors results in a new partial sequence in the worklist.

As an example, if we assume the dependency set is \{income\} and we consider the event handler in Figure 9.3, there exist multiple paths through the event handler that all have the same effect on income, so giving them the same priority would lead to redundant work.

Our first reprioritization function exploits this observation by lowering the priority of all the involved partial sequences, except one that we pick arbitrarily. Finding the anchors that have an equivalent effect relative to the dependency set can be done by comparing the constant values and symbolic values in their symbolic states.

Connector Reprioritization

There may exist multiple sequences of connectors between a given anchor and a partial sequence. Recall our observation in Section 9.4 that these connectors only navigate between screens in the application, without affecting the program state that the partial sequence depends on. In many cases, any of these paths will suffice, and it would only lead to a path explosion if we try to follow all of them.

Based on this, we introduce a second reprioritization function that exploits this observation by lowering the priority of all the involved partial sequences, except one that we pick arbitrarily, similar to the previous reprioritization function.

Increment-Decrement Reprioritization

Another common pattern is pairs of path summaries, in which one changes some state, and another reverts those changes. This general pattern manifests itself in a number of concrete instances, such as add/remove buttons that mutate a collection of items, buttons for incrementing or decrementing a number, or buttons toggling a value.

Extending partial sequences with path summaries that simply undo changes is not productive and may lead to unnecessary exploration of paths. Note that we only care about parts of the program state that are in the dependency set of the current partial sequence. Our third reprioritization function aims at decreasing the priority of any partial sequence where this pattern is found. This is not trivial to detect precisely, however. A simple approximation is to consider only numeric counters and boolean flags. Whenever the reprioritization function identifies a pair of path summaries where one increments
some variable the other decrements the same variable, then the priority of event sequences that mix the two path summaries is lowered, and similarly for boolean flags.

9.7 Evaluation

To evaluate the practical usefulness of our approach, we have implemented the proposed event sequence generation algorithm and supporting infrastructure in a tool called Collider. We now consider the following research questions:

Q1. Is our algorithm able to generate test cases for challenging targets in real-world Android applications? We view a target as being “challenging” if it cannot be reached with traditional random testing or model-based testing techniques.

Q2. Does the use of anchors and connectors have an effect on the ability to reach the targets? A simple alternative would be a backward breadth-first search in the UI model.

Q3. Do the prioritization heuristics have an effect on the ability to reach the targets? If that mechanism is disabled, the partial sequences in the worklist will be treated in a random order.

9.7.1 Implementation

Collider is implemented with approximately 8,000 lines of Java code excluding libraries. The part implementing the sequence generation phase closely follows the pseudo-code from Section 9.6, whereas the part for symbolic summarization requires more explanation.

A central part of Collider is the concolic execution engine for symbolically summarizing event handlers as described in Section 9.5. As the concolic execution is performed at the level of event handlers (including methods called in the process), application state that may be shared between event handlers, in particular, all object fields, are initialized with symbolic values. Collider operates directly on the Dalvik bytecode of the compiled Android applications. We do not differentiate between application code, Android library code, and the Java standard library; however, the symbolic execution uses mocks for more precise treatment of some basic library methods.

The concolic execution engine must be able to evaluate Android applications concretely, inspect the evaluation and program state, and modify the program state in order to explore new branches. In Collider, concrete execution is handled by the Android emulator provided by the Android SDK, which ensures a correct execution of the application. All interaction between Collider and the Android emulator is handled by a combination of the ordinary instrumentation framework for testing Android applications and the
debugging interface in the Android VM. Via the debugger, breakpoints are inserted after each bytecode instruction, such that the symbolic execution can be performed in parallel with the concrete execution in a lock-step manner. Using this technique, neither the application nor the emulator needs to be modified in any way, which simplifies the implementation.

For the symbolic execution, we reuse parts of the solver infrastructure from Symbolic Java PathFinder\(^1\), which in turn relies on underlying solvers, such as, Yices\(^2\). The solver infrastructure is also used to check the feasibility of partial sequences, as described in Section 9.6. This implementation currently supports basic constraints on numbers, booleans, strings, and arrays. The Smali\(^3\) disassembler is used for extracting various pieces of information about the Dalvik bytecode, and the testing library Robotium\(^4\) is used for simulating user interactions with the application.

### 9.7.2 Benchmark Applications and Targets

Our evaluation has been conducted on five Android applications selected using the following criteria: (1) the source code for the applications must be available to allow us to manually inspect the application behavior, (2) we only consider applications that are UI driven and not computationally intensive, so we exclude games and system services, (3) to get interesting targets, the applications must contain branches that depend on previous events or event parameters, and (4) the applications should represent different application categories, such as productivity, entertainment, and tools, and from differ-

---

\(^1\) [http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc](http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc)

\(^2\) [http://yices.csl.sri.com/](http://yices.csl.sri.com/)

\(^3\) [http://code.google.com/p/smali/](http://code.google.com/p/smali/)

The five applications are: TippyTipper (1,800 LOC), a tip percentage calculator including tax calculation and functionality for splitting a bill between a group of people; ConnectBot (33,000 LOC), a SSH client with support for public/private key management; MunchLife (400 LOC), a utility for keeping score in a card game; OpenManager (2,500 LOC), a file manager with support for viewing, moving, and copying files; and DieDroid (1,900 LOC), an application for virtual dice rolling using a number of different systems and conditions on the die rolls.

We have manually built a UI model of each application. This could in principle be done automatically, as discussed in Section 9.8, but no suitable tool was available to us when conducting the experiments. These UI models are sound in the sense described in Section 9.3.

To obtain a baseline for comparison, we combine two existing approaches. First, we use a simple crawler that produces events systematically based entirely on the UI models, without considering the application code. Second, we use the random testing tool Monkey provided by the Android SDK.

This tool fires a large number of random events and periodically restarts the application. When we run these tools on the benchmark applications, we observe that coverage first rises rapidly and then stabilizes. We select a time budget that allows the stabilization to be reached. As a result, the different benchmarks have been exercised using 3,000 to 6,000 events each. Now, we define that a branch in a benchmark application is “challenging” if neither of these two tools is capable of producing event sequences that reach the target. Although the tools involve randomization, this classification appears to be reasonably robust.

For our evaluation of Collider, we only consider the challenging targets. In practice, this means that all targets of interest depend on the sequencing of events beyond what is expressible in the UI models alone.

We focus on branches in the application code, not on those in the Android SDK and external libraries. Moreover, we exclude dead branches, i.e. those that cannot be reached with any sequence of events according to a manual inspection. As our focus is on user events, we also exclude branches that depend on external data, such as, the file system or device configuration. For the same reason, we populate the initial application states with meaningful data, such as, files for OpenManager and a valid SSH connection for ConnectBot.

The first two columns of numbers in Table 9.1 show for each benchmark (1) the number of targets of interest that depend on event sequencing, but not on event parameters, and (2) the number of targets that depend on both parameters.
event sequencing and event parameters. This classification is obtained by a manual inspection of each target.

9.7.3 Results

Q1: We answer research question Q1 by applying our event sequence generation algorithm on the selected targets. The column named Reached in Table 9.1 shows the number of targets where a successful test case is generated. For example, 7 of the 16 targets of interest in TippyTipper are reached. As we only consider targets where the baseline tools fail, we conclude that our proposed algorithm is capable of producing test cases for challenging targets.

As an example of a challenging target, the DieDroid benchmark contains a screen that shows a number of rolled dice, marked red, green or gray depending on user-defined winning and failure thresholds. In one event handler, a particular branch can only be reached if the winning threshold is larger than the loosing threshold. To reach this target branch, our algorithm identifies a path summary exercising the branch. The algorithm then continues to extend the partial sequence backward, finding anchor points in two separate dialogs that set these thresholds, and inserting connector events as necessary. Each of these anchors is parameterized by user input, for which the solver identifies two values that satisfy the path condition. After extending the partial sequence to the application entry point, the tool outputs a concrete event sequence that reaches the target.

We have manually inspected all the targets that were not reached by Collider to investigate whether the reason is due to limitations in the symbolic summarization phase or due to the assumptions we make in the sequence generation phase. In the former case, the limitations can perhaps be remedied by improving the concolic execution engine; in the latter case, more fundamental changes to our approach might be necessary to increase the coverage further. The number of missed targets in the first category are shown for each benchmark in the column named Potential in Table 9.1. We observe that none of the missed targets are in the second category. Our prototype implementation only supports symbolic reasoning of numeric values and booleans, resulting in imprecise treatment of, for example, strings and objects. A closer inspection reveals that this particular source of imprecision is a dominant cause of missed targets. For this reason, we believe that many of the challenging targets that are not reachable with our current implementation can potentially be reached with realistic improvements of the symbolic analysis, without requiring modifications of the main sequence generation phase. Naturally, we will pursue this in our further work.

