Automated Testing Techniques for
Event-Driven and Dynamically Typed
Software Applications

Christoffer Quist Adamsen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark
Automated Testing Techniques for Event-Driven and Dynamically Typed Software 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
Christoffer Quist Adamsen
March 5, 2018
Abstract

Software testing is the process of executing a software application on a set of inputs, and determining if the application behaves as intended on these inputs. This thesis focuses on testing techniques for event-driven and dynamically typed applications. This is an important class of software, which includes mobile and web applications, that can be challenging to test thoroughly. When developers test their applications, they face several challenges. In particular, they need to: (i) generate a set of inputs to their application, along with oracles that specify intended behavior, (ii) react to test failures by repairing the application, and (iii) determine when the testing is adequate.

The goal of this thesis is to design new techniques that can help developers in addressing these challenges. The thesis identifies opportunities for improving over state-of-the-art techniques, and proposes new techniques to address each of the challenges. We present a new methodology that extends the error detection capabilities of existing, manually written Android test suites. In the context of JavaScript web applications, we present practical race detectors for detecting AJAX and initialization races, and a technique that can prevent event race errors by restricting the nondeterminism. Finally, we present a notion of test completeness for dynamic languages, along with a hybrid static/dynamic analysis framework that approximates test completeness, and demonstrate the usefulness of test completeness facts. To demonstrate the usefulness of the proposed techniques, they have been implemented in publicly available prototypes, which have been evaluated on real-world software applications.
Resumé

Begrebet software testing dækker over at køre en software applikation på en mængde af inputs, og beslutte om applikationen opfører sig som forventet på disse inputs. Denne afhandling fokuserer på teknikker der har til formål at teste event-drevne og dynamisk typede applikationer. Dette er en vigtig klasse af software, som inkluderer mobil- og webapplikationer, som kan være udfordrende at teste grundigt. Når udviklere skal teste deres applikationer, står de over for en række af udfordringer. I særlighed skal de: (i) generere en mængde af inputs til deres applikation samt udarbejde orakler, der beskriver applikationens forventede opførsel, (ii) reparere deres applikation når en test fejler, og (iii) beslutte hvornår deres betænkelser på at teste deres applikation er tilstrækkelige.

Målet med denne afhandling er at designe nye teknikker, der kan hjælpe udviklere med at adressere ovenstående udfordringer. Afhandlingen identificerer muligheder for at forbedre eksisterende løsninger, og foreslår nye teknikker, der adresserer hver af de ovenstående udfordringer. Vi præsenterer en ny metodologi, som kan anvendes til at forbedre eksisterende Android test suiter, som er blevet skrevet manuelt, i forhold til deres evne til at finde fejl. I relation til JavaScript webapplikationer præsenterer vi praktiske værktøjer til at finde AJAX- og initialiseringsræs, samt en teknik der kan forhindre fejl, der opstår på grund af eventræs, ved at restringere nondeterminismen. Desuden præsenterer vi et testfuldstændighedsbegreb for dynamiske sprog samt en hybrid statistik/dynamisk analyse, der approksimerer testfuldstændighed, ligesom vi demonstrerer nyttigheden af testfuldstændighedsfakta. For at demonstrere brugbarheden af de foreslåede teknikker er de blevet implementeret i offentligt tilgængelige prototyper, som er blevet evalueret på softwareapplikationer fra den virkelige verden.
Acknowledgments

I would like to thank my advisor, Anders Møller, for giving me the confidence to pursue a PhD degree in Computer Science, for all of the great and rewarding experiences that I have had during my PhD studies, and for the many opportunities he has given to me. I could not have hoped for a better advisor than Anders. I would also like to express my thanks to all the people in the Programming Languages research group at Aarhus University for the many interesting and fruitful discussions we have had, and for contributing to a work environment where I have always been looking forward to going to work. I am especially thankful for the close collaboration I have had with Gianluca Mezzetti since I started on my PhD studies, and for all the experiences that we have shared. I would also like to thank Esben Sparre Andreasen, especially for his guidance during the early stages of my PhD. I also thank Gianluca and Esben for their feedback on this thesis. I am also thankful to Frank Tip for a pleasant and productive stay at Samsung Research America in 2016, for his encouragements to enjoy my stay and explore California, and for our collaboration and his advice since then. I also thank Gogul Balakrishnan and Domagoj Babic for their support and guidance during my internship at Google in 2017. Finally, I am very grateful to Julie for all of her love and support.

Christoffer Quist Adamsen,
Aarhus, March 5, 2018.
Contents

Abstract i
Resumé iii
Acknowledgments v
Contents vii

1 Overview 1

1 Introduction 3
  1.1 Contributions .......................... 5
  1.2 Published Papers and Manuscripts ........ 6
  1.3 Outline .................................. 7

2 Event-Driven and Dynamically Typed Software Applications 9
  2.1 The Event-Driven Execution Model ........ 9
  2.2 Dynamically Typed Programming Languages 16

3 Test Generation for Event-Driven Software Applications 19
  3.1 Automated User Interface Testing .......... 20
  3.2 Test Amplification ........................ 27
  3.3 Test Amplification using Adverse Conditions 29

4 Race Detection 33
  4.1 Race Detection for JavaScript Web Applications 33
  4.2 Race Detection for Mobile Applications .... 38
  4.3 Practical Race Detection for JavaScript Web Applications ... 39

5 Program Repair 45
  5.1 Preventing Data Races in Concurrent, Shared-Memory Programs 46
  5.2 Preventing Event Races in JavaScript Web Applications .... 47
  5.3 Controlling the Nondeterminism in JavaScript Web Applications 48
# CONTENTS

6 Test Adequacy 51
  6.1 Test Adequacy Criteria .................................. 52
  6.2 Test Completeness ........................................ 54
  6.3 Analyzing Test Completeness for Dynamic Languages .... 57

7 Conclusions 61

II Publications 67

8 Systematic Execution of Android Test Suites 69
  8.1 Introduction .............................................. 70
  8.2 Motivating Example .................................... 72
  8.3 Methodology ............................................ 76
  8.4 Implementation ....................................... 80
  8.5 Experimental Evaluation ............................... 83
  8.6 Related Work .......................................... 91
  8.7 Conclusion .............................................. 92

9 Analyzing Test Completeness for Dynamic Languages 93
  9.1 Introduction .............................................. 93
  9.2 The Dart Language ..................................... 97
  9.3 Overview ............................................... 97
  9.4 Dependence Analysis .................................. 102
  9.5 Type analysis .......................................... 106
  9.6 Test completeness ...................................... 107
  9.7 Experimental Evaluation ............................... 109
  9.8 Related Work .......................................... 115
  9.9 Conclusion .............................................. 116

10 Repairing Event Race Errors by Controlling Nondeterminism 117
  10.1 Introduction .............................................. 118
  10.2 Motivating Example .................................... 120
  10.3 Background on Event Races ............................. 123
  10.4 A Framework for Specifying Repair Policies .......... 124
  10.5 Repair Policies ........................................ 126
  10.6 Implementation ....................................... 132
  10.7 Evaluation ............................................. 133
  10.8 Related work .......................................... 138
  10.9 Conclusion .............................................. 140
  10.A Unreparable Event Races ................................. 141

11 Practical Initialization Race Detection 143
  for JavaScript Web Applications
Part I

Overview
Chapter 1

Introduction

An overwhelming amount of software is being developed and used every single day. One of the key properties of a piece of software is that of robustness, i.e., the ability of an application to avoid abrupt failures. In order for an application to be considered robust, it must correctly handle erroneous inputs, unexpected use cases and events, as well as environment failures and nondeterminism. Accounting for all possible cases is a challenging problem for developers.

Developers typically address this problem by testing their application. Software testing is the process of executing an application on a set of inputs, and determining, based on the outputs of the tests, whether the resulting executions match the expectations of the developers. It is widely accepted that testing is essential to improve software robustness.

Developers often face three main challenges while trying to increase the robustness of their application using testing. First, developers need to design a representative set of inputs that covers some or all relevant behaviors of the application that they are testing, along with oracles that specify the intended application behavior under these inputs. When executing the application on these inputs, the encountered behavior may not adhere to the oracles, which gives rise to test failures. Developers need to react to such test failures, by updating their application in a way that fixes the underlying root causes of the failures. Finally, they also need to determine when their testing efforts are sufficiently good for them to have confidence in the correctness and robustness of their application.

This thesis is about these three main challenges, which we will refer to as test generation, program repair, and test adequacy. Program testing can be carried out at different granularities of a program. For example, in unit testing, the purpose is to exercise small units of the program in isolation, such as the behavior of a single function, whereas end-to-end testing techniques instead interact with the program from the end-user’s point of view. In this thesis we consider end-to-end testing, where the application is tested as a whole.

The focus of this thesis is on improving over state-of-the-art techniques for
CHAPTER 1. INTRODUCTION

event-driven applications. Event-driven applications are an important class of applications, which is challenging to test thoroughly. Notably, both mobile and web applications are examples of event-driven applications. In event-driven applications, event handlers execute in response to events, which originate, for example, from the user clicking on the screen. Therefore, the input to such applications can be thought of as a sequence of events. In the context of mobile and web applications, developers often use end-to-end testing frameworks, such as Espresso [67], Protractor [6], and Robotium [8], to write tests that simulate a sequence of events, and check that the application behaves as expected. The purpose of such tests, also called regression tests, is to prevent that changes to the application introduce errors into functionality that was previously known to be correct. Although this is tedious and time consuming, it is an effective way to test that long and complicated use cases work properly.

Many event-driven applications are being implemented using dynamically typed programming languages. The main distinguishing feature of these languages is that all syntactically correct programs can be executed. In comparison, programs written in a statically typed programming language may be rejected by the type checker, if they are ill-typed. For many years, JavaScript code has been used to drive almost every modern client-side web application, and in more recent years, it has also gained momentum in the development of event-driven server-side applications. Today, these kinds of event-driven applications are also being built using other dynamically typed languages that can be compiled to JavaScript, such as Dart and TypeScript, both of which support optional static typing. From a testing perspective, the use of dynamically typed programming languages presents developers with additional challenges, since the lack of a static type system that guarantees the absence of type errors enables a new class of errors that needs to be tested for.

It is difficult to design a set of inputs that cover many interesting behaviors of an application. Furthermore, due to nondeterminism, even the same user event sequence may lead to significantly different program behavior. This can happen if two events may arrive in either order, and the final state of the application depends on the order in which the two events arrive. Such situations are also known as event races. Ordinary manual testing is generally inadequate to reveal errors that are caused by nondeterminism. These challenges make it difficult for developers to thoroughly test event-driven applications, and thereby to gain confidence that there are no erroneous behaviors. This has motivated the development of better testing techniques for event-driven applications. As an example, there exists a wide range of literature on automated testing techniques that attempt to explore the application as an end-user would. These techniques have generally not been adopted in practice, though. It is an open challenge to design better, automated testing techniques, which are practically useful for event-driven applications, as well as techniques that can be used to leverage some of the application-specific knowledge that is present in manually written tests, such as inputs and oracles, since such techniques will enable
developers to get even more mileage from their testing efforts.

Testing is used with the intent of identifying errors in a given program, as also noted by Myers [132]. Therefore, developers typically need to diagnose and fix the root cause of a test failure themselves, by debugging the application. This can be a highly nontrivial and error-prone task. In addition to the challenge of test generation for event-driven applications, another main challenge is therefore to design techniques that can assist developers in fixing the root causes of errors that have been detected from testing. In this context, it is interesting to note that techniques for program repair have had reasonable success in domains such as concurrency related errors in multi-threaded programs. Unfortunately, these techniques are not immediately applicable to event-driven applications.

Finally, developers need to determine when enough testing has been carried out. A famous quote from Dijkstra is: “Program testing can be used to show the presence of bugs, but never to show their absence” [45]. Although this is the case in general, additional testing efforts may not necessarily improve the overall capabilities to demonstrate that a given property holds. Therefore, a third main challenge is the question of test adequacy: when is the amount of testing adequate for a given property of interest?

1.1 Contributions

This thesis focuses on three main challenges that arise when trying to increase the robustness of event-driven and dynamically typed software applications: the design of new techniques for test generation, program repair, and test adequacy. The thesis identifies opportunities for improving over state-of-the-art techniques for each of the main challenges, proposes new techniques to address each of the challenges, and evaluates these techniques on real-world software applications.

In summary, this thesis makes the following contributions.

**Test generation** We propose a new methodology for running existing, manually written test suites for mobile applications, where unexpected event sequences get injected systematically and aggressively, in a way that does not alter the expected outcomes of the tests. The methodology has been implemented in a tool named **Thor** that works for Android test suites. An evaluation on real-world mobile applications demonstrates that the methodology increases the error detection capabilities of existing test suites.

In the context of JavaScript web applications, we present a new light-weight testing technique for detecting common kinds of event race errors that only manifest during the loading of a web page. A key insight is that these common kinds of errors can be detected by simulating an adverse scenario, where events are injected eagerly into the execution. In an evaluation on popular web applications, we find that our implementation, **InitRacer**, is capable of
detecting harmful, real-world initialization race errors, meanwhile reporting relatively few harmless races.

We also present a technique for detecting race conditions that arise when user interactions cause JavaScript web applications to update based on the results of asynchronous client-server communication. From an analysis of a given user event sequence, the technique plans a set of tests that are likely to expose event races with an observable effect on the screen. Experiments on a set of real-world web applications show that our implementation, AjaxRacer, can effectively detect observable event races errors in practice.

**Program repair** We present a new technique that dynamically restricts the nondeterministic scheduling of events in JavaScript web applications according to a given repair policy, in order to prevent erroneous executions from manifesting. The technique is implemented in EventRaceCommander. Our evaluation shows that most event race errors can be prevented by the use of application-independent repair policies, which intuitively enforce different expectations that developers tend to have to the scheduling of events.

**Test adequacy** We present a notion of test completeness for dynamically typed languages, that determines if a test suite has sufficient coverage to prove a given type-related property, along with a light-weight static analysis that conservatively approximates test completeness. Using our implementation Goodenough, which has been developed for the Dart programming language, we demonstrate that our analysis is capable of inferring test completeness facts from a set of benchmarks, and that these facts can be used to prove the absence of some type errors, which cannot be proven using a traditional and light-weight static type analysis.

### 1.2 Published Papers and Manuscripts

The following papers are included in their original, published versions in Part II of this thesis. Some minor adjustments have been made to the layout of these papers to accommodate for the format of the thesis.


- **Analyzing Test Completeness for Dynamic Languages.** Christoffer Quist Adansen, Gianluca Mezzetti, and Anders Møller. Published in Proc. 25th International Symposium on Software Testing and
1.3. OUTLINE


• Repairing Event Race Errors by Controlling Nondeterminism. Christoffer Quist Adamsen, Anders Møller, Rezwna Karim, Manu Sridharan, Frank Tip, and Koukshik Sen. Published in Proc. 39th International Conference on Software Engineering (ICSE), May 2017 [11]. Included in Chapter 10. An appendix, which elaborates on the experiments, has been added in this thesis.


In addition to these published papers, this thesis also includes the following unpublished manuscript.


The author of this thesis has contributed significantly to all phases of the corresponding research projects—from the development of early ideas to prototyping, implementation, evaluation, and paper writing. Furthermore, the author has been main developer of the implementations for each of the published papers (the author exclusively implemented the tools AjaxRacer [13], Event-RaceCommander [11], and InitRacer [12]).

1.3 Outline

This thesis is divided into two parts: Part I gives an overview of the research that has been carried out by the author, and Part II consists of the co-authored research papers, as detailed in the previous section.

Part I is structured as follows. Chapter 2 gives some background on event-driven and dynamically typed software applications, which are the main focus of this thesis. Chapters 3, 4, 5, and 6 are dedicated to the three main challenges that have been motivated in this introduction: test generation, program repair, and test adequacy. Each of these chapters gives an overview to state-of-the-art, states the research challenges that have been identified by this thesis, and introduces the contributions that have been made by the research papers in Part II to these challenges.
Chapters 3 and 4 both consider the problem of automated test generation, but with different focuses. Chapter 3 focuses on techniques that explore the user interface of an application, with the purpose of finding errors, and first gives an introduction to state-of-the-art for automated test generation. Next, the chapter motivates the need for techniques that amplify existing, manually written test cases, based on the limitations of fully automated techniques. Finally, the chapter presents the key insights from our methodology for Android testing (Thor [9]).

The subsequent chapter, Chapter 4, focuses on event race detection. This chapter gives an introduction to state-of-the-art race detectors for JavaScript web applications and mobile applications. Then, we motivate the need for race detectors that are more practical, and summarize the main contributions from our work on practical race detectors for JavaScript web applications (InitRacer [12] and AjaxRacer [13]).

Chapter 5 addresses the problem of program repair, with a focus on techniques that aim to prevent concurrency related errors. The chapter first introduces related work from a shared-memory, multi-threaded setting. Then, the chapter motivates the design of techniques that prevent race errors in a single-threaded, event-driven setting. Finally, we present our technique for controlling the nondeterminism in JavaScript web applications along with the key observations and contributions from our work (EventRaceCommander [11]).

Chapter 6 considers the challenge involved in determining whether a test suite is adequate. The chapter first gives an introduction to various test adequacy criteria. A weakness of many test adequacy criteria is the lack of guarantees that they can provide. Therefore, the chapter also gives an overview of techniques that attempt to give guarantees based on observations from a finite set of concrete executions. Finally, we present a notion of test completeness for dynamically typed languages and our hybrid static/dynamic analysis framework for conservatively approximating test completeness (Goodenough [10]).

Chapter 7 summarizes the conclusions of this thesis.
Chapter 2

Event-Driven and Dynamically Typed Software Applications

This chapter gives an introduction to the event-driven execution model and dynamically typed programming languages. This background is essential for the remainder of Part I. Specifically, the subsequent chapters will use this as a basis for describing some of the main challenges related to testing of event-driven and dynamically typed software applications. This chapter additionally describes two event-driven systems that are being used heavily, JavaScript web applications and Android mobile applications.

2.1 The Event-Driven Execution Model

The primary feature that distinguishes the event-driven execution model from other programming paradigms is that application code executes in response to events. These events may originate from a user that interacts with the application (e.g., by clicking on a button), or from the underlying system (e.g., due to an incoming network message, information being read by a sensor, or a timer that expires). Some of these events may ultimately have been created by the application itself. For example, many event-driven systems enable applications to schedule a function for execution after a given delay. Similarly, applications may create a network event in the future by sending a request to a server, although the application has no control over the time at which the corresponding network response (or error) event arrives. In comparison, the arrival of other types of events is completely independent of the application (e.g., events initiated by a user).

It is only possible for an application to process one event at a time per thread. Therefore event-driven systems usually place newly arrived events in an event queue, and asynchronously dispatch events to the application from
the event queue, when the application is ready. In order for an application to execute code in response to events, the application should register a function as an event handler for the event type of interest. The processing of the event queue, which is performed by the underlying system, thus involves repeatedly removing an event from the event queue, and then invoking all of the event handlers that have been registered by the application for the event’s type, if any.

The event-driven execution model is a particularly good fit for graphical user interface (GUI) applications, since these applications enable the user to perform actions, and then react in response to them. Two prominent examples of such applications are JavaScript web applications and mobile applications, which are a main focus of this thesis. The remainder of this section provides details on the event-driven execution models that are being used in these applications.

JavaScript web applications  JavaScript web applications are built from HTML, CSS, and JavaScript code. When a user requests a browser to display the web page at a given URL, the browser fetches and parses the corresponding HTML code. During parsing, the browser builds the Document Object Model (DOM) of the HTML code (i.e., a tree representation of the HTML), which is used for rendering the web page.

The HTML code may contain `<script>` tags that have JavaScript code associated with them. The JavaScript code may either be declared inline (i.e., inside a `<script>` tag), or at an external URL, specified using the `src` attribute. When the browser encounters a `<script>` tag during parsing, it executes the corresponding JavaScript code without preemption, i.e., the script executes until its completion without the interleaving of other events. Note that, if the script is external, the browser must first fetch the code from the given URL. By default, the browser executes the JavaScript code that belongs to a given `<script>` synchronously, when it encounters the tag during parsing. However, it is also possible for developers to indicate that an external script should execute asynchronously, using the `async` attribute on the `<script>` tag, or that a script should only execute after the web page has been parsed, using the `defer` attribute. The use of these attributes enables the browser to continue its processing of the remainder of the web page, thereby speeding up the performance as perceived by the user.

The JavaScript code can use the DOM API to access and mutate the current state of the DOM, to send AJAX requests, and to register event handlers. Event listeners can be registered by assigning functions to special property names of DOM objects (e.g., `onclick`, `onload`), or by calling built-in functions from the DOM API, such as the `addEventListener` and `setTimeout` functions. It is also possible to register event handlers by setting special attributes of HTML tags directly in the HTML code.
2.1. THE EVENT-DRIVEN EXECUTION MODEL

index.html

1  <html>
2  <head>...</head>
3  <body>
4      <input id="search-field" type="text"
5              placeholder="Enter search terms" />
6  <div id="search-results">...</div>
7  <script src="script.js"></script>
8  </body>
9  </html>

script.js

10  var searchField = document.getElementById('search-field');
11  var searchResults = document.getElementById('search-results');
12  function handleKeyPress(event) {
13      var xhr = new XMLHttpRequest();
14      xhr.open(`/method="/GET",
15              `/url="/search?q=' + encodeURIComponent(searchField.value),
16              `/async="/true`);
17      xhr.onreadystatechange = function () {
18          if (xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {
19              searchResults.innerHTML = xhr.responseText;
20          }
21      };
22      xhr.send(/data="/null`);
23  }
24  searchField.addEventListener('keypress', handleKeyPress);

Figure 2.1: A simple JavaScript web application.

Example Figure 2.1 illustrates the implementation of a simple JavaScript web application that dynamically updates its UI in response to user events, by registering appropriate event handlers and communicating with a server using AJAX. The UI of the web application contains a search field (declared in line 4). When the browser reaches the external script in line 7, it synchronously fetches and executes script.js before proceeding. This script retrieves a reference to the search field (line 10), and then registers the function handleKeyPress (line 12) as an event handler for keypress events on the search field using the built-in function addEventListener (line 24). When the user enters a character into the search field, the web application sends an asynchronous AJAX request to a server. This is done by creating an XMLHttpRequest object (line 13), and initializing it, by calling the open method (line 14) with the relevant HTTP method and URL, and with a boolean true to indicate that the request should be asynchronous. Finally, the request is sent with an empty body by calling the send method of the XMLHttpRequest object with null (line 22). In order to listen for the response event corresponding to the AJAX request, an event
CHAPTER 2. EVENT-DRIVEN AND DYNAMICALLY TYPED APPS

Intermediate readyStateChange events (i.e., those where the response has not been received entirely) have been omitted for simplicity.

Figure 2.2: Possible interleavings in the example from Figure 2.1.
2.1. THE EVENT-DRIVEN EXECUTION MODEL

time, if the user enters a character into the search field before the response corresponding to the previous AJAX request has arrived. It is nondeterministic which AJAX response event will arrive first. This may lead to a situation where the displayed search results are inconsistent with the search terms entered by the user. To illustrate this, Figure 2.2 shows two schedules that may arise when the user searches for the term “js”. Both schedules correspond to a situation where the user has pressed the keys ‘j’ and ‘s’ before any of the corresponding AJAX response events have arrived. In the first schedule, the AJAX requests and their corresponding response events are in first-in-first-out order. As a result, the search result ends up being consistent with the user’s request. The second schedule instead leads to a situation where the search results corresponding to the search term “j” are displayed, although the user searched for “js”, because the AJAX requests and their response events are not in first-in-first-out order.

Chapter 4 is dedicated to techniques whose goals are to detect errors that manifest nondeterministically due to event races.

Android mobile applications Google Android is a mobile operating system that primarily targets smartphones and tablets. One of the key features of Android, and smartphones in general, is that they allow the user to install third-party applications. As of January 2018 more than 3.5 million Android applications have been published at the Google Play Store.

Android applications are implemented primarily using Java and Kotlin, and feature shared-memory, multi-threaded programming. Many modern applications also implement functionality using C and C++ with the Android NDK. In addition to these languages, which are being used to implement the logic of an application, XML configuration files are used to declare various pieces of essential information about an application, such as its name, components, and layouts.

Android applications consist of four different types of components: activities, broadcast receivers, content providers, and services. An activity represents a single screen and its user interface, and is implemented by a class that inherits from the Activity class in the Android framework. All activities must be declared in a special file, AndroidManifest.xml, along with the entry points of the application, i.e., the activity that gets started when the user opens the application, and activities that can be started directly by other applications.

Every activity follows the Android activity lifecycle, which is shown in Figure 2.3. When the user launches an activity it causes the activity to transition to the “Created” state. The Android framework signals this to the application by calling the onCreate method of the activity. In response, the application is supposed to carry out initialization that should only be done once for the activity. The activity transitions to the “Started” state

1 1http://www.appbrain.com/stats/number-of-android-apps
CHAPTER 2. EVENT-DRIVEN AND DYNAMICALLY TYPED APPS

Figure 2.3: The Android activity lifecycle (from http://developer.android.com/).

when it becomes visible to the user, and to the “Resumed” state when the activity goes to the foreground, where the user can start interacting with the activity. Thus, the application should register the appropriate user event handlers in the callbacks corresponding to these state changes. Event handlers can also be registered in the XML layout files, for elements that are not created dynamically. At a later point, when a user event causes a new activity to start, or a system activity needs to start (e.g., due to an incoming phone call), the current activity transitions from the “Resumed” state to the “Paused” state, and then to the “Stopped” state, where the activity is no longer visible. In response to the corresponding callbacks, the activity should persist data and release resources. Finally, the activity may get destroyed entirely, if another component requests so, or by the Android system to save resources.

Example Figure 2.4 presents a partial implementation of an Android activity that downloads and shows a random image when the user clicks on a button on the screen. When the activity is launched, the onCreate method (line 5) is called by the Android framework. This method updates the view hierarchy of the activity to the one that has been declared in the layout file activity_main.xml, by calling the method setContentView (line 7). The layout file declares a button, and also specifies that the method onClick (line 10) should be registered as an event handler for click events on the button. When the user clicks on the button, a DownloadImageTask object is created (line 11). The class DownloadImageTask inherits from a class from the Android framework called AsyncTask, which provides functionality for performing asynchronous tasks on a background thread (to avoid blocking the UI). When the activity invokes the method execute on the DownloadImageTask object in line 12, a random image is downloaded, but on a different thread, and then a Bitmap object is constructed.
2.1. THE EVENT-DRIVEN EXECUTION MODEL

MainActivity.java

```java
public class MainActivity extends Activity {
    private DownloadImageTask task = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void onButtonClick(View view) {
        task = new DownloadImageTask(this);
        task.execute();
    }

    public void onDownloadFinished(Bitmap bitmap) {
        ImageView imageView = (ImageView) findViewById(...);
        imageView.setImageBitmap(bitmap);
    }

    @Override
    protected void onStop() {
        if (task != null && task.getStatus() != FINISHED) {
            task.cancel(true);
        }
    }
}
```

DownloadImageTask.java

```java
public class DownloadImageTask extends AsyncTask<...> { ... }
```

activity_main.xml

```xml
... <Button android:onClick="onButtonClick" ... />
```

Figure 2.4: A simple Android activity.

from the raw bytes (the corresponding code is omitted from Figure 2.4). After that, the Bitmap image is passed to the method `onDownloadFinished` (line 15), but now on the UI thread. This method displays the image in the UI (lines 16-17). The activity may be stopped at any time. When this happens, the Android framework invokes the method `onStop` (line 21), which tells the `DownloadImageTask` object to stop the current download (if any) in line 23.

Android applications can use broadcast receivers to listen to system events, such as sensor changes, or events from other applications. A broadcast receiver is a component that receives intents that have been broadcasted. An intent is an object that describes an operation to be performed, by means of a unique
identifier for the operation, and a map of additional key-value pairs. These
intents can be broadcasted implicitly, meaning that the receiver of the intent
is not specified, and that all broadcast receivers can receive and react to the
intent, or explicitly, where an intent is sent directly to the broadcast receiver
with a given identifier.

An Android service is a component that runs in the background, with
no user interface. Activities and other components can, for example, start
a service to perform a long-running task on a different thread, and get the
result back asynchronously (e.g., once a file has been downloaded). Similarly
to activities, services have their own lifecycle and are implemented by reacting
to callbacks that happen in response to state changes.

The last type of component is content providers. A content provider
manages and exposes a set of data, which can be stored, for example, on the
file system, or in a database, to the application that owns the content provider,
as well as other applications, depending on the permissions.

Compared to web applications, Android applications are subject to a much
wider variety of events. For example, at any time, the user may interact with
the screen, lifecycle events may cause the state of an activity to change, active
services may finish with a result, broadcast receivers may receive notifications
from other applications, or the system due to sensor changes, etc. It is
challenging for developers to correctly anticipate all the possible scenarios that
can arise in presence of all these events. Chapter 3 is dedicated to techniques
that address this problem by automated test generation. In addition to
these problems, the execution model of Android applications is such that
applications are subject to traditional data races from shared-memory, multi-
threaded programs, and event races, as known from JavaScript web applications.

Section 4.2 gives a brief introduction to race detection for mobile applications.

2.2 Dynamically Typed Programming Languages

Many event-driven applications are being implemented using dynamically
typed programming languages. Notably, almost all modern client-side web
applications are driven by JavaScript, which is among the most popular
programming languages with dynamic typing. A significant amount of event-
driven, command-line and server-side applications are also implemented in
JavaScript.

The main feature that sets dynamically typed languages apart from statically
typed languages (such as C, C++, C#, Java, and Scala) is that all programs
that are syntactically correct can be executed. In comparison, developers that
use a statically typed programming language are required to satisfy the type
checker, often by carefully inserting type annotations, or their programs will
be rejected.
Many event-driven applications are also being built using other dynamically typed languages that can be compiled to JavaScript, such as Dart and TypeScript. Both of these languages support optional static typing, which offers programmers the possibility to annotate parts of the programs with types, so that they can benefit from type warnings produced by the type checker. The type systems in these languages are unsound, though, so even fully typed programs with no type warnings may suffer from type errors at runtime. Also, unlike fully statically typed programming languages, it is possible to execute Dart and TypeScript programs even when the type checker reports type warnings.

The flexibility offered by dynamically typed programming languages generally supports rapid development, but also comes at a price. These languages generally suffer from poor IDE support, since functionalities such as auto-completion, code navigation, and refactoring tools rely on the availability of type information. Furthermore, from a testing perspective, the use of these languages poses new challenges for developers, since the lack of a static type system enables a class of runtime type errors that needs to be tested for.