The event sequence generation phase, which is the central part of our algorithm, typically takes less than one minute per target where a matching event sequence is found (running on an ordinary i5 3.1GHz PC). A few outliers, for example in TippyTipper, require up to 30 minutes. This is caused by a
large number of connectors between partial event sequences and anchors. An adjustment of the prioritization mechanism could perhaps give a performance improvement in this case, but a more thorough experimental study would be necessary to investigate this further.

The symbolic summarization phase is more time consuming. Our simple prototype implementation runs for 3 to 5 hours on each benchmark. However, note that this phase is doing preprocessing, independent of the choice of targets. Moreover, the current implementation naively analyzes each event handler in isolation, without taking into account that event handler methods often share common functionality via other methods. Thus a considerable amount of time is spent re-analyzing shared methods. With additional implementation effort, we believe that well-known techniques can be adapted to avoid this redundancy, which we return to in Section 9.8. Another reason is the implementation approach: Basing the concolic execution engine on single-stepping via the Android VM debugger does lead to a relatively simple implementation, but it naturally incurs a substantial overhead.

Q2: To answer research question Q2, we investigate how the use of anchors and connectors reduces the search space compared to a simple backward breadth-first search.

The distribution of observed test case lengths is shown in Figure 9.5. The sequence lengths range from 4 to 102 events, however, with the test case containing 102 events being an outlier. On average, a test case consists of 10 events if we exclude this outlier. The average test input length for each benchmark is shown in the ‘Average size Test case’ column in Table 9.1. With such relatively long event sequences, a simple backward breadth-first search would lead to an explosion of possible paths.

Of all the generated test cases excluding the outlier, 53% of the events are connector events. The average number of connector events per test case is
shown in the Connectors column in Table 9.1. Since a considerable part of the events are connectors, the ability to jump between anchors is an advantage compared to a backward breadth-first search.

The idea of pruning potential anchors by checking consistency of the symbolic states and the path conditions further reduces the search space. The rightmost column in Table 9.1 lists the pruning of anchors for each benchmark. The pruning eliminates 38%-71% of the anchors. Since this happens in each step of the backward search, it adds up to a substantial reduction of the search space.

Q3: For research question Q3, we disable the prioritization functions and run Collider again. In theory, we are still able to reach the same targets, however, we expect a slower pace due to the larger number of paths that need to be considered before finding test cases that reach the targets. We want to test if Collider is still able to reach the same targets, even if we allow ten times as many iterations of the worklist algorithm compared to number of iterations used when the prioritization functions are enabled.

Running our algorithm again, 21 of the 46 targets are now unreachable. Moreover, for the remaining 25 branches that are still reached, the total running time for the sequence generation has increased from 45 seconds to 2.5 hours. Thus, we conclude that the prioritization heuristics have a considerable impact on the ability to reach targets within reasonable time.

A possible threat to validity in our evaluation is whether the selected benchmarks represent the range of real-world applications in use. All of the selected benchmarks are real-world applications publicly available in the Android marketplace, and they have been selected in accordance with the criteria stated in Section 9.7.2. The nature of our evaluation, involving manual inspection of the benchmarks and manual construction of UI models, reduces the feasibility of scaling the evaluation to a larger number of benchmarks. However, these preliminary experiments demonstrate the potential of our algorithm.

9.8 Related Work

Our work builds on a significant body of work in symbolic execution and model-based testing. We first discuss related work involving symbolic execution for event-driven programs.

As in our approach, the ACTEve technique by Anand et al. [7] performs automated testing for Android applications using concolic execution. However, their approach explores the application starting from its entry point, not aiming for particular targets. Concolic execution is applied at the level of the entire application rather than on individual event handlers. Moreover, concolic execution is used for reasoning about low-level properties of events, such as coordinates for tap events, which we can treat more abstractly by
the use of UI models. Despite applying a subsumption mechanism to filter away certain event sequences, their approach apparently does not scale beyond event sequences consisting of more than four events.

The Barad framework by Ganov et al. [39] performs automated testing for SWT GUI applications, which are also event-driven. It first symbolically executes each event handler, not to produce path summaries as in our approach, but to discover registered event handlers and build a model of the application similar to the UI models we use. Next, a set of abstract event sequences are produced from the model, and symbolic execution is performed on each sequence to produce concrete test inputs. Mirzaei et al. [85] generate tests for Android applications using a similar approach by first producing abstract event sequences based on application models and then running Symbolic PathFinder to perform symbolic execution on each sequence. In contrast, our approach utilizes information from symbolic execution also when constructing the sequencing of events. Several other symbolic execution tools have been build specifically for Android [62, 108]. Related tools for automated testing of web applications, which are also driven by user events, include Apollo [11] for PHP and Kudzu [100] for JavaScript. Common to these frameworks and tools is that they do not create event sequences in a targeted manner but explore the given application from its entry point.

As mentioned in Section 9.1, our targeted approach to generation of event sequences resembles call-chain-backward symbolic execution by Ma et al. [67], although we consider relations between events rather than function calls. In their approach, call sequences are generated backward from the target one function at a time. We also construct event sequences backward, but using anchors and connectors to narrow the search, as explained in Section 9.6.

The idea of guiding automated testing using data dependence appears in many techniques [7, 9, 11, 16, 36]. Of particular relevance is the one by Arlt et al. [9] that we also mentioned in Section 9.1. In their technique, abstract event sequences are constructed based on how event handlers read and write shared state and subsequently concretized using a UI model, but reasoning at the level of entire event handlers rather than individual paths through event handlers. A novel feature of our approach is that path-specific data dependence information is extracted from event handler summaries that have been created using concolic execution.

For the symbolic summarization phase, we currently use traditional concolic execution, also called dynamic symbolic execution, or directed automated random testing [44], at the level of event handlers. We can in principle benefit from the numerous improvements that have been proposed to that basic technique. Specifically, we suspect that performance of the symbolic summarization phase can be improved using compositional dynamic test generation [43], which involves method summaries, orthogonal to our use of event handler summaries.

Alternatives to symbolic execution for automated testing include random
testing, search-based testing, and model-based testing. Monkey, which we used for the experiments in Section 9.7, is a popular random testing tool for Android that has been shown to be effective for bug finding [56]. The tools A²T² [3], AndroidRipper [6], iCrawler [63], and EXSYST [47] enhance random testing by using the application GUI to guide the testing. These light-weight techniques can be a good starting point for automated testing. However, as they have a black-box view on the application code, they are generally unable to reach the challenging targets that require many events and with constrained event parameters and specific execution paths in the event handlers, as shown in Section 9.7.

Tools such as Artemis [11] and to some extent also Dynodroid [68] employ feedback-directed automated testing, which is based on random testing but prioritizing using information gathered during the testing. Such techniques can often obtain good coverage with fewer test inputs than traditional random testing and faster than techniques that involve symbolic execution, yet they are not suitable for the more challenging targets that we focus on here.

Model-based testing approaches [97] organize the testing around a model of the application under test. For the event-driven applications we consider, the models express over-approximations of the relevant event sequences by abstracting away from the event parameters and the different execution paths that exist in the event handlers. Some tools extract tests for Android applications directly from such models using random or combinatorial approaches [92, 107], without involving symbolic execution.

The models used in model-based testing may be specified manually or generated automatically. The GUITAR tool by Memon et al. [77] is among the earliest and most well known approaches for reverse engineering models of GUI applications. It extracts the model using automated crawling. A recent extension, AndroidGUITAR, supports Android applications. The ORBIT tool by Yang et al. [111] is a variant that builds models that are tailored to the Android event system. Several of the other techniques that we have mentioned above also automatically construct models [7, 39, 85]. Although the various techniques involve different kinds of models, each of them can in principle provide the information we need for the UI models described in Section 9.3.

We distinguish between anchor events and connector events, however, other classifications exist. As an example, Xie and Memon [110] categorize events according to whether they manipulate the GUI while we focus on how the events modify data.

## 9.9 Conclusion

We have presented a targeted algorithm for automated testing of event-driven systems, in particular Android applications. The algorithm is tailored to targets that require long event sequences and reasoning about event parameters.
We have evaluated the effectiveness of this algorithm on a small suite of real-world Android applications, aiming for targets that are beyond reach for traditional random testing and model-based testing techniques. Our prototype implementation, Collider, successfully produces event sequences for many of the challenging targets.

Moreover, we believe that a large part of the remaining targets can also be reached using the algorithm, provided that the symbolic constraint solver component is extended with better support for, in particular, strings and arrays. We leave that for future work. Also, we plan to apply some of the techniques suggested in the literature on concolic execution, for example, compositional dynamic test generation, to improve performance of the symbolic summarization phase. Another practical limitation of our current prototype is that it requires UI models as input. This can in principle be remedied by integrating existing algorithms for automatic UI model construction. Such an extension of the implementation would enable a larger scale experiment in which Android applications are automatically analyzed and tested. For this purpose, it is practical that our approach works on bytecode and does not need access to the source code of the applications.
Chapter 10

Server Interface Descriptions for Automated Testing of JavaScript Web Applications

Abstract

Automated testing of JavaScript web applications is complicated by the communication with servers. Specifically, it is difficult to test the JavaScript code in isolation from the server code and database contents. We present a practical solution to this problem. First, we demonstrate that formal server interface descriptions are useful in automated testing of JavaScript web applications for separating the concerns of the client and the server. Second, to support the construction of server interface descriptions for existing applications, we introduce an effective inference technique that learns communication patterns from sample data.

By incorporating interface descriptions into the testing tool Artemis, our experimental results show that we increase the level of automation for high-coverage testing on a collection of JavaScript web applications that exchange JSON data between the clients and servers. Moreover, we demonstrate that the inference technique can quickly and accurately learn useful server interface descriptions.

10.1 Introduction

Many modern web applications run in browsers as HTML-embedded JavaScript programs that communicate with a server. The JavaScript code reacts to user events and asynchronously sends HTTP requests to the server for updating or retrieving data. The response from the server is used for example to dynamically modify the HTML page. With this so-called Ajax style of structuring web applications, the server mostly acts as a central database seen from the client’s point of view. The server interface comprises a collection of opera-
tions, identified by URLs, that accept input and produce output typically in XML or JSON data formats.

Some web service providers have public APIs, such as Google, Twitter, and Facebook, that are well documented and used by many client applications, for example in mashups that each use small parts of different APIs. In contrast, in many other web applications, the server-side and the client-side are developed in conjunction within the same organization. In such web applications, the programming interface of the server is often not described in a formal way, if documented at all. This can make it difficult to modify or extend the code, even for small web applications. More concretely, we have observed that it limits the possibility of applying automated testing on the JavaScript code in isolation from the server code and database contents.

It is well known that precisely specified interfaces can act as contracts between the server code and the client code, thus supporting a clean separation of concerns and providing useful documentation for the developers. In this work, we show that having formal descriptions of the programming interfaces of the server code in Ajax web applications is instrumental when conducting automated testing of the JavaScript code in such applications. In addition, we present a technique for automatically learning server interface descriptions from sample data for pre-existing web applications.

As an example, consider the JavaScript code in Figure 10.1, which is part of a web application that manages attendance lists for meetings. When the function `goto_page` is called, an Ajax request is sent to the server via the jQuery library.\(^1\) This request takes the form of an HTTP GET request with a specific URL and the parameters `page` and `query`. The `dataType` value `"json"` on line 15 indicates that the response data is expected to be formatted using JSON, a widely used format because it integrates well with JavaScript.\(^2\)

When the response data arrives, the function `populate_table` is called via line 17. By inspecting that function (lines 21–43) we see that the JSON data is expected to consist of an array of objects with specific properties: `id`, `name`, `email`, `department`, and `checkedin`. Moreover, their values cannot be arbitrary. For example, the `checkedin` property is used in a branch condition, so it probably holds a boolean value, and the other properties appear to hold strings that should not contain special HTML characters, such as `<` or `&`, since that could lead to malformed HTML when inserted into the page on line 41.

Figure 10.2 shows an example of an actual JSON response that may appear.

In this example—as in many JavaScript web applications in general—the interface between the server and the client is not made explicit. As a consequence, the server code and the client code become tightly coupled, so it becomes difficult to change either part without carefully checking the consequences to the other part. For instance, the server code could safely omit

---

\(^1\)http://jquery.com/
\(^2\)http://json.org/
12 function goto_page(id, q) {
13  jQuery.ajax(GET_PAGE_URL + '?page=' + id +
14  '&query=' + q,
15  {'dataType': 'json',
16  'success': function(result) {
17    populate_table(result);
18  }});
19 }

20 function populate_table(attendees) {
21  var table = $('#attendees');
22  table.html('');
23  for (i = 0; i < attendees.length; i++) {
24    var a = attendees[i];
25    var style = '';
26    if (a.checkedin) {
27      style = ' style="background-color: #B6EDB8;"';
28    }
29    ahtml = '<tr id="row' + a.id + style + '"><td><b>' + a.name + '</b> - ' +
30      a.email + '<br/>' + a.department + '</td>' +
31      '<td><a href="#" onclick="info(' + a.id + ')">[info]</a> ' +
32      '<a href="#" onclick="checkin(' + a.id + ')">[checkin]" +
33      '<a href="#" onclick="del(' + a.id + ')">[delete]</a>' +
34      '</tr>);
35  table.append(ahtml);
36  }
37 }

Figure 10.1: A typical example of Ajax in JavaScript.

Figure 10.2: Example JSON response for the Ajax interaction from Figure 10.1.

the checkedin property when the value is false without breaking the client code, since a .checkedin on line 27 would then evaluate to undefined, which is coerced to false, however, the necessity for such subtle reasoning makes the application fragile to modifications. Also, the client code implicitly assumes that escaping of special HTML characters has been performed on the server, but this may not have been communicated to the server programmer.

One aim of our work is to advocate the use of formal interface descriptions as contracts between the client code and the server code. In the example above, an interface description could specify what are valid request parameters and the details of the response data format, such that the server code and the client code to a larger extent can be developed separately. Interface descriptions are the key to solve a substantial practical problem that we have observed in our work related to the tool Artemis that performs automated testing of JavaScript web applications [11]: It can be difficult to set up
servers and populate databases to be able to test the client-side JavaScript code. Moreover, an automated tester, that focuses on testing the JavaScript code and has a black-box view on the server, is often not able to produce high coverage tests within a given time budget. With interface descriptions, we can automatically construct mock servers that can be integrated into such an automated tester in place of the real servers.

To illustrate this idea, consider again the example from Figure 10.1. If we wish to apply automated testing to the JavaScript code, two approaches could be considered at first: (1) We could ignore the server and simply assume that any response is possible to any Ajax request. Automated testing could then reveal that the JavaScript code will throw a runtime exception if the response data is not an array or if the array contains a null value (on line 24 and line 27, respectively), and malformed HTML would be generated if the object properties contain special HTML characters. However, this does not imply that there are errors in the JavaScript code—implicitly it may be the server’s responsibility to ensure that the Ajax response does not contain such values. (2) Alternatively, we could use a live server with realistic database content. This would eliminate the problem with false positives in the first approach. However, two drawbacks arise: first, it requires deep insight into the application to be able to provide realistic database content [19]; second, the testing capabilities become fixed to that particular database content, which may limit the coverage of the client code. Interface descriptions give us another alternative: (3) With a description of what requests the server accepts and the responses it may produce, an automated testing tool such as Artemis becomes able to focus on testing the JavaScript code on meaningful inputs.

To alleviate the burden of writing interface descriptions for pre-existing applications, we additionally propose an automated learning technique. Our hypothesis is that practically useful interface descriptions can be created using only sample request and response data. The sample data can be obtained by users exercising the functionality of the application without requiring detailed knowledge of the server code. This makes the learning technique independent of the specific programming languages and frameworks (PHP, JSF, ASP.NET, Ruby, etc.) that may be used on the server and thereby be more generally applicable.

The idea of using interface description languages (IDLs) to specify the interfaces of software components has proven successful in many other contexts. Prominent examples in related domains include Web IDL for the interface between browsers and JavaScript application code [30], WSDL for web service communication [21], and OMG IDL for interprocess communication with CORBA [93]. Nevertheless, IDLs are still not widely used in the context of client-server interactions in Ajax web applications, despite the existence of languages, such as WADL [49]. We suspect that one reason is that writing the interface descriptions is a laborious task. To this end, our work is the first to propose an automatic technique to learn interface descriptions for Ajax web
In summary, our contributions are as follows:

- We first introduce a simple Ajax server interface description language, named AIL, inspired by WADL (Section 10.2). This language can describe HTTP operations involving JSON and XML data as commonly seen in Ajax web applications.

- We demonstrate how the interface descriptions can be incorporated into automated testing of JavaScript web applications to be able to test client code without involving live servers. Specifically, we extend the automated testing tool Artemis with support for AIL (Section 10.3) by introducing a generic mock server component that is configured using AIL descriptions.

- We provide an algorithm for learning AIL descriptions of Ajax web applications through dynamic analysis of network traffic between clients and servers (Section 10.4).

- We experimentally evaluate our approach by investigating how AIL descriptions affect the code coverage obtained by Artemis with our extensions and by comparing the inferred AIL descriptions with manually crafted ones (Section 10.5). Our results show that (1) by using the descriptions, Artemis can obtain as good coverage with the mock server as with real servers and manually populated databases and (2) the learning algorithm is capable of producing useful AIL descriptions.

Testing Ajax applications is recognized as a difficult problem [72, 81] and interface descriptions have proven useful for testing classical web applications [2, 57, 50, 52, 53, 73], but no previous work has combined interface descriptions and testing of Ajax applications. Related work on interface description languages, learning algorithms, and automated testing is discussed in Section 10.6.