Two kinds of type errors that can occur at runtime are message-not-understood and subtype-violation errors. A message-not-understood error manifests if the program attempts to access a field or method that does not exist. As a simple example, consider the following piece of code, where a method called \texttt{m} is being invoked on a value that has been obtained by calling function \texttt{f}.

\begin{verbatim}
var x = f(y);
x.m();
\end{verbatim}

If the runtime type of \texttt{x} at this point does not have a method \texttt{m} during some execution, then the execution fails with a message-not-understood error. Note that it may depend on the value of \texttt{y} (which is passed as an argument to the function \texttt{f}) whether \texttt{x} has such a method, if \texttt{f} is overloaded. A subtype-violation error manifests if a type cast fails. Interestingly, there are no subtype-violation errors in JavaScript and TypeScript applications, since there is no notion of static types in JavaScript, and because the TypeScript compiler does not insert any type checks when compiling to JavaScript. The Dart programming language has two different runtime modes, production mode and checked mode. In production mode, type annotations and type casts are ignored, meaning that subtype-violations can also not occur in Dart when this execution mode is being used. As a simple example, consider the following line of code, where an expression is being assigned to a variable \texttt{x}, which has type \texttt{A}.

\begin{verbatim}
A x = f(y);
\end{verbatim}

In Dart’s production mode, this code succeeds even if the function \texttt{f} happens to return a value whose type is not a subtype of \texttt{A}. However, in checked mode, Dart performs a subtype check at every assignment to a typed variable or field,
and at every explicit type cast. Thus, if an execution of a Dart program does not comply with the type annotations and explicit type casts of the program, then the execution will fail in checked mode with a subtype-violation error. In particular, the above line of code would fail at runtime if $f$ returns a value whose type is not a subtype of $A$.

Developers can use testing to cover instructions that perform field lookups, method lookups, and type casts, and thereby gain confidence that these instructions cannot fail with a message-not-understood or subtype-violation error. However, this approach is generally inadequate to prove the absence of errors, as noted by Dijkstra [45], since there may exist an unseen execution that exposes an error. In Chapter 6, we address the problem of test adequacy, and present a novel program analysis that determines if the coverage provided by a test suite suffices to prove the absence of type-related errors.
Chapter 3

Test Generation for Event-Driven Software Applications

Testing is the process of executing a program on a set of inputs, and then determining if the resulting executions are good or bad, depending on the expectations of the developers. If the execution of a test is good, the test is said to succeed, otherwise it is said to fail. It is widely accepted that testing can be used to improve the robustness of an application.

Testing can be performed at many different granularities of the program. For example, unit testing focuses on exercising an individual unit (e.g., a function) in isolation, whereas end-to-end testing focuses on testing the application as a whole, which, for applications with a user interface (UI), involves traversing the UI by performing user events. The focus of this thesis is on end-to-end testing of event-driven applications, and in the following we therefore (mostly) restrict the discussion to testing of event-driven applications as a whole.

It is inherently difficult to test event-driven applications thoroughly. Many applications continuously enable the user to perform new actions, or listen for new system events, making it impossible to test all possible event sequences. Developers often focus on writing test cases using end-to-end testing frameworks, such as Robotium [8], Espresso [67], and Protractor [6], that simulate how an end-user would perform one of the use cases that are supported by the application. However, a recent empirical study of manually written JavaScript tests by Fard and Mesbah [50] finds that many tests do not adequately cover asynchronous and event-dependent callbacks. In practice, users may deviate from the expected use cases, and system events may arrive at unexpected times. Such scenarios are difficult for developers to anticipate and are a major source of programming errors, as demonstrated by recent efforts on designing better techniques for UI testing [9, 83, 84, 115] and event race detection [12, 31, 81, 118, 146].
Developers are also faced with a great deal of competitiveness from other applications. For example, in the domain of mobile applications, many applications serve the same purpose, meaning that users often have the freedom to choose between several applications to fulfill a need. This makes users more likely to stop using an application if it does not work properly. In a user study from 2013, only 79% of users were willing to retry a mobile application if it failed to work the first time [167], and in a study from 2011 as many as 26% of mobile applications were found to be used just once [166].

The challenges involved in testing event-driven applications thoroughly motivate the development of automated testing techniques, as well as techniques for boosting the effectiveness of existing, developer-written UI tests, in order to help developers perform more thorough testing, and thereby increase the robustness of their applications.

**Outline** The remainder of this chapter is structured as follows. Section 3.1 provides an overview of state-of-the-art techniques for automated UI testing of event-driven applications, with a focus on mobile applications. Section 3.2 motivates and introduces state-of-the-art techniques for leveraging existing, manually written test cases. Finally, Section 3.3 presents the research challenge that we address in our ISSTA 2015 paper (THOR [9]) and summarizes the contributions and key ideas from our work.

### 3.1 Automated User Interface Testing

The academic community has developed a wide range of testing techniques that automatically explore the UI of mobile and web applications. The main advantage of these techniques is that they can be applied to thousands of applications with little effort. These techniques aim to detect errors with little or no help from the developers, and typically explore the UI of the application under test by iteratively triggering one of the enabled events. Being fully automated, they operate with no knowledge of the intended application behavior, and therefore use application-independent oracles that detect problems that applications should clearly not suffer from, such as uncaught exceptions.

Despite all the work on automated test generation for mobile applications, the field is relatively scattered, in the sense that few techniques build on top of existing ones. In the following, we give an overview of some of all the test generation techniques that have been proposed in the literature. Finally, we discuss the state-of-the-art and highlight general limitations of fully automated testing techniques.

**Random testing** One of the simplest ways to test an application is via random testing [74], in which the application is executed on randomly chosen inputs. There exist a wide range of extensions to random testing, such as
feedback-directed random testing [137]. One of the advantages of random testing techniques is that they (typically) treat the application under test as a black box and therefore do not require any knowledge of the application—unlike, for example, some model-based testing techniques that require to be given a model of the application.

The input to an event-driven application can be thought of as a sequence of events. Random testing techniques for event-driven applications therefore tend to continuously generate an event at random and send it to the application under test. The Android Monkey [65] is a random testing tool that has been developed by Google and is included in the Android SDK. Monkey supports a wide range of user and system events. It is a naive, random testing tool in the sense that it is oblivious to the widgets on the screen: it generates user events at random coordinates on the screen, without considering whether the widget at a given location has an event handler or not. This may lead to many events that do not cause any application code to execute, as well as redundant events (e.g., clicking on two different coordinates that cause the same event handler to execute). Its simple design, however, makes Monkey capable of generating many events in a short period of time, since there is barely no decision making involved in the generation of a new event.

Various kinds of random testing techniques have also been proposed for mobile applications in the academic community [76, 83, 110, 115]. Liu et al. [110] extend adaptive random testing, which is a technique for improving the effectiveness of ordinary random testing techniques (such as Monkey), to mobile applications. Their technique, SmartMonkey, incrementally generates a test suite $T$, by iteratively generating a set of candidate tests $S$, and selecting the test from $S$ that has the largest distance to the nearest test case in $T$. To this end, Liu et al. propose a new distance metric for Android test cases, which uses the Levenshtein string distance metric [106] to compute the distance between two sequences of event types. Hu and Neamtiu [83] propose to use Monkey for sending a sequence of random events to the application, and then look for patterns that identify potential errors in the Android log files. Machiry et al. [115] propose a widget aware random testing tool, Dynodroid, which relies on a modified version of the Android framework (Gingerbread, version 2.3.5) in order to extract the widgets on a screen for which an event handler is actually registered. Dynodroid supports three different strategies for choosing which of the enabled events from the current state to trigger next: one selects the event that has been selected least frequently, another selects uniformly at random, and the last one randomly selects an event in a way that prioritizes events that have not previously been triggered in the current context. Finally, PUMA [76] is a framework by Hao et al. that can be used to implement a

---

1This thesis uses the term `widget` to refer to an element on the screen, rather than a view that can be embedded in other applications. This is consistent with existing literature on automated testing for Android applications (e.g., [16] [115] [120] [191] [192]).
wide variety of dynamic analyses. To demonstrate the expressiveness of the framework, the authors implement several dynamic analyses using PUMA, including a random testing technique, which injects null HTTP responses into the HTTPClient library.

In the context of Windows Phone applications, Ravindranath et al. [145] present a system called VANARSENA that tests an application using several concurrent, randomized monkeys. According to Ravindranath et al., this alleviates the problem of having to remember which actions that have previously lead to crashes, in order to explore deeper paths into the application. VANARSENA instruments the application binary to improve effectiveness: (i) a technique called hit testing makes it possible to avoid performing redundant events by determining the events that will lead to the execution of different event handlers in the source code, (ii) by signaling exactly when the application has finished processing an event (possibly asynchronous), VANARSENA can speed up event generation, and (iii) by intercepting certain API calls VANARSENA mimics various environment failures.

Ermuth and Pradel [47] infer so-called macro events from the interactions of human users with a web application. A macro event is a sequence of low-level events that correspond to a logical interaction of a user with a web page (e.g., hovering a widget, in order to open a dropdown menu, and then clicking on a menu item). Ermuth and Pradel use the inferred macro events during random testing to alleviate the problem that purely random testing techniques are unlikely to explore complicated user event sequences.

Model-based testing  Model-based testing techniques rely on a model of the application under test to systematically test the given application. In the literature, some model-based testing techniques require that a model is provided by the developers, while others are capable of learning a model automatically. In the context of mobile application testing, most model-based testing techniques build a finite model of the application on-the-fly during testing, and use the model to guide further exploration. The use of a model has several advantages. First of all, it enables systematical exploration of an application’s UI. Furthermore, the model can be used to avoid the generation of redundant events. Another advantage is that model-based testing techniques can make use of a well-defined stopping criterion, by testing the application until a given property on the model holds (e.g., until all transitions in the model have been covered). This is in contrast to random testing techniques, which typically operate with a bound on the number of events to generate, or a time budget.

Amalfitano et al. [16] present a framework for automated test generation called MOBIGUITAR, which is based on the GUITAR framework [123]. MOBIGUITAR infers a state-machine model of the application under test using an extended version of ANDROIDRIPPER [15], and then generates test
cases from the model in a way such that all pairs of adjacent events are covered by a test. Mirzaei et al. [128] also propose to generate tests from a model of the application, though in a combinatorial way, and from a model that has been extracted statically by an analysis that is based on the Soot [170] program analysis framework. To avoid a combinatorial explosion of tests, they propose to identify widgets that are independent, and therefore do not need to be tested in combination, using a static dependence analysis.

Yang et al. [184] and Azim and Neamtiu [25] present two automated testing techniques, ORBIT and A$^3$E, respectively, which systematically traverse the application in a depth-first based fashion. ORBIT extracts the events that are supported by every widget using static analysis, and uses this information to trigger an event only if the corresponding widget may actually have an event handler for that event. In comparison, A$^3$E uses a fully dynamic approach. A$^3$E also supports a different strategy that uses a static activity transition graph of the application in order to cover more activities, by exploiting that exported activities can be started directly from third-party applications using intents. Choi et al. [38] argue that it is expensive to restart an application, in order to continue exploration from the initial state. Their technique, SWIFTHAND, infers a model of the application on-the-fly, and continuously selects a user event sequence in the model that leads to a state with an unexplored transition, to avoid restarting the application. Unlike other testing techniques, SWIFTHAND uses automata learning to refine the model when it finds a discrepancy between the abstraction of the current state and the state that is expected according to the model.

Hu et al. [84] present a technique called AppDoctor, which quickly scans for errors using so-called approximate execution. In approximate execution, user and lifecycle event handlers are invoked directly (as opposed to triggering, say, a long-press event), which can significantly speed up testing. The increased speed comes at the cost of false positives, though. Therefore, AppDoctor attempts to verify that event sequences that crash in approximate execution also crash when being replayed in faithful mode. AppDoctor supports several exploration strategies, such as breadth- and depth-first exploration.

One of the challenges in model-based testing is to find the right level of abstraction for the model. A very fine-grained abstraction (e.g., one that directly compares screenshots) will often lead to a state explosion, whereas a course-grained abstraction (e.g., one that merely compares activity names) may prevent automated exploration from reaching interesting application behaviors. Baek and Bae [26] experiment with various levels of abstraction. On a set of open-source applications, the authors find that the use of more fine-grained abstractions leads to a 16% increase in statement coverage on average, in comparison to an abstraction that is based solely on activity names.

Model-based testing has been adopted in industry by Google. Robo [66] is an automated testing tool from Firebase Test Lab, which is a cloud-based infrastructure where developers can upload their application binary for testing.
CHAPTER 3. TEST GENERATION

Robo automatically explores the UI of an application up to a given maximum depth and allows developers to specify inputs to text fields. Notably, Robo is able to identify login screens and automatically sign in using an account that has been provided by developers.

Evolutionary testing  In evolutionary testing, a population of individuals is evolved in order to maximize a given metric (e.g., code coverage) by crossing and mutating existing individuals. Mahmood et al. [117] propose the first evolutionary technique for testing Android applications, EvoDroid, in which test cases are represented by individuals that consist of input genes (e.g., form input) and event genes (e.g., click event). EvoDroid aims to cover as many unique paths as possible in a call graph of the application that has been extracted using static analysis. It also uses static analysis to compute a model of the application. This model is used to avoid situations where the crossing of two individuals would lead to an infeasible test, and to extend individuals with new genes. Mao et al. [120] adapt a multi-objective evolutionary search algorithm to test Android applications. Their technique, SAPIENZ, maximizes coverage and fault detection, while simultaneously minimizing the length of test cases, to aid debugging. SAPIENZ uses pre-defined patterns of low-level events that capture complicated interactions with the application in order to achieve higher coverage, similar to the random, macro-based testing framework from Ermuth and Pradel [47].

Fuzz testing  Fuzzing is a testing technique in which unexpected inputs are supplied to the application under test. Such inputs are often constructed by “fuzzing” a valid input (i.e., by making creative variations of it) to test if edge cases are handled correctly by the given application. The term fuzz testing is sometimes also used to refer to random testing. In this thesis, however, we only use the term to refer to testing techniques that do not merely simulate random user and system events.

In the context of mobile applications, developers need to account for the fact that their application will run on devices with a wide variety of form factors and screen sizes, that the network may change during use or drop entirely, that unexpected system events may occur at any time, that external inputs may be invalid, etc. To this end, Liang et al. [108] synthesize a large set of environmental contexts from real-world executions. In order to expose an application to contexts that are more likely to be of relevance to that application, their system CAIIPA uses machine learning to prioritize the contexts that have have had an impact on similar applications that have already been tested.

Ye et al. [185] focus entirely on activities that accept MIME data as input. Their technique, DROIDFUZZER, extracts the activities that accept MIME data as input, and then generates abnormal data input for these activities, by making variations of a valid input. Ye et al. report that typical errors found
by DroidFuzzer include crashes (e.g., due to an out-of-memory error) and the application becoming unresponsive. IntentFuzzer by Sasnauskas and Regehr [151] extracts the components that are exported by an application (entry points of the application), and then uses static analysis to approximate the keys that each entry point reads from its input, which is an intent object. For every entry point, IntentFuzzer then iteratively generates an intent, which has a random value for every key that might be read by the application, according to the the static analysis.

Symbolic and concolic testing  Symbolic and concolic execution is a disciplined way to generate inputs that explore different program paths. The main idea in classical symbolic execution [99] is to execute the program on symbolic values, and to fork the execution at every branch, meanwhile maintaining a path condition that describes the set of inputs that follows the same execution path, in terms of constraints on the symbolic values imposed by the branches. During the exploration, satisfiability is checked to prune infeasible paths, and in the end, a constraint solver can be used to find concrete inputs that follow the same path. Concolic execution [61, 156] explores one program path at a time, starting from an execution of the program on some concrete input. Alongside the concrete execution, a path condition is recorded as in symbolic execution. One of the main advantages of concolic execution is that symbolic constraints in the path condition can be replaced by concrete values, when the constraints become too complex for the theorem prover to handle. When the execution finishes, concolic execution generates a new concrete input that follows a different execution path, by negating a constraint corresponding to a single branch in the path condition, and solving the resulting path condition.

Anand et al. [18] use concolic execution to generate sequences of events, starting from the main activity of an Android application. Their technique, ACTEve, uses an instrumented version of the Android framework and the application under test, such that it can track events from their origin in the Android framework to the place where they are eventually handled in the given application. ACTEve records constraints on the event parameters (such as the coordinate of a click event), and then uses a theorem prover to generate events that follow other execution paths. In comparison to the other test generation techniques that have been described in this section, ACTEve is capable of generating events that exercise more execution paths in event handlers whose behavior depends on the parameters of the events.

Jensen et al. [90] propose a technique called Collider, which complements other testing techniques by trying to generate an event sequence that reaches a given target, such as a statement in the application under test. Collider starts from the target and repeatedly prepends events to an event sequence, until the sequence reaches the target from the main activity. One of the key insights of Jensen et al. is that the construction of such an event sequence
can benefit from two main ingredients: (i) *connector events* that navigate between different screens of the application, and (ii) *anchor events* that mutate parts of the program state that influence the branches that guard the target. COLLIDER searches for anchor events that cause the last event handler to reach the target, using symbolic summaries for the event handlers, which include information about the (symbolic) effect that event handler execution paths have on variables. When an anchor event has been selected, COLLIDER finds a sequence of connector events that connects the anchor event with the first event of the event sequence, by leveraging a UI model of the application.

**State-of-the-art** As is evident from the previous discussion, there exists a wide range of automated testing techniques for mobile applications. In a recent study, Choudhary et al. [39] compare several of the previous techniques in order to understand if any of the approaches are more promising than others, and to highlight general strengths and weaknesses of these techniques for future research. The authors select 68 open-source applications for the comparison, which have been used in the evaluation of at least one of the involved techniques. Surprisingly, they find that the random testing techniques MONKEY and DYNODROID perform significantly better than the other study subjects A3E, ACTEVE, ANDROIDRIPPER, PUMA, and SwiftHAND (ORBIT and EvoDroid are not available, and thus not included in the comparison). According to the study, MONKEY and DYNODROID perform almost equally well. For this reason, Choudhary et al. choose to declare MONKEY as a clear winner, motivated by the fact that MONKEY works with any version of the Android framework, unlike DYNODROID.

Zeng et al. [191] and Zheng et al. [196] extend the study of Choudhary et al. [39] by reporting on the limitations of MONKEY when applied to a large industrial-strength application, WeChat. On this application, the authors find that MONKEY achieves low coverage, even when manually logging in to the application. They argue that this can be attributed to MONKEY being widget oblivious (as addressed in previous work) and state oblivious (i.e., it does not prioritize events that are more likely to change the state). To this end, they design a simple random testing technique, which is similar to MONKEY, but implements widget and state awareness to improve over MONKEY. The latter is achieved by prioritizing events that have previously caused the application to transition to a new activity, or the current screen to change.

**The oracle problem** Most automated testing techniques fail to detect the more subtle functionality errors, because they typically operate without any knowledge of the expected behavior of the application under test, which is often referred to as the oracle problem. To some degree, the oracle problem can be alleviated by the design of better oracles that are not only restricted to application crashes. For example, CAIPA [108] uses anomaly detection: it
monitors the network traffic, CPU utilization, and memory usage during testing, and then compares the resulting data to that obtained in previous runs of the same application, but also to data obtained from testing of similar applications. Anomaly detection has also been used in static analysis to detect malicious application behavior and situations where an application behaves different from the way users expect it to behave [23, 24, 70]. Yang et al. [183] search for responsiveness problems in Android applications that may lead to “Application Not Responding” errors, by inserting artificial long delays at problematic operations (e.g., network calls). EventBreak [139] searches for slowdown pairs in web applications, which are pairs of events where triggering one event increases the execution time of the other event. When executed repeatedly, these event pairs may make the application unresponsive. LeakDroid [182] repeatedly executes event cycles that should have a neutral effect on the resource usage, while monitoring the execution to identify potential memory leaks. Finally, Zaeem et al. [188] define a notion of user-interaction features, which are actions associated with a common sense expectation of how the application should respond, and use this for generating oracles to QUANTUM.

3.2 Test Amplification

Fully automated techniques have been successful in detecting many application-agnostic errors, such as crashes, but have not yet been heavily adopted in practice. One concern is the limited amount of confidence that developers can gain from application-independent oracles, since these oracles are incapable of checking that the application under test is functionally correct. Therefore, automated testing is mostly used to complement manually written test suites in practice.

In practice, it is highly popular among mobile and web developers to use end-to-end testing frameworks for writing tests manually, as well as capture-replay techniques that can be used to record a test scenario from user interactions (e.g., Espresso Test Recorder [68]). This is witnessed by the fact that the GitHub repositories for Protractor[2] and Robotium[3] at the time of writing have been starred more than 2250 and 7150 times, respectively, and that a repository containing examples that demonstrate different frameworks and techniques for manual testing of Android applications[4] has more than 4850 stars.

An end-to-end test case is a script that opens the application under test, and then sends events to the application one at a time. The instructions in the script that send events can be interleaved by test assertions, which check that the UI or the internal state of the application matches the expectations...
of the developers, as well as instructions that wait for a given condition (e.g., to wait for an asynchronous task to finish before proceeding). A common pattern is that a test first initializes the application in a concrete state (either programmaticaly, or via a sequence of events), and then checks that a given use case works as intended, by simulating a sequence of user events.

Since a significant amount of the software development time is often devoted to testing \[107\], it is not unusual that test suites achieve high coverage of the source code, and that they implicitly incorporate a deep knowledge of the application being tested. Interestingly, this application-specific knowledge can in principle be used to mitigate two problems that are present in fully automated testing. First, manually written test suites tend to contain interesting user event sequences, with specific and carefully chosen input values, that provide execution paths that lead deep into the application under test. Techniques that are able to leverage manual tests are therefore less dependent on sophisticated, automated exploration tools. Second, a manually written test suite and its assertions specify the intended behavior of the application and can potentially be used to alleviate the oracle problem.

These observations have lead to the development of techniques that attempt to leverage existing tests, such as test amplification techniques \[51\, 139\, 183\, 194\]. The goal of these techniques is to automatically extend the fault detection capabilities of existing test suites. The Testilizer tool for web application testing by Fard et al. \[51\] extends manually written test suites with automated crawling and uses heuristics and machine learning for generating assertions to achieve an improvement in code coverage and fault detection. More specifically, Fard et al. observe an execution of the test cases in order to build a model of the application, and then pass this model to an automated, model-based testing technique. In order to generate application-specific assertions for new execution paths, Fard et al. exploit that assertions in UI tests tend to be written using a well-known API from a testing framework. This makes it possible to determine what the assertions of a test is checking at runtime (e.g., that a DOM element has a certain attribute) and to reuse these assertions in new states where the same assertions should hold.

Yang et al. \[183\] propose a test amplification technique for detecting poor responsiveness in Android applications. They generate end-to-end test cases from a UI model of the application under test, but their technique can also be applied to readily available end-to-end test cases. Their technique first observes an ordinary execution of every test, in order to collect the response times for every user event. Then every test is executed in an environment where delays are injected during the use of certain APIs (e.g., upon blocking network communication). Finally, they compare the response times for the user events from the two executions of every test, and report an issue if the injected delays cause an observable effect on the response times of the user events. EventBreak \[139\] also searches for responsiveness issues, but in web applications \[Section 3.1\]. More specifically, EventBreak first observes an
3.3. TEST AMPLIFICATION USING ADVERSE CONDITIONS

initial execution to record a trace, which stores information about the events and their response times. Based on the trace, EVENTBREAK infers potential slowdown pairs, and then investigates the individual slowdown pairs in more detail using targeted exploration. By using an existing, manually written test suite to generate a set of initial executions, EVENTBREAK provides a way to amplify the tests.

Zhang and Elbaum [194] observe that developers of Android applications often get exception handling wrong, when dealing with APIs for interacting with external resources, such as Bluetooth and GPS. Furthermore, they argue that such code is also difficult to validate. To this end, the authors propose to amplify manually written unit test suites, by instrumenting relevant API calls, to make it possible to simulate that these calls fail with an exception. Then, for each test, they systematically explore all possible exception scenarios for the first $k$ API calls, by repeatedly executing the same test in different environments. In principle, the technique should also be applicable to end-to-end test suites. However, end-to-end tests tend to be significantly slower than unit tests. As a result, it is likely only possible to systematically expose every test to a small fraction of the possible exception scenarios. In particular, it is worth noting that the technique by Zhang and Elbaum, from a suite of just 21 test cases, generates a total of 17,921 amplified test cases.

Other techniques also focus on unit testing. These are not immediately applicable to end-to-end testing of event-driven applications, though. Xie and Notkin [181] infer operational abstractions from existing unit tests and then generate new tests that attempt to violate these abstractions, Fraser and Zeller [60] infer parameterized unit tests from ordinary unit tests, and Mirshokraie et al. [127] leverage UI tests to infer assertions that can be used for unit testing.

3.3 Test Amplification using Adverse Conditions

The fact that the use of manually written tests is widely used by developers means that there is a large potential for techniques that can leverage existing tests. For this reason, it is an interesting research challenge to design test amplification techniques that extend the error detection capabilities of manually written end-to-end test suites for event-driven applications. This section presents our contributions to this research challenge.

One of our key observations is that Android applications largely remain untested in the presence of system events, even when they are accompanied by a thorough end-to-end test suite. These system events may arise in response to sensor changes, changes in the environment, from other applications, or actions that have been initiated by the user (e.g., due to the device being docked). One problem is that developers tend to focus on expected event sequences, which simulate a use case that should be supported by the application. Therefore, it
is not unusual that test suites mostly interact directly with the application UI, rather than, for example, rotating the device or triggering other system events, although these kinds of events are also frequent in real use. In a study of 5 open-source Android applications and their test suites, we found that the execution of the test suites only caused 7 device rotations, which is a negligible amount compared to the approximately 3500 events that were performed in total. One likely reason for this behavior is that the developers may be less aware of those kinds of events, or that there is no obvious place to test them among the ordinary test cases. Another reason may be that some testing frameworks do not provide the necessary primitives to trigger such events.

To combat this problem, we propose a methodology that can be used to amplify manually written end-to-end test suites for Android applications, to test for errors that would otherwise remain unnoticed. The methodology systematically exposes every test case to adverse conditions. Since we wish to extend the error detection capabilities with respect to system events, we use as adverse conditions sequences of events that are likely to be unexpected from the developer’s point of view. A key insight is that existing tests can be exposed to such adverse conditions in a way that does not alter the expected outcome of the tests. This way, it is possible to leverage not only the deep and interesting execution paths that are provided by a test suite, but also its application-specific assertions, by reusing them directly.

An important dimension of our methodology is which event sequences to inject during the execution of a test. To this end, we define a notion of neutral event sequences. An event sequence is said to be neutral if it can be injected during the execution of a test without altering the expected outcome. By using neutral event sequences as adverse conditions, we can safely report a warning to the developers in the case of a failing test, without introducing false positives. Note that test failures may happen for two reasons. One possibility is that an assertion in the test starts failing. However, the application UI may also have changed in response to a neutral event sequence, which may cause the test to get stuck, because the next event in the test is not enabled (e.g., if the test attempts to click on a widget that is no longer present on the screen).

Another important dimension is where neutral event sequences should be injected. For every test, we define a set of injection points, i.e., points during the execution of a test, where neutral event sequences can be injected. More specifically, we define one injection point for every event that gets triggered by a test case, at the point where all the event handlers of that event have terminated. This definition guarantees that the set of injection points does not change across multiple executions of a test, which is important in order for the resulting tests to be reproducible.

The definition of neutral event sequences and injection points enables our testing methodology in the context of Android applications: During the execution of a test case, we aggressively execute every event sequence from a catalog of neutral event sequences, at every injection point. In our experience,
most test failures can be attributed to the injection of a single neutral event sequence, at a single injection point. This is also confirmed by our experiments. To aid debugging, we therefore propose to use a variant of delta debugging \[190\] that automatically isolates the minimal, erroneous configuration.

We instantiate our methodology for testing of Android applications by designing a catalog of neutral event sequences. The classification of whether an event sequence is neutral or not can be subjective. Therefore, we use the observation from Zaeem et al. \[188\] that some event sequences are associated with a common sense expectation to their impact on the application. For example, a rotation of the device, followed by another rotation, is generally not expected to change the application state. The reader is referred to the paper (\[9\], Chapter 8) for the full catalog.

Experimental results from an evaluation of our implementation for Android testing, THOR, show that the proposed methodology can significantly increase the error detection capabilities of existing test suites: when applied to the test suites of 5 open-source Android applications, 429 of 507 test cases start failing! The 429 test failures reveal a total of 66 distinct problems. By using neutral event sequences, THOR is able to leverage the application-specific assertions that have been written manually by developers, by reusing them directly. Interestingly, 18 of the 66 distinct problems would be challenging to detect using automated testing techniques, since they are caught by such application-specific assertions. Other test amplification techniques also struggle to detect these errors, since most of these techniques alter the expected outcome of a test, and therefore cannot reuse assertions directly. For example, it is quite likely that the act of injecting exceptions to test exception handling code (like the technique by Yang et al. \[183\]) alters the expected outcome of a test.

The idea of aggressively exposing an application to adverse conditions has also proven to be useful for detecting event race errors. In our work on automatically detecting initialization race errors in JavaScript web applications \(12\), Section 4.3, we propose a technique in which events are injected aggressively and eagerly during the loading of a web page. In this context, the adverse conditions are comprised of events that occur unexpectedly early during loading. The next chapter is devoted to race detection.
Chapter 4

Race Detection

Due to nondeterminism it is possible to have event races in event-driven applications. An event race is a situation where two events $u$ and $v$ may occur in either order, the event handlers of $u$ and $v$ access the same memory location, and at least one of the accesses is a write. The existence of event races makes it possible for the same user event sequence to lead to significantly different program behavior, and in particular, to have subtle errors that do not manifest in ordinary situations (e.g., if an error only occurs if the network is slow) and cannot easily be reproduced. For this reason, errors that manifest nondeterministically due to races are typically not covered by ordinary UI testing techniques—neither manual nor automated techniques. This limitation has lead to a wide range of work on specialized techniques for detecting event race errors in event-driven applications.

Outline  Section 4.1 gives an introduction to existing race detection techniques for web applications. Section 4.2 focuses on race detection for mobile applications, which builds on some of the techniques that have been designed for web applications. Finally, Section 4.3 motivates and introduces our work on a practical initialization race detector for JavaScript web applications (InitRacer [12]).

4.1 Race Detection for JavaScript Web Applications

The fact that race conditions, as known from multi-threaded concurrency programming, also arise in the single-threaded, non-preemptive, and event-driven execution model of JavaScript was first observed in a blog post from 2009 by Steen [163]. In the JavaScript execution model, such race conditions arise due to the nondeterministic ordering of events, rather than fine-grained thread interleavings as in multi-threaded, shared-memory programs. For this reason, these race conditions are often referred to as event races. In the
same year as Steen’s blog post, Ide et al. [87] presented different sources of event races, including asynchronous AJAX communication with a server and HTML parsing, but they did not propose any analysis techniques for detecting problems caused by event races. These observations have generated an interest in the development of techniques for detecting situations where JavaScript applications behave nondeterministically depending on the order in which event handlers execute.