In this paper, we use the term Ajax [40] in a broad sense, covering different technologies for client-server communication in JavaScript-based web applications. In current web applications this typically involves the XMLHttpRequest API, but our general approach in principle also encompasses the more recent WebSocket API. The data being transmitted may involve different formats including XML and JSON that are supported by AIL, although our current learning algorithm and experiments focus on JSON.

---

3 http://www.w3.org/TR/XMLHttpRequest/
4 http://www.w3.org/TR/websockets/
10.2 An Interface Description Language for Ajax

Our first step is to design a formal language, AIL (Ajax server Interface description Language), for describing the interfaces of servers in Ajax-style web applications. The communication in such web applications consists of HTTP client-server interactions where the JavaScript code running on an HTML page in a browser sends requests and receives responses. An HTTP request contains a method (typically GET or POST), a URL path, and a number of name-value parameter pairs. For simplicity, we abstract away the other information in the HTTP requests, such as the protocol and headers. We design AIL as a simple language that concisely captures the essence of WADL [49] and integrates with JSON.

An AIL description consists of a base URL and a collection of operation descriptors, each of the form

\[
\text{request} : \text{response}
\]

where request is a pattern that describes a set of possible HTTP requests, and response describes the possible responses that may be generated by the server for those requests. Within an AIL description, the request patterns must be disjoint in the sense that every possible HTTP request can match at most one of the patterns, which ensures a deterministic behavior.

An AIL description establishes a contract between the clients and the server: The clients are responsible for ensuring that each request matches one of the request patterns, and it is the server’s responsibility that the response matches the corresponding response schema. Below we describe the syntax and matching semantics of request patterns and response schemas.

Figure 10.3 shows an AIL description (without JSON Schema files) for a simple JSON news server that makes five operations available for JavaScript applications. The first three operations provide access to news items, author information, and authentication. The last two operations can be used for submitting news items to the server and for registering new users. All operations use HTTP and JSON. The description refers to external JSON Schema files that specify the data formats involved in the operations. Such an AIL description evidently characterizes the structure of the operations that are supported by the server while abstracting away from the actual data being transmitted at runtime.
The initial version of AIL supports two kinds of data formats: JSON and XML. AIL simply relies on JSON Schema\(^5\) and RELAX NG\(^6\) for describing the structure of data.

A request pattern consists of an HTTP *method* (GET, POST, etc.), a *path*, and a comma-separated list of *parameters*:

```
method path(parameters)
```

A path consists of *path fragments* separated by ‘/’, each being a string or a parameter. Each parameter has the form `name:cardinality datatype`, where `cardinality` is either absent (meaning precisely one occurrence), ‘?’ (optional) or ‘+’ (zero or more).

A *datatype* is written as a constant string (e.g. "start"), the wildcard ‘*’, the keyword `void`, a reference to an external JSON Schema file (e.g. `@token.json`), a reference to a RELAX NG schema file (e.g. `@person.rng`) or a type defined in such a file (e.g. `@types.rng#person`), or a list of datatypes separated by ‘|’. The datatypes of parameters that occur in paths are restricted to simple string types, such as numerals or string enumerations, and the special datatype `void` is never used in request patterns.

A datatype matches strings in the obvious way: a constant string matches that string and no others, ‘*’ matches any value, `void` is used for describing the empty response, a reference to a schema type matches according to the semantics of JSON Schema and RELAX NG, respectively, and ‘|’ represents union.

An HTTP request matches a request pattern if each constituent matches. A response pattern is simply a datatype with matching defined accordingly. We omit the precise semantics of pattern matching due to the limited space, but the intuition should be clear from this brief description.

The example shown in Figure 10.4 is a part of an AIL description of the application *The Bug Genie*\(^7\). This application uses REST-style naming where some parameters appear in the path, not in the HTTP request body or the query string. The responses use JSON in both application; we omit the details of the associated JSON schemas.

The AIL language as presented above is capable of expressing the basic properties of server interfaces. One straightforward extension is to support other data formats, such as, HTML, plain text, or JSONP (JSON with padding), credentials for HTTP Basic authentication, and request content types (i.e. MIME types). In some situations it can also be useful both for documentation and testing purposes to describe error responses, that is, non-“200 OK” HTTP response codes, and HTTP content negotiation. For now, AIL cannot describe temporal properties, for example that operation A must be invoked before operation B, simply because such properties have not been significant in any of the web applications we have studied. Another possible

\(^5\)http://json-schema.org/
\(^6\)http://relaxng.org/
\(^7\)http://www.thebuggenie.com/
extension is support for WebSockets, which unlike HTTP involves connection-oriented communication and thereby does not fit directly into the simple request-response model.

10.3 Using Server Interface Descriptions in Automated Testing

Server interface descriptions are not only useful for documenting the server interface for the client programmer; they also make it possible to test the client code in isolation from the server code, which provides new opportunities for practical automated testing. In Section 10.3.1 we give a brief overview of the Artemis tool from earlier work by Artzi et al. [11], with a focus on the complications caused by Ajax communication. In Section 10.3.2 we show how a new mock server component can exploit AIL descriptions to improve the level of automation.

10.3.1 Automated Testing with Artemis

A JavaScript web application is driven by events, such as the initial page load event, mouse clicks, keyboard presses, and timeouts. Event handlers are executed single threaded and non-preemptively. A test input to an application is thus given by a sequence of parameterized events. Of particular relevance here are the events that are triggered by Ajax response where the event parameter contains the HTTP response data from the server.

Figure 10.5 shows a use of the XMLHttpRequest API, which provides low-level Ajax functionality (in contrast to the example in Figure 10.1 that uses the jQuery library). The call to x.send on line 56 initiates the request to the server, in this case an HTTP GET request to the relative URL news/read, which matches the AIL description in Figure 10.3. An event handler for receiving the response is set up on line 47. When the response content has finished loading, x.readyState will have the value 4, and the event handler function is called. If the response status code is 200 the response content is then available as a text string in x.responseText. For this example, the challenge for an automated tester is how to produce meaningful server response

---

8http://www.w3.org/TR/XMLHttpRequest/
data that will thoroughly exercise the response event handler including the `showItems` function being called on line 50.

The Artemis tool uses feedback-directed random testing of JavaScript web applications to produce high-coverage test inputs. That is, it executes the application on a test input and monitors the execution to collect interesting information that can be used to generate new test inputs to improve coverage. The heuristics used for collecting information, producing new test inputs, and prioritizing the work list of test inputs are described by Artzi et al. [11], and we here focus on the interactions with the server.

Although our goal is to test the JavaScript code, not the server, we face the problem that the JavaScript code in Ajax-style applications closely depends on the server. As discussed in Section 10.1 it is often difficult to populate the server database with useful data that is required to ensure high coverage of the JavaScript code. A simple example is line 27 in Figure 10.1, where both branches can only be covered if the server database contains a nonempty list of attendees, whereof at least one is marked as `checkedin` and another is not—no matter how other events, such as mouse clicks, are being triggered in the browser. On top of this, even a well populated database may not suffice. Reaching one part of the JavaScript code may require certain values in the database where another part may require different values, so multiple database snapshots may be necessary to enable high coverage of the JavaScript code, which makes the burden even higher.

Yet another problem for automated testing appears when important server responses can only be triggered by request values that are practically impossible to guess by the testing tool. Consider for example the operation `users/login` for the server described in Figure 10.3. A successful response requires the client to provide a valid user name and password, which is (hopefully) impossible to guess, so a considerable part of the client code will remain unexplored. A common workaround is to ask the user for help in such situations [10]. The consequence of these problems is that “automated” testing may require a considerable manual effort.

We observe that when testing client code, many execution paths require data from the server that is structurally well-formed but not necessarily semantically valid. As an example, for testing the `populate_table` response handler function in Figure 10.1, we do not need server response data that contains actual attendee names and email addresses, but we do need JSON data with a certain structure. This observation allows us to use AIL descriptions instead of actual servers and live data for testing client code.

### 10.3.2 Extending Artemis with an AIL Mock Server

To alleviate the problems described above, we have extended the Artemis tool with a mock server component that is configured by an AIL description. Whenever the JavaScript program under test initiates Ajax communication,
function getNewsItems() {
    var x = new XMLHttpRequest();
    x.open("GET", "news/read");
    x.onreadystatechange = function () {
        if (x.readyState == 4)
            if (x.status == 200) {
                showItems(x.responseText);
            } else {
                alert("An error occurred :-(");
            }
    }
    x.send(null);
}

Figure 10.5: A simple use of XMLHttpRequest to perform an Ajax call to get news items from the server from Figure 10.3.

the mock server intercepts the request such that the actual server is never contacted during the testing.

Given an HTTP request, the mock server performs the following steps: (1) It searches through the AIL description to find an operation descriptor with a request pattern that matches the HTTP request. If one is found, a random response that matches the corresponding response datatype is prepared; otherwise, the response to the client is a generic “404 Not Found”. (2) The response data is then sent to the test input generator component in Artemis, which will subsequently produce new test inputs that include an Ajax response event containing the response data.