**Static race detection**  Zheng et al. [197] were the first to propose a technique for automatically detecting race conditions in JavaScript applications. Their technique focuses on atomicity violations, where an event handler unexpectedly executes in-between the sending of an AJAX request and the receipt of the corresponding response (e.g., due to the user clicking a button), and consistency problems, where a user event handler happens to operate on other data than expected by the user. Zheng et al. propose a static analysis for detecting such errors, which relies on the WALA program analysis framework [86] to construct call graphs and compute points-to information. Their static analysis approximates the accesses made by event handlers to global variables and reports a consistency problem if a user event handler reads a global variable that is being written by an AJAX response event handler (since the user event handler may behave differently depending on whether the AJAX response event has arrived or not). To focus on races that are more likely to be harmful, Zheng et al. propose to raise a race warning only when the memory location of the given race is used to, for example, compose an AJAX POST request, since this makes the corresponding race more likely to have an impact on persistent storage and thereby make permanent damage.

One of the main limitations of the approach proposed by Zheng et al. is the use of static analysis, which is notoriously difficult for JavaScript [19]. Even with a state-of-the-art modeling of the DOM, such as the one presented by Jensen et al. [92], static analysis is unable to precisely detect races on individual HTML elements. To make things worse, modern web applications often make heavy use of dynamic code loading (e.g., by calling the built-in function `eval`), which can be very challenging to reason about statically.

**Dynamic race detection** Petrov et al. [138] were the first to present a happens-before relation for JavaScript web applications, and to give a formal definition of a race in such applications. They proposed the first dynamic race detector for JavaScript web applications, WebRacer, which monitors the execution using a modified version of the WebKit JavaScript engine, and provides capabilities for simulating certain user actions, in order to expose more races. For every memory location, WebRacer keeps track of the last operation that read and wrote that memory location, respectively. WebRacer uses this information in the following way: Upon a new read or write operation,
a warning is reported if the operation is unordered according to the happens-
before relation with the last operation that wrote the given memory location
(or the last read of the given memory location, if the new operation is a write).

The design of WebRacer sacrifices completeness (relative to the given
execution) for efficiency. Raychev et al. [146] present a dynamic race detector,
EventRacer, that is capable of detecting all event races relative to a given
user event sequence, unlike WebRacer, without sacrificing performance. One
of their main contributions is an algorithm for scaling vector clocks [103]
to event-driven applications with thousands of events, which is based on a
notion of chain decomposition. This algorithm is important for enabling
EventRacer.

Both WebRacer and EventRacer use predictive analysis [157] to gener-
alize a single concrete execution to many other possible executions, using the
happens-before relation. This way, these tools get a lot of mileage from just
one execution. At the same time, though, this kind of analysis is the source
of many false positives and harmless warnings in the context of JavaScript
web applications. One of the main problems is that it is difficult to predict
what the effect of a race is without trying to execute one of the supposedly
erroneous interleavings. In fact, it may not even be possible to reorder the two
events from a race warning at all, although they are unordered according to the
happens-before relation, since this relation is typically incomplete in practice.
If the race is between events $a$ and $b$, then such a situation could happen if
event $a$ is generated by a user event on an HTML element that only becomes
visible after event $b$ has been executed. Neither WebRacer nor EventRacer
account for the visibility of HTML elements when building the happens-before
relation. Another situation that tends to lead to many harmless race warnings
is when the web application uses ad-hoc synchronization to account for both
orderings of the corresponding events, such that none of the orderings are
erroneous.

As a countermeasure against spurious warnings, Petrov et al. and Raychev
et al. propose the use of ad-hoc filters that focus on certain kinds of races.
Raychev et al. also define a notion of race coverage to deal with races that
are infeasible due to the presence of ad-hoc synchronization. Intuitively, a
race $a$ is said to cover another race $b$ if treating $a$ as synchronization (i.e.,
inserting a happens-before edge between the two events in $a$) eliminates $b$
as a race. Focusing on uncovered races significantly reduces the number of
warnings to consider. However, it may also hide harmful races. Another
problem is that many instances of ad-hoc synchronization lead to uncovered
races. Thus, it is not unlikely for harmful, uncovered races to drown in such
harmless, uncovered synchronization races. EventRacer has no mechanism
for distinguishing uncovered synchronization races from others. Finally, if a
developer uses ad-hoc synchronization to fix a harmful uncovered race that has
been reported by EventRacer, then the original race is covered, but it also
introduces a new uncovered synchronization race, which is completely harmless.
Detecting event races that matter  The observation that WebRacer and EventRacer tend to report an overwhelming number of spurious warnings has generated an interest in automatically classifying event races as harmful or benign.

In their position paper, Mutlu et al. [130] advocate the notion of observable races that lead to states that are visually distinguishable by the end-user, and propose to systematically explore all possible AJAX schedules in an attempt to detect such observable races. Mutlu et al. subsequently present a technique for detecting races that, similarly to the races detected by Zheng et al. [197], have an observable effect on persistent state such as cookies, local storage, and session storage [131]. Their technique observes a specific execution to record a trace (using a modified version of the Firefox browser and its SpiderMonkey JavaScript engine), and then performs a lightweight data flow analysis on the trace to determine situations where values created by two event handlers may both reach a sensitive location. Similar to WebRacer and EventRacer, their analysis is an instance of a predictive analysis, making it susceptible to spurious warnings. Furthermore, their technique can (albeit intentionally) not be used to detect observable races that do not impact persistent state, but still damage the user experience.

Hong et al. [78] present WAVE, an automated testing framework that is designed to detect observable races by exploring different event schedules. Similarly to the technique for exploring AJAX schedules by Mutlu et al., this technique does not consider races as such. WAVE first extracts an execution model that specifies constraints on the ordering of events from an initial, concrete execution that uses a given user event sequence. WAVE then creates a test suite by enumerating all the combinatorial sequences of events that satisfy the happens-before relation. These tests are subsequently executed in an environment that dynamically enforces the ordering of events specified by the individual test. To increase the likelihood of detecting errors early on, Hong et al. propose to prioritize tests according to the number of previously unexplored event orderings they will cover. A warning is reported if a test has different behavior from the initial execution in terms of: (i) exceptions being thrown, (ii) the resulting pages being visually different, or (iii) the browser becoming unresponsive, due to the inability of replaying the event sequence specified by the test.

Although WAVE does not use a predictive approach, it suffers from some of the same problems that arise with predictive analyses in the context of JavaScript web applications. In particular, WAVE reports a warning whenever it is unable to replay the exact order of events from a test (case (iii)). As previously discussed, though, it is unreasonable to expect that events can be reordered without this having an impact on the rest of the execution (e.g., due
to ad-hoc synchronization). In an experimental study by Jensen et al. [91], an approach similar to WAVE is reported to flag 100% of the tests as erroneous, for event sequences with thousands of events. Another problem with WAVE is scalability: It is not practically feasible to enumerate all the combinatorial sequences of events for realistic event sequences – for a web application with $n$ pairwise unordered events, the time and space complexity is $O(n!)$.

Jensen et al. [91] present the first model checker for event-driven applications, R$^4$, which explores all the nondeterminism relative to a given user event sequence, looking for observable races. More specifically, R$^4$ reports a warning if two schedules, with the same user events, differ in terms of: (i) exceptions being thrown, (ii) the resulting pages being visually different, or (iii) AJAX network communication. R$^4$ is stateless, meaning it does not store a representation of the entire execution tree, and thereby avoids the scalability issues of WAVE. In addition, it leverages EventRacer to perform dynamic partial order reduction [58] to avoid exploring redundant schedules. Still, it is typically infeasible to explore the entire state space relative to a given user event sequence. To this end, Jensen et al. propose to explore the state space only up to a given bound on the number of event reorderings, based on the hypothesis that most errors can be found with only a few event reorderings. A similar hypothesis has previously been used in the context of concurrency related errors [129]. In order to deal with the fact that the execution may change significantly when events are reordered, Jensen et al. propose an execution mode, approximate replay, where an event is skipped if it is not enabled. This way R$^4$ can make progress even when the execution differs from the expected event sequence, unlike WAVE.

From a usability point of view, every warning reported by R$^4$ clearly identifies the primary cause of the nondeterminism: Every warning consists of two schedules, where one can be derived from the other by only a single event reordering. In comparison, the test prioritization in WAVE has the undesirable side-effect that erroneous schedules may differ from the initial execution in many ways, making it difficult for developers to understand the problem.

Finally, Zhang and Wang [192] propose a technique called RClassify, which classifies race warnings from EventRacer as harmful, harmless, or false positives. Given a race, RClassify attempts to execute both orderings of the race. If RClassify is unable to enforce both orderings, then the race is classified as being a false positive. Otherwise, the two resulting program states are compared to determine if the race is harmful. Unlike WAVE and R$^4$, RClassify considers a race as harmful even when the internal JavaScript state differs. To this end, the authors developed a mechanism for developers to provide a set of fields that should not be considered in the comparison.
4.2 Race Detection for Mobile Applications

The problems caused by event races have also lead to an interest in the development of race detection techniques for mobile applications. Due to the complexity of the execution model in mobile applications, which also features multi-threaded concurrency programming, the existing techniques for event race detection in JavaScript web applications do not immediately apply. In particular, these techniques are incapable of establishing inter-thread happens-before relationships, which originate due to, e.g., thread creation. As a result, JavaScript race detection techniques would suffer from an overwhelming number of spurious warnings if they were applied directly to mobile applications. On the other hand, traditional data race detection techniques for multi-threaded concurrent programs implicitly treat operations that execute on the same thread as being ordered according to the happens-before relation, which is not true for event-driven applications. Thus, these techniques are unsuited as well.

Maiya et al. [118] and Hsiao et al. [81] present a happens-before relation for mobile applications, and use this to design a dynamic event race detector (DroidRacer and CAFA, respectively). The happens-before relations simultaneously reason about thread-local and inter-thread orderings, and thus generalize the happens-before relations known from single-threaded event race detection and multi-threaded data race detection. There are multiple practical challenges involved in tracking the happens-before relation in Android. For example, these tools are unable to infer happens-before orderings that result from native application code. Maiya et al. also report that it is complicated and inefficient to track inter-process communication. Instead, they propose to model the effects of the Android runtime environment, including the life-cycle of an application, by emitting so-called enable operations in the trace to signal that a given event has been enabled.

Similar to the experiences with race detection for JavaScript web applications, Hsiao et al. find that there are an overwhelming number of harmless low-level races in Android applications. In order to focus on races that are more likely to be harmful, they choose to focus on use-after-free races, where a read operation is unordered with a write operation that sets the same memory location to null. In order to deal with ad-hoc synchronization, and the possibility of a memory location being reassigned inbetween the two accesses in a use-after-free race, the authors propose the use of ad-hoc filters.

Bielik et al. [31] present a dynamic race detector, named EventRacer for Android, which improves over DroidRacer and CAFA in terms of scalability, infers more happens-before relationships, and is capable of detecting races between the application and framework code. Bielik et al. design an algorithm that can build the happens-before relation in quadratic time. Using this algorithm, EventRacer can analyze the trace that results from a minute of end-user interactions in less than 30 seconds. In comparison, DroidRacer and CAFA take hours to complete, since the algorithms they use to build and
query the happens-before relation are too simplistic. For example, DROID-RACER computes the transitive closure of the individual happens-before edges that follows from its trace analysis, which has a cubic time complexity. To combat the complexity of the Android system, Bielik et al. also propose to use a set of speculative happens-before rules. These rules are based on empirical observations of how the Android system behaves.

4.3 Practical Race Detection for JavaScript Web Applications

Despite all the work on race detection for mobile and web applications, existing techniques still suffer from limitations that make them unsuitable for use in practice. In the context of JavaScript web applications, EventRacer and R4 are currently considered as state-of-the-art. However, due to limitations of predictive analysis, EventRacer tends to report an overwhelming number of spurious warnings. R4 instead uses model checking, which is a heavyweight and expensive solution. Furthermore, both EventRacer and R4 use modified browsers, meaning that their prototype implementations are now far behind the latest browsers and therefore incapable of processing many modern web applications. These problems motivate the need for techniques that are able to detect event race errors in JavaScript web applications, but also practically useful. The remainder of this section presents our contributions to this research challenge.

There is no clear definition of what it means for a race detector to be practically useful, and it is likely also subjective. Instead of trying to give a definition, we instead highlight a number of properties that we argue a practically useful race detector should have. First and foremost, it should be able to detect event race errors, which developers care about, in modern web applications, and with reasonable precision and speed. The warnings reported by the race detector should also be actionable. More specifically, they should contain information about the effects of a race, and support the developers in diagnosing the root cause of the problem, by enabling them to reproduce the problem. Finally, it is also important that the race detector is platform independent and maintainable in order for it to be adopted in practice. Therefore, it should (preferably) not require browser modifications, or heavyweight solutions such as model checking or static analysis, which are expensive to maintain as the browsers evolve (e.g., due to changes in the JavaScript language, or in the browser APIs).

**Practical initialization race detection** We take a pragmatic and practical approach to race detection and propose a dynamic race detector called Init-Racer ([12], Chapter 11), which is designed to detect initialization race errors with few spurious warnings. An initialization race in a JavaScript web
CHAPTER 4. RACE DETECTION

application is an event race that may only manifest during the loading of the web page, i.e., until the entire web page has been parsed by the browser, and all asynchronous actions that have started during the loading of the web page have finished (e.g., the loading of external resources, or timers that have been registered). Many of the harmful races that have been reported in previous work (e.g., [138, 146]) are initialization races. A common source of initialization race errors is when the user interacts with the web application before it has been fully loaded.

The design of InitRacer is inspired by the idea of executing an application in “adverse conditions” from THOR ([9], Section 3.3). One of our key observations is that there exist three kinds of initialization race errors that can be exposed by designing an adverse input for the web application, which leads to an execution of the web page that developers will not normally encounter during ordinary testing. Notably, InitRacer executes the web page on this adverse input and compares the result to that of a “good execution.” It is thereby able to (i) confirm the feasibility of the harmful interleavings, and (ii) report on the consequences of the individual races, unlike predictive race detectors such as EventRacer.

The three kinds of initialization race errors that InitRacer aims to detect are called access-before-definition errors, form-input-overwritten errors, and late-event-handler-registration errors. A access-before-definition error manifests when a variable or property of an object is read before it has been initialized. A late-event-handler-registration error manifests when an event arrives before a corresponding event handler has been registered (and there exists an execution where the event handler registration happens prior to the arrival of the same event), and a form-input-overwritten error manifests when a piece of JavaScript overwrites an input of the user during the loading of the web page (e.g., if a JavaScript instruction updates the value of an HTML form input element after the user has already started typing into the same input element). The reader is referred to Section 11.2 for examples.

InitRacer works in three phases. The first phase is called observation mode, and aims to detect late-event-handler-registration errors and form-input-overwritten errors. In observation mode, InitRacer observes an execution of the web page to generate a trace of operations, and then searches for matches with trace patterns that characterize late-event-handler-registration errors and form-input-overwritten errors. In this mode, InitRacer considers an adverse scenario where the user aggressively and eagerly triggers all possible events as soon as possible: whenever an HTML element gets declared, InitRacer triggers all the events that have just been enabled (e.g., click, keydown), to detect late-event-handler-registration errors, and for every HTML form input element, InitRacer changes the value of the input field as soon as the field becomes visible, to detect form-input-overwritten errors.

Developers are only going to care about event race errors that are likely to manifest in practice. Despite of this, all previous race detectors for JavaScript
web applications ignore the likelihood of the erroneous interleavings of a race. In this work, we propose to focus only on those races, where it is guaranteed that there is a potentially long delay (e.g., due to network communication) from the point in time where the user event, which is responsible for the race error, becomes enabled, until the race error can no longer manifest. This ensures that reported race errors will be likely to manifest when the network is slow.

The second phase, adverse mode, is dedicated to exposing the last kind of initialization race errors, namely access-before-definition errors. In our experience, these errors typically manifest when event handlers execute earlier than anticipated by the developers. To this end, InitRacer simulates an adverse scenario where the user triggers every event handler as soon as the event handler gets registered, looking for uncaught exceptions. Again, InitRacer records a trace from the resulting execution and searches for matches with a trace pattern that characterizes access-before-definition errors. The errors that get exposed from the adverse mode execution may require multiple, specific user events during the loading of the web page to manifest, making them unlikely in practice. To focus on errors that developers are more likely to care about, a separate phase called validation mode automatically attempts to reproduce each access-before-definition error by triggering only a single user event during the loading of the web page. Only errors that can be reproduced this way are presented to the developers of the web page.

Experiments with InitRacer on 100 popular web applications show that it is capable of detecting harmful, real-world initialization race errors, meanwhile reporting relatively few harmless races. Furthermore, InitRacer produces informative error descriptions that support further debugging. InitRacer is pragmatic in the sense that it is designed specifically to detect three kinds of initialization errors with relatively high precision. In comparison, EventRacer is general enough to detect any event race, but on the other hand reports many spurious warnings.

**Practical AJAX race detection**  
InitRacer by design only targets event races that manifest during the loading of the web page. However, many JavaScript web applications also suffer from harmful event races after the page has been fully loaded. In particular, there exists a large class of interactive web applications that dynamically update their content in response to user interactions, without imposing any page reloads, to provide a smooth user experience. A common way to implement such interactive features is to fetch the new content from a web service using AJAX (see Section 2.1) and update the UI when the corresponding AJAX response event arrives. This general pattern is prone to harmful AJAX event races, though, since the user may interact with the web page before the processing of the previous user event has finished. Notably, this scenario may lead to a situation where multiple AJAX
requests are active at the same time, and their corresponding AJAX response events may arrive in any order. As a result of such races, web applications may reach an inconsistent state, which can be frustrating to their end-users. In Chapter 12 we give examples of observable AJAX event races from real-world web applications.

We propose a light-weight, two-phased approach to detect observable AJAX event races in modern web applications. The purpose of the first phase is to detect pairs of user events that are potentially AJAX conflicting. Intuitively, two user events are potentially AJAX conflicting if their event handlers use AJAX to update content in the same part of the screen. To compute this set of pairs, our technique, AjaxRacer, monitors an execution of a given user event sequence, to record a trace for each of the user events. These traces are analyzed for potential AJAX conflicts, similar to the way predictive race detectors work (e.g., EventRacer [146]). A key difference is that AjaxRacer uses a model of memory accesses that only considers writes to the individual pixels on the screen, as opposed to low-level read and write operations of JavaScript variables and properties. This memory model is well-suited for detecting event races that have an observable effect on the screen.

In principle, AjaxRacer could output a race warning for each of the potentially AJAX conflicting pairs of user events that have been detected in phase 1. However, such an approach is unlikely to be adopted in practice, since it would suffer from spurious warnings, similar to EventRacer and other predictive race detectors. Instead, AjaxRacer plans a test for each pair \((u_1, u_2)\), which is expected to fail only if the corresponding AJAX event race has an observable effect on the screen. Each test loads the web page twice. In both cases, the test triggers the user event sequence \(u_1 \cdot u_2\), and takes a screenshot when the processing of the two events have finished. Building on ideas from THOR and INITRACER, we expose one of the executions to adverse conditions, by postponing all AJAX response events that have been derived from \(u_1\), until all events that have been derived from \(u_2\) have been processed. AjaxRacer uses a mechanism similar to the event controller from EVENTRACECOMMANDER [111, Section 5.3] to intercept and postpone the relevant AJAX response events. In the other execution, AjaxRacer simulates a scenario that has likely already been tested by the developers, by waiting for all of the derived events from \(u_1\) to finish before triggering \(u_2\). If the two resulting screenshots are not identical, the test fails, signaling that the corresponding AJAX event race has observable effects.

We evaluated AjaxRacer on 20 widely used web pages that use AJAX. Our results show that AjaxRacer is an effective technique for exposing observable AJAX event races, and that AJAX event race errors are a common problem in real-world web applications. From the 20 web pages, AjaxRacer found observable AJAX races in 15 of them, with only few false positives. Furthermore, the reports that were automatically generated by AjaxRacer enabled us to easily determine the effects of the individual AJAX races, despite
the fact that we did not have prior experience with the subject applications.
Chapter 5

Program Repair

Until now we have been looking at different ways to expose errors in the application under test, to address the problem of test generation. When an error is detected, developers need to debug their application to determine what is the root cause of the error, so that they can repair the application in a way that prevents the error from manifesting in future executions. This can be a highly nontrivial and error-prone task, which motivates the design of techniques that can be used to help developers in this process.

From a high-level point of view, the problem of designing techniques for program repair appears to be significantly more challenging than that of test generation. For example, it is clear that program repair requires some knowledge of the application, the root causes of errors and the intended behavior. In comparison, many test generation techniques are black box. Another concern is that some techniques might change the application in order to fix an error. This could in principle introduce new errors. For this reason, it is necessary to reason about all the possible execution paths that are affected by a change, when applying a fix.

Due to these challenges, it is interesting, and somewhat surprising, that there exist techniques for program repair, which have had reasonable success. In particular, there is a line of work in the literature on techniques that attempt to prevent concurrency related errors in multi-threaded programs. Unfortunately, these techniques are not immediately applicable to event-driven applications, because the nature of races in event-driven applications is different from that of classical data races in multi-threaded, shared-memory programs (Chapter 4). In this thesis, we focus on program repair techniques that prevent nondeterministic errors from manifesting in event-driven applications.

Outline This chapter is structured as follows. First, Section 5.1 gives a brief overview of some of the different techniques that have been proposed for preventing concurrency errors in multi-threaded, shared-memory programs. Then Section 5.2 motivates the research challenge of program repair in the
context of JavaScript web applications, and presents techniques that have been proposed in the literature to address this research challenge. Finally, Section 5.3 presents our contribution to this research challenge, EVENTRACECOMMANDER ([11], Chapter 10), which is a system for restricting the nondeterminism in JavaScript web applications according to a predefined policy.

5.1 Preventing Data Races in Concurrent, Shared-Memory Programs

A data race in concurrent, shared-memory programs is defined as two concurrent accesses to the same memory location, where at least one of the accesses is a write. It is often a nontrivial task to diagnose and fix data race errors, because such errors only manifest under subtle interleavings, and often require the use of synchronization, which may lead to, for example, deadlocks and liveness problems.

Ratanaworabhan et al. [143, 144] were among the first to propose a technique that tolerates data races. The notion of “tolerating” a race has been used in the literature to describe repair techniques that do not technically eliminate any races in the given program, but instead attempt to prevent some of the harmful interleavings by altering the runtime. Ratanaworabhan et al. design a technique called TOLERACE, which targets asymmetric races in C programs. An asymmetric race is a data race where the memory location corresponding to the race is associated with a lock, and only one of the two accesses is guarded by the lock. TOLERACE simulates a sort of mutual exclusion by taking thread-local copies of shared variables upon entering a critical section. This mechanism provides a way to tolerate asymmetric races where only one of the accesses is a write.

Concurrency errors are generally more amenable to automated repair than sequential errors, because concurrency errors can be prevented by merely restricting the nondeterminism of the overall system. In the extreme case, a concurrent program could effectively be restricted to a single-threaded one. Although this is clearly undesirable, it suffices to guarantee the absence of errors that manifest due to the fine-grained interleavings of threads. This high-level observation has lead to a wide range of techniques that focus on preventing data races by influencing the scheduler or introducing synchronization. Some of these techniques rely on warnings that have been reported by race detectors. Krena et al. [112] propose several mechanisms to prevent data races in Java programs, including the creation of critical sections to prevent atomicity violations, the use of signal and wait operations to prevent order violations, as well as ways to lower the probability of certain interleavings by inserting context switches and changing thread priorities. Weeratunge et al. [177] infer atomic regions from profiling of correct executions and then insert locks to avoid concurrency errors. EVENTHEALER by Tchamgoue et al. [165] also infers
5.2 Preventing Javascript Event Races

atomic regions, though from static analysis of binaries, and then enforces these regions using synchronization. LOOM by Wu et al. [180] enforce so-called execution filters that have been written by developers. Tchamgoue et al. focus on event-driven applications that have a main thread, where event handlers execute with higher priority than the main thread and preemption is possible. Jin et al. [94] focus on fully automated repair where developers do not have to be involved. Their technique, CFix, combines techniques for preventing atomicity violations (AFix [93]) and order violations (OFix [94]), and uses testing to explore the effect of different repair strategies. AFix enforces mutual exclusion by carefully inserting lock and unlock operations based on control flow analysis, in order to avoid introducing double-lock, double-unlock, and unlock-before-lock errors. OFix uses signal and wait operations to enforce so-called all-A-B and first-A-B orderings, where A and B refer to instructions that have been described statically using a call stack and a thread stack (i.e., a sequence of call stacks that leads to the creation of a thread). Liu and Zhang [109] and Khoshnood et al. [97] propose techniques that are based on constraint solving (Axis and CONCBUGASSIST, respectively). Axis inserts synchronization on the basis of constraint solving on Petri net models, whereas CONCBUGASSIST finds a minimal set of inter-thread ordering constraints that are needed to prevent a concurrency error, and inserts synchronization to enforce these orderings.

Unlike the previous approaches, Zhang et al. [193] take a proactive approach. They observe that, in many cases, it is possible to anticipate that the harmful order of a race is about to manifest during execution, and propose to tolerate the race by stalling the relevant thread until the harmful ordering is no longer possible. This enables their technique to circumvent races that are unknown to developers. Races that are unknown to the developers can also be prevented using reactive approaches, such as techniques that rely on rollback-recovery [140, 161, 171, 195]. These techniques generally work by taking checkpoints during the execution, and then rolling back the program in case of an error. Although this approach does not guarantee that the same error will not occur again, it is often successful since most concurrency related errors manifest only with a small probability. The main disadvantage of this approach is that it is significantly more heavyweight than other techniques.

5.2 Preventing Event Races in JavaScript Web Applications

The work on preventing data races in concurrent, shared-memory programs is not directly applicable to single-threaded, event-driven applications, such as JavaScript web applications. In particular, the concurrency errors that are being addressed by the techniques in the previous section generally involve two threads that concurrently access the same memory location. This scenario
Mutlu et al. [130] were among the first to propose a technique to avoid races in JavaScript web applications. They discuss the idea of “schedule shepherding,” where the execution is steered towards well-known, correct schedules, similar to the way some techniques prevent data races in concurrent, shared-memory programs (e.g., [177]). More specifically, Mutlu et al. discuss to control the order of AJAX events in order to prevent observable races, but do not present a technique that actually does this.

Wang et al. [176] propose a technique called Arrow that is based on static analysis and constraint solving. Arrow uses static analysis to extract a causal graph that encodes happens-before orderings between various events of the web page (such as the parsing of HTML elements, and the definition of global variables). Arrow also uses static analysis to determine the intended behavior of the developers, based on the hypothesis that developers’ intentions are reflected in the order of the source code. The intended behavior is specified as a set of definition-use pairs that are must-aliased according to the static analysis, to prevent accidentally repairing spurious races. Finally, Arrow leverages constraint solving to find ordering relationships that guarantee that the definitions of global variables execute prior to their uses, and then enforces these orderings by changing the order of HTML elements and adding ad-hoc synchronization. One of the main limitations of Arrow is the use of static analysis, which is notoriously difficult for real-world JavaScript code. The authors manually write models for third-party JavaScript libraries to alleviate this problem. This is an impractical solution, though, since it is a very tedious and error-prone task, especially due to the existence of a vast amount of libraries in many different versions. Furthermore, the focus on definition-use races means that Arrow is unable to repair many race errors, even with more precise static analysis.

5.3 Controlling the Nondeterminism in JavaScript Web Applications

Despite the fact that event race errors are prevalent in JavaScript web applications, there has not been much work on preventing event races in such applications. However, the lack of language-level synchronization primitives in JavaScript means that developers often need to resort to ad-hoc synchronization, which is error-prone. It is an interesting challenge to design techniques that can assist developers of single-threaded, event-driven applications in preventing event race errors.

We address this research challenge in our ICSE 2017 paper (ł11, Chapter 10), by presenting a technique that can be used to prevent harmful event races in JavaScript web applications. The technique, EventRaceCommander, dynamically restricts the nondeterminism of a given application
5.3. CONTROLLING THE NONDETERMINISM IN WEB APPS

according to a predefined repair policy. The technique is enabled by an event controller, which dynamically intercepts events, before any of the corresponding event handlers in the web application get to execute, and then decides whether to forward each event to the web application (possibly causing one or more event handlers to execute), postpone it for later, or discard it entirely, based on the repair policy.

One of our key observations is that most event race errors manifest due to the violation of a few implicit assumptions that developers tend to make about the feasible schedules. In the evaluation of InitRacer ([12], Section 4.3) we find that developers often do not anticipate situations where the user starts interacting with the application before it has been fully loaded. It is also common that developers forget to test for scenarios where the user interacts with the application while an AJAX response event is pending, or where AJAX response events do not arrive in first-in-first-out order.

We use that observation to define a catalog of five application-independent repair policies, which can be used to enforce these implicit assumptions, thereby providing a method to prevent most event race errors. In addition, we show that the dynamic race detector EventRacer can be extended in a way such that every reported race is augmented with all the applicable repair policies from our catalog (i.e., the policies that can be used to prevent the given race). This information is also useful from a debugging point of view, since it points out the root cause of the underlying race.

In cases where the application-independent repair policies are too intrusive, and therefore damage the user experience on the repaired website, it is possible to define application-specific repair policies that do not expose the same problems. The use of application-specific repair policies can improve over application-independent ones along two dimensions. First, it is possible to discard or postpone only those events that are involved in a harmful race. Second, it is possible to reduce the duration for which these events are postponed or discarded. In our experience, it is typically straightforward to design an application-specific repair policy that prevents a given race error, once the root cause of the error is known.

Unlike the technique for automated repair of event races by Wang et al., EventRaceCommander is not limited to definition-use races. EventRaceCommander also has sufficient control over the nondeterminism to prevent races whose root causes are unrelated to the order of AJAX events, unlike the technique discussed by Mutlu et al. To this end, it is worth emphasizing that initialization race errors generally cannot be prevented by merely controlling the order in which AJAX events arrive, since most initialization race errors manifest due to user events that arrive before the web application is ready to process them.

Experiments with EventRaceCommander on 20 popular web applications show that most event races can be prevented using our catalog of application-independent repair policies, and that the instrumentation only
has little effect on the loading time of the considered web applications. From a total of 117 event races in our study, 94 become infeasible. Furthermore, we find that it is possible to design application-specific repair policies that do not damage the user experience, when application-independent ones yield suboptimal results.
Chapter 6

Test Adequacy

A challenging problem that developers face when testing their application is whether a current set of tests is adequate for them to have confidence in the correctness of the application, or additional tests need to be added.

Researchers and practitioners have addressed this problem by the development of a wide range of test adequacy criteria that can be used to determine if a test suite adequately exercises the application under test. Many test adequacy criteria make this determination by comparing information that has been recorded from an execution of the test suite (e.g., the coverage), with preestablished requirements that are based on the structure of the program (e.g., that all statements should be covered).