As result, we obtain a nondeterministic model of how the server may behave according to the AIL description, and the JavaScript code can be explored without the need of a real server and database.

Now, several observations can be made. First, using the mock server solves the problem of populating databases since it automatically returns a wide range of possible responses, as specified by the AIL description. This means that the client code is effectively tested on a variety of structurally meaningful inputs. The response data generated by the mock server may of course not be semantically valid, but as argued above, structurally correct response data will suffice for testing many properties of client code. This approach also elegantly handles the issue with the users/login operation mentioned above: the mock server component will skip the actual password check and automatically produce a meaningful response representing successful login.

Second, our approach makes it easy to model the asynchronous nature of Ajax, which is a source of intricate race errors [96, 113]: Even though the AIL mock server component only produces a single response for each request in step 1, the Artemis test input generator component may create multiple new inputs in step 2 to test different event orders.

Third, the construction of responses in step 1 may not be entirely random. We can exploit the existing feedback mechanism in Artemis such that information that has been collected by Artemis in previous executions of the
10.4 Automatic Learning of AIL Descriptions

We have shown that AIL offers a simple, formal mechanism for documenting client-server communications in Ajax-style web applications and that AIL descriptions are useful in automated testing of the client code. However, despite the many advantages of having server interface descriptions, constructing such descriptions for pre-existing applications can be a nontrivial task. To support the construction of AIL descriptions, we show how to automatically learn descriptions from samples of client-server communication traffic through a black-box, dynamic approach. This approach has been chosen for generality and independence of particular server technologies used. We imagine that such a learning algorithm can be used when a development team realizes that their web applications have grown from being small and manageable to become larger and more difficult to maintain without proper separation of concerns and without the ability to apply automated testing techniques. Automated learning makes it easier to retrofit server interface descriptions to existing applications, thereby supporting automated testing for the further development of the applications.

We assume that the AIL descriptions being used in automated testing as described in Section 10.3 have been written manually or with support from the learning algorithm. The automatically generated descriptions may naturally require some manual adjustments since they are generated on the basis of sample data.

We first introduce our learning problem. The input $I$ denotes a finite set of concrete HTTP request and response pairs $\langle r, s \rangle$. The output $d$ denotes an AIL description that expresses a possibly infinite relation $\mathcal{J}_d$ of request and response pairs. Since we perform black-box learning, we assume that sufficient samples are available for learning.

Given a set of input samples, there are many valid AIL descriptions that "generalize" it. Thus, it is important to define which AIL descriptions are the most desired. At the high level, we want a learned AIL description to closely match the server programmer’s view of the interface—a set of independent operations each with its own meaning and purpose. The central challenge is to identify these operations from the given observations without any white-box knowledge of the server and client.

To guide our learning algorithm, we specify the following desirable properties that a learned description should have:
Completeness The input $I$ is covered by the learned AIL description $d$, i.e. $I \subseteq \lfloor d \rfloor$.

Disjointness The request patterns of $d$ must be disjoint.

Precision $d$ should be as close to $I$ as possible, i.e. $\lfloor d \rfloor \setminus I$ should be small. We say that $d_1$ is more precise than $d_2$ iff $\lfloor d_1 \rfloor \setminus I \subseteq \lfloor d_2 \rfloor \setminus I$.

Conciseness $d$ should be small. We say that $d_1$ is more concise than $d_2$ iff $|d_1| \leq |d_2|$, where $|d|$ denotes some appropriate notion of the size of an AIL description $d$.

With these properties in mind, we devise an algorithm to learn AIL descriptions from input samples. Our algorithm has two phases: data clustering and pattern generation. The data clustering phase is the key step, organizing the input samples into distinct clusters such that each cluster corresponds to a “likely” operation descriptor, and these together form an AIL description with the aforementioned properties. Once the appropriate clustering has been determined, the pattern generation phase transforms the clusters into actual AIL descriptions and JSON schemas. This last step is straightforward and will not be described in this paper due to space constraints.

For the clustering phase we make two observations. First, identifying responses that are structurally similar can be a good starting point. For example, two JSON values that have the same object structure but contain different strings or numbers can be considered “similar” and hence likely belong to the same operation. Second, we can infer important information for clustering from the path fragments and parameters that occur in the request data. As an example, consider requests to the first operation from Figure 10.4:

```
POST comment/delete/comment_id:*() : @delete.json
```

A request consists of path fragments and GET/POST parameters, which we will denote features. The features for this operation are three path fragments, i.e. the constant strings `comment` and `delete` and a comment ID value. These can be divided into key features, which are characterized by having relatively few possible values that together identify the operation for the request, and non-key features, with a higher number of possible values, which do not identify operations. For this particular operation, the key features are the first two, i.e. `comment` and `delete`, and we can expect that our sample data will contain a higher number of comment ID values than the number of distinct operations.

These observations motivate us to further divide the clustering phase into two steps: (1) construct an initial clustering by considering only the response data and grouping the responses into distinct clusters with respect to their response types (Section 10.4.1); and (2) restructure the clustering using request data by identifying the likely key features (Section 10.4.2). After the
10.4. AUTOMATIC LEARNING OF AIL DESCRIPTIONS

In the clustering phase, we construct AIL descriptions that satisfy the completeness property by ensuring that each sample is associated with a cluster and giving the cluster a request pattern and a response pattern that match all samples in the cluster.

10.4.1 Response Data Clustering

We first cluster the input set \( I \) using HTTP response data. Although AIL can describe both XML and JSON data, we describe our algorithm for JSON, which is the most widely used data interchange format for Ajax web applications. A JSON response is a JavaScript data structure containing primitive values (strings, numbers, booleans, and null), objects, and arrays.

For each request and response pair \( \langle r, s \rangle \in I \), the response \( s \) contains JSON data. We map \( s \) to its type abstraction:

- a primitive value is mapped to its respective primitive type (e.g. String, Number, Boolean, or Null);
- an object value \( \{p_1 : v_1, \ldots, p_k : v_k\} \) is mapped to a record type \( \{p_1 : t_1, \ldots, p_k : t_k\} \) by replacing each object property value with its type, where \( t_i \) denotes the type of the value \( v_i \); and
- an array \( [v_1, \ldots, v_k] \) is mapped to a union type \( \bigcup_{i=1}^{k} t_i \), where \( t_i \) denotes the type of \( v_i \).

We now cluster all sample pairs from \( I \) according to structural equivalence of the response type abstractions. For example, the five sample responses shown in Figure 10.6 will be clustered together into three clusters. The first two samples have the same type abstraction \( \{id : \text{Number}, \ name : \text{String}, \ stories : \text{Number}\} \) and are thus grouped together, while the next two contain an additional property, resulting in the type abstraction \( \{id : \text{Number}, \ name : \text{String}, \ email : \text{String}, \ stories : \text{Number}\} \) and their own cluster. Similarly, the type abstraction of the last response \( \{id : \text{Number}, \ title : \text{String}\} \) does not match the first or the second cluster, so it will be placed in a third cluster.

10.4.2 Request Data Clustering

Using the distinction between key and non-key features, we want our learning algorithm to construct request patterns in which key features are represented using constant strings, and non-key features are represented using wildcards. However, deciding on the division between key and non-key features may require restructure of the clustering to ensure that the disjointness property is satisfied. In the example shown in Figure 10.6, the first four responses are initially put into two clusters. If the name parameter is classified as a key feature, then we need to split the two clusters into four, one for each
sample. On the other hand, if it is classified as a non-key feature, then we need to merge the two clusters into one to ensure disjointness. To generate the desired request patterns using constant strings and wildcards, this example shows that we may need to merge clusters together, using a wildcard, or split them into separate clusters, using constant strings.

To describe in more detail how we merge and split clusters, we first introduce some additional terminology. As stated, each path fragment and parameter of a request is a feature. The set of features in a request forms its signature, denoted by $S$. As an example, a request with URL `foo/bar` and a parameter `baz=1` has the signature \{#0, #1, #baz\} where path fragments are identified by their positions in the URL and parameters are identified by their names. Since we wish to construct one request pattern for each cluster, we first split clusters that contain requests with different signatures. Request patterns that are constructed from clusters with different signatures are trivially disjoint. Next, we need to decide on a suitable partition of $S$ into key and non-key features, corresponding to constant strings and wildcards, respectively.

There are two obvious extremes when selecting the partition: (1) assign wildcards to all features, thereby merging all clusters with the same signature into a single one, which is likely to be highly imprecise, and (2) assign constant strings to all features, thereby splitting all clusters into singletons (i.e. simply the input set $I$), which is neither concise nor very useful. These two extremes relate to operation descriptions being concise and precise respectively, which are conflicting requirements that we must reconcile.