Test adequacy criteria can, in principle, be used as a stopping criterion for testing, i.e., a way for developers to determine when it is no longer necessary to add new tests. However, most test adequacy criteria are not finitely applicable \cite{198}, which means that the requirements established by these criteria cannot, in general, be satisfied by any finite test suite. For example, it may be impossible to meet a given coverage requirement in the presence of dead code. Furthermore, even when the requirements of a test adequacy criterion can be satisfied, it is typically overly expensive in terms of manual effort. For these reasons, test adequacy criteria are rarely used as stopping criteria. Instead, developers are typically presented with a score that indicates the adequacy of their test suite, in terms of a percentage, along with insights into how this score can be improved. This information can be used to guide further testing, and notably, to avoid the addition of redundant tests.

Dijkstra famously stated in 1970: “Program testing can be used to show the presence of bugs, but never to show their absence” \cite{45}. This is coherent with the fact that most test adequacy criteria do not give any guarantees about the absence of errors, even when they are fully satisfied, although the criteria have originally been designed to help developers evaluate if a test suite is sufficiently thorough. Dijkstra’s statement is true in general, but at the same time, it is clearly possible for tests to demonstrate the absence of errors, as
also noted by Goodenough and Gerhart in 1975 [64]. For example, it is trivial that a property of an application can be guaranteed to hold by a test suite that exercises all feasible execution paths of the application (when internal branching from the primitives of the underlying programming language are also taken into account). There is an interesting line of work that addresses the challenge of generalizing observations from a finite set of concrete executions to other unseen executions (e.g., [17] [153] [157]).

Applications that have been implemented using dynamically typed programming languages may suffer from runtime type errors, due to the lack of a static type checker that can guarantee the absence of such errors. As discussed in Section 2.2, two common kinds of errors are message-not-understood errors, which occur if lookup operations attempt to access non-existent fields or methods, and subtype-violation errors, which manifest at runtime when unsafe cast operations fail. Notably, some of the work that generalizes observations from concrete executions to other unseen executions has been done in the context of dynamically typed programming languages (in particular, [17] [153]).

Outline This chapter is structured as follows. Section 6.1 gives an introduction to popular test adequacy criteria and discusses their strengths and weaknesses. Test adequacy criteria generally fail to give any guarantees, even when they have been fully satisfied. Section 6.2 focuses on techniques that infer properties that are guaranteed to hold from concrete executions. Finally, Section 6.3 summarizes our contributions from our ISSTA 2016 paper, which presents a technique for inferring type-related completeness facts from concrete executions (Goodenough [10]).

6.1 Test Adequacy Criteria

A test adequacy criterion is a procedure that given a program and its test suite outputs a percentage, which indicates how thorough the test suite is [198]. Such a criterion is said to be satisfied when all of the requirements that have been established by the criterion are met by the test suite, i.e., in which case the output of the criterion is 100%. Some test adequacy criteria also establish requirements based on a specification of the program—in this thesis, however, we only consider criteria that establish coverage requirements based on the program code itself.

A simple, yet very popular, test adequacy criterion is statement coverage [198], which compares the number of statements covered by a test suite to the total number of statements in the application under test. In addition to its simplicity, the success of statement coverage can likely be explained by the following observations: (i) it is usually valuable to cover at least every statement of the application under test, and (ii) statement coverage reports are actionable, in the sense that it is easy for developers to understand that
a statement has not been covered, and to design an input that covers the
given statement (at least in most cases, with application-specific knowledge).
The criterion is not finitely applicable, since the application under test may
contain statements that are unreachable. This problem can be alleviated by
considering only the reachable statements of the application. However, this
is mostly of theoretical interest. It is well-known that it is undecidable in
general whether a statement is reachable, and therefore it is also undecidable
(in general) to determine if a test suite satisfies this variant of the statement
coverage criterion.

Another popular criterion is the branch coverage criterion \cite{198}, which
measures the percentage of edges in the control flow graph that have been
covered. This criterion naturally subsumes statement coverage\footnote{A test adequacy criterion \( A \) is said to subsume another criterion \( B \) if a test suite that satisfies \( A \) automatically also satisfies \( B \).}, since full
branch coverage implies that all nodes in the control flow graph have been
covered. Path coverage is an even stronger test adequacy criterion (i.e., path
coverage subsumes branch coverage), which requires that all paths in the
program have been covered. The path coverage criterion \cite{198} is not well-
suited for use in practice, however, since most applications have an infinite
number of paths (thus, the criterion is clearly not finitely applicable).

Unlike the test adequacy criteria that have been described above, some
criteria that are based on the possible data flows in the program, have also
been proposed in the literature. For example, Frankl and Weyuker \cite{59} define
the all-definitions and all-uses criteria. The all-definitions criterion measures
the percentage of variable definition sites (that have a corresponding, reachable
use site) for which there exists an execution path that propagates a definition
to one of its corresponding use sites. The all-uses criterion requires that
every variable definition reaches every corresponding, reachable use site, and
thus subsumes the all-definitions criterion. These criteria are well-suited for
identifying situations where a variable is defined incorrectly.

Mutation testing is a methodology, which was proposed by DeMillo et al. \cite{44},
that can be used to measure the adequacy of a test suite. Instead of using
coverage directly as a proxy for test adequacy, mutation testing focuses on the
ability of a test suite to identify artificial errors that have been injected into the
program. The main idea in mutation testing is to evaluate if a test suite is able
to distinguish the program under test from a set of automatically generated
programs, called mutants, which differ slightly from the original program. A
large set of mutants are typically generated by applying a predefined catalog of
mutation operators to the original program (as a simple example, a mutation
operator can change a greater-than-or-equal operator in the program into a
less-than operator). This tends to make mutation testing computationally
expensive. If the test suite fails on one of the mutant programs, the mutant is
said to be killed. Mutants that survive the execution of a test suite motivate the
addition of new tests that are capable of killing the given mutants. The mutant adequacy criterion measures the percentage of mutants that are killed by a test suite. A well-known problem with this criterion is that generated mutants may be semantically equivalent to the original program, meaning they cannot be killed, or other mutants, which can lead to biases in the ability of a test suite to kill a set of mutant programs. Therefore, it is desirable to consider only non-equivalent mutants, which has lead to literature on eliminating equivalent mutants (e.g., [27, 77, 135, 136, 154, 155]).

In the context of JavaScript web applications and Android mobile applications, developers are often concerned about testing the individual screens of their application. In Android applications, covering more screens roughly means covering more activities (recall from Section 2.1 that every screen in an Android application is represented by a component called an activity). This motivates the activity coverage criterion, which measures the percentage of activities (as declared in the AndroidManifest.xml file) that have been covered by the tests. Note that this criterion is subsumed by the statement coverage criterion. The activity coverage criterion has also been used to evaluate the effectiveness of several automated UI testing techniques for Android (e.g., [25, 191, 196]). In a recent study of an industrial-strength application, Zheng et al. [196] report that their automated testing technique achieves approximately 32% activity coverage. Surprisingly, the authors find that as many as 40% of the remaining, uncovered activities are dead, due to the existence of old activities that are no longer being used, and new yet-to-be-released activities. This suggests that it is not necessarily reasonable to expect that most of the activities that have been declared in AndroidManifest.xml can be covered.

A single activity in an Android application can support many different use cases, in the sense that the user interactions supported by that activity do not necessarily cause the current activity to change. These user interactions do not necessarily contribute to higher activity coverage. For this reason, it may be possible to achieve high activity coverage with just a shallow exploration of the application under test. To carry out more thorough testing, some model-based testing techniques (e.g., [26, 76, 188]) therefore use a more fine-grained abstraction of the current state, which—in addition to the name of the current activity—accounts for the view hierarchy (i.e., an XML representation of the UI) and the set of enabled events.

6.2 Test Completeness

A test suite can be complete with respect to a program property in the sense that the test suite suffices to guarantee that the property holds in any execution. This was already observed by Goodenough and Gerhart [64] back in 1975. They gave a simple proof that a test suite, which fully satisfies a reliable and valid test adequacy criterion, suffices to guarantee the absence of errors. In
their terminology, a test adequacy criterion is \textit{reliable} if all test suites satisfying the criterion lead to the the same test result, and \textit{valid} if every erroneous input is detected by a test suite that satisfies the criterion. These requirements are mostly interesting from a theoretical point of view, though.

Most of the test adequacy criteria that have been discussed in the previous section only give weak guarantees when the criteria are satisfied. As a simple example, consider a test suite that satisfies the statement coverage criterion for a dynamically typed program. Such a test suite is insufficient to guarantee the absence of type errors, since there may still exist an execution where a field or method lookup operation fails. If the program is a JavaScript application, though, the test suite suffices to guarantee the absence of syntax errors.

The path coverage criterion can, in principle, give stronger guarantees. If all feasible paths have been executed and no errors have been detected, then this guarantees the absence of errors. This only holds, though, when internal branching from the primitives of the underlying programming language (e.g., due to implicit safety checks such as a divide-by-zero check) have been accounted for. The path coverage criterion can only be satisfied if the set of feasible execution paths is finite. Furthermore, to determine if a test suite satisfies the criterion, it is necessary to reason about the feasibility of execution paths, which requires reasoning about, for example, correlated branches.

King \cite{King1999} presented symbolic execution and its relation to program verification. As noted by King, there is a direct correspondence between exhaustive symbolic execution and the act of giving a proof of correctness: If the program has been augmented with the properties of interest, then theorem proving can be used to verify them under the different path conditions. However, exhaustive symbolic execution is only possible when the symbolic execution tree is finite, and hinges on the capabilities of a theorem prover.

Yorsh et al. \cite{Yorsh2018} show that it is possible to reach the result of abstract interpretation by combining testing, abstraction, and theorem proving. Given an abstraction function that maps a set of concrete states to a set of abstract states, Yorsh et al. define an abstraction-based adequacy criterion, which is satisfied when the abstract states that have been covered by the test suite cover all reachable abstract states. The covered abstract states are obtained by applying the abstraction function to the set of concrete states that have been covered by the test suite. Their technique repeatedly fabricates new concrete states (which are not necessarily feasible) until the adequacy criterion is satisfied, using a theorem prover that produces counterexamples. When the adequacy criterion is satisfied it is possible to verify the absence of runtime errors based on the abstract states, as in abstract interpretation \cite{Wong1993}.

A recent and promising approach to obtaining guarantees from concrete executions is the idea of sandbox mining by Jamrozik et al. \cite{Jamrozik2018}. Sandbox mining is a way to address the problem that ordinary testing cannot give any guarantees about unseen executions. One of the key insights is the concept of \textit{test complement exclusion} \cite{Yorsh2018}: by excluding the use of API resources that
have not been accessed during testing (using a sandbox), the incompleteness of testing is turned into a guarantee, in the sense that no unseen behavior will ever happen. Thus, in this context, the concrete executions are used for mining the rules that should be enforced by the sandbox. The ability of a sandbox to prevent malicious behavior is heavily dependent on the design of the sandbox, and in particular, the kind of rules that it enforces. In their implementation for Android, Jamrozik et al. consider calls to sensitive APIs that are governed by a permission. To prevent the sandbox rules from becoming too course-grained, the authors qualify the API usages by information about the widget that lead to a given API usage.

This thesis is concerned with test completeness for dynamically typed programming languages, and the ability of a test suite to prove the absence of runtime type errors. Sandbox mining is not immediately applicable in this context. As a simple example, consider the problem of guaranteeing that a field lookup operation \( x.f \) will never fail with a message-not-understood error. Using the idea of sandbox mining, one option would be to mine the types of the expression \( x \) (i.e., record the types that \( x \) evaluates to during test suite execution), and then design a sandbox that disallows \( x \) from having a type that was not observed during testing. However, if the sandbox is violated at runtime, there is no way for the user to resolve the problem (as opposed to use cases where the sandbox intercepts accesses to sensitive APIs, and the user can choose to grant the permission to a given API). Instead, the sandbox can stop the program execution, but this is likely as bad, and potentially even worse, than the type error itself.

In the context of dynamically typed programming languages, An et al. [17] propose a dynamic type inference for Ruby, which is based on constraint solving. Their implementation, called Rubydust, instruments the program in a way that causes subtyping constraints to be generated when values are assigned to variables, returned from a function, etc. An et al. prove that the inferred types are sound when the set of concrete executions cover all possible paths in every method. These guarantees are mostly of theoretical interest, though, since the criterion suffers from the same problems as the path coverage criterion.

Schäfer et al. [153] infer dynamic determinacy facts from a concrete execution, i.e., facts saying that a variable or a field at a given program point (possibly qualified by an execution context) has a so-called determinate value. As a trivial example, a variable which is assigned a constant is determinate in any context. In order to infer more determinacy facts, Schäfer et al. propose a technique called counterfactual execution, which accounts for effects of executions that have not been seen, by force-executing both branches of conditional statements. Dynamic determinacy facts can, in principle, be used to demonstrate the absence of type errors. In JavaScript, if a variable is determinate at an operation that reads a field from the variable, and the value is an object, then the field lookup operation is guaranteed not to throw a type error. However, many determinacy facts are qualified by a context, so it is
non-trivial to generalize such guarantees to all possible executions.

6.3 Analyzing Test Completeness for Dynamic Languages

As mentioned in Section 2.2, the use of dynamically typed programming languages introduces the risk of a class of runtime type errors, including message-not-understood errors and subtype-violation errors at implicit casts, which are not present in statically typed programs. As a consequence, developers also need to test for these kinds of errors. It is an interesting research challenge whether it is possible to design techniques that can leverage concrete executions from test suites to prove the absence of type errors in dynamically typed programs.

The remainder of this section summarizes our contributions to this research challenge. In our ISSTA 2016 paper ([10], Chapter 9), we present an analysis that conservatively approximates test completeness for type-related properties. We say that a test suite is type complete for a given expression, if the test suite is complete with respect to the type of the expression (i.e., the test suite covers all the possible types that the expression may have in any execution). Type completeness can be used to prove the absence of type errors, such as message-not-understood errors and subtype-violation errors. For example, if a test suite is type complete for the receiver in a field lookup operation, and all the types that have been observed for the receiver have the corresponding field, then the field lookup operation is guaranteed to succeed in any execution.

It is straightforward to design a simple technique that can approximate type completeness. As a simple example, consider an expression \( x \), and say that the only possible types that \( x \) may evaluate to is \( A \) or \( B \), according to a sound static type analysis. It is trivial that a test suite is type complete for the expression \( x \) if the test suite covers all of the possible types that \( x \) may have according to the static type analysis (i.e., \( A \) and \( B \)). Although this is a quite naive approach, it is still interesting that a test suite can be type complete from a testing perspective. In particular, if a test suite is type complete for the receiver of a method invocation, then all of the outgoing edges in the call graph at the method invocation have been covered by the test suite. From a static analysis perspective, the fact that an expression can be proven type complete demonstrates that there are no spurious types in the set of types computed by the static analysis. However, the strategy is rather pointless for proving the absence of errors, since the type completeness facts cannot be used to infer more precise types than the static type analysis. Thus, the strategy is of no help for code that is challenging to analyze statically, for example, due to the use of overloading. A main challenge is to be able to establish type completeness for expressions where the test suite only covers a subset of the types that have been inferred by a static type analysis.
CHAPTER 6. TEST ADEQUACY

One of our key insights is that, although a test suite may not cover all the possible types of an expression according to a static type analysis, it may cover everything that the type of the expression depends on. To this end, we define a static analysis that over-approximates the dependencies of every expression. The purpose of the dependence analysis is to make it possible to infer more type completeness facts, to guarantee the absence of type errors. We use this knowledge in the design of the dependence analysis by computing also the type dependence of every expression, in addition to the value dependence, unlike more traditional dependence analyses used for optimization and program slicing. As a simple example, our analysis may conclude that the return type of a function $f$ depends (only) on the type of one of the parameters of $f$, say $p$. In such cases, it is possible to prove type completeness for call expressions that invoke $f$ by proving type completeness for the arguments that are passed to the parameter $p$, using the principle of substitution. Chapter 9 explains the analysis in more detail, and also contains examples that illustrate situations where the substitution principle is useful.

A completeness analysis is responsible for inferring type completeness facts from the executions that have been obtained by running the test suite. This analysis uses a small set of proof rules that compares the coverage of the concrete executions with the results of the static type analysis and the static dependence analysis. The resulting type completeness facts can be used directly to prove the absence of type errors, as well as to improve the precision of the static type analysis. For example, if an expression only evaluates to the types $A$ and $B$ during the execution of a test suite, and the expression can be proven to be type complete, then the static type analysis can discard all values that are not of type $A$ or $B$ using a type filter [102].

We have implemented our hybrid static/dynamic analysis framework in a tool called GOODENOUGH that works for the Dart programming language. Experiments on a set of command-line and web applications demonstrate that GOODENOUGH is able to prove type completeness facts from concrete executions. This is not too surprising, though, since it is not unusual that expressions have a fixed type. More importantly, our experiments show that it is possible to improve the precision of a baseline static type analysis on the majority of benchmarks, by using the type completeness facts that have been inferred by GOODENOUGH for type filtering. Notably, GOODENOUGH improves the ability of the static type analysis to prove the absence of message-not-understood errors and subtype-violation errors.

Unlike the RUBYDUST technique [17], GOODENOUGH is capable of giving guarantees with only limited coverage. One of the fundamental differences is that GOODENOUGH makes it possible to prove a type-related property by considering only what is required to prove that property. In comparison, RUBYDUST requires full path coverage of all methods, but then guarantees that all the inferred types are sound. In dynamic determinacy analysis [153], most dynamic determinacy facts are qualified by a call context, in which the
facts hold. In comparison, the completeness facts inferred by Goodenough hold in any execution.
Chapter 7

Conclusions

This thesis addresses three main challenges that developers face when testing their programs. Test generation is the challenge of designing a thorough set of inputs to the application under test, as well as oracles that specify intended behavior. Program repair is the challenge of updating the application appropriately in case of test failures. Finally, test adequacy is the challenge of determining when a sufficient amount of testing has been carried out. The goal of this thesis is to make contributions to each of these challenges in the context of event-driven and dynamically typed software applications. From an analysis of the strengths and weaknesses of existing solutions, we have identified opportunities for contributing to the state-of-the-art for test generation, program repair, and test adequacy, and proposed new techniques that explore these opportunities.

In summary, this thesis contributes with three results that address the challenge of test generation for Android mobile applications and JavaScript web applications:

• A methodology for extending the error detection capabilities of manually written end-to-end test cases, along with an implementation for Android, called Thor ([9], Chapter 8). Thor systematically exposes end-to-end test cases to neutral event sequences. In case of a test failure, Thor automatically isolates a minimal configuration, consisting of a single injection point and a single neutral event sequence, that is responsible for the failure. By using neutral event sequences, Thor is able to leverage the application-specific assertions that have been written by developers, by reusing them directly, unlike most other test amplification techniques. Experiments with Thor show that our methodology is an effective way to increase the error detection capabilities of existing test suites.

• A lightweight technique for detecting initialization race errors in JavaScript web applications ([12], Chapter 11). Our implementation, InitRacer, is a pragmatic race detector in the sense that it is designed specifically
to detect common kinds of initialization errors, with relatively high precision (as opposed to other techniques, such as EventRacer [116], which is general enough to detect any event race, but has many spurious warnings). The design of InitRacer builds on ideas from Thor, which exposes Android applications to adverse conditions. In InitRacer, the adverse conditions are comprised of events that occur unexpectedly early during loading. In particular, InitRacer works by injecting events aggressively and eagerly into the execution to create an adversarial scenario. Experiments with InitRacer on 100 popular web applications show that it is capable of detecting harmful, real-world initialization race errors, while reporting relatively few harmless races. Furthermore, InitRacer produces informative error descriptions that support further debugging.

- A practical race detector for JavaScript web applications, called AjaxRacer, which uses a two-phased approach to detect observable AJAX races ([13], Chapter 12). In the first phase, AjaxRacer analyzes an execution of a given user event sequence to identify pairs of user events that are potentially AJAX conflicting. Building on ideas from InitRacer and Thor, AjaxRacer then creates an adverse execution for each of these pairs, in which all AJAX events that are derived from the first user event have been postponed. This execution is created using a mechanism for controlling the execution, similar to the event controller in EventRaceCommander, and is subsequently compared to a “good” execution where the event handlers of the two user events do not interleave. An evaluation on a set of real-world web applications demonstrates that AjaxRacer is an effective technique for detecting observable AJAX race errors, with only few false positives.

This thesis also contributes with the following result to the challenge of program repair for JavaScript web applications.

- A technique that can be used to steer away from erroneous schedules in JavaScript web applications, by restricting the nondeterminism according to a given repair policy ([11], Chapter 10). Based on a study of event race errors in real-world web applications, we observe that many errors manifest for similar reasons and can be prevented by a small set of application-independent repair policies. The policies generally prevent schedules that are unlikely to show up during ordinary testing. Experiments with our implementation, EventRaceCommander, show that application-independent policies can be used to prevent most event race errors, and that it is easy to design application-specific policies when the application-independent ones damage the user experience.

Finally, this thesis makes the following contribution to the test adequacy challenge, in the context of dynamically typed programs.
• A notion of test completeness for dynamically typed programs, along with a hybrid static/dynamic analysis framework called Goodenough for conservatively approximating test completeness \cite{goodenough}. We identify that a set of concrete executions, provided by a test suite, can be complete with respect to the type of an expression, and that type completeness facts can be used to guarantee the absence of type-related errors, such as message-not-understood errors and subtype-violation errors. To this end, we propose a completeness analysis, which infers type completeness facts by comparing the coverage of the concrete executions with the results of a lightweight static type analysis and static dependence analysis. Experiments with our implementation for the Dart programming language demonstrate that our framework can be used to infer type completeness facts from a set of command-line and web applications, and that these facts can be used to improve the precision of a baseline static type analysis via type filtering. Similar to Thor, Goodenough leverages existing tests, but in a way that allows more precise static type analysis of the application under test, thereby refuting spurious type warnings.

In summary, these techniques improve over existing state-of-the-art solutions for test generation, program repair, and test adequacy. At the same time, it is important to note that these contributions do not solve all the challenges that have been raised in this thesis.

There is an active research community working on automated test generation and program repair for event-driven applications, especially for mobile and web applications. For example, the observation from our ISSTA 2015 paper \cite{zeng} that many Android applications appear to suffer from poor persistence has already lead to techniques that automatically attempt to detect such errors using static analysis \cite{zheng}. The same observation has also inspired work that automatically identifies and heals data loss problems when they occur at runtime \cite{choudhary}. Choudhary et al. \cite{choudhary} surprisingly came to the conclusion that the Android Monkey \cite{monkey} is superior to several of the automated testing techniques that have been presented in the literature. Zeng et al. \cite{zeng} and Zheng et al. \cite{zheng} later studied the effectiveness of random testing when applied to an industrial-strength application. Their observations lead the authors to highlight several limitations of random testing and to propose promising directions for future research. We emphasize that several of these limitations can in principle be addressed by leveraging existing, manually written test cases, as Thor \cite{thor}. For example, Zheng et al. observe that many activities can only be reached in certain states (e.g., if a feature has been enabled in the settings) and that other activities require long event sequences or text inputs that pass validation mechanisms. Manually written test cases often initialize the application in different states, which could be exploited by automated testing techniques. Furthermore, such test cases often contain long event
sequences and specific user inputs to navigate deep into the application under test. This thesis may help increase the focus on manually written test cases and inspire new techniques that leverage the application-specific knowledge that is embedded in manually written tests.

In the context of JavaScript web applications, we have advocated the need for more practical race detectors, and taken a step in that direction with the design of InitRacer [12] and AjaxRacer [13]. To get even further, we believe it is necessary to get more experience with how these tools work in practice, when they are applied to large-scale, industrial-strength applications, and to get feedback from developers. Presumably, many web application developers are not sufficiently aware of event race errors, hence another important task is also to increase the awareness of these errors.

Another interesting research challenge is to design techniques for detecting event race errors in Node.js applications. Madsen et al. [116] presented a static analysis that approximates a so-called event-based call graph, which can be used to identify certain event-related errors, such as dead event listeners. A recent study by Wang et al. [175] identifies challenges and opportunities involved in extending existing race detection techniques to Node.js applications. In the context of race detection for JavaScript web applications, a significant challenge is to classify event race errors as harmful or harmless, since they mostly tend to annoy the user. Mutlu et al. [131] argue that only races that have an impact on persistent storage should be considered harmful, since for all other races, the user can simply obtain the desired behavior by reloading the web page. In comparison, the challenge of classifying event race errors in Node.js applications as harmful or harmless is less likely to be a problem. For example, it is much more likely for an uncaught exception in a Node.js application to have severe consequences, since many applications are meant to be long-running, and because there is no user that can simply repeat his or her action, or reload the application entirely.

EventRaceCommander [11] can prevent event race errors by restricting the nondeterminism of a JavaScript web application. We envision that EventRaceCommander can be used as a quick fix for developers to prevent harmful event race errors when they are first detected, until the developers have a proper solution to the underlying problem. At the same time, we also acknowledge that EventRaceCommander is unlikely to be used on large industrial-strength applications, since the intrusiveness of the instrumentation and the (albeit small) overhead may be intolerable. In the evaluation of EventRaceCommander, we observed that most event race errors can be prevented using a small catalog of repair policies that prevent schedules that tend not to be anticipated by developers. This inspired the idea of trying to design an adverse schedule, which breaks the expectations of the developers, in InitRacer. In the future, EventRaceCommander could inspire other work that enforces schedules that are likely to be erroneous, with the purpose of detecting errors.

Our hybrid static/dynamic analysis framework, Goodenough [10], can
increase the awareness about the idea of test completeness, and may inspire the design of new analyses that obtain guarantees from concrete executions, for example by generalizing GOODENOUGH to other properties that are not necessarily type-related. In relation to the challenge of guaranteeing the absence of runtime type errors in dynamically typed programming languages, it is an interesting research challenge to extend GOODENOUGH to other languages than Dart, such as JavaScript and TypeScript. GOODENOUGH is not immediately applicable to these languages, since its ability to infer type completeness facts relies on the precision of a static type analysis and static dependence analysis. It is still an open research challenge to design sound and precise static analysis for JavaScript, and therefore, it is probably necessary to sacrifice the soundness of the completeness analysis in order to make the analysis framework practically useful for JavaScript and TypeScript applications.
Part II

Publications
Chapter 8

Systematic Execution of Android Test Suites in Adverse Conditions


Abstract

Event-driven applications, such as, mobile apps, are difficult to test thoroughly. The application programmers often put significant effort into writing end-to-end test suites. Even though such tests often have high coverage of the source code, we find that they often focus on the expected behavior, not on occurrences of unusual events. On the other hand, automated testing tools may be capable of exploring the state space more systematically, but this is mostly without knowledge of the intended behavior of the individual applications. As a consequence, many programming errors remain unnoticed until they are encountered by the users.

We propose a new methodology for testing by leveraging existing test suites such that each test case is systematically exposed to adverse conditions where certain unexpected events may interfere with the execution. In this way, we explore the interesting execution paths and take advantage of the assertions in the manually written test suite, while ensuring that the injected events do not affect the expected outcome. The main challenge that we address is how to accomplish this systematically and efficiently.

We have evaluated the approach by implementing a tool, Thor, working on Android. The results on four real-world apps with existing test suites demonstrate that apps are often fragile with respect to certain unexpected events and that our methodology effectively increases the testing quality: Of
507 individual tests, 429 fail when exposed to adverse conditions, which reveals
66 distinct problems that are not detected by ordinary execution of the tests.

8.1 Introduction

As of May 2015 more than 1.5 million Android apps have been published in
the Google Play Store. Execution of such apps is driven by events, such as,
user events caused by physical interaction with the device. One of the primary
techniques developers apply for detecting programming errors is to create end-
to-end test suites (also called UI tests) that explore the UI programmatically,
mimicking user behavior while checking for problems. Testing frameworks,
such as, Robotium, Calabash and Espresso are highly popular among
Android app developers. As a significant amount of the software development
time is often devoted to testing, it is not unusual that test suites have
high coverage of the source code and incorporate a deep knowledge of the app
UI and logic. Furthermore, the result of each single test can be of critical
importance to sanction the success of the entire development process, as tests
may be used for verifying scenarios in the business requirements.

Nevertheless, due to the event-driven model, only a tiny fraction of the
possible inputs is typically explored by such test suites. As the test cases are
written manually, they tend to concentrate on the expected event sequences,
not on the unusual ones that may occur in real use environments. In other
words, although the purpose of writing test suites is to detect errors, the tests
are traditionally run in “good weather” conditions where no surprises occur.

Our goal is to improve testing of apps also under adverse conditions. Such
conditions may arise from events that can occur at any time, comprising
notifications from the operating system due to sensor status changes (e.g.
GPS location change), operating system interference (e.g. low memory), or
interference by another app that concurrently accesses the same resource (e.g.
audio). It is well known that Android apps can be difficult to program when
such events may occur at any time and change the app state. A typical example of bad behavior is that the value of a form field is lost when
the screen is rotated.

As a supplement or alternative to manually written test suites, many
automated testing techniques have been created aiming to find bugs with little
or no help from the developer. The primary advantage of such techniques is that they can, in principle,
explore the state space more extensively, including the unusual event sequences. However, these techniques generally cannot provide as high coverage as manual
techniques. Moreover, automated techniques mostly operate without any knowledge of the expected behavior of the app, so they typically test only generic correctness properties (such as, the app should not crash with a null dereference exception) and fail to notice more subtle functionality errors.

In contrast to those approaches, we wish to take advantage of existing, manually written end-to-end test suites that are already widely used by app developers. We present an algorithm that systematically injects special events in existing tests to check the robustness in adverse conditions. We thereby leverage the application-specific knowledge and amplify the tests [183, 194].

As observed by Zaeem et al. [188], certain events in mobile apps are associated with a common sense expectation of how the app should respond. For example, suspending and then resuming an app should typically be allowed anytime without affecting the behavior of the app. We use a similar idea to select which events to inject. Zaeem et al. exploit their observation in a model-based testing technique. A limitation of that approach is that it requires a UI model of the app under test and a suitable abstraction of the execution states to determine whether the events cause substantial changes.

By leveraging existing test suites, we avoid both problems.

To our knowledge, no previous work has exploited existing tests of mobile apps to get assurance about the app behavior when executed in such adverse conditions. Fard et al. [51] combine existing tests and crawling for web applications, but in a way that requires heuristic regeneration of assertions, whereas we use the existing test assertions unmodified and focus on injecting events that are typically not mentioned in the tests. More fundamentally, our aim is to obtain a systematic exploration of the possible consequences of injecting such events in each test case.

Although the basic idea in our approach is simple, making it work in practice involves several design challenges. Which events are relevant to inject, and when should they be injected in each test case? Ideally, the technique should (1) increase the ability to detect bugs as much as possible, (2) run without a significant slowdown compared to ordinary test suite execution, and (3) provide error messages that precisely indicate the cause of each error being detected, to aid debugging. The first step of our approach is to establish a notion of neutral event sequences that may be tailored to individual apps or test cases, with sensible defaults for the Android platform. We then present an algorithm for injecting neutral event sequences, supplemented by strategies for isolating the causes of failures and reducing redundancy.