Our algorithm is given in Figure 10.7. Let $D$ be initial response data clustering $D$ from Section 10.4.1. For each signature $S$ in $D$, the algorithm iterates over all clusters $D'$ that match $S$. It then iterates over all possible partitions $\rho$ that divide $S$ into key and non-key features, selecting the partition
with the minimal cost with respect to a cost function $C$. This partition is used to restructure the clusters $D$. The end result, after iterating over all signatures, is $D$ restructured in accordance with its request data. What remains is to define the cost function $C(\rho, D')$, where $\rho$ is a partition of $S$ into key and non-key features and $D'$ is a set of clusters with the same signature.

Recall our observation that clustering based on response types typically yields a good baseline clustering. Thus, we favor partitions that result in the smallest number of splits and merges compared to the baseline clustering. This strategy is further supported by the other observation that key features have few unique values, so our goal is to find a partition that leads to the smallest number of splits and merges.

We define the cost $C(\rho, D')$ as the total number of splits and merges necessary to get from $D'$ to the restructured clustering. Intuitively, a least cost partition helps avoid merging too much, for precision, and avoid splitting too much, for conciseness. In case of a tie, we choose a partition that minimizes the number of wildcards.

To illustrate the cost calculation, consider the two initial clusters that were created from the first four sample responses in Figure 10.6. Those two clusters are a result of different response structures, however, we cannot ensure disjointness of the request patterns without a reorganization. Both clusters have the signatures $S = \{#0, #name\}$ corresponding to the author URL path fragment and the name parameter. The cost function is applied to all possible partitions $\rho$ of $S$, in this example the following four partitions:

1. Neither #0 nor #name is considered a key feature, causing the two clusters to be merged at a cost of 1 into a cluster with request pattern *?name=*.
2. Only #0 is a key feature, which also causes a single merge operation, hence the cost is 1, but the resulting cluster now has request pattern author?name=*.
3. Only \#name is a key feature, which means that the two clusters are split into four singleton clusters at a total cost of 2, resulting in the four request patterns *?name=alice, *?name=bob, *?name=charlie, and *?name=eve.

4. Both \#0 and \#name are key features, which also results in four singleton clusters at a total cost of 2, but the request patterns are now author?name=alice, author?name=bob, author?name=charlie, and author?name=eve.

We choose the second partition since it has minimal cost and minimal number of wildcards.

Finally, we have constructed clusters with the desired properties. Each cluster can be turned into an AIL operation descriptor, as hinted earlier. Its request pattern is generated from the employed partition \( \rho \), and JSON schemas for the response patterns are generated from the type abstractions of the response samples in the cluster. The close connection between JSON schemas and the type abstraction we use for response data leads to a straightforward construction.

### 10.5 Evaluation

We have argued that server interface descriptions can provide separation of concerns, which enables testing of JavaScript code in isolation from the server code. When conducting automated testing of the JavaScript code, the use of AIL and a mock server removes the burden of setting up actual servers with appropriate database contents. To find out how this may influence other aspects of automated testing, we first consider the following research questions:

- Q1. How is the running time of automated testing affected when replacing the real server by the mock server for a fixed number of test inputs?

- Q2. Does the use of AIL in place of live servers affect the code coverage obtained by automated testing?

To evaluate how our learning algorithm from Section 10.4 can be useful when creating AIL descriptions for existing applications, we consider two additional research questions:

- Q3. To what extent is the learning algorithm capable of producing AIL descriptions that correctly describe the servers in actual JavaScript web applications?

- Q4. How much effort is required for producing request and response data for the learning algorithm, and how fast is the learning algorithm?

To answer these questions we have implemented three tools[^9]:

1. a web proxy for recording the HTTP communication between clients and servers,

[^9]: Our tools are available at [http://www.brics.dk/artemis/](http://www.brics.dk/artemis/)
10.5. EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Client Framework</th>
<th>Server Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleajax</td>
<td>79</td>
<td>jQuery</td>
<td>Python (Django)</td>
</tr>
<tr>
<td>resume</td>
<td>244</td>
<td>Flapjax</td>
<td>Python</td>
</tr>
<tr>
<td>globetrotter</td>
<td>347</td>
<td>jQuery</td>
<td>Java (JWIG)</td>
</tr>
<tr>
<td>impresspages</td>
<td>558</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
<tr>
<td>buggenie</td>
<td>3,716</td>
<td>Prototype</td>
<td>PHP</td>
</tr>
<tr>
<td>elfinder</td>
<td>6,724</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
<tr>
<td>tomatocart</td>
<td>8,817</td>
<td>Prototype</td>
<td>PHP</td>
</tr>
<tr>
<td>eyeos</td>
<td>17,629</td>
<td>jQuery</td>
<td>PHP</td>
</tr>
</tbody>
</table>

Figure 10.8: Benchmark applications.

(2) the learning algorithm from Section 10.4, which reads the data recorded by the web proxy and outputs AIL descriptions and JSON schemas, and (3) the AIL mock server for Artemis.

We have collected 8 benchmark applications that use JavaScript for their client-side logic and Ajax for communicating between the client and the server, and where the source code for the entire application has been available, including the server code: simpleajax is a small home-built test application for event registrations; resume is an application management system; globetrotter is a travel application system; impresspages is a CMS system; elfinder is an online file explorer; buggenie is a project management tool that we also used as example in Section 10.2; tomatocart is an e-commerce platform; and eyeos is an online desktop environment.

Figure 10.8 contains a list of the applications together with the number of lines of JavaScript code (excluding frameworks), the framework they use for JavaScript if any, and the language or framework used on the server side.

Our experiments are performed on a 3.1GHz i5 machine with 4GB of memory.

10.5.1 Using AIL in Automated Testing

To be able to answer Q1 and Q2 we run Artemis on our benchmark applications using various configurations: EmptyDB with real servers but with empty databases, FullDB with real servers where the databases are populated with realistic data, Random with a fully generic mock server that accepts all re-
quests and produces random JSON responses, and \textit{AIL} with the mock server using the AIL description.

The database contents used in the FullDB configuration are selected as snapshots obtained when we exercised the applications to collect sample request and response pairs. For the AIL configuration, we use manually constructed AIL descriptions, or equivalently, descriptions that were produced by the automated learning and subsequently manually adjusted to properly model the servers. Three of the larger benchmark applications are unfortunately beyond the capabilities of the latest version of Artemis for reasons that are unrelated to AIL and Ajax communication, so our experiments are conducted on the remaining five applications.

The execution time of Artemis depends on a number of factors, one of course being the time budget, which is expressed as a maximum number of test input executions. Other factors are the specific data that the JavaScript application receives from the server in Ajax interactions and the response time of the server. Replacing the live server with a mock server affects the two latter factors. Responses that are randomly generated from the AIL response patterns may trigger long running loops in the JavaScript code, however, the work performed by the mock server is presumably simpler than that of the real server in most cases.

The first columns in Figure 10.9 show the total running time of Artemis with the two configurations AIL and FullDB using a budget of 100 test input executions for each application. This gives an answer to Q1: for these applications, the running time is not affected notably by the AIL mock server.

The remaining columns show the code coverage (measured as number of lines of JavaScript code) for 300 test input executions of each application using all four configurations. The extra column, Init, shows the coverage obtained by loading the application without triggering any events, which can be viewed as a baseline for the coverage comparisons. The \textit{globetrotter} and \textit{buggenie} applications have not been tested with empty databases since this did not make sense for those cases. (Please note that the LOC column in Figure 10.8 should not be compared with the coverage numbers in Figure 10.9 since the latter only include lines with executable code.)

We observe that the use of the AIL mock server yields similar coverage results as when using the real servers populated with realistic data, which partially answers Q2.

For \textit{globetrotter}, \textit{elfinder}, and \textit{resume}, coverage is slightly improved when using AIL. In each case, the increased coverage is caused by conditions in the JavaScript code that are only triggered with specific Ajax response data, for example an empty array or a certain boolean value somewhere in a JSON structure. These are examples of application behavior that depend on the precise contents of the server database, as discussed in Section 10.3. In \textit{globetrotter}, for example, the program state describes a travel application that can be at different workflow stages. The mock server quickly generates JSON
responses that cover all the workflow stages, while the FullDB configuration only manages to cover a single one.

The lower code coverage for \textit{buggenie} is caused by an animation not being triggered in the AIL configuration due to the heuristics used internally by Artemis. For \textit{elfinder}, a few lines are reached with FullDB but not with the AIL configuration. The data in this application contains a tree-like structure of files and directories that are linked through hash and parent hash values that refer to each other. This invariant cannot be expressed with the current design of AIL, so the mock server is not able to produce the right response.

Several additional observations can be made from the coverage numbers. The EmptyDB, FullDB and AIL measurements show higher coverage than Init, demonstrating that we actually test additional functionality besides simply loading the page. Interestingly, the Random measurements for \textit{resume}, \textit{globetrotter}, and \textit{elfinder} show considerably less coverage, which demonstrates that meaningful response data is important. In all cases, this is caused by the initialization of the web applications depending on correctly structured Ajax responses. As expected, populating the database (i.e. FullDB) results in higher or equal coverage than using the empty database (i.e. EmptyDB).

Although we did not expect to find bugs in the benchmark applications, the use of the AIL mock server revealed one in \textit{resume} that was not found with any of the other configurations. The bug is triggered by a sequence of events that involve sending an empty array to the server and back to the client ending up in \texttt{obj.values} at the following snippet of code where it leads to a runtime error:

```javascript
var ln =
A({href: 'javascript:undefined'},
  ''+obj.values[0]['number']+' - ' +obj.values[obj.values.length-1]['number']);
```

This example illustrates how unclear assumptions between client and server developers can end up creating errors in the applications.

In other situations, similar unclear assumptions do not cause errors but lead to fragile code that may break in future revisions made by programmers
who are not aware of subtle invariants that must be satisfied. The use of AIL in Artemis also revealed such a situation. The elfinder application contains the following snippet of code where dir and files originate from an Ajax response:

```java
while (dir && dir.phash) {
    dir = files[dir.phash]
}
```

The purpose of this code is to traverse a directory structure where files are represented in an array indexed by file hash values. Running Artemis with the AIL configuration discovered that if this data structure contains loops then the while loop never terminates. The required invariant—that the data structure sent in the Ajax response never contains such loops—is not documented in the application source code. AIL is not expressive enough to capture such invariants, but one could argue anyway that the application would be more robust if its correctness did not depend on such intricate invariants involving the server state.

Concluding these experiments, our answer to Q2 is that the use of AIL leads to good coverage compared to using a server with an empty database, a server with a populated database, or a mock server that generates random responses. The experiments also pointed us to examples where correctness of the applications depends on subtle, undocumented invariants.

### 10.5.2 Automated Learning of AIL Descriptions

To obtain the training data for the learning algorithm, we install and exercise each application by manually clicking on visual elements and entering data into forms for a few minutes, while the web proxy monitors all Ajax communication. This is done without detailed knowledge of each application and entirely without looking at the server code. This gives us between 70 and 611 sample request and response pairs, depending on the amount of information exchanged.

We now run the learning algorithm on the data obtained for each application, which in each case takes less than a second. The request data clustering process described in Section 10.4.2 performs altogether 18 splits and 43 merges when searching for the partitions with the minimal cost. This results in a total of 130 AIL operation descriptors and 9,550 lines of JSON schema—all generated automatically.

Figure 10.10 shows the amount of sample data, the time used for collecting the sample data, and the time used by the AIL learning algorithm for each application. From these numbers we can give a rough answer to Q4: the effort required for using automated AIL learning is clearly manageable, compared to the time otherwise spent developing the web applications.

Producing AIL descriptions is of course not enough; they also need to capture the actual patterns of the Ajax communication. Recall from Section 10.4
10.5. EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Samples</th>
<th>Sampling</th>
<th>Learning</th>
<th>Match</th>
<th>1→N</th>
<th>N→1</th>
</tr>
</thead>
<tbody>
<tr>
<td>simpleajax</td>
<td>70</td>
<td>3m</td>
<td>74ms</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>resume</td>
<td>128</td>
<td>9m</td>
<td>111ms</td>
<td>12</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>globetrotter</td>
<td>97</td>
<td>8m</td>
<td>84ms</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>impresspages</td>
<td>179</td>
<td>6m</td>
<td>224ms</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>buggenie</td>
<td>210</td>
<td>6m</td>
<td>118ms</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>elfinder</td>
<td>181</td>
<td>6m</td>
<td>124ms</td>
<td>11</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>tomatocart</td>
<td>370</td>
<td>8m</td>
<td>153ms</td>
<td>22</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>eyeos</td>
<td>611</td>
<td>6m</td>
<td>213ms</td>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td>17</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 10.10: Number of sample request and response pairs used for AIL learning, time used for collecting sample data and learning AIL descriptions, and results from comparing the learned AIL descriptions with the manually written ones.

that the constructed AIL description is complete by construction, relative to the training data. However, the training data may not cover the entire application, which might result in incomplete AIL descriptions where some operations supported by the server are missing in its AIL description. Another potential source of mismatches between automatically constructed AIL descriptions and manually written ones is that the learning algorithm may not be sufficiently precise or concise (using the terminology from Section 10.4). Furthermore, as there is no canonical “best” AIL description for a given Ajax server, we must settle for a subjective baseline for comparison, which we decide to construct as follows: For each benchmark application, we manually write an AIL description based on an inspection of the source code for the server part of the application. This process can take hours, but this work is of course only required to be able to measure the quality of the learning algorithm in our experiments.

Next, we need a measure of the difference between the automatically constructed AIL descriptions and the manually constructed ones. The first aspect of this measure is how the individual operation descriptions match between the two. Figure [10.10] also shows the results of this comparison. The Match column counts the number of learned operations that map directly to the actual server operations, while 1→N and N→1 count the number of server operations that map to multiple learned operations and vice versa, which indicate mismatches between the two descriptions. A second aspect is to what extent the individual datatypes of parameters and response patterns differ between the two descriptions.

We get a total of 85 matches, 17 occurrences of 1→N, and a single N→1. The high number of matches is already encouraging, but a closer inspection of the other cases reveal that they are relatively benign. In all the 1→N cases,
a simple manual merging of the relevant operation descriptors suffices to resolve the discrepancy. This is acceptable since the learned AIL description is only intended as a starting point for the programmer, as an alternative to writing the AIL description from scratch. An example is delete operation in resume, which can be called both with and without an id parameter resulting in different responses, causing the learning algorithm to produce two separate AIL operation descriptors. Another example of a 1→N case appears in buggenie. In this case, a specific server operation runIssueRevertField performs multiple tasks and dispatches internally, based on a parameter field, in a way where one may argue that the AIL description produced by the learning algorithm, where these sub-operations are divided into separate descriptors, is in fact just as good as the manually constructed one.

The single N→1 case appears in tomatocart and is caused by our merging heuristic being slightly too aggressive. Two operations for deleting images and setting default images, respectively, both take an id parameter and return a trivial response, and the operations are only distinguished by the value of an action parameter. The similarity of the data causes the two operations to be merged by our current heuristic.

Regarding the quality of the inferred datatypes in request and response patterns, we notice that many of our benchmarks use JSON in responses but not in requests. For request patterns, the main question then is whether wildcards are introduced appropriately. The learning algorithm needs at least two distinct values of a path fragment or parameter to conclude that it is not constant. For example, resume represents session IDs in parameters, so the training data must involve multiple sessions. Incompleteness of our sample data in some cases results in missing wildcards, however, this is easy to adjust manually after the learning phase.

Most JSON response data in the benchmark applications is built using arrays and simple objects with fixed collections of properties. For these common cases the learning algorithm is able to generate JSON schemas correctly. Differences between the JSON schemas constructed by the learning algorithm and the manually constructed ones are mostly due to incomplete sample data. However, we observed two interesting cases—in impresspages and globetrotter, respectively—that could be improved. Some responses in impresspages have a recursive structure of objects and arrays. More specifically, the data represents a list of page and directory objects where each directory object itself contains a list of page and directory objects. Our current learning algorithm is not able to produce the desired concise JSON schema. In globetrotter, a specific JSON structure contains information about a list of countries. Each country is represented by an object where the country name appears as a property name, not as a property value, which causes the learning algorithm to view each country object as being distinct.

Based on these experiments, our answer to Q3 is that the learning algorithm is able to produce AIL descriptions that are reasonably close to manually
constructed ones. This suggests that automated learning can be a good start-
ing point for creating AIL descriptions for pre-existing applications, and that sufficient sample data can be obtained by someone who is familiar with the functionality of the applications but does not have knowledge of the server code.

10.5.3 Threats to Validity

A possible threat to validity of our experimental results is that the manually constructed AIL descriptions that we use as baseline for the comparisons have been made by ourselves without expert knowledge of most of the benchmark applications. More solid results could perhaps be obtained by performing user studies with the developers of the applications. Also, our benchmark applications do not reflect all possible uses of Ajax and JSON, and they may not be representative of typical usage patterns although we have striven toward including a wide variety of applications.

10.6 Related Work

Our work touches on several areas of work on interface descriptions, automated testing of web applications, and learning algorithms.

10.6.1 Interface Descriptions for Separation of Concerns

The idea of design-by-contract is a fundamental principle in modern software engineering for separating the concerns of individual software components. Even in web programming, which often involves dynamic scripting languages both on the client and the server, interface description languages play an important role: Similar to AIL, WSDL [21] allows description of operations and their argument and return types, however, WSDL is tailored to XML-based web services and has no support for JSON, and we are not aware of uses of WSDL for describing server interfaces in Ajax-style JavaScript web applications. As mentioned in Section 10.2, AIL is by design conceptually closer to the language WADL [49], although AIL has a compact non-XML syntax and supports JSON. The Web IDL language is used for describing the API that web browsers provide to JavaScript programs [30] for accessing the HTML DOM and other parts of the browser state, however, unlike AIL, each Web IDL description is common to all JavaScript web applications and cannot describe the interfaces of individual Ajax servers. Web IDL has its roots in the OMG IDL interface definition language for CORBA [93].