By generalizing from existing tests, the classification of the problems being detected as either bugs or false positives is naturally subjective. Indeed, in some cases the developer might think that the problem is not important enough to be fixed, for example, if a dialog window disappears when the phone is rotated. To this end, our approach can help the developer by revealing implicit assumptions of test cases that concern the special events.
In summary, our contributions are:

- A methodology for leveraging existing tests to detect bugs that involve unexpected events (Section 8.3). The methodology relies on the insight that existing tests can be run in adverse conditions to increase the ability to detect bugs in the apps and identify hidden assumptions of the tests.

- An implementation, Thor, designed for Android apps with Espresso or Robotium test suites (Section 8.4), which includes a selection of neutral event sequences.

- An experimental evaluation (Section 8.5). We show using 4 real Android apps with existing test suites that our methodology is able to detect bugs and identify hidden assumptions that are not exposed by ordinary test executions. In particular, our technique causes 429 otherwise succeeding tests to fail in adverse conditions out of a total of 507 tests. From those failing tests we have manually identified 66 distinct problems. We estimate that 22 of the 66 problems are critical bugs from the user perspective, where the remaining ones are likely unintended by the app developers but not harmful to the overall functionality of the apps. Among the 22 critical bugs, 18 affect the functional behavior of the app without causing it to crash, thereby demonstrating the advantage of exploiting the application-specific knowledge available in the test suites. Our experiments also show the effectiveness of the failure isolation and redundancy reduction strategies.

8.2 Motivating Example

This section explains our methodology using a concrete, motivating example.

An Android app is structured in various screens, each called an activity and representing a focused component for user interaction. Consider the code in Figure 8.1 showing a snippet of code from an activity in Pocket Code[5] for Android, an educational app for teaching visual programming.

The ProjectsListFragment allows users to manage their projects in a list. A Fragment is a piece of an app’s UI that can be placed inside an Activity. When the activity is put into the foreground, the onResume method (line 2) is called on the fragment, which loads the projects from the disk and creates a ProjectAdapter for them (lines 6-7). This adapter holds the list of projects and provides a UI element for each entry through the getView method that is shown by the activity to the user (see Figure 8.2a).

In order to delete a project, the user should long press the entry associated with it, triggering a call to onCreateContextMenu (line 9). This method adds the selected project to the checked projects of the adapter (line 10) and displays
8.2. MOTIVATING EXAMPLE

```java
class ProjectsListFragment extends ... {
  void onResume() {
    initAdapter();
  }
  void initAdapter() {
    projects = loadListFromDisk();
    adapter = new ProjectAdapter(projects);
  }
  void onCreateContextMenu(MenuInfo info) {
    adapter.addCheckedProject((...)info.pos);
  }
  void showConfirmDeleteDialog() {
    list = new OnClickListener() {
      void onClick(...) {
        deleteCheckedProjects();
      }
    }
    ....setPositiveButton(yes , list);
  }
  void deleteCheckedProjects() {
    for (int pos:adapter.checkedProjects()) {
      deleteProject((ProjectData)
        getListView().getItemAtPos(pos));
    }
  }
  class ProjectAdapter extends ... {
    Set<Integer> checkedProjects = new ...
    View getView(final int pos , ...) {
      findViewById(PROJECT_CHECKBOX)
        .setOnCheckedChangeListener(
          new OnCheckedChangeListener() {
            void onCheckedChanged(boolean checked) {
              if (checked) {
                checkedProjects.add(pos);
              } else {
                checkedProjects.remove(pos);
              }
            }
          }
      );
      ...
    }
  }
  }
```

Figure 8.1: Snippet from the Pocket Code app.
CHAPTER 8. TESTING IN ADVERSE CONDITIONS

Figure 8.2: Snapshots from Pocket Code during the deletion of a project (a-c), and a home button long-press to show the open apps (d).

The above use case is taken into account by the test `testDeleteCurrentProject` in Figure 8.3, taken from the original app repository. The test is
8.2. MOTIVATING EXAMPLE

```java
public void testDeleteCurrentProject() {
  createProjects();
  clickOnButton("Programs");
  longClickOnTextInList(DEFAULT_PROJECT);
  clickOnText("Delete");
  clickOnText("Yes");
  assertFalse("project still visible",
              searchText(DEFAULT_PROJECT));
  assertTrue("project not visible",
             searchText(OTHER_PROJECT));
  String newCurrent =
           ProjectManager.getCurrentProject().getName();
  assertNotSame("project not deleted",
                DEFAULT_PROJECT, newCurrent);
}
```

Figure 8.3: A test from Pocket Code that checks the deletion feature of the projects list (simplified for presentation).

written using the Robotium test framework. The actions are interleaved with assertions, which check that the state of the UI (lines 7–10) and the app (line 13) conforms with the expected one. A failure of the test reveals an unexpected behavior.

However, a test may succeed when executed in an ordinary manner, giving the developer a peace in mind, although the app may behave differently in a real-world scenario. As we shall see, this is the case for the test in Figure 8.3.

The behavior of an app depends not only on the sequence of ordinary UI actions (as simulated by the test), but also on external events that may interfere. Such events arise from incoming calls, clicks on hardware buttons, such as, the home button or the earphone media keys, device rotations, other apps trying to acquire the audio focus, the user plugging out his earphones, etc. Manually written test cases rarely take such events into account.

For this example, consider what happens during the execution of test-DeleteCurrentProject if the app is sent to the background and resumed, as the result of, for example, the user long pressing the home button and returning to the app (see Figure 8.2d). During this process, the current activity is paused, which technically means that the activity can still be partially visible, but can be left soon or, as in this case, simply resumed. The app is notified of this change by means of a series of method calls, and it is supposed to commit all the changes and release all its resources. The app will be resumed when it is put back to the foreground.

This sequence of events reveals a bug in the implementation of the deletion feature. If the app is paused while the confirmation dialog is shown, it will show the same dialog when resumed. However, the referenced adapter gets [http://developer.android.com/reference/android/app/Activity.html](http://developer.android.com/reference/android/app/Activity.html)
recreated from scratch (line 3) with a fresh set of checkedProjects (line 28), causing the project deletion to fail silently.

This problem is not revealed by the test in an ordinary execution. However, it is caught by the assertion on line 7 in Figure 8.3 if the test is exposed to an adverse condition immediately after the UI action on line 5, which simulates the app being sent to the background and resumed.

8.3 Methodology

We assume to be given an app with a UI test suite. A UI test is a small program that uses the primitives offered by a test framework to interact with the UI of an app. The instructions usually contained in a test may trigger events on the UI (as in lines 3–6 of Figure 8.3), inspect the UI or directly interact with the internal state (as in line 2 where projects are created and added to the app, circumventing the UI), await timers or other conditions (typically to make sure some asynchronous task is completed before the test proceeds), and assert desired properties of the internal state or the UI (as in lines 7, 9, and 13).

Mobile apps involve many kinds of events: system events that are triggered by sensors, by changes in the environment, or by other apps, as well as actions initiated by the user (e.g. docking, tapping). Events that are triggered in typical test suites tend to concentrate on user events that directly concern the graphical UI, not on other kinds of user interactions (e.g. rotating the device) or system events. For example, although device rotations are common in real use, they are rare in test suites. Among the approx. 3 500 events that appear when executing the 507 tests that we consider in Section 8.5, only 7 are of this kind. This means that the behavior of the apps in presence of such kinds of events largely remains untested. One likely reason is that the programmers may be less aware of those kinds of events, or that there is no obvious place to test them among the ordinary test cases. Another reason may be that some testing frameworks do not provide the necessary primitives to trigger such events; for example, Robotium does not support simulating a click on the home button.

Neutral event sequences We say that a sequence $n$ of events is neutral with respect to a given test if injecting $n$ during the test is not expected to affect the outcome.

As an example, the event sequence Pause–Stop–Restart, which consists of the Android life-cycle events that occur when the user long presses the home button and then returns to the app, is neutral with respect to the test in Figure 8.3. Whether an event sequence is neutral or not is of course subjective—and must in the end be decided by the app developer—but a common sense expectation often exists, as observed by Zaeem et al. [188] who use a related notion called user-interaction features (see Section 8.6).
8.3. METHODOLOGY

Notice that the property of being neutral may depend on the individual test. For example, a loss of WiFi connection followed by a 3G signal recovery can be neutral for most tests but not for one that checks that data is only uploaded when WiFi is available.

The identification of neutral event sequences is pivotal for our methodology. Our idea is to systematically issue neutral sequences of events in specific moments of a test execution. If a test assertion fails, it is safe to issue a warning reporting that the app is misbehaving. The neutral event sequences of interest are those that complement the ordinary user events that concern the graphical UI of the app.

**Injection points** Not all program points in a test are suitable locations for injecting events. First, there is no reason to inject the same events in consecutive instructions that merely inspect the UI or internal app state. Second, to ensure that the simulated event sequence with the injected events is realizable in practice, we should not inject events in the middle of a sequence of instructions that programmatically modify the app state. Third, to avoid introducing nondeterministic outcomes of the tests, we should not inject events right after sleep instructions. Altogether, this leaves us with the following design choice, which is also simple to implement: Injection of events takes place immediately after test instructions that trigger events, identified by the use of primitive operations from the test framework. This corresponds to the program points after each of lines 3–6 in our example in Figure 8.3. More precisely, we inject events after a test has triggered an event and the corresponding event handlers have completed (i.e. when the event queue becomes empty), and we delay execution of the remaining test instructions until the event handlers of the injected events have completed.

**The basic algorithm** Given a set of test cases $T$ and a list of neutral event sequences $N$, we execute each test in $T$ using a modified testing framework that injects every event sequence from $N$ (in the given order) at every injection point (as defined above). In other words, we combine all the event sequences in $N$ into one neutral event sequence (neutral event sequences are trivially closed under concatenation) and inject it aggressively. Naturally, we only consider test cases that do not fail in ordinary executions.

In the test from Section 8.2 an injection point is reached after the clicks on the “Programs”, “Delete”, and “Yes” buttons and after the long click on the project (lines 3–6 in Figure 8.3). Our algorithm injects the neutral event sequences at each of these injection points, in particular, at the delete confirmation dialog (Figure 8.2c), thereby triggering the error caught by the assertion in line 7 in Figure 8.3.

**Detecting multiple errors with each test** A potential limitation of the basic algorithm is that a test stops as soon as an assertion fails or the app crashes, which may shadow other errors. For example, the basic algorithm never
reaches beyond line 7 in the test in Figure 8.3, although the later assertions might potentially fail with another choice of injections. For this reason, we slightly extend the basic algorithm: Whenever an assertion fails or the app crashes, we rerun the test but only perform event injections at program points after the failed assertion or app crash. We keep rerunning as long as the test fails.

With the algorithm presented above, and assuming no assertion failures or app crashes, every test is subjected to a large number of additional events, but we still only execute each test once. We choose this approach to minimize the slowdown compared to ordinary test execution, as restarting tests is likely more time consuming than executing the injected events [38]. On the other hand, it is possible that we thereby miss errors that could be detected with a less aggressive strategy. Some errors may only manifest with very specific combinations of injections. We hypothesize that this is mostly a theoretical concern: Few additional errors will be detected in practice if we inject only a subset of the neutral event sequences and use only a subset of the injection points. We test this hypothesis experimentally in Section 8.5.

Isolating the causes of failures As stated in Section 8.1, we aim not only to detect as many errors as possible and avoid a significant slowdown compared to ordinary test execution; we also want useful error messages to help the developer understand the causes of the errors being found. Although each failing test execution comes with a concrete trace of the events involved, the fact that we have chosen an aggressive strategy for injecting events means that it may not be clear which of the injections are the critical ones. Our approach to address this problem is based on the following small-scope hypothesis [88]: Most errors that can be detected by injecting neutral event sequences can be found by injecting only one of the neutral event sequences and at only one injection point. We also test this hypothesis in Section 8.5.

On this basis, we choose to apply a simple variant of delta debugging [190]. Whenever a test fails (using either the basic or the extended algorithm presented above), we perform the following steps that attempt to isolate a single neutral event sequence and a single injection point to blame:

1. We first perform a binary search through \( N \), in each step eliminating half of the neutral event sequences, and each time with the same injection points, until we find a single neutral event sequence \( n \) that leads to the same failure.

2. If such an \( n \) exists, we perform a binary search through the sequence of injection points, repeatedly eliminating half of them and each time injecting only \( n \), until we find a single injection point triggering the failure. As relevant injection points are likely close to the failure, in each iteration we inject \( n \) in the half that is closest to the failure point.

Of course, many heuristic variations can be conceived, for example, replacing the first binary search with a linear scan if \( N \) is small, or first trying neutral
public void testDeleteProject() {
    createProjects();
    clickOnButton("Programs");
    longClickOnTextInList(PROJECT_1);
    clickOnText("Delete");
    assertTrue("Dialog title is wrong!",
                searchText("Delete this program?"));
    clickOnText("Yes");
    assertFalse("Project still visible",
                searchText(PROJECT_1));
    ArrayList<String> projectList =
        UtilFile.getProjectNames(...);
    boolean deleted = true;
    for (String project : projectList) {
        if (project.equalsIgnoreCase(PROJECT_1)) {
            deleted = false;
        }
    }
    assertTrue("Project not deleted", deleted);
}

Figure 8.4: Another test from Pocket Code, illustrating potential redundancy (cf. Figure 8.3) when injecting neutral event sequences.

event sequences that are known to be involved in many errors in prior test executions.

The isolation algorithm may fail to find a single injection to blame. The small-scope hypothesis might not apply because a combination of injections is needed to expose the error, and flakiness of tests (see [114]) may divert the search from the failure injection point. Still, our approach reduces the sets of injections to consider during debugging.

Reducing redundancy The choice of rerunning the tests to enable detection of multiple errors with each test has two potential drawbacks. Although more assertion failures or app crashes may be encountered, and hence more error messages are emitted, some of these may have the same ultimate cause and thereby do not reveal any additional bugs in the app. Also, rerunning tests takes additional time.

As an example, consider the test in Figure 8.4 which is also part of the test suite for Pocket Code. The test is checking a similar use case to that of Figure 8.3, but deletes another project than the currently selected one, and also checks that the project is in fact deleted from the disk. This test fails for the same reason as the test from our motivating example when executed using our basic algorithm, resulting in redundant warnings. Furthermore, for the extended algorithm that aims to detect multiple errors with each test, it also
leads to superfluous test executions, because the technique reruns a test for every warning being produced.

We propose the following mechanism to heuristically omit certain injections, for use in situations where the developer would like to concentrate on the error messages that are likely caused by distinct bugs and to get the error messages faster. During execution of the tests, we build a cache of abstract states. An abstract state is added each time some event $e$ has been processed and some neutral event sequence $n$ is about to be injected. Each abstract state consists of an abstraction of the UI state together with $e$ and $n$. (Such abstractions of UI states are common in the literature on model-based testing of mobile, web, and GUI applications [16, 25, 38, 49, 51, 76, 125, 188].) Now, we simply skip the injection if the abstract state already appears in the cache. For the example, this means that the injection, that cause the redundant warning from the test in Figure 8.4 to be triggered, is omitted. The intuition behind this choice is that any bug that may be found by that injection would likely have been found already when the abstract state was added to the cache.

This mechanism may obviously cause some bugs to be missed. Injecting a single neutral event sequence in some abstract state may falsify multiple assertions, so a bug can be missed if an injection is skipped due to the cache mechanism. In Section 8.5 we quantify the trade-off between reducing redundancy and maintaining the ability to detect bugs.

8.4 Implementation

We have implemented the testing technique in a tool Thor7 designed for Android apps and Robotium test suites. The tool lets the app developer select the set of tests to run, the set of neutral event sequences to take into account, and whether or not to enable the different variations of the algorithm presented in Section 8.3. During the execution, the tool provides an interactive visualization of the issues found, including all the necessary information to reproduce them, allowing further investigation.

Thor executes the tests on an emulator running Android KitKat 4.4.3 (from the Android Open Source Project). A controller component on the host machine guides the executions of the tests. The test execution is parallelized by using multiple Android emulator instances. The controller is implemented in Node.js8 and is managed via a web interface. We have manually instrumented the Android framework by adding hooks that allow the tester component to control the execution of a test, detect injection points, and perform the injection of neutral event sequences.

The redundancy reduction strategy requires a way to abstract runtime states. Much like previous work on model-based testing, our implementation

7 http://brics.dk/thor/; named after the storm god.
8 http://nodejs.org/
abstracts away from data stored in activities, on disk, and by content providers, preserving only information about the structure of the UI as represented by the Android view hierarchy. More precisely, our abstraction consists of the tree of objects, which are all instances of the View class, represented by their class names and associated event handlers, while ignoring all concrete string values, screen coordinates, timestamps, etc. A convenient property of this choice of abstraction is that it can be computed quickly, which is important for its use in reducing redundancy.

Selecting neutral event sequences The Android framework provides ample opportunities for selecting neutral event sequences. It contains well over 60 different services to manage the different resources available on the system (e.g. IActivityManager, IAudioService, IAlarmManager, IBackupManager, ICameraService, IDropBoxManagerService, ILocationManager, IMountService, ITelephony, IUsbManager, IVibratorService, IWifiManager, and IWindowManager). Most importantly, the activity manager is responsible for the life-cycle of the activities running on the system, which includes creating, pausing, resuming, and destroying activities, depending on external factors (e.g. low memory) and user interactions (e.g. home button click, screen rotation). As another example, the audio service manages the speakers and earphones, including granting permission to emit sounds (also called audio focus). Apps are allowed to use the functionalities offered by some of the services using remote procedure calls with a thread-migration programming model. In this way, apps can invoke service methods as if they were running on a thread of the service process. Moreover, services can invoke methods of apps. This opens for many different ways in which apps can be influenced by events from services, or even of other apps. Any event that causes the internal state of a service to change may affect the app as well: the service may call a method on the app process depending on its state, or the app may call a method on the service process that depends on the service state. Hence, internal service state changes can potentially alter the behavior of an app. Any sequence of events that should not influence the outcome of a test case is of interest.

For our experiments we focus on events concerning the activity manager and the audio service as these are widely used. The activity manager is particularly relevant, as all apps must interact with it and cannot ignore the life-cycle events. Many common high-level events (e.g. incoming call, device rotation, docking) cause life-cycle events. According to the Android documentation, the activity manager always issues events to an app as a consequence of the changes of the activity status. The possible events and the associated state changes are shown in Figure 8.5.

Example. For our example in Section 8.2 the activity manager triggers a pause event by calling the method schedulePause on ApplicationThreadProxy, which eventually dispatches the event to the current activity, whose default
CHAPTER 8. TESTING IN ADVERSE CONDITIONS

Figure 8.5: The Android activity lifecycle (from http://developer.android.com/).

implementation dispatches it further to its contained fragment by calling `onPause` on the `Fragment` class.

Other services, including the audio service, are only relevant for apps that use particular hardware components. Some of the events that are concerned with the audio service are `audio focus request`, which causes a loss event to be sent to an app whenever another app is requesting to use the audio, and `abandon audio focus`, which causes an app to be notified that the audio is available.

We choose an initial collection of neutral event sequences for Thor. Concerning the events related to the activity manager we select the sequences of events in Figure 8.5 that cycle back to the `Resumed` state (assuming an implicit edge from `Destroyed` to `Created` due to the user restarting the app).

- **Pause–Resume (PR)** – issued in some devices when the screen is turned off and on.
- **Pause–Stop–Restart (PSR)** – issued when the user long-presses the home button to show the open apps and then returns to the app.
- **Pause–Stop–Destroy–Create (PSDC)** – issued when the phone is rotated or docked.

Technically, PSR also contains a Resume event, and PSDC also contains Start and Resume, but we omit these in the event sequence names for presentation.

We have excluded other cycles because they are not neutral, e.g. Pause-Stop-Create. Indeed they cause the app to be killed by Android, therefore invalidating any expectations on the outcome of the test.

Concerning the audio service, the following examples illustrate neutral event sequences:
8.5 Experimental Evaluation

We evaluate THOR by conducting an empirical study on real Android apps to investigate whether THOR is capable of exposing bugs that are undetected by ordinary executions of existing test suites, and at what extra cost in testing time. We divide this into five research questions:

Q1 (error detection) Several aspects of the error detection capability deserve attention. To what extent is it possible to trigger failures in existing test suites by injecting neutral event sequences? Different failures may have the same ultimate cause. How many distinct problems in the apps do the failures identify? How many of the problems are likely perceived as critical bugs from the user’s perspective, and how many of these affect the functional behavior of the app but without causing it to crash? How many failures are missed if disabling the rerun of a test at each failure?

Q2 (execution time) What is the slowdown for the different modes of THOR compared to ordinary test execution? Specifically, how much extra time is spent when enabling the rerun of a test at each failure, and how much time is saved when enabling the redundancy reduction mechanism?

Q3 (redundancy reduction) When enabling the redundancy reduction mechanism, what is the effect on the number of test failures and critical bugs being detected?
CHAPTER 8. TESTING IN ADVERSE CONDITIONS

Table 8.1: The apps used in the evaluation of Thor.

<table>
<thead>
<tr>
<th>App</th>
<th>Version</th>
<th>Rating</th>
<th>Downloads</th>
<th>LOC Source</th>
<th>UI Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pocket Code (PC)</td>
<td>Aug 07, 2014</td>
<td>93/5</td>
<td>50K–100K</td>
<td>34K</td>
<td>19.4K</td>
</tr>
<tr>
<td>Pocket Paint (PP)</td>
<td>Sep 19, 2014</td>
<td>4/5</td>
<td>10K–50K</td>
<td>6.6K</td>
<td>3.8K</td>
</tr>
<tr>
<td>Car Cast (CC)</td>
<td>Jul 11, 2014</td>
<td>4.1/5</td>
<td>100K–500K</td>
<td>6K</td>
<td>0.5K</td>
</tr>
<tr>
<td>AnyMemo (AM)</td>
<td>Apr 03, 2014</td>
<td>4.5/5</td>
<td>100K–500K</td>
<td>20.1K</td>
<td>1.2K</td>
</tr>
</tbody>
</table>

For each app, we show its version, the rating on the Android Play Store, and the number of downloads. The LOC column shows the number of code lines for the app source and the UI tests. The Tests column shows the number of tests that were stable vs. unstable (i.e. consistently or randomly failing) in our test environment. The last two columns show the code coverage and the time it takes to execute the stable tests.

Q4 (failure isolation) Is the failure isolation mechanism effective in finding a neutral event sequence and injection point for each failure?

Q5 (hypotheses) Our design in Section 8.3 was based on two hypotheses. Is it correct that few additional errors will be detected if we inject only a subset of the neutral event sequences and use only a subset of the injection points? Also, is it correct that most of the errors can be found by injecting a single neutral event sequence in a single injection point?

Experiments

UI testing frameworks for Android are popular: Robotium\(^9\) and Calabash\(^{10}\) each count more than 850 stars and 400 forks on GitHub, and Espresso\(^{11}\) has recently been added to the Android Support Library. In addition, several companies (e.g. Appurify, Xamarin, TestDroid) are offering Android cloud testing facilities. This gives evidence that UI testing is widely used in practice. Nevertheless, we regrettably only have access to a few nontrivial apps with UI test suites, since most are closed source as also noted by Fard et al.\(^{51}\). Our experiments are therefore based on a case study of 4 open-source apps. The

\(^9\)http://code.google.com/p/robotium
\(^{10}\)http://calabash
\(^{11}\)http://developer.android.com/tools/testing-support-library/index.html#Espresso
experiments are performed on a 2.4 GHz Intel Core i5 laptop with 8 GB RAM using the x86 Android emulator and a pool of 3 emulator instances.

Table 8.1 shows some characteristics of our benchmark apps. AnyMemo (AM) is an app designed for learning different languages, computer related topics, etc. through flashcards. Car Cast (CC) is a simple podcast player that uses the media player facilities and reacts to several audio related external events. Pocket Code (PC) is the app we used in Section 8.2 as motivating example. Finally, Pocket Paint (PP) is a paint program with various tools for editing images.

As evident from the size of the UI test suites, the app developers have put a considerable effort into writing UI tests. However, since running all tests can take a considerable amount of time (e.g. more than 2 hours in the case of PC), it is not uncommon that parts of a test suite are out of sync with the app code as the app is being developed. Moreover, the developers may run the tests in specific environments, which are unknown to us, for example, with a particular version and configuration of the Android emulator ecosystem. For these reasons, it is not surprising that some of the tests fail when we execute the test suites even without injecting any new events. We mark those tests as unstable and exclude them from the evaluation of THOR. When executing the resulting stable tests with THOR, we encounter no failures that can be attributed to environment settings.

In order to answer Q1–Q5 we conduct three experiments.

**Experiment 1** We run THOR on each test suite, using all the variations presented in Section 8.3 (i.e. enabling or disabling the rerunning of a test when a failure occurs, the failure isolation technique, and the redundancy reduction mechanism). The configuration uses the neutral event sequences concerning the activity manager and the audio service mentioned in Section 8.4. We manually classify each failing test to determine the root cause, in order to group failures that are caused by the same bug and identify which bugs are likely critical from the user’s perspective. As such manual classification is time consuming, we settle for a large representative subset of the failures. This experiment allows us to answer Q1–Q3.

**Experiment 2** To answer Q4 we measure the success rate of the failure isolation strategy when applied to all tests that fail in adverse conditions.

**Experiment 3** To test the hypotheses in Q5 we develop an algorithm that injects a random subset of the neutral event sequences and use a random subset of the injection points (using a uniform distribution), and execute it 50 times per test case. For each test, we compare the failures found by these random runs with the ones found by the ordinary execution of THOR in Experiment 1.
Table 8.2: Failures and bug categories.

<table>
<thead>
<tr>
<th>App</th>
<th>Logical Crash</th>
<th>Silent fail</th>
<th>Not persisted</th>
<th>User setting lost</th>
<th>Critical UI Operation ignored</th>
<th>Unexpected screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pocket Code</td>
<td>1 (9)</td>
<td>7 (42)</td>
<td>-</td>
<td>1 (6)</td>
<td>-</td>
<td>4 (51)</td>
</tr>
<tr>
<td>Pocket Paint</td>
<td>2 (45)</td>
<td>-</td>
<td>1 (4)</td>
<td>4 (42)</td>
<td>1 (25)</td>
<td>-</td>
</tr>
<tr>
<td>Car Cast</td>
<td>1 (7)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AnyMemo</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The number of bugs in each category (with the corresponding number of failures shown in parentheses). **Logical.** Crash: e.g. NullPointerException or bad use of Android Support Library. **Silent fail:** e.g. the example from Section 8.2. **Not persisted:** e.g. internal static field savedPictureUri is reset in PP. **User setting lost:** e.g. the selected brush size is lost in PP. **Critical UI.** **Operation ignored:** e.g. cannot draw in PP. **Unexpected screen:** e.g. navigation to unexpected screen, activity, or fragment. **UI.** **Not persisted:** e.g. button disabled, text cleared, or checkboxes unchecked. **Element disappears:** e.g. menu or dialog disappears, or dropdown closes. **Spec. fail:** less important assertions, e.g. a wrong text such as “Delete these programs” instead of “Delete this program”. **Other.** **Brittle test:** invalid reference to a View in the test due to app relaunch. **Emulator issue:** e.g. emulator unable to perform an action.

Next, for each failure detected by THOR where the failure isolation mechanism was unable to isolate a single neutral event sequence and a single injection point, we try every possible single neutral event sequence and single injection point in turn to check whether one actually exists. To limit the time for conducting this experiment, we use a randomly selected subset of the tests.

**Results**

**Q1** The results from running THOR on the four apps in Experiment 1 show that as many as 429 tests fail out of a total of 507 when run in adverse conditions. This amounts to 1770 failures counted as distinct failing test assertions or app crashes, none of which appear in ordinary execution of the tests. These numbers witness the good weather assumption about the environment in which tests are traditionally executed, and clearly demonstrate that our technique is able to trigger failures in existing test suites.
The majority of the failures are due to the PSDC event sequence (AM: 91%, CC: 100%, PC: 61%, PP: 98%), whereas PR accounts for the lowest amount of warnings (AM: 5%, CC: 0%, PC: 16%, PP: 1%). This is expected, since the three neutral event sequences that consist of life-cycle events in turn invoke more event handlers in the app. For example, PSDC is the only one that causes the activity to be destroyed and recreated (see Figure 8.5), meaning that, for example, inadequate persistence is more likely to expose a problem.

In the experiment, no failures originate from audio related event sequences. Thor is able to dynamically detect the specific tests that uses the audio so that it only injects audio related events when relevant (similar to the relevant events detection by Machiry et al. [115]). Hence, we only inject audio related events in few tests of CC, which is the only app that uses audio.

Our manual classification of a random selection of 682 of the 1770 failures gives the categorization depicted in Table 8.2. As discussed, a single bug may cause multiple test failures. The table presents the number of distinct bugs and the corresponding number of failures (in parentheses) in each category that we have identified. The categories Logical, Critical UI, and UI correspond to real problems, whereas Other corresponds to technical issues originating from a specific coding style used in some tests that is brittle towards app relaunches (this can be avoided with minor rewriting efforts) or from the flaky nature of UI tests. The table further divides the different categories and also includes examples of typical kinds of errors being found.

From the table it follows that we have identified 66 distinct problems in the apps from the 1770 generated failures. On average, each problem is spotted approx. 10 times. This is not surprising; different tests often visit the same UI components and will therefore encounter the same failures, if, for example, one of these components disappears when the device is rotated. Later in this section, we separately evaluate how the redundancy reduction mechanism affects these numbers (cf. Q4).

Table 8.2 also shows that 22 out of 66 distinct problems detected by Thor fall into the categories Logical and Critical UI, which we conservatively classify as being critical bugs from the user’s perspective.

The bug in PC that we described in Section 8.2 is among the ones detected by Thor in the category Silent fail. Another bug in PC causes the app to navigate to a wrong screen, because the activity does not persist the active fragment. In PP, user settings, such as, the currently selected tool and brush size, are not properly persisted, causing them to be reset under certain life-cycle events. A list of some other bugs that Thor has revealed is shown in Table 8.3.

Overall, the number of errors is dominated by the UI category, comprising, for example, simple UI widgets that disappear from the screen. Many of these problems are likely to be ignored by app developers, since the improvement of the app does not match the implementation effort needed to solve the problems. As an example, it is typically not considered a serious problem that a menu or dialog closes upon e.g. rotation. On the other hand, some of the bugs in the
Table 8.3: Examples of errors detected by Thor.