Interface descriptions have also been proposed for HTML form-based web applications without JavaScript. The WebAppSleuth methodology by Fisher et al. [37] works by submitting requests to a server and analyzing the responses to infer parts of its interface. The resulting interface descriptions are related
to AIL descriptions but tailored to HTML forms, not JSON or XML. Each form is described by its set of mandatory and optional fields together with simple value constraints and dependencies between the fields.

The extensive survey by Alalfi et al. [2] covers many modeling methods used in web site verification and testing, but without JavaScript and Ajax. To our knowledge, the only existing work involving interface descriptions for Ajax communication is that by Hallé et al. [52]. They propose a contract language based on interface grammars, linear temporal logic, and XPath expressions for specifying the order of HTTP interactions that exchange XML data in long-running sessions. We believe the data formats of requests and responses are more important in typical Ajax applications than restrictions on the order of operations, so we have chosen to ignore the temporal aspect in our first version of AIL. Their paper discusses how the contracts can be used for runtime monitoring. They also ask the important question “who should write the contracts?” To this end, we take the approach of using machine learning on sample execution traces, as explained in Section 10.4. A range of well-documented web services that fit into the design of AIL can be found at Google’s APIs Explorer website.\footnote{http://code.google.com/apis/explorer/}

10.6.2 Automated Testing of Web Applications

Besides the Artemis tool [11] that we discussed in Section 10.3.1, we are aware of a few other tools for automatically testing JavaScript web applications. The Kudzu tool by Saxena et al. [100] performs automated testing by a combination of symbolic execution with a string constraint solver for value space exploration and random exploration of the event space, whereas Artemis uses a more light-weight feedback-directed approach. The state-based testing technique by Marchetto et al. [72, 73] builds finite-state machines that model Ajax web pages from concrete execution traces. As in our approach, a subsequent manual validation or refinement step is required to ensure that the extracted model is correct before the model can be used for automated testing. The key difference to our approach is that the models in state-based testing describe the DOM states of the JavaScript execution, not the interactions with the server. A closely related tool is Crawljax by Mesbah et al. [81, 82] that also aims to derive models of the user interface states of Ajax applications and use these models as a foundation for testing. AJAX Crawl by Duda et al. [29] similarly performs dynamic exploration of Ajax applications, but for the purpose of crawling the applications by search engines, not aiming at testing.
A common limitation of Kudzu, Crawljax, and AJAX Crawl is that the exploration of the JavaScript applications is done with little focus on the client-server communication, simply using live servers, which leads to the problems discussed in Section 10.3.1 about how to properly populate the server databases. On top of this, most tools, with Artemis as an exception, do not restore the server database state after each test input execution, which affects testing reproducibility.

The JSConTest tool by Heidegger and Thiemann performs random testing for JavaScript programs that are annotated with type contracts [53]. These function annotations play a similar role as AIL descriptions, but at the level of function calls rather than Ajax client-server interactions. Due to the JavaScript-oriented design of JSON Schema that we use in AIL, it is natural that the basic contract language in JSConTest has similar expressiveness. However, JSConTest also supports function types, which are not relevant for client-server exchange of JSON or XML data. Another difference is that JSConTest permits user-defined contracts written in JavaScript, which might be useful to consider for a future version of AIL to address the limitations identified in Section 10.5.

Several tools have been developed for automatically testing server-based web applications. The Apollo tool by Artzi et al. [10] and the tool by Wasser- man et al. [109] perform directed automated random testing for PHP code, but JavaScript is not considered. With our proposal of using a server interface description language for separating the concerns of server code and client code, we have so far focused on testing the client code, but an interesting direction of future work is to develop testing or analysis techniques that can also check whether the servers fulfill their part of the contracts.

Elbaum et al. [31] have proposed the use of user session data for black-box testing of web applications. They record concrete user sessions and replay the sessions in various ways to test the server code, not aiming for testing client code and not involving explicit server interface descriptions.

The WAM tool by Halfond and Orso [50, 51] automatically discovers interfaces of web applications using static analysis or symbolic execution of the server code. The interfaces are subsequently used in automated testing, similar to our approach, although WAM considers classical web applications without Ajax and JSON. The notion of interfaces used by WAM is similar to that in WebAppSleuth. WAM is restricted to Java Servlets, unlike our approach, which is in principle independent of the languages and frameworks used on the server.

10.6.3 Learning Algorithms

The learning algorithm presented in Section 10.4 has been developed specifically for AIL, but related algorithms exist for other domains.
WebAppSleuth \cite{37}, which we also mentioned in Section \ref{10.6.1}, uses learning techniques to identify interfaces of server-based web applications that receive HTML forms. That approach does not involve learning the structure of server response data, and a single server operation is considered at a time, while our learning algorithm needs to work for multiple operations.

The latest version of WAM \cite{51} likewise uses a learning algorithm to produce interface descriptions. The WAM algorithm operates on path constraints constructed through symbolic execution of the server code, which differs from our learning algorithm that is based on sample request and response data and has a black-box view on the server. Furthermore, WAM does not consider response types, unlike our learning algorithm.

The clustering problem that we face in Section \ref{10.4} is related to the work by Broder et al. on clustering web pages that are syntactically similar \cite{17}. Their approach is to define a distance measure between two web pages, using a distance threshold to cluster similar pages. This approach could be transferred to JSON responses and our learning algorithm, but we found the results from initially clustering only entirely equal structures to be sufficient for our purposes.

We are not aware of existing work on JSON Schema inference. The closest related work has been centered around DTD and XML Schema inference. This problem is described by Chidlovskii as being reducible to grammar inference \cite{20}. Others improve on this line of work \cite{14}, however, the differences between XML and JSON make their algorithms unsuitable for JSON Schema.

\section{10.7 Conclusion}

Server interface descriptions for Ajax-style web applications enable separation of concerns of the server code and the client code. The AIL language has been designed to capture the essence of the existing proposals WADL and allow concise description of the basic properties of server operations, in particular involving JSON data. Our experimental validation suggests that the expressiveness of AIL suffices for typical Ajax communication patterns, but also that it might be useful in future work to add support for user-defined contracts to specify more fine-grained invariants.

One key contribution is that we demonstrate that server interface descriptions are useful in automated testing. No previous work has combined server interface descriptions with testing of Ajax applications. Our experimental results show that this approach can improve the level of automation by eliminating the need for carefully populated databases on the servers, while maintaining the quality of the testing of the client code. Another key contribution of our work is the automated learning algorithm that can produce server interface descriptions from sample request and response data. The experiments show that AIL learning can be performed with a modest effort, and that the
resulting descriptions are a good starting point when programmers wish to construct AIL descriptions for pre-existing web applications.

In addition to the suggestions about possible extensions of AIL, several directions of future work appear. As an alternative or supplement to our AIL learning approach that has a black-box view on the server, it would be interesting to infer or validate AIL descriptions by static or dynamic analysis of the server code for the most widely used server web frameworks. Additionally, AIL may also be useful for static analysis of JavaScript applications to enable more precise reasoning of Ajax interactions than currently possible. Specifically, we wish to incorporate AIL into the JavaScript analysis tool TAJS [61].
Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sag- 
SIGPLAN-SIGACT Symposium on Principles of Programming Lan-
guages, 2014. 31, 39, 96, 97

methods for web application verification and testing: state of the art. 
Software Testing, Verification and Reliability, 19(4), 2009. 129, 148

Reverse engineering finite state machines from rich internet applications. 
In Proc. 15th Working Conference on Reverse Engineering, 2008. 20

Experimenting a reverse engineering technique for modelling the be-
haviour of rich internet applications. In IEEE International Conference 
on Software Maintenance, 2009. 20

A GUI crawling-based technique for Android mobile application testing. 
In Proc. 3rd International Workshop on Testing Techniques and 
Experimentation Benchmarks for Event-Driven Software, 2011. 19, 122

testing of Android applications. In Proc. 27th International Conference 
on Automated Software Engineering, 2012. 19, 102, 122

Automated concolic testing of smartphone apps. In Proc. 20th ACM 
SIGSOFT Symposium on the Foundations of Software Engineering, 
2012. 21, 22, 53, 55, 102, 120, 121, 122

application record and replay. In Proc. 41st Annual IEEE/IFIP Interna-
tional Conference on Dependable Systems & Networks, 2011. 27, 38, 
40, 96

153


[23] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications. In *Proc. 5th International Conference on Software Testing, Verification and Validation*, 2012. [18]


[108] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying Android applications using Java PathFinder. ACM SIGSOFT Software Engineering Notes, 37(6), 2012. [121]


[110] Qing Xie and Atif M. Memon. Using a pilot study to derive a GUI model for automated testing. ACM Transactions on Software Engineering and Methodology, 2008. [122]