<table>
<thead>
<tr>
<th>Pocket Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Null pointer crash in the copy program dialog.</td>
</tr>
<tr>
<td>- The OK button gets disabled on the new program dialog.</td>
</tr>
<tr>
<td>- A fragment (Scripts/Looks/Sounds) is randomly opened.</td>
</tr>
<tr>
<td>- The user setting Show Details is lost.</td>
</tr>
<tr>
<td>- The selected project name disappears from the action bar title.</td>
</tr>
<tr>
<td>- The app randomly navigates to the program list screen and opens the copy program dialog.</td>
</tr>
<tr>
<td>- Clicking “OK” on the copy dialog of a script has no effect.</td>
</tr>
<tr>
<td>- Clicking on “+” in the bottom bar navigates to a wrong screen.</td>
</tr>
<tr>
<td>- Title changes from “Delete this program?” to “Delete these programs?”.</td>
</tr>
<tr>
<td>- The bottom bar appears.</td>
</tr>
<tr>
<td>- Created program element disappears when dragged.</td>
</tr>
<tr>
<td>- Deletion of various elements fails silently.</td>
</tr>
<tr>
<td>- Various dialogs close inadvertently.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pocket Paint</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The dialogs tool info and tool settings crash due to missing empty constructors.</td>
</tr>
<tr>
<td>- The position of the canvas on the screen is reset.</td>
</tr>
<tr>
<td>- When drawing on the canvas, the strokes are not retained.</td>
</tr>
<tr>
<td>- The selected tool, color, and brush is not persisted.</td>
</tr>
<tr>
<td>- The full-screen mode gets disabled.</td>
</tr>
<tr>
<td>- The “Undo” button gets enabled although no drawing actions have been performed.</td>
</tr>
<tr>
<td>- Various dialogs close inadvertently.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Car-Cast</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Crashes when an open dialog (loader) is dismissed.</td>
</tr>
<tr>
<td>- Various dialogs close inadvertently.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AnyMemo</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Content of a text field disappears.</td>
</tr>
<tr>
<td>- Review dialog closes.</td>
</tr>
<tr>
<td>- Buttons (“Forgot”, “Easy”, etc.) disappear.</td>
</tr>
</tbody>
</table>

UI category involve text fields being reset, which can result in a negative user experience. In a few cases, the choice of the neutral event sequences to inject may depend on the individual test cases. For example, PP has a full-screen mode, which is closed when the device is rotated. This situation is detected by Thor. If this behavior is intended, the PSDC event sequence is not neutral, meaning that the test failure concerned has revealed a hidden assumption of the test. We emphasize that all the issues in the UI category are conservatively counted as non-critical.

The fact that Thor is able to detect functional bugs, such as, incorrect persistence of user settings, proves that our approach is capable of exposing bugs that are undetected by ordinary execution of the tests. Also notice that among the 22 distinct bugs that damage the user experience, only 4 are crashes. This shows the value of leveraging existing tests, compared to automated testing techniques that focus entirely on app-agnostic error conditions.
Table 8.4: The slowdown when running in adverse conditions.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>AM</th>
<th>CC</th>
<th>PC</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>1.05x</td>
<td>1.21x</td>
<td>1.38x</td>
<td>0.99x</td>
</tr>
<tr>
<td>Enable rerun failing tests</td>
<td>2.11x</td>
<td>3.09x</td>
<td>4.70x</td>
<td>3.70x</td>
</tr>
<tr>
<td>Enable redundancy reduction</td>
<td>1.02x</td>
<td>1.30x</td>
<td>1.57x</td>
<td>1.17x</td>
</tr>
<tr>
<td>Enable both</td>
<td>1.73x</td>
<td>1.93x</td>
<td>3.46x</td>
<td>2.04x</td>
</tr>
</tbody>
</table>

The use of existing tests also has advantages compared to the approach by Zaeem et al. [188] (see Section 8.6). For example, the bug presented in Section 8.2 does not cause the UI to change and is hence unnoticed by their tool. As a small additional experiment, we run THOR on CC in a mode where it also raises warnings if the UI changes due to an injection of a neutral event sequence (based on a comparison of screenshots). This roughly doubles the number of warnings but reveals no new bugs.

In order to determine the number of failures that are missed by disabling the rerun of a test at each failure, we perform a case study on PP where we manually classify each failing test (with rerunning disabled) to determine the root cause. Our results show that the basic algorithm without the rerunning extension detects only 8 of the 17 distinct bugs.

Q2 Table 8.4 presents the slowdown for all the variations of our algorithm compared to ordinary test executions (cf. Table 8.1). The slowdown of our basic strategy is 0.99–1.38x, which is competitive to an ordinary test execution. We note that there is a small overhead for injecting events during the test execution, but nonetheless, running test suites in adverse conditions can be faster (as in the case for PP), because failures cause test cases to exit early.

When THOR is run with the rerun extension, the execution time increases significantly (2.11x–4.70x). This increase is expected, since test suites with many failures will have many reruns. Such reruns are expensive, especially when the test cases are long (some tests from our study trigger more than 100 UI events). However, this mode is suited for situations where the developer is interested in learning all the failures of the test suite in one execution. In many cases the developer will be satisfied with learning one failure in each test case, as provided by the basic algorithm. Eventually, when the initial bugs have been fixed and only few failures remain, the rerun mode can be used with little overhead.

The redundancy reduction mechanism aims to eliminate redundant failures. In doing so, it trades part of the bug detection capability for performance. When few duplicates are present, the overhead of building the cache of abstract states outweighs the benefits gained by reduction. From Table 8.4 it follows that the mechanism is successful in improving the performance when there are many duplicates, thereby alleviating much of the overhead of the rerun extension.
Q3 According to Experiment 1, the redundancy reduction mechanism works best when combined with the rerun extension, which tends to find more failures that have the same cause. For example, PP suffers from duplication of PSDC-related failures when executed using this algorithm (recall that 98% percent of the failures in PP are due to PSDC). However, when executed using the redundancy reduction mechanism, the number of PSDC-related failures reduces from 350 to 75, resulting in significantly fewer reruns. Importantly, a 79% reduction of the number of failures does not cause a similar degradation in the number of distinct bugs being detected: 14 of the 17 distinct problems in PP are detected when redundancy reduction is enabled. In combination with the results showing the speedup obtained by this mechanism (as discussed in relation to Q2), we conclude that the redundancy reduction mechanism provides an effective compromise between speedup and bug detection capability.

Q4 Our results from conducting Experiment 2 show that the failure isolation mechanism is capable of successfully identifying one single neutral event sequence and injection point for all except 5 failures. Three of the unsuccessful isolations succeeded in identifying the neutral event sequence, but was unable to reduce the set of injection points. This situation occurs when the binary search fails to spot the error in any iteration. The remaining two attempts also successfully identified the relevant neutral event sequence, and was able to reduce the number of injection points from 49 and 27 to 3 and 14, respectively.

Q5 Experiment 3 provides us with the necessary data to test our two hypotheses. From a total of 13,784 random runs coming from 144 different test cases and running for a total of 94 hours (compared to 4 hours and 24 minutes for our ordinary execution of THOR on all the 507 test cases), we observe a total of 7,810 failures, 2,783 of which are duplicates. We here consider two failures to be the same if they have the same exception and they are raised in the same test case. Only 56 of the 5,027 distinct failures obtained in random runs are not already detected by our basic algorithm. This number reduces to 26 if we consider the total number of distinct failures without distinguishing between the test cases that raise them, but only look at the actual exception. We point out that this can only give a rough estimate of the actual number of bugs being detected by the two techniques, because the same failure may concern distinct problems. Still, these results support our first hypothesis, namely that only few more errors would be detected if we inject all the possible subsets of neutral sequences in all the possible subsets of injection points.

Concerning the second hypothesis, the results of Experiment 3 show that a high percentage of the failures raised by the basic algorithm can be eventually reduced to a single event and a single injection point to blame: only 26 out of 324 failures used in the experiment need a more complex combination of injected events.
8.6 Related Work

Numerous techniques and tools have been developed to support software testing, specifically for mobile apps, and we here discuss the most closely related work.

The use of existing test suites is a key property of our technique for guiding exploration and providing test assertions that specify the intended behavior of the app being tested. This idea of leveraging existing tests has been applied before. Xie and Notkin [181] infer operational abstractions from existing unit tests and then generate new tests that attempt to violate these abstractions. Fraser and Zeller [60] similarly infer parameterized tests from ordinary unit tests. These techniques are not immediately applicable to our setting with event-driven applications and errors that involve events that are typically not mentioned in the original tests.

The Testilizer tool for web application testing by Fard et al. [51] uses a technique more closely related to ours. By extending human-written test suites with automated crawling and heuristics for assertion regeneration, they achieve an improvement in fault detection and code coverage. In comparison, our use of neutral event sequences does not require random crawling and assertion regeneration. Zhang and Elbaum [194] use test amplification for validating exception handling, but do not benefit from the existing assertions in the tests.

Many automated testing techniques aim to explore apps by variations of random testing, or crawling, without the use of existing test suites [16, 18, 25, 38, 82, 84, 115, 119, 145, 183, 188]. This has the practical advantage that it is easier to perform large-scale experiments on 1000s of apps, as these are immediately available, unlike their UI tests. More fundamentally, since these testing techniques cannot take advantage of the test-case specific assertions, they often use code coverage as a proxy for error detection capability and are restricted to detecting application-agnostic error conditions, such as, app crashes. Assuming that our preliminary experimental results generalize—recall that 18 of the 22 critical bugs found by Thor are not crashes—this means that random testing techniques can only see the tip of the iceberg regarding app errors that arise in adverse conditions.

Other error detection techniques use notions of neutrality similar to ours, either implicitly or explicitly. LeakDroid [182] repeatedly executes event cycles that should have a neutral effect on the resource usage, while monitoring the execution to identify potential leaks. Orion [104] performs semantics preserving mutations of a program to test the correctness of optimizing compilers. The Quantum tool by Zaaem et al. [188] is based on a notion of user-interaction features, which are actions that are associated with a common sense expectation of how the app should respond. Neutral event sequences are related to user-interaction features where the common sense expectation is that the app behavior should be unaffected by the actions. As discussed in Section 8.1, Quantum requires a UI model of the app and a suitable abstraction of the execution states, instead of leveraging UI tests.
The fact that life-cycle events are a major source of programming errors is also noted by Hu et al. [84]. Their tool, AppDOCTOR, performs fast random testing by triggering low-level event handlers directly and then attempts to eliminate false positives by faithfully simulating the high-level user events (e.g. emitting a touch event Down, waiting 3 seconds, and then emitting a touch event Up, instead of directly invoking the Long-press event handler). THOR injects events via the Android framework, not by invoking event handlers directly. This may in principle still lead to false positives, however, we have found no false positives among the 1770 warnings in our experiments.

Another kind of “adverse conditions” is environment failures, which are the focus of stress testing tools, such as, VANARSENA [115], PUMA [70], and CAHPA [108]. These tools aim to expose bugs by inducing faults in network response (e.g. replacing an actual response by HTTP 404), and fuzzing device configuration, wireless network conditions, etc. Some of the device specific events triggered by THOR are related to this approach, for example, the neutral event sequence 3G–WiFi–3G.

Errors in event-driven apps may also be caused by unexpected nondeterministic ordering of events, as studied in recent work on race detection [82, 119]. Such techniques require manual investigation to identify the harmful races and tend to produce many false positives. None of the bugs discovered by THOR involve races.

Other techniques use symbolic execution [18, 90], which can potentially reach the challenging targets in the app, but are difficult to scale and do not exploit the information present in UI tests. Finally, existing tools that are based on static analysis (e.g. [1, 2, 22, 98]) focus on security vulnerabilities, not on the kinds of errors being detected by THOR.

8.7 Conclusion

We have presented a light-weight methodology for leveraging existing test suites by executing them in adverse conditions, systematically injecting event sequences that should not affect the outcome of the tests. Our evaluation on a small collection of Android apps demonstrates that the approach is effective in finding critical bugs, many of which are functionality errors that are difficult to detect with other automated testing techniques.

In addition, our results show that the cost in additional testing time is low relative to the number of bugs found, and that the technique is capable of isolating the causes of failures to support debugging.

Acknowledgments

This work was supported by the Danish Research Council for Technology and Production.
Chapter 9

Analyzing Test Completeness for Dynamic Languages

By Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Published in Proc. 25th International Symposium on Software Testing and Analysis (ISSTA), July 2016. ACM SIGSOFT Distinguished Paper Award.

Abstract

In dynamically typed programming languages, type errors can occur at runtime. Executing the test suites that often accompany programs may provide some confidence about absence of such errors, but generally without any guarantee. We present a program analysis that can check whether a test suite has sufficient coverage to prove a given type-related property, which is particularly challenging for program code with overloading and value dependent types. The analysis achieves a synergy between scalable static analysis and dynamic analysis that goes beyond what can be accomplished by the static analysis alone. Additionally, the analysis provides a new coverage adequacy metric for the completeness of a test suite regarding a family of type-related properties.

Based on an implementation for Dart, we demonstrate how such a hybrid static/dynamic program analysis can be used for measuring the quality of a test suite with respect to showing absence of type errors and inferring sound call graph information, specifically for program code that is difficult to handle by traditional static analysis techniques.

9.1 Introduction

A well-known quote from Dijkstra is: “Program testing can be used to show the presence of bugs, but never to show their absence” [45]. This is of course true in general, but, as also observed decades ago [64], there are situations where test suites are complete in the sense that they allow verification of correctness
properties that hold in any execution. As a trivial example, for a straight-line Java program that takes a string as input, a single test execution suffices to detect any null pointer bug that may exist. More generally, test suites may be complete with respect to local properties at specific program points. In this paper, we explore how such a notion of test completeness can be exploited for soundly type checking challenging programming patterns in dynamically typed languages.

One of the advantages of programming languages with dynamic or optional typing, such as, Dart \[16\], TypeScript \[126\], Typed Racket \[169\], and Reticulated Python \[173\], is flexibility: they allow dynamic programming patterns, such as, value-based overloading \[14, 172\] and other forms of value dependent types, that do not fit into traditional static type systems. The cost of this flexibility is that type-related errors are not detected until runtime and that type information is not available statically for guiding auto-completion and navigation in IDEs or optimizers in compilers. Our goal is to provide program analysis techniques that can automatically infer sound and precise type information for such code. More specifically, we focus on Dart and aim to ensure absence of runtime type errors and to infer precise call graph information. Two kinds of runtime type errors are particularly interesting in Dart programs: subtype-violation errors at implicit downcasts and message-not-understood errors at field and method lookup operations. Programmers can benefit from knowing to what extent their tests guarantee absence of such runtime errors. Likewise, call graphs are useful for enabling dead code elimination and other optimizations, as well as for various kinds of IDE support \[52\].

Example Figure 9.1 shows some Dart code from a sample application shipped with the box2d library\[1\] that uses the vector_math library\[2\] which illustrates a style of programming that is common with dynamically typed languages and that is challenging for static type analysis. The cross function is overloaded in the sense that its behavior and return type depend on the runtime types of its parameters: the branch in line 23 returns type vec3 or the runtime type of the parameter out, line 26 returns type double, and line 35 returns type vec2 or the type of out. Moreover, if the types of the parameters at a call do not match any of the cases, then a failure occurs in line 39 if assertions are enabled and otherwise null is returned, which will likely trigger a runtime error later in the program. The cross function is called, for example, in lines 51 and 52. In Dart’s checked mode execution, the assignments to dp2perp and dp1perp will fail if the values being returned are not of type vec3. In production mode execution, the assignments will succeed, but the applications of * in line 53 will fail if dp2perp or dp1perp is of type double and the other argument is of type vec2 or vec3, and + will fail if the left-hand argument and the right-hand argument do not have the same type. (Lines 15-46 show the implementation

\[1\]https://github.com/financeCoding/dartbox2d
\[2\]https://github.com/google/vector_math.dart
of + for the case where the left-hand argument is of type vec2.) How can the programmer be certain that none of these potential type-related errors can occur in the application? Running a few tests may give some confidence, but it is difficult to know when enough testing has been done.

Traditional static analysis is not suitable for reasoning about such code since very high precision (e.g. context sensitivity and path sensitivity) would be needed, which is difficult to achieve together with sufficient scalability. For the example described above, a context-insensitive analysis may be able to

---

```plaintext
9.1. INTRODUCTION

21  dynamic cross(dynamic x, dynamic y, [dynamic out=null]) {
22    if (x is vec3 && y is vec3) {
23      return x.cross(y, out);
24    } else if (x is vec2 && y is vec2) {
25      assert(out == null);
26      return x.cross(y);
27    } else if (x is num && y is vec2) {
28      x = x.toDouble(); 
29      if (out == null) {
30        out = new vec2.zero(); 
31      }
32      var tempy = x * y.x;
33      out.x = -x * y.y;
34      out.y = tempy;
35      return out;
36    } else if (x is vec2 && y is num) {
37      ...
38    } else {
39      assert(false);
40    }
41    return null;
42  }
43  
44  class vec2 {
45    ...
46    vec2 operator+(vec2 other) =>
47      new vec2.raw(x + other.x, y + other.y);
48    ...
49  }
50  
51  // solve the linear system
52  vec3 dp2perp = cross(dp2, normal);
53  vec3 dp1perp = cross(normal, dp1);
54  vec3 tangent = dp2perp * duv1.x + dp1perp * duv2.x;
55  ...
```

Figure 9.1: Code from the `vector_math` library (lines 21–48) and the `box2d` library (lines 50–53).
show that \textbf{cross} can return values of type \texttt{vec3}, \texttt{vec2}, or \texttt{double} (assuming that \texttt{out} is always also of one of those types), but that information is not precise enough to rule out the various type-related errors. Two-phase typing \cite{172} has been proposed to address similar challenges but is not yet practical for realistic programs. Conversely, traditional dynamic analysis is also not suitable as it does not give any soundness guarantees. One notable exception, which has inspired our work, is the constraint-based dynamic type inference technique by An et al. \cite{17}. It can infer sound type information for Ruby programs using dynamic analysis, however, it requires full path coverage of every function in the program being analyzed, which is rarely realistic.

**This work** We propose a hybrid of lightweight static analysis and dynamic execution of test suites. Our key insight is that such a combination of static and dynamic techniques can determine when test suites have sufficient coverage to guarantee type-related correctness properties as in the example.

The static analysis has two parts: a dependence analysis and a type analysis (technically, a points-to analysis). It is context- and path-insensitive and thereby scales to large programs, and it is relatively easy to implement; notably it requires simpler modeling of native functions than what would be required by a fully static analysis approach.

In summary, the contributions of this paper are:

- We define a notion of test completeness as a sufficient condition for a test suite to have enough coverage to guarantee a given type-related correctness property. Using a lightweight static analysis to approximate test completeness, we then demonstrate how a hybrid static/dynamic analysis can produce sound type information for challenging dynamic programming patterns that resist traditional static analysis.

- Based on an implementation, \textsc{goodenough}\textsuperscript{3} for the Dart programming language, we evaluate the ability to ensure absence of runtime type errors and to produce precise call graphs, compared to a more traditional technique that uses the static analysis information only. Across a variety of Dart programs, we find numerous examples where precision is improved. Specifically, the analysis is able to guarantee for 81 % of the expressions that all types that can possibly appear at runtime are in fact observed by execution of the test suite. The experiments additionally show that the limiting factor of the precision in some cases is the test coverage and in other cases the precision of the dependence analysis, which suggests opportunities for future improvements of the technique.

\textsuperscript{3}Named in honor of J. B. Goodenough \cite{64}.
9.2 The Dart Language

In this paper we use Dart as subject for presenting and evaluating our technique. The Dart programming language was introduced by Google in 2013 and has later been standardized by Ecma [46]. The language is now widely used by Google and elsewhere, primarily as an alternative to JavaScript but also for server-side high performance computation and, more recently, embedded devices. Dart supports both object oriented and functional programming using a Java-like syntax.

From our perspective, the most interesting aspect of the language is its type system. Type annotations are optional (the default is called dynamic). The static type checker can warn about likely type errors in the parts of the program where type annotations have been provided, while ignoring the parts without type annotations. Warnings produced by the type checker do not preclude executing the program. Moreover, even for fully annotated programs, the type checker is deliberately unsound. These design choices give Dart the flexibility of dynamically typed programming languages, but they also mean that type errors can occur at runtime.

Dart programs run in either production mode where type annotations are entirely ignored, or checked mode where implicit type casts are performed at assignments to non-dynamic variables or fields, checking that the runtime type is a subtype of the annotated type. Two kinds of runtime type errors can occur: a subtype-violation error occurs if a type cast fails, and a message-not-understood error occurs if attempting to access a field or method that does not exist. We here use the terminology from Ernst et al. [48] who recently presented a formalization of a core of Dart with a focus on its type system and the causes of unsoundness.

The example code in Figure 9.1 shows how programmers use optional types in practice. Some variables and methods have type annotations (e.g. dp2persp in line 51 and operator+ in line 45), whereas in other parts (e.g. cross) the programmer chose to use dynamic (that annotation could in fact be omitted without changing the meaning of the code).

9.3 Overview

Given a Dart program with a test suite, we wish to know for any expression $e$ whether the test suite has sufficient coverage to explore all possible types $e$ may evaluate to. In our setting, a test suite is simply a finite set of program inputs. To simplify the discussion we assume that program execution is deterministically determined by the input.
CHAPTER 9. TEST COMPLETENESS

Definition 9.3.1 (Test completeness). A test suite $T$ is complete with respect to the type of an expression $e$, written $\text{COMPLETE}_T(e)$, if execution of $T$ covers all possible types $e$ may have at runtime.

The analysis we present conservatively approximates completeness (which is clearly undecidable): if the analysis reports that the completeness condition is satisfied, then this is indeed the case, but the analysis may be unable to prove completeness although the condition semantically holds.

Our approach involves four components: 1) a dynamic execution of the test suite, 2) a static dependence analysis, 3) a static type analysis, and 4) a test completeness analysis. In this section we present an overview of the approach and explain the purpose of each component, while the subsequent sections give more details about the components.

Combining Over- and Under-approximation

Our starting point is a well-known fact about dynamic execution and static analysis: executing a test suite constitutes an under-approximation of the program behavior, whereas a static analysis (in the abstract interpretation style) can provide an over-approximation. If the two agree, for example regarding the possible types of an expression $e$, then the completeness condition is trivially satisfied. Obviously, no further tests can then possibly reveal new types of $e$.

For the program code shown in lines 55–56, a simple static analysis is able to show that the only possible value of $x$ in line 56 is the $A$ object created in line 55.

```
55 x = new A();
56 x.m();
```

That information tells us not only that line 56 cannot fail with a message-not-understood runtime error, but also that the only possible callee is the $m$ method in class $A$.

For scalability reasons we wish to use only context-insensitive and path-insensitive static analysis. (An analysis is context-insensitive if it does not distinguish between different calling contexts inter-procedurally [160], and it is path-insensitive if it does not distinguish between the different paths that may lead to a given program point intra-procedurally [29].) The precision of such an analysis may suffice for large parts of a program, but programs written in dynamically typed languages often contain code that require more heavyweight program analysis techniques, such as, refinement typing [172] or context sensitivity [19]. The example in Figure 9.2 shows a typical case of value-dependent types. Here, $A$ has a $m$ method and $B$ does not. The function $g$ is overloaded as its argument determines whether it returns an object of type $A$ or $B$ (assume that some other code not shown here contains a call to $g$ with argument 0). The call in line 64 clearly always returns an $A$ object, but this fact cannot be obtained by a context-insensitive static analysis alone.
9.3. **OVERVIEW**

```plaintext
57  class A {
58      m() { ... }
59  }
60  class B {}
61
62  f() {
63      var t = 42;
64      A x = g(t);
65      x.m();
66  }
```

**Figure 9.2:** Example of value-dependent types.

(it would infer that the type is either A or B). Nor is it obvious by executing a test suite covering `f` that A is the only possibility. If the call to `g` instead returned a B object, then the program would fail at runtime, in checked mode with a subtype-violation error in line 64 and in production mode with a message-not-understood error in line 65.

### Exploiting Tests and Dependencies

Our key insight is that it is possible through a combination of lightweight static analysis and execution of a test suite to obtain completeness guarantees for the kind of code shown in the previous example.

**The dependence analysis component** One component in our system is a context-insensitive and path-insensitive dependence analysis that over-approximates the dependencies of each expression in the given program. Unlike traditional dependence analyses, this one considers both value and type dependencies. (This dependence analysis is described in more detail in Section 9.4.) For example, it infers that the type of `r` in line 74 depends (only) on the value of the parameter `a`. It also tells us that the parameter passed to `g` in line 64 has no dependencies (it is a constant). By combining these pieces of information, we see that a single concrete execution of line 64 suffices for learning all the possible types of the return value at that call. Thus, we run the test suite, and if it covers line 64 we learn that the only possible type is A—in other words, the test suite is complete with respect to the type of the return value of this particular call. Notice that the static analysis alone does not have this information; we need the concrete execution too.

**The type analysis component** The `bar` function shown in lines 76–88 is a Dart version of a Ruby

```plaintext
67  g(a) {
68      var r;
69      if (a > 0) {
70          r = new A();
71      } else {
72          r = new B();
73      }
74      return r;
75  }
```

76  var y;
77  if (p) {
78      y = 3;
79  } else {
80      y = "hello";
81  }
82  } else {
83      if (p) {
84          y + 6;
85      } else {
86          y.length;
87  }
88  }
```
example by An et al. [17]. Assume that elsewhere in the program, there are calls to bar with arguments true and false. A purely static analysis would require path sensitivity to be able to prove that y is always a number in line 84 (so that the + operation is guaranteed to succeed) and a string in line 86 (so that it has a length field).

We now show how we can obtain the necessary precision without path sensitive static analysis. The dependence analysis gives us that the type of y in line 84 and the type of y in line 86 can only depend on the value of the parameter p. As the actual value of p is unknown to the dependence analysis, we need more information to prove type safety of lines 84 and 86. For this reason, we include another component in our system: a context-insensitive and path-insensitive type analysis that provides an over-approximation of the possible types of all expressions. For the bar example, the type analysis tells us that the value of p can only be true or false. Now, notice that by combining this information with the dependence information we see that if executing the test suite includes call to bar with both these two values then the test suite is complete with respect to the type of y in lines 84 and 86. We thereby know that runtime type errors cannot occur in those lines.

The RUBYDUST technique by An et al. [17] is able to infer sound types for the bar function if all possible paths inside the function are executed by the tests. For this particular example, RUBYDUST can therefore, like our technique, infer sound and precise types using only two executions of the function. However, our technique differs in several important aspects: (1) RUBYDUST infers sound types if all possible paths are executed, but it does not know whether or not that condition is satisfied (in this example, the control-flow of the function suggests that there are four paths, but only two are possible because of the branch correlation); in contrast, our technique is able to tell that the two executions suffice. (2) In this example, the fact that the two branches are correlated is quite obvious and could be inferred by a very simple static analysis, and that information could easily be incorporated into RUBYDUST. However, our technique is capable of reaching the conclusion about the type of y without reasoning explicitly about branch correlations. (3) If bar contained additional branches with code not affecting y, then the RUBYDUST soundness criterion would require more than two executions, whereas our technique would still only require two, to show type safety for the operations on y.

As these examples suggest, we use two lightweight static analyses: a dependence analysis and a type analysis. A central part of our approach for inferring test completeness facts is combining the information provided by these analyses with the information obtained from executing the test suite. In Section 9.6 we explain how this can be achieved, a key step being a mechanism for substituting type properties according to dependencies.
Using the Inferred Completeness Facts

**Program correctness** For Dart programs we are particularly interested in test completeness at assignments, calls, and field lookup operations. As mentioned in Section 9.2, running Dart programs may encounter message-not-understood errors at property lookup operations and subtype-violation errors at implicit casts, even in programs that are accepted without warnings by the static type checker. For a programmer, knowing that his test suite is complete for such properties gives confidence about the correctness of the program.

**Test adequacy** The notion of test completeness with respect to type properties directly gives rise to a new metric for test adequacy alongside statement coverage, branch coverage, path coverage, etc. [198):

**Definition 9.3.2** (Type coverage). For a given set of expressions \( X \) in a program and a test suite \( T \), the type coverage of \( T \), denoted \( C_T(X) \), is computed as

\[
C_T(X) = \frac{|\{x \in X \mid \text{COMPLETE}_T(x)\}|}{|X|}
\]

As \( X \), one may select, for example, the set of expressions that are subjected to implicit casts in a unit of code of interest, e.g. a method, class, or library. A type coverage of 100% would ensure that the test suite is adequate for revealing all cast errors that are possible in that unit of code.

Traditional coverage metrics are not suitable for giving such guarantees. For example, full statement coverage or full branch coverage is not always sufficient, and full path coverage is impossible to achieve whenever loops are involved [198]. Other techniques that focus on branches and paths, such as, basis path testing using cyclomatic complexity [122], have similar limitations.

By selecting \( X \) as the set of receiver expressions of method calls (e.g. \( x \) in line 65) and the function expressions at function calls (e.g. \( g \) in line 64) in a unit of code, then 100% type coverage implies that the call graph has been fully exercised in that code. A programmer may use such information to guide further testing. For example, if a test suite has full statement coverage for two classes \( C \) and \( D \), maybe only 30% of the part of the call graph that involves \( C \) has been covered, while the number is 95% for \( D \), in which case the programmer should perhaps prioritize adding tests targeting \( C \) rather than \( D \). We leave it to future work to evaluate the practical usefulness of reporting such type coverage numbers to developers and how the type coverage metric correlates with other metrics and with errors; for the rest of this paper we focus on the use of test completeness in checking type safety and inferring call graphs.

**Type filtering** A well-known trick in points-to analysis and dataflow analysis is to use type tests (e.g., casts) that appear in the program as *type filters* [19, 69, 162]. The type-related completeness facts inferred by our analysis can of course be exploited for such type filtering: if we have inferred that, for example,
CHAPTER 9. TEST COMPLETENESS

y is definitely a string in line 86 then a subsequent analysis may use that fact to filter away spurious values that are not strings. As part of our evaluation in Section 9.7 we demonstrate that performing type filtering based on the completeness facts inferred by our analysis can have a substantial impact on the precision of type analysis and call graph analysis.

The lightweight type analysis we use as one of our components can directly use this mechanism itself, and increasing precision of this component may lead to more completeness facts being inferred. It therefore makes sense to repeat the entire analysis, boosting precision using type filtering based on type information inferred in the previous iteration. For now, this remains an intriguing observation, however; although it is possible to construct examples where the completeness analysis becomes more precise by such a feedback mechanism, we have not yet encountered much need in practice.

Optimizations The inferred completeness facts may also be used for optimizations, for example, removal of runtime type checks, dead-code elimination, replacement of indirect calls by direct calls, inlining, and type specialization [35, 69, 73, 96, 111]; we leave such opportunities for future work.

9.4 Dependence Analysis

As motivated in the preceding section, a key component of our technique is a static dependence analysis. Unlike traditional dependence analyses as used in, for example, optimization [53], information flow analysis [148] and program slicing [168] we are interested in both value and type dependencies. We therefore introduce a general dependence relation denoted \(<\), which is a binary relation over abstractions of runtime states at different program points (inspired by the notion of abstract dependence by Mastroeni and Zanardini [121]). For example, we have seen that the type of \(y\) at the program point after line 84 depends on the value of \(p\) in line 76, which we write as \(\text{TYPE}_{y}[84] < \text{VAL}_{p}[76]\). The dependence relation is computed using a whole-program analysis, but all dependencies are intra-procedural in the sense that they relate variables and parameters within the same function.

More generally, the dependence information we need can be expressed via different abstractions of runtime states:

Definition 9.4.1 (Type, value, and top abstraction). The type abstraction \(\text{TYPE}_{x}\) for a variable \(x\) maps a state \(\sigma\) to the runtime type of \(x\) in \(\sigma\). The value abstraction \(\text{VAL}_{x}\) instead maps \(\sigma\) to the value of \(x\) in \(\sigma\). The top abstraction \(\top\) is the identity function on program states (we use this abstraction later to express dependencies that are unknown due to analysis approximations).

The dependence relation can now be expressed as a relation of the form \(\pi[c] < \pi'[c']\) such that \(\pi, \pi' \in \Pi\) and \(c \in \mathbb{C}\) where \(\Pi\) is a family of state abstractions and \(\mathbb{C}\) is a set of program points. (The program point associated with a
9.4. DEPENDENCE ANALYSIS

entry[f, p₁,..., pₙ]  the entry of function f with parameters p₁,..., pₙ

call[f, x₁,..., xₙ]  calls function f with arguments x₁,..., xₙ

aftercall[x]  writes the returned value to x

const[x, p]  writes constant value p to x

new[x, D]  writes new D instance to x

assign[x, y]  writes value of y to x

load[x, y, f]  writes value of y.f to x

store[x, f, y]  writes value of y to x.f

is[x, D, y]  writes true to x if value of y is a subtype of D, false otherwise

binop[x, y, ⊕, z]  writes the result of the operation y ⊕ z to x

if[x]  a branch on condition x

phi[x, x₁, x₂, B]  writes φ(x₁, x₂) to x after branch

return[x]  returns the value stored in x

Figure 9.3: Primitive instructions.

given line is the one immediately after that line.) We want the dependence relation being computed to conservatively approximate all dependencies in the following sense.

Property 1 (Valid dependence relation). Given any two executions of the program that both reach a program point c inside a function f with entry program point c₀, let σ and σ' be the states at c for the two executions, respectively, and similarly let σₐ and σₐ₀ be the states at c₀ when f was entered. If there exists some state abstraction π ∈ Π where π(σ) ≠ π(σ') then there must exist some π' ∈ Π where π[c] ⪯ π'[c₀] and π'(σₐ) ≠ π'(σₐ₀).

Intuitively, if two executions disagree on π at program point c, then they must also disagree on π' at the entry of the function containing c, where π' is some state abstraction that π depends on. For example, if two executions disagree on the type of r in line 74 then they must also disagree on the value of a in line 67, so for our choice of state abstractions, the dependence relation must include the fact that the type of r depends on the value of a.

Program representation To concisely explain how the dependence analysis works we represent a program as a control flow graph (C, →) where C is now a set of nodes corresponding to primitive instructions of the different kinds as shown in Figure 9.3 and → ⊆ C × C is the intra-procedural control flow relation between nodes. We let →⁺ be the transitive closure of →.

We assume nested expressions have been flattened, so all primitive instructions operate directly on local variables (which include function parameters) and nodes are uniquely identified by the line number. Every call is represented by two nodes: a call node and an associated aftercall node. The call node is labeled by the name of the function being called (we explain in Section 9.5 how indirect function/method calls are resolved), and for simplicity we assume every call has a single callee. The program representation is in SSA
CHAPTER 9. TEST COMPLETENESS

\begin{align*}
\text{(Assign)} & \quad \text{assign}[x, y] \in C \quad \pi \in \{\text{TYPE, VAL}\} \\
& \quad \pi_x[x] \prec \pi_y[y] \\
\text{(BinOp)} & \quad \text{binop}[x, y, z] \in C \quad l \in \{y, z\} \\
& \quad \text{VAL}_x[x] \prec \text{VAL}_l[l] \\
\text{(Is)} & \quad \text{is}[x, D, y] \in C \\
& \quad \text{VAL}_x[x] \prec \text{TYPE}_y[y] \\
\text{(Load)} & \quad c = \text{load}[x, y, f] \in C \quad \pi \in \{\text{TYPE, VAL}\} \\
& \quad \pi_x[x] \prec \top[\text{entry}_c] \\
\text{(New)} & \quad c = \text{new}[x, D] \in C \\
& \quad \text{VAL}_x[x] \prec \top[\text{entry}_c] \\
\text{(Def)} & \quad \pi \in \{\text{TYPE, VAL}\} \\
& \quad \pi_x[x] \prec \top[\text{entry}_c] \\
\text{(Phi)} & \quad \phi[x, x_1, x_2, B] \in C \quad \pi \in \{\text{TYPE, VAL}\} \\
& \quad \text{if}[y] \in B \quad l \in \{x_1, x_2, y\} \\
& \quad d = \begin{cases} 
\pi_l[l] & \text{if } l \in \{x_1, x_2\} \\
\text{VAL}_l[l] & \text{if } l = y 
\end{cases} \\
& \quad \pi_x[x] \prec d \\
\text{(Call)} & \quad c = \text{call}[f, x_1, \ldots, x_n] \in C \quad c_e = \text{entry}[f, p_1, \ldots, p_n] \in C \\
& \quad c_a = \text{aftercall}[x] \in C \quad c_r = \text{return}[y] \in C \\
& \quad c \mapsto c_a \\
& \quad c_e \mapsto c_r \\
& \quad d = \begin{cases} 
\text{VAL}_x[x_i] & \text{if } \pi_y[y] \prec \text{VAL}_p_i[c_e] \\
\text{TYPE}_x[x_i] & \text{if } \pi_y[y] \prec \text{TYPE}_p_i[c_e] \\
\top[\text{entry}_c] & \text{if } \pi_y[y] \prec \top[c_e] 
\end{cases} \\
& \quad \pi_x[x] \prec d 
\end{align*}

Figure 9.4: Rules for type and value dependence.

form \cite{43}, and every \(\phi\)-node \(\phi[x, x_1, x_2, B]\) is associated with the set \(B\) of all \(\text{if}\)-nodes that dominate \(x_1\) or \(x_2\). Due to the use of SSA, each variable name uniquely corresponds to a node where the variable is assigned its value (where parameters are considered to be assigned at the function entry node), and we sometimes use the variable name to denote the program point after that node. Furthermore, we let \(\text{entry}_c\) denote the entry program point of the function containing node \(c\). The meaning of the other kinds of nodes is straightforward.

**The dependence analysis** We compute the dependence relation \(\ll\) as the smallest fixpoint of the rules shown in Figure 9.4 (in addition, \(\ll\) is reflexive and transitive).
The **Assign** rule for an assignment \texttt{assign[x,y]} establishes that the type and value of \(x\) depend on the type and value of \(y\), respectively. More precisely, for \(\pi = \text{TYPE}_x\) we see that the type of \(x\) at the program point where \(x\) has been defined, i.e. \(\text{TYPE}_x[x]\), depends on the type of \(y\) at the program point where \(y\) has been defined, i.e. \(\text{TYPE}_y[y]\), and similarly for the value. The **BinOP** rule shows that the value of the result depends on the values of the two operands; however, the *type* of the result is always fixed by the operator so it does not depend on the operands. The **Is** rule for the instruction \texttt{is[x,D,y]} shows that the *value* of \(x\) depends on the *type* of \(y\). (The *type* of \(x\) is always boolean.)

A rule for \texttt{const[x,p]} can be entirely omitted, since neither the type nor the value of constants depend on anything.

To keep the analysis lightweight we choose to entirely avoid tracking dependencies on the heap. This is modeled using the \(\top\) abstraction. In the **Load** rule for \texttt{load[x,y,f]} the type and value of \(x\) conservatively depend on the entire state at the entry of the function, i.e. \(\top[\text{entry}_f]\). With this coarse model of the heap, it is safe to omit a rule for \texttt{store[x,f,y]}.

The **New** rule captures the fact that the value being produced by \texttt{new[x,D]} is unknown to the analysis (whereas the *type* is trivially \(D\), without any dependencies).

The **Phi** rule models dependencies that are due to the data and control flow at branches, which are represented by \texttt{if[y]} and \texttt{phi[x,x_1,x_2,B]} nodes. First, the type and value of \(x\) depend on the type and value, respectively, of both \(x_1\) and \(x_2\) (as reflected by the case \(l \in \{x_1,x_2\}\)). Second, the type and value of \(x\) also depend on the value of the branch conditions in \(B\) (corresponding to the case where \(l = y\)). It is standard to compute control flow dependence using post-dominance information [43]. A node \(c \in C\) is control flow dependent on \(c' \in C\) if \(c'\) determines whether \(c\) is executed, i.e. (i) there is a path from \(c\) to \(c'\) where every node in the path is post-dominated by \(c'\), and (ii) \(c\) is not post-dominated by \(c'\). Since we assume the control flow graph to be on SSA form, control flow dependence for variables is already explicit in the \(\phi\)-nodes.

The **Call** rule exploits the dependencies computed between the return value and the parameters of the callee, to obtain dependencies between the resulting value and the arguments at the call site. In this rule, \(c\) is a **call** node and \(c_a\) is the associated **aftercall** node at the call site, while \(c_e\) is the callee **entry** node and \(c_r\) is the **return** node. The dependencies for the result \(x\) are found simply by substituting the dependencies of the return value \(y\) according to the parameter binding.

Finally, the **Def** rule has the consequence that the type or value of a variable \(x\) at any program point \(c\) can always be determined from its type or value at the definition of \(x\), and the top abstraction at \(c\) depends on the top abstraction at the entry of the function. This rule is strictly not necessary, but it simplifies the use of \(<\) in Section 9.6.

Notice that even though type abstractions are conservative approximations of value abstractions, all combinations of dependencies between types and values can occur; in particular, it is possible that the *value* of a variable \(x\)
CHAPTER 9. TEST COMPLETENESS

\[
\forall (a_1, \ldots, a_n) \in \pi_1(\sigma_c) \times \cdots \times \pi_n(\sigma_c) : \\
\exists t \in T, \sigma \in [t]_c : \forall i = 1, \ldots, n : \pi_i(\sigma) = a_i \\
T \vdash \pi_1, \ldots, \pi_n[c]
\]

\[\text{(Inductive-ToEntry)}\]  
\[\{\pi'_1, \ldots, \pi'_m\} = \{\pi' | \pi_i[c] \triangleleft \pi'[\text{entry}_c]\} \quad T \vdash \pi'_1, \ldots, \pi'_m[\text{entry}_c]\]

\[\text{(Inductive-ToCall)}\]  
\[c = \text{call}[f, x_1, \ldots, x_a] \in \mathbb{C} \quad \pi \in \{\text{TYPE}, \text{VAL}\} \]
\[c_a = \text{aftercall}[x] \in \mathbb{C} \quad T \vdash \pi'_1, \ldots, \pi'_m[c]\]
\[c_e = \text{entry}[f, p_1, \ldots, p_a] \in \mathbb{C} \quad \{\pi'_1, \ldots, \pi'_m\} = \{\text{VAL}_{x_i} | \pi_y[y] \triangleleft \text{VAL}_{p_i}[c_e]\} \cup \]
\[\{\text{TYPE}_{x_i} | \pi_y[y] \triangleleft \text{TYPE}_{p_i}[c_e]\} \cup \{\top | \pi_y[y] \triangleleft \top[c_e]\}\]
\[c \rightarrow c_a \quad \{\text{val}_x \rightarrow \text{val_a}\} \quad \{\text{null} \rightarrow \text{null}\}\]
\[c_e \rightarrow^+ c_r \quad T \vdash \pi_x[c_a]\]

Figure 9.5: Rules for test completeness.

depends on the type of another variable y (due to the Is rule), and that the type of x depends on the value of y (due to the PHI rule).

**Proposition 1** (Soundness of computed dependencies). For any program, the relation \(\triangleleft\) defined by the least fixpoint of the dependence rules satisfies Property 1.

**Example** For the example in lines 57–75, the dependence analysis infers, in particular, that \(\text{TYPE}_x[74] \triangleleft \text{VAL}_a[67]\) but not \(\text{TYPE}_x[74] \triangleleft \top[67]\), and that there is no \(\pi\) such that \(\text{TYPE}_x[64] \triangleleft \pi[62]\). Therefore the CALL rule gives that there is also no \(\pi'\) such that \(\text{TYPE}_x[x] \triangleleft \pi'[62]\), i.e., the type of x does not depend on the input to f.

9.5 Type analysis

The static type analysis component serves two purposes: it resolves indirect calls for the dependence analysis in Section 9.4 and it computes an overapproximation of the possible types of every expression, which we use in the test completeness analysis in Section 9.6. The type analysis is simply a context-insensitive and path-insensitive subset-based analysis that simultaneously
tracks functions/methods and types. This is a well-known analysis technique (see e.g. the survey by Sridharan et al. [162]), so due to the limited space we omit a detailed description of how this component works. The heap is modeled using allocation-site abstraction [36], and we use flow-sensitivity only for local variables. We choose a lightweight analysis for the reasons given in Sections 9.1 and 9.3.

In Dart, all values are objects, including primitive values and functions. The analysis abstracts primitive values by their type (e.g. bool or String) and treats each function and method as having its own type. As output, for every program point $c$ the analysis returns an abstract state $\hat{\sigma}_c$ that over-approximates all concrete states that may appear at $c$ when the program is executed. We define $\hat{\sigma}_c(e)$ to be the set of types the expression $e$ may evaluate to according to $\hat{\sigma}_c$.

**Example** Consider a call to the `cross` function from Figure 9.1

```plaintext
89 x = cross(y,z);
```

in a context where the type analysis finds that the type of $y$ is either `vec3` or `vec2` and the type of $z$ is `vec3`. That is, $\hat{\sigma}_c(y) = \{\text{vec3}, \text{vec2}\}$ and $\hat{\sigma}_c(z) = \{\text{vec3}\}$ where $c$ is the program point at the call. (This example uses a direct call, so the connection between the call and the callee is trivial.) Assuming there are other calls to `cross` elsewhere in the program, the context-insensitive type analysis has only imprecise information about the possible return types for this function. However, the dependence analysis has inferred that the return type only depends on the types of the parameters. This allows the test completeness analysis, presented in the following section, to conclude that two executions suffice to cover all possible types of $x$: one where $y$ has type `vec3` and one where it has type `vec2`. This example demonstrates the power of combining dependence analysis and type analysis.

Now consider a slight modification of the example, where the type of both $y$ and $z$ is either `vec3` or `vec2`. Our technique then requires four executions of the call to ensure that all relevant combinations are covered. This suggests that relational dependence and type information may be valuable: since we only obtain test completeness guarantees at a call when all combinations of dependencies have been exercised, it may be beneficial to know that only some combinations are relevant. This is an opportunity to explore in future work.

### 9.6 Test completeness

In this section we present a proof system for test completeness, i.e., for proving $\text{complete}_T(x)$ for a given variable $x$ in some program with a test suite $T$ (see [Definition 9.3.1]). (We use the program representation from Section 9.4 so we assume without loss of generality that the expression of interest is simply a variable $x$.)
CHAPTER 9. TEST COMPLETENESS

The proof rules, shown in Figure 9.5, rely on three ingredients: the execution of $T$, the dependence relation $\prec$ from Section 9.4, and for each program point $c$ the abstract state $\hat{\sigma}_c$ produced by the type analysis from Section 9.5. Each test input $t \in T$ gives rise to a program execution, which can be seen as a sequence of concrete states at different program points. We write $\llbracket t \rrbracket_c$ for the set of states encountered at program point $c$ when $t$ is executed.

A simple judgment $T \vdash \pi[c]$ intuitively means that the test suite $T$ is complete with respect to the state abstraction $\pi$ (see Definition 9.4.1) at program point $c$. In other words, the judgment holds if for every abstract value in $\pi(\sigma_c)$ where $\sigma_c$ is a runtime state at $c$ in some execution of the program, there exists a test in $T$ that also encounters that abstract value at $c$. In particular, we are interested in the type abstraction $\text{TYPE}_x$ and the program point where $x$ is defined. We therefore aim for the following connection between proof judgments and test completeness for type properties.

**Proposition 2** (Soundness of test completeness analysis).

$$T \vdash \text{TYPE}_x[x] \text{ implies } \text{COMPLETE}_T(x)$$

To show $\text{COMPLETE}_T(x)$ for some variable $x$, we thus attempt to derive $T \vdash \text{TYPE}_x[x]$.

More generally, judgments may involve multiple state abstractions: $T \vdash \pi_1, \ldots, \pi_n[c]$. The intuitive interpretation of such a judgment is that $T$ is complete with respect to the product of the state abstractions $\pi_1, \ldots, \pi_n$ at program point $c$.

We now briefly describe the rules from Figure 9.5.

The Base rule corresponds to the observation in Section 9.3 that completeness sometimes can be shown using the information from the type analysis. To understand the rule in its full generality, consider first this special case:

$$\forall a \in \hat{\sigma}_c(x) : \exists t \in T, \sigma \in \llbracket t \rrbracket_c : \text{TYPE}_x(\sigma) = a$$

This rule states that a test $T$ is complete for the type of $x$ if for all the types $a \in \hat{\sigma}_c(x)$ that can be observed according to the type analysis (Section 9.5), there exists an execution that reaches program point $c$ with a concrete state $\sigma$ where $x$ has type $a$.

A first step from this special case toward the general Base rule is to generalize it from using the type abstraction $\text{TYPE}$ to use any state abstraction $\pi \in \Pi$. For this, we introduce notation for lifting a state abstraction $\pi$ to operate on abstract states produced by the type analysis: we define $\pi(\hat{\sigma}_c) = \{ \pi(\sigma_c) | \hat{\sigma}_c \text{ is the type abstraction of } \sigma_c \}$. Note that $\pi(\hat{\sigma}_c)$ can be infinite, for example if $\pi = \top$. This corresponds to a completeness condition that requires an infinite set of executions, so in that situation we can simply give up. Another interesting case is when $\pi = \text{VAL}_x$ for some variable $x$. This
occurs when our analysis is required to prove $T \vdash \text{val}_p[c]$ in the example in lines 76–88 in Section 9.3 (where $c$ is the program point at the entry of \texttt{bar}). Because $p$ is a boolean according to the type analysis, $\text{val}_p(\hat{\sigma}_c)$ contains only the two values \texttt{true} and \texttt{false}, so two executions suffice to prove $T \vdash \text{val}_p[c]$.

The last step to the general \textsc{Base} rule is to account for judgments with multiple state abstractions $\pi_1, \ldots, \pi_n$. For this case, we simply require observations of all combinations of abstract values in $\pi_1(\hat{\sigma}_c) \times \cdots \times \pi_n(\hat{\sigma}_c)$.

The \textsc{Inductive-ToEntry} rule uses the dependence relation $\prec$ to prove a completeness property at a program point $c$ by a completeness property at the function entry $\text{entry}_c$.

The \textsc{Inductive-ToCall} rule mimics the \textsc{Call} rule from the dependence analysis (Figure 9.4), substituting completeness properties at calls according to the dependencies of the callee. This corresponds to the reasoning used for the example in Section 9.3 for the call in line 64.

Completeness proofs can be obtained by proving ground facts using the \textsc{Base} rule, and then deriving all the others using the inductive rules whenever their premises are satisfied. This procedure terminates, as it works intra-procedurally and the induction proceeds in program order.

Using test completeness for type filtering As suggested in Section 9.3, the completeness facts being inferred can be plugged in as type filters in a type analysis. For example, we can run the type analysis described in Section 9.3 a second time, now using type filtering.

Let $X$ be the set of observed runtime types for a variable $x$ where $\text{complete}_T(x)$. By Proposition 2 this is the set of all the possible types $x$ may have in any execution. During the type analysis, for every abstract state, we can filter out those types of $x$ that are not in $X$. Removing those spurious types may improve precision throughout the program.

### 9.7 Experimental Evaluation

Our experimental evaluation addresses the following four research questions.

**Q1** To what extent is the technique capable of showing \textit{test completeness} for realistic Dart programs and test suites? More specifically, what is the \textit{type coverage} being computed for such programs and test suites?

**Q2** Does the hybrid static/dynamic approach result in better precision for soundly checking absence of runtime type errors (message-not-understood and subtype-violation errors) and producing precise call graph information (outdegree of calls, reachable functions), compared to an analysis that does not exploit the test suites?

**Q3** How important is the dependence analysis, which is a central component in our technique, for the precision of the computed type coverage?
Q4 In situations where the analysis cannot show test completeness, is the cause likely to be insufficient tests or lack of precision in the static analysis components?

Implementation and benchmarks Our implementation, GOODENOUGH, consists of the four components listed in Section 9.3, including the type filtering technique that uses completeness facts to improve the type analysis. For logging the runtime types of expressions we use a small runtime instrumentation framework inspired by Jalangi [158]. We keep the runtime overhead low by only observing types at variable reads and calls in the application code, and we do not instrument libraries.

The evaluation is based on 27 benchmarks, ranging from small command-line and web applications to sample applications that demonstrate the use of libraries. The benchmarks are listed in the first column of Table 9.1. The second column shows the size of each benchmark in number of lines of code (excluding unreachable library code). We use the existing test suites available for the command-line applications; for the web applications we obtain test suites by manually exercising the applications via a browser for one minute each. To obtain some degree of confidence that the static analysis components are sound, we check that all runtime observations agree with the static analysis results.

The implementation, benchmarks, test suites, and details of the experimental results are available online.

Q1: Type coverage

To answer the first question we run GOODENOUGH on each benchmark with its test suite $T$ and measure the type coverage $C_T(X)$, where $X$ is chosen as the set of all the expressions in the program. (Due to Proposition 2, the type coverage numbers we report are safe in the sense that they under-approximate the semantic definition of type coverage given by Definition 9.3.2.) The results are shown in the third column of Table 9.1.

Type coverage is generally high, with an average of 81%. In other words, the analysis is able to guarantee for 81% of the expressions that all types that can possibly appear at runtime are in fact observed by execution of the test suite.

Q2: Type errors and call graphs

To investigate whether our hybrid static/dynamic approach to test completeness can be useful for showing absence of type errors and inferring sound call graph information, we compare GOODENOUGH with a variant that does not exploit the

\[http://www.brics.dk/goodenough/\]
Table 9.1: Results.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>$C_T(X)$</th>
<th>MNU (%)</th>
<th>UIC (%)</th>
<th>RF (%)</th>
<th>CGS (%)</th>
<th>PI (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_star</td>
<td>1338</td>
<td>83.6</td>
<td>-</td>
<td>-</td>
<td>0.31</td>
<td>7.53</td>
<td>77.27</td>
</tr>
<tr>
<td>archive</td>
<td>2139</td>
<td>87.8</td>
<td>11.11</td>
<td>-</td>
<td>-</td>
<td>0.17</td>
<td>3.45</td>
</tr>
<tr>
<td>bad_aliens</td>
<td>1716</td>
<td>77.6</td>
<td>4.26</td>
<td>18.18</td>
<td>-</td>
<td>0.16</td>
<td>22.22</td>
</tr>
<tr>
<td>dartbox2d</td>
<td>8732</td>
<td>70.9</td>
<td>4.13</td>
<td>10.10</td>
<td>3.87</td>
<td>3.32</td>
<td>22.52</td>
</tr>
<tr>
<td>csslib</td>
<td>6972</td>
<td>82.4</td>
<td>6.45</td>
<td>3.08</td>
<td>1.16</td>
<td>1.85</td>
<td>8.95</td>
</tr>
<tr>
<td>Dark</td>
<td>1506</td>
<td>78.4</td>
<td>25.88</td>
<td>1.41</td>
<td>-</td>
<td>0.55</td>
<td>10.63</td>
</tr>
<tr>
<td>dart_regexp_tester</td>
<td>3095</td>
<td>84.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dartrix</td>
<td>1594</td>
<td>80.2</td>
<td>-</td>
<td>0.50</td>
<td>1.49</td>
<td>66.67</td>
<td>-</td>
</tr>
<tr>
<td>dartrocket</td>
<td>11651</td>
<td>74.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>frappe</td>
<td>5712</td>
<td>77.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>graphviz</td>
<td>3275</td>
<td>80.7</td>
<td>-</td>
<td>6.25</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>markdown</td>
<td>1901</td>
<td>89.9</td>
<td>-</td>
<td>1.98</td>
<td>8.57</td>
<td>63.72</td>
<td>-</td>
</tr>
<tr>
<td>qr</td>
<td>6118</td>
<td>87.3</td>
<td>-</td>
<td>-</td>
<td>0.88</td>
<td>16.00</td>
<td>-</td>
</tr>
<tr>
<td>petitparser</td>
<td>3882</td>
<td>69.0</td>
<td>-</td>
<td>0.31</td>
<td>3.68</td>
<td>33.62</td>
<td>-</td>
</tr>
<tr>
<td>pop_pop_win</td>
<td>14590</td>
<td>71.9</td>
<td>-</td>
<td>0.04</td>
<td>0.86</td>
<td>17.45</td>
<td>-</td>
</tr>
<tr>
<td>rgb_cube</td>
<td>1079</td>
<td>86.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>solar</td>
<td>657</td>
<td>85.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>solar3d</td>
<td>7813</td>
<td>86.5</td>
<td>71.74</td>
<td>6.90</td>
<td>2.89</td>
<td>4.65</td>
<td>17.65</td>
</tr>
<tr>
<td>spaceinvaders</td>
<td>1292</td>
<td>75.9</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>speedometer</td>
<td>357</td>
<td>91.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>spirodraw</td>
<td>1220</td>
<td>86.5</td>
<td>4.76</td>
<td>5.56</td>
<td>-</td>
<td>4.54</td>
<td>-</td>
</tr>
<tr>
<td>sunflower</td>
<td>526</td>
<td>86.4</td>
<td>75.00</td>
<td>33.33</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>todomvc_vanilladart</td>
<td>3967</td>
<td>78.4</td>
<td>2.08</td>
<td>5.00</td>
<td>0.15</td>
<td>0.93</td>
<td>7.58</td>
</tr>
<tr>
<td>vote</td>
<td>7867</td>
<td>76.7</td>
<td>17.29</td>
<td>7.69</td>
<td>15.16</td>
<td>18.09</td>
<td>31.10</td>
</tr>
<tr>
<td>three</td>
<td>7432</td>
<td>83.1</td>
<td>42.05</td>
<td>5.29</td>
<td>20.93</td>
<td>7.91</td>
<td>8.00</td>
</tr>
<tr>
<td>stats</td>
<td>1560</td>
<td>84.2</td>
<td>35.71</td>
<td>9.09</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>tutorial_kanban</td>
<td>940</td>
<td>70.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Runtime observations from the test suite. We select five metrics for measuring the precision:

MNU (*message-not-understood*): number of warnings about possibly failing access to a field or a method;

UIC (*unsafe implicit cast*): number of warnings about potential subtype violations at implicit downcasts (only relevant for checked mode execution);

CGS (*call-graph size*): number of edges in the call graph;

RF (*reached functions*): number of possibly reached functions;

PI (*polymorphic invocation*): number of call sites with multiple potential callees.

The columns MNU, UIC, RF, CGS, and PI in Table 9.1 show the percentage change of the resulting numbers for the hybrid analysis compared to the fully-static analysis.
CHAPTER 9. TEST COMPLETENESS

We see that the hybrid analysis is able to greatly improve the precision for some benchmarks, while it gives marginal or no improvements in others. This is not surprising, considering the different nature of the benchmarks and their programming style. For a few benchmarks, the naive fully-static analysis already obtains optimal precision for MNU or UIC, leaving no room for improvements with the hybrid technique. Interestingly, 19 out of 27 benchmarks improve for at least one metric, meaning that the hybrid approach shows advantages across a variety of programs. This confirms that type filters based on inferred completeness facts can have a substantial impact on the precision of the type analysis, as discussed in Section 9.3.

An important design decision has been that the static analysis components are context- and path-insensitive to keep the system lightweight. By manually studying some of the examples where our hybrid approach obtains high precision, we see that a fully-static alternative would require a high degree of context sensitivity to reach the same conclusions, and it is well known that such analyses do not scale well.

For example, a context-insensitive static analysis is insufficient to reason precisely about the return type of functions similar to `cross` from the `vector_math` library discussed in Section 9.1. In contrast, Goodenough finds 19 calls to `cross` where the test suite is complete, which enables the static type analysis to filter the inferred types for the return of the calls and thereby avoid several spurious warnings.

Another example is found the Dark benchmark:

```java
var canvas;
GL.RenderingContext gl;
void startup() {
  canvas = querySelector("#game");
  gl = canvas.getContext("webgl");
  if (gl==null)
    gl = canvas.getContext("experimental-webgl");
  if (gl==null) {
    crash("No webgl", "Go to [...]”);
    return;
  }...
}
void start(Level _level) {
  ...;
  GL.Texture colorLookupTexture = gl.createTexture();
  ...;
}
```

According to the type analysis, `canvas` can be any subtype of the `HTML Element` class, and the calls to `getContext` return objects of type `CanvasRenderingContext2D` or `RenderingContext`. With this information, one would conclude that in checked mode execution subtype-violation errors may occur at the two
9.7. EXPERIMENTAL EVALUATION

assignments to gl, and in production mode the `createTexture` invocation may result in a message-not-understood error since `createTexture` is not declared by `CanvasRenderingContext2D`. Although `canvas` can be an arbitrary HTML element, the call is monomorphic. The results of the calls to `getContext` only depend on the value of the arguments, which are constant. Executing the two branches dynamically is enough to prove type completeness for both calls, which reveals that both calls return objects of type `RenderingContext`, not `CanvasRenderingContext2D`. Type filtering uses this fact to remove spurious dataflow. This example also shows the importance of test coverage: a single execution does not suffice to cover all types at both calls.

While lightweight type analysis (e.g. the one from Section 9.5) cannot reach the conclusion that only `RenderingContext` objects may be returned by `getContext`, a more precise analysis could. In this case, 2-CFA context sensitivity would be needed if we did not exploit the dynamic executions, since the argument to `getContext` is passed to the native function `getContext_Callback_2_`. In other situations, 5-CFA would be necessary to reach the same precision as our hybrid approach.

To correctly handle the example above with a fully static analysis, it would additionally be necessary to precisely model the implementation of `getContext_Callback_2_`: it returns a `RenderingContext` object when the argument is "webgl" or "experimental-webgl", and a `CanvasRenderingContext2D` object when the argument is "2d". The Dart SDK contains more than 10 000 external functions, many of which are highly overloaded in such ways. As an example, the method `getParameter` of `RenderingContext` from the WebGL API returns 15 (unrelated) types depending on which of 87 constants the argument matches. Another example is the library function `Element.tag`, which calls a native function that has more than 200 cases involving different types, which a fully static analysis would need detailed models for. Thus, our hybrid approach avoids a massive task of modeling such details, as it only needs the dependence information.

We also observe that test completeness is useful for reasoning about the types of numeric expressions without the need for context sensitivity, constant folding and constant propagation. Indeed, the type of many numerical expressions only depends on the types of the operands. The following simple example from Dark is one of many where cast failures may occur in checked mode execution according to a naive type analysis:

```dart
void move(double iX, double iY, double passedTime) {
  ...
  double frictionXZ = pow(0.0005, passedTime);
  double frictionY = pow(0.2, passedTime);
}```

The declared return type of `pow` is `num`, which has subtypes `int` and `double`. Covering the two calls to `bar` with a single execution is enough to ensure completeness.

Q3: Dependence analysis

The dependence analysis finds that 73% of all function return expressions have no type dependencies on the function entry state. This means that a single execution of those expressions is enough to cover all their possible types. At other 16% of the return expressions, the return type depends on the types or values of one or more parameters but not on the heap, and for the remaining 11% the dependence analysis finds dependencies on the heap.

To investigate the importance of the dependence analysis, we have repeated the experiments from Q1 and Q2 using a weaker dependence analysis that does not exploit the call graph produced by the type analysis. Instead, the type and value dependencies of the return value at the call site of all indirect calls are conservatively set to top. This single restriction causes a reduction in the number of completeness facts being proven and a significant degradation in the precision metrics: type coverage drops from 81% to 77%, and the precision of type error detection and call graph construction drops to almost the same as the naive fully-static analysis (leaving only a 0.7% improvement of the UIC metric for the Dark benchmark). These results demonstrate that the dependence analysis is indeed a critical component.

Q4: Reasons for inability to show test completeness

To answer Q4, we have investigated the behavior of our analysis in selected places where GOODENOUGH is not able to prove test completeness. It is naturally difficult to quantify the reasons, yet some patterns are clear. Our preliminary investigation has identified two main reasons: missing coverage by the test suites, and too coarse heap modeling in the dependence analysis. On the other hand, imprecision in the type analysis does not appear to be significant.

As we have already seen in the Dark example in the answer to Q2, coverage is indeed important to prove test completeness. In that case two executions on specific browser versions were needed. In many other cases, we see that simply improving the statement coverage of the test suite would likely improve the type coverage significantly.

For this first design of our technique, we have chosen a simplistic approach to dependence analysis, in particular, by using the top abstraction at all field load operations. Consider the following example, which uses the `parse` function from a Dart HTML5 parser.

---

The `parse` function always returns a tree of HTML elements whose structure is determined solely by the input to `parse`. A dependence analysis that is able to track dependencies on the heap could in principle determine that the lookup of `firstChild` always has the same type, or equivalently, that the expression has no type dependencies. We see similar cases in many of the libraries being used in the benchmarks. This observation suggests that extending the dependence analysis to also track dependencies involving the heap may be a promising direction for future work.

### 9.8 Related Work

#### Types in dynamic languages

Several techniques have been proposed to statically reconstruct precise type information for dynamic languages, for example, for optimization purposes \cite{73, 96}. The type inference approach by An et al. \cite{17} is discussed in Section 9.3. Dependent types and flow-based type checking can deal with common dynamic patterns, such as, value-based overloading \cite{40, 71, 105, 112, 164, 172}. These techniques require programmers to provide detailed type annotations. Advanced static analysis has been used to precisely infer and check types in JavaScript \cite{19}, however, this has not yet been proven to scale to realistic programs.

#### Test adequacy and coverage metrics

Numerous criteria for deciding whether a test suite is adequate and metrics for measuring coverage have been proposed (the key concepts being described by Zhu et al. \cite{198}), so due to the limited space we can only mention the high-level relations to our work. The focus in the literature is typically on using coverage metrics to guide the effort of testing programs. Common to most of those techniques, including the seminal work by Goodenough and Gerhart \cite{64}, is that they do not come with tool support for checking whether a test suite has adequate coverage to guarantee properties like absence of runtime type errors. The general idea of “test completeness” has a long history \cite{34, 64, 79, 80}, but until now without the connection to types in dynamic languages.

#### Hybrid analysis

Other hybrids of static and dynamic analysis have been developed to combine the strengths of both parts \cite{30, 42, 72, 100, 187}, or to use dynamic executions to guide static analysis \cite{134, 153, 178}. Notably, the method by Yorsh et al. \cite{187} uses automated theorem proving to check that a generalized set of concrete states obtained by dynamic execution covers...
all possible executions. This involves a notion of abstraction-based adequacy, which is reminiscent of our notion of type coverage. Predicate-complete testing is another related idea [28]. Most hybrid analysis create tests on-the-fly and do not exploit preexisting test suites, but for the dynamic programming patterns that are our primary target, automatically creating useful tests is by itself a considerable challenge. One approach that does use test suites and is designed for a dynamic programming language is blended analysis [178], however, it is unsound in contrast to our technique.

**Dependence analysis** The concept of dependence used in Section 9.4 appears in many variations in the literature. From a theoretical point of view, our definitions fit nicely into the framework proposed by Mastroeni and Zanardini [121]. In the field of information flow [62], dependence plays an important role in reasoning about non-interference. In program slicing [179] and compiler optimization (e.g. [53]), program dependence graphs [101] model the dependencies between values used in different statements. The novelty of the dependence analysis in Section 9.4 is to capture dependencies not only between values but also between types.

### 9.9 Conclusion

We have presented the notions of test completeness and type coverage together with a hybrid program analysis for reasoning about adequacy of test suites and proving type-related properties. Moreover, we have demonstrated using our implementation Goodenough how this analysis technique is suitable for showing absence of type errors and inferring sound call graph information, in Dart code that is challenging for fully-static analysis.

Many interesting opportunities for future work exist. We plan to explore how better heap modeling in the dependence analysis and relational dependence and type analysis can improve precision further. In another direction, we intend to perform an empirical study of how the type coverage metric correlates with other metrics and with programming errors, and to use type coverage to guide automated test generation. It may also be interesting to use the type analysis results for program optimizations and to apply our approach to other dynamic languages.

**Acknowledgments**

This work was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 647544).
Chapter 10

Repairing Event Race Errors by Controlling Nondeterminism


Abstract

Modern web applications are written in an event-driven style, in which event handlers execute asynchronously in response to user or system events. The nondeterminism arising from this programming style can lead to pernicious errors. Recent work focuses on detecting event races and classifying them as harmful or harmless. However, since modifying the source code to prevent harmful races can be a difficult and error-prone task, it may be preferable to steer away from the bad executions.

In this paper, we present a technique for automated repair of event race errors in JavaScript web applications. Our approach relies on an event controller that restricts event handler scheduling in the browser according to a specified repair policy, by intercepting and carefully postponing or discarding selected events. We have implemented the technique in a tool called EVENTRACECOMMANDER, which relies entirely on source code instrumentation, and evaluated it by repairing more than 100 event race errors that occur in the web applications from the largest 20 of the Fortune 500 companies. Our results show that application-independent repair policies usually suffice to repair event race errors without excessive negative impact on performance or user experience, though application-specific repair policies that target specific event races are sometimes desirable.
10.1 Introduction

Modern application development has largely moved to platforms requiring event-driven programming, using web browsers and mobile platforms. The event-driven model is well-suited to the needs of today’s interactive programs, which must perform high-latency network requests to send and receive requested data while remaining responsive to user input. However, as studied in recent work [78, 91, 131, 138, 146, 197], this programming style can cause pernicious nondeterminism errors, which can lead to crashes, lost user data, and malfunctioning user interfaces.

Recent work has attacked this nondeterminism problem through tools for detecting event races, where application behavior may differ depending on the order in which event handlers are executed. For web applications, event race detectors are capable of finding errors in real-world, deployed web applications [146]. Further, tools such as R4 [91] can filter away warnings about races that do not affect the external behavior of web applications.

Despite these advances, the output of state-of-the-art event race detectors is often still not practical. Diagnosing the root cause of an event race in a real-world web application can require a significant effort—it often requires deciphering complex event sequences, and it can be difficult to classify how harmful a reported race is, especially for non-expert users of the tools. In addition, preventing such races may require introducing complex synchronization into the code, an arduous task since the web platform provides few mechanisms for synchronizing across event handlers. Manually devising such a fix is often not worth the effort, particularly for minor errors, when considering that fixes sometimes have unforeseen consequences [186]. Better techniques are needed to reduce the cost of fixing event race errors.

In this work, we explore automated repair of event race errors in web applications. Automated repair holds great promise for addressing the aforementioned drawbacks of event race detectors. If event race errors can be automatically repaired, without requiring developers to deeply understand root causes, the errors may be avoided more often. There is a wide body of work on repairing races in multi-threaded programs [37, 93, 94, 97, 102, 109, 112, 113, 141, 144, 171, 177, 180, 193, 195], but relatively little work on repair for event races. Wang et al. [176] have proposed a repair technique for event races in web applications, but it has significant limitations in the types of races it can handle (see Section 10.8).

Our approach builds on an event controller that restricts event handler scheduling in the browser according to a specified repair policy, by intercepting and carefully postponing or discarding selected events. Restricting schedules dynamically to avoid bad orderings is a well-known approach to automated repair of races in the context of shared-memory concurrency races, but to our knowledge it has not previously been applied to event-driven applications. An important property of our approach is that the event controller is built entirely
by instrumenting the web application code. Most event race detection tools for JavaScript work using modified browsers, which is reasonable for detecting races, but not for automated repair, as the code must run on end-user browsers. In spite of relying solely on light-weight instrumentation, our approach is general enough to repair common types of event races, although some event race errors cannot be repaired by merely restricting the nondeterminism (see Section 10.5).

Given this event controller, the question remains of what policies are required to repair races in practice. A policy specifies which events to postpone or discard, so choosing an appropriate policy requires knowledge about which event orders are good and which are bad. We observe that many races in web applications can be prevented with a small collection of application-independent policies. For example, application developers often expect AJAX response handlers to execute in a first-in first-out (FIFO) order, and that the page completes loading before the user interacts with it: many errors occur when these assumptions are violated. Our application-independent policies enforce these assumptions, yielding a simple method for avoiding many races.

Application-independent policies are easy to apply, but may impact performance or user experience negatively. For example, delaying all user events until after the page has loaded may make the page appear sluggish, and in fact many user interactions during page load may be perfectly safe (i.e., they cannot lead to harmful races). We show that these problems can be alleviated using application-specific policies, which can be designed, for example, by specializing an application-independent policy to a particular web application.

In summary, the contributions of this paper are as follows.

- We demonstrate that most errors involving event races in JavaScript web applications can be repaired automatically, using light-weight instrumentation to steer the nondeterminism according to a specified repair policy.

- We propose the use of application-independent policies, which can be specialized as needed to avoid excessive delay in event processing, or to target specific event races reported by existing race detection tools.

- We evaluate our approach based on an implementation called EVENT-RACECOMMANDER, by repairing 117 event race errors in the websites of the 20 largest companies from the Fortune 500 list. Our results show that 94 of the errors can be repaired using application-independent policies, mostly without excessive negative impact, and that application-specific policies can alleviate the undesirable effects when this is not the case.
10.2 Motivating Example

Figure 10.1 shows a small web application for browsing through galleries of images, consisting of three files. File index.html defines a top-level page with two buttons, labeled “Gallery 1” and “Gallery 2.” Clicking each button causes function loadThumbs (lines 131–142) to be invoked with the gallery name “g1” or “g2,” depending on the gallery being selected. Executing loadThumbs will send an AJAX request to the server (lines 133–141). When the server responds, the readystatechange callback function (lines 134–139) is invoked. This callback parses the response to retrieve an array of thumbnail images and stores them in variable thumbs (line 136), and then invokes showThumbs with the same gallery name as before. Function showThumbs (lines 143–155) iterates through thumbs and creates a 'Delete' button for each image that, when clicked, will invoke deleteImg with the gallery name and index of the image. Function deleteImg (lines 156–159) creates another AJAX request, requesting the selected image to be deleted from the server (lines 157–158).

Event Races

The example application exhibits three event races that may cause runtime exceptions or other unexpected behavior, depending on the order in which event handlers execute. The corresponding undesirable schedules are illustrated in Figure 10.2 and discussed in detail below.

A If the user clicks the “Gallery” buttons before init.js has executed, then the user event is lost, since the click event handlers are not yet registered.

B If the user clicks the “Gallery” buttons after executing init.js, but before script.js has executed, then an event handler is associated with the click event, but function loadThumbs is still undeclared. Hence, executing either of the click event handlers on lines 127 and 129 triggers an uncaught ReferenceError.

C Assume the user clicks the “Gallery 1” button after the entire page has been parsed. This causes loadThumbs (lines 131–142) to execute with argument “g1,” generating an AJAX request. When the server responds, the event handler on lines 134–139 executes, causing showThumbs to execute (lines 143–155). If the user then clicks the “Gallery 2” button, loadThumbs runs again (now with argument “g2”) assigning an empty array to thumbs before making a second AJAX request. Now, say the user clicks the “Delete” button for an image that is still on the screen, before the response to the second AJAX request is received. Then, the click handler on line 151 invokes deleteImg (lines 156–159), causing the expression thumbs[pos].id to be evaluated (line 158). But thumbs is still...
10.2. MOTIVATING EXAMPLE

index.html

```html
<html>
  ...
  <div id="container" ...>
  <button id="g1">Gallery 1</button>
  <button id="g2">Gallery 2</button>
  <script src="init.js"></script>
  <script src="script.js"></script>
</html>
```

init.js

```javascript
document.getElementById('g1').addEventListener('click', function () { loadThumbs('g1'); }, false);
document.getElementById('g2').addEventListener('click', function () { loadThumbs('g2'); }, false);
```

script.js

```javascript
var thumbs;
function loadThumbs(name) {
  thumbs = [];
  var xhr = new XMLHttpRequest();
  xhr.onreadystatechange = function () {
    if (xhr.readyState === XMLHttpRequest.DONE) {
      thumbs = JSON.parse(xhr.responseText);
      showThumbs(name);
    }
  }
  xhr.open('GET', 'gallery?name=' + name, true);
  xhr.send(null);
}
```

```javascript
function showThumbs(name) {
  container.innerHTML = '';
  for (var pos = 0; pos < thumbs.length; ++pos) {
    ... // display thumbnail image
    var b = document.createElement('button');
    b.textContent = 'Delete';
    (function (pos) {
      b.addEventListener('click', function (e) { deleteImg(name, pos); }, false);
    })(pos);
    container.appendChild(b);
  }
}
```

```javascript
function deleteImg(name, pos) {
  ... xhr.open('POST', 'gallery?action=delete&name=' + name + '&img=' + thumbs[pos].id, true); ...
}
```

Figure 10.1: Motivating example (inspired by Zheng et al. [197]).
empty! So, `thumbs[pos]` evaluates to `undefined`, and accessing the `id` property of `undefined` yields an uncaught `TypeError`.

We will refer to scenarios where user events interfere with initializations performed during page loading (e.g., scenarios A and B) as *initialization races*. Races such as the one in scenario C will be referred to as *post-initialization races*.

**Repairing Event Race Errors**

The types of problems discussed above commonly occur when a schedule differs from developers’ expectations. For example, developers typically test their code in environments where the parsing and loading of a page is fast and where user actions do not occur until page loading is complete. Scenarios like A and B violate this assumption, causing various sorts of errors to arise when user events arrive at inopportune moments. Similarly, developers commonly assume the network to be fast, so that responses to AJAX requests are received before the user performs additional actions. Scenario C, where the user clicks on “Delete” before the response for the click on “Gallery 2” is received, violates this assumption, resulting in a runtime error.

Our approach for preventing undesirable schedules relies on code instrumentation, and takes as input a repair policy that specifies constraints on the scheduling of event handlers. In particular, the web application is in-
10.3. BACKGROUND ON EVENT RACES

scheduler(\(\sigma, \tau, P\)) := (\(\tau \cdot \text{extend}(\sigma, \tau, P)\), update(\(\sigma, \tau, P\)))

update(\(\sigma, \tau, P\)) := \(P \setminus \{(q, s, 1, a, r) \in P_A(\sigma, \tau, S_\tau)\}\)

\[\bigcup \{r_i \in r(\sigma) \mid (q, s, t, a, r) \in P_A(\sigma, \tau, S_\tau)\}\]

action(\(u, \tau, P\)) := \(\max \{a \mid (q, s, t, a, r) \in P_A(\text{begin}(u), \tau, S_\tau)\}\), \(\text{where}\)

\[\text{Dispatch} < \text{Postpone} < \text{Discard}\]

\[\text{extend}(\sigma, \tau, P) :=
\begin{cases}
\text{discard}(u) & \text{if } \sigma \equiv \text{begin}(u) \land \text{action}(u, \tau, P) \equiv \text{Discard} \\
\text{postpone}(u) & \text{if } \sigma \equiv \text{begin}(u) \land \text{action}(u, \tau, P) \equiv \text{Postpone} \\
\sigma & \text{otherwise}
\end{cases}\]

Figure 10.3: The effect of a repair policy on the execution.

instrumented so that all events are intercepted and monitored by a runtime controller. At runtime, when an event arrives that is not in accordance with the repair policy, it is either discarded or postponed until the execution of the associated event handlers agrees with the policy. For example, scenarios A and B can be prevented in our approach by an application-independent policy that postpones user events until all statically declared scripts are loaded, by intercepting the events and regenerating them later. (In cases where this policy blocks harmless user events, one can easily create a policy that only postpones clicks on the “Gallery” buttons.) Likewise, scenario C can be prevented by an application-independent policy that discards user events after an AJAX request until the response arrives. In such cases, EventRaceCommander shows a “spinner” on the screen to inform the end-user that user events are temporarily blocked.

While the three scenarios discussed here can be repaired using application-independent policies, application-specific policies may be preferable, as we shall see in Section 10.7.

10.3 Background on Event Races

This section defines event races and related concepts using a simplified version of the formalism of Raychev et al. [146].

We instrument an event-based program to generate a sequence of \textit{operations}, called a \textit{trace}, for a given execution. An operation can be of the following kinds (assuming each event is given a unique identifier \(u\)):

- \textit{read}(u, x) and \textit{write}(u, x) denote that an event handler of \(u\) reads and writes, respectively, a shared variable \(x\).
• fork$(u, v)$ denotes that an event handler of $u$ creates a new event $v$ that will be dispatched later.

• begin$(u)$ and end$(u)$ denote the beginning and ending, respectively, of the execution of $u$’s event handlers.

We denote the set of all operations by $Op$, and the event to which an operation $\sigma$ belongs by $\text{evt}(\sigma)$. The execution of a program generates a finite trace $\tau = \sigma_0 \cdots \sigma_n \in Op^*$. In event-based programs, all event handlers of an event execute atomically without interleaving with the execution of any handler of another event. Therefore, if an event $u$ gets dispatched, then all the operations from the event handlers of $u$ appear as a contiguous subsequence in the trace, where the first and last operations of the subsequence are $\text{begin}(u)$ and $\text{end}(u)$, respectively. If the trace contains an operation fork$(u, v)$, then $\text{begin}(u)$ appears before $\text{begin}(v)$, i.e., an event must be created before it gets dispatched.

A trace $\tau$ defines a linear relation $<$, where $\sigma < \sigma'$ if the operation $\sigma$ appears before $\sigma'$ in the trace $\tau$. As in traditional concurrent programs, we can define a happens-before relation $\preceq$ as the minimal partial order (i.e., a reflexive, anti-symmetric, transitive relation) over the events of a trace such that $u \preceq v$ if fork$(u, v) \in \tau$ or if $u$ and $v$ are user events where $\text{begin}(u) < \text{begin}(v)$. Two events $u$ and $v$ are unordered, denoted by $u \parallel v$, if they are not related by the happens-before relation. We are now in a position to define the notion of an event race.

An event race $(\sigma, \sigma')$ is a pair of operations from a trace $\tau$ where $\sigma < \sigma'$, $\text{evt}(\sigma) \parallel \text{evt}(\sigma')$, both $\sigma$ and $\sigma'$ access (i.e., read or write) the same shared variable, and at least one of $\sigma$ and $\sigma'$ is a write.

Recent work has focused on classifying event races as either harmful or harmless [91, 130, 131, 146]. In general, such classification is a subjective matter. In many cases, lost user events or uncaught exceptions do not significantly affect the user experience and do not require remediation, though for some web sites such errors are intolerable. Our approach side-steps this ambiguity by relying on the user of EVENTRACECOMMANDER to distinguish desirable schedules from undesirable ones, and applying a repair policy that prevents undesirable schedules from taking place. In other words, our approach does not rely on a particular definition of harmfulness, nor is it limited to races that are considered harmful.

10.4 A Framework for Specifying Repair Policies

This section presents a framework for constraining the order in which event handlers are executed using a specified repair policy. A repair policy $P$ consists of a set of rules, which upon their activation (determined by the current trace $\tau$ and program state $S_\tau$ of the web application) may discard or postpone
10.4. A FRAMEWORK FOR SPECIFYING REPAIR POLICIES

events occurring in the execution. To this end, we add the following types of operations.

- **discard(u)** denotes the discarding of event u (i.e., the event handlers for event u will not be invoked, and no begin(u) operation will ever appear in the trace).

- **postpone(u)** denotes the postponement of event u (i.e., u will be re-emitted later, and at least one of the operations begin(u), discard(u), postpone(u) will appear in the trace once u is re-emitted).

A key contribution of our work is that we—based on a study of many event races in real-world web applications—observe that harmful races mostly arise for similar reasons, and can be repaired using application-independent policies. A *rule* is a quintuple of the form \((q, s, t, a, r) \in Q\) where:

- **q** is an *operation predicate* over the operations in \(Op\) that specifies a necessary condition for the rule to be activated upon the current operation of a scheduler.

- **s** is an *expiration status*, which is a predicate over the pairs of traces and program states that determines if the rule is enabled or expired.

- **t \in \{1, \infty\}** is a *scope*. If \(t = 1\), then the rule expires the first time it is activated, otherwise it remains enabled until the expiration status \(s\) becomes true.

- **a \in \{Dispatch, Discard, Postpone\}** is an *action* in response to the event that the rule was activated by.

- **r : Op \rightarrow 2^Q** is an *extension function* that maps an operation to a policy, which is used for dynamically adding new rules to the existing policy.

For the sake of presentation, we will use \((q, s, t, a) \Rightarrow r_0, \ldots, r_n\) to denote the policy \(\{(q, s, t, a, \lambda \sigma. \bigcup_{0 \leq i \leq n} r_i)\}\), and \((q, s, t, a)\) to denote the policy \(\{(q, s, t, a, \emptyset)\}\).

A rule \((q, s, t, a, r) \in P\) is *activated* by an operation \(\sigma\) in state \((\tau, S_\tau)\) if \(q(\sigma)\) and \(\neg s(\tau, S_\tau)\) hold, i.e., the rule matches the operation and is not expired. We denote by \(P_A(\sigma, \tau, S_\tau)\) the set of rules in \(P\) that are activated by \(\sigma\) in \((\tau, S_\tau)\). The definition of activated rules enables us to describe the effect of a repair policy on the execution by means of a function, \(scheduler(\sigma, \tau, P)\) \[Figure 10.3\], that maps an operation, a trace, and a repair policy to an extended trace and updated policy. The auxiliary function \(extend(\sigma, \tau, P)\) \[Figure 10.3\] determines whether events should be discarded or postponed, by computing the action for \(\sigma\) as a maximum over the activated actions (multiple rules may be activated simultaneously). The ordering among actions is defined in \[Figure 10.3\]. If \(\sigma\)
is not a begin operation, then \( \tau \) is simply extended with \( \sigma \). Hence, a policy cannot discard or postpone an event based on specific operations within an event handler. Rules can, however, match on specific operations and use them to modify the policy via the extension function.

In addition to extending the trace \( \tau \), the current repair policy \( \mathcal{P} \) is replaced by \( \mathcal{P}' = \text{update}(\sigma, \tau, \mathcal{P}) \) (Figure 10.3), which differs from \( \mathcal{P} \) as follows.

1. All rules with scope 1 that were activated by \( \sigma \) in \((\tau, S_\tau)\) are removed from \( \mathcal{P} \).

2. \( \mathcal{P} \) is extended with the rules in \( r(\sigma) \) for every activated rule \((q, s, t, a, r)\).

We emphasize that postponing one event may require other events to be postponed as well, due to the happens-before relation of the original web application. For example, the load event of a script always follows the execution of the same script. Our framework automatically enforces such constraints, and additionally preserves the order of user events.

### 10.5 Repair Policies

We identify five classes of event races that cause many problems in practice. Section 10.5 defines application-independent policies in terms of the framework presented in Section 10.4. Then, Section 10.5 shows how such general policies can be specialized to particular web applications, for improved performance and user experience.

#### Application-Independent Repair Policies

**User events before DOMContentLoaded**  
Scenarios A and B from Section 10.2 illustrate initialization races that lead to undesirable behavior when a user interacts with a web page before it has been fully parsed. The errors induced by these races can be repaired by enforcing the policy \( \mathcal{P}_{\text{init, user}} \) from Figure 10.4d where \( q_{\text{user}} \) is an operation predicate that matches any user event. Due to the definition of the policy’s expiration status, PARSED (Figure 10.4b), this policy postpones any user event until the event handlers of DOMContentLoaded have been executed. It is easy to see how this policy prevents the errors in scenarios A and B from Section 10.2. By preventing click events on the “Gallery” buttons until the page has been parsed, the click event handlers will be registered in time, and the loadThumbs function will be defined before it is invoked, thereby preventing the ReferenceError.

In this policy, DISCARD could be used instead of POSTPONE. The DISCARD action is intended for user events only, since users can always simply repeat their inputs when the policy allows it, which is not possible for system events.
10.5. REPAIR POLICIES

\[ q_{\text{user}}(\sigma) := \sigma = \text{begin}(u) \land \text{type}(u) \in \{\text{keydown, mousedown, ...} \} \]
\[ q_{\text{callback}}(\sigma) := \sigma = \text{begin}(u) \land (\text{type}(u) = \text{timer} \lor (\text{type}(u) = \text{load} \land \text{tagName}(\text{target}(u)) \in \{\text{iframe, img}\})) \]
\[ q_{\text{fork}}(\sigma) := \sigma = \text{fork}(\cdot, v) \land (\text{type}(v) \in \{\text{script-exec, timer} \lor (\text{type}(v) = \text{readystatechange} \land \text{readyState}(\text{target}(v)) = 4)) \]
\[ q_{\text{begin}}(u, \sigma) := \sigma = \text{begin}(u) \]

(a) Operation predicates.

\[ \text{ARRIVED}(u, \tau, S_{\tau}) := \text{begin}(u) \in \tau \]
\[ \text{PARSED}(\tau, S_{\tau}) := \text{DOMContentLoaded} \in \bigcup_{\text{begin}(u) \in \tau} \text{type}(u) \]

(b) Expiration status utilities.

\[ \text{WaitFor}(u) := (q_{\text{user}}, \text{ARRIVED}(u), \infty, \text{Discard}) \]
\[ \text{WaitRec}(u) := (q_{\text{fork}}, \text{ARRIVED}(u), \infty, \text{Dispatch}) \]
\[ \frac{\text{fork}(u, w)}{\text{WaitFor}(w), \text{WaitRec}(w)} \]
\[ \text{ORDER}(u, v) := (q_{\text{begin}}, \text{ARRIVED}(u), \infty, \text{Postpone}) \]
\[ \frac{\text{fork}(v, w)}{\text{ORDER}(u, w)} \]

(c) Utility functions.

\[ P_{\text{init-user}} := (q_{\text{user}}, \text{PARSED}, \infty, \text{Postpone}) \]
\[ P_{\text{init-system}} := (q_{\text{callback}}, \text{PARSED}, \infty, \text{Postpone}) \]
\[ P_{\text{async-user}} := (q_{\text{fork}}, \top, \infty, \text{Dispatch}) \]
\[ \frac{\text{fork}(u, v)}{\text{WaitFor}(v)} \]
\[ P_{\text{async-fifo}} := (q_{\text{fork}}, \top, \infty, \text{Dispatch}) \]
\[ \frac{\text{fork}(u, v)}{\text{ORDERNext}(v)} \]
\[ P_{\text{init-user}}^{+} := P_{\text{init-user}} \cup \left( (q_{\text{fork}}, \text{PARSED}, \infty, \text{Dispatch}) \right) \]
\[ \frac{\text{fork}(u, v)}{\text{WaitFor}(v), \text{WaitRec}(v)} \]

(d) Application-independent repair policies.

Figure 10.4: Repair policies.
CHAPTER 10. REPAIRING EVENT RACE ERRORS

System events before DOMContentLoaded Harmful initialization races also arise when system events fire unexpectedly early. In the following example, which is based on code from exxon.com, the load event listener attached by the script will never run if the iframe loads prior to the execution of the script.

```html
<iframe src="..." id="iframe"></iframe>
```

```javascript
$("#iframe").load(function (e) { /* adjust iframe height */ });
</script>
```

Such errors can be repaired using the policy $\mathcal{P}_{init,\text{system}}$ from Figure 10.4d, which postpones system events, such as the load event of the iframe in line 160, until the page has been parsed. $\mathcal{P}_{init,\text{system}}$ matches any iframe or img load event, and any timer event, with the operation predicate $q_{\text{callback}}$.

User events while async event is pending Scenario C in Section 10.2 represents a situation where the application logic implicitly assumes that asynchronously forked events are handled atomically, without being interrupted by user events. Such post-initialization race errors can be prevented using policy $\mathcal{P}_{async,\text{user}}$ of Figure 10.4d. Informally, this policy adds the rule $\text{WaitFor}(v)$ (Figure 10.4c) to the policy whenever an operation forks an asynchronous event $v$ (e.g., AJAX request, asynchronous script request, setTimeout). This rule discards user events until $v$ is observed in the trace.

AJAX FIFO Sometimes programmers implicitly assume that the responses to multiple AJAX requests arrive in the same order as the requests were made. Consider the following example, which captures the essence of a race from gazzetta.it:

```javascript
ajax('POST', url1, function (a) { document.cookie = f(a); });
ajax('POST', url2, function (b) { document.cookie = g(b); });
```

The two callback functions are executed in response to the first and second AJAX request, respectively. Both functions assign some data from the server’s response to the same document.cookie key. Therefore, the value of this key depends on the order in which AJAX responses arrive.

To prevent such races, we use policy $\mathcal{P}_{async,fifo}$ of Figure 10.4d to postpone AJAX response events that would break FIFO order: Upon each AJAX request operation $\text{fork}(\cdot, v)$, the policy starts listening for the next AJAX request operation $\text{fork}(\cdot, w)$ by adding the rule $\text{OrderNext}(v)$ (Figure 10.4c) to the policy whenever an operation forks an asynchronous event $v$ (e.g., AJAX request, asynchronous script request, setTimeout). This rule discards user events until $v$ is observed in the trace. Furthermore, $\text{OrderNext}(w)$ is added (by the
10.5. REPAIR POLICIES

rule in $P_{\text{async,fifo}}$ to order $\text{begin}(w)$ with the response of the AJAX request operation that follows $\text{fork}(:, w)$ (if any).

**User events before async initialization** Sometimes initialization actions are being performed by asynchronously executed code. Consider the following snippet, which was extracted from flysas.com.

```html
<input id="from-airport" /><input id="to-airport" />
<script>
var lastFrom = ..., lastTo = ...; // inspect cookie
$.get('/service?code=' + lastFrom, function (from) {
  $.get('/service?code=' + lastTo, function (to) {
    $('#from-airport').val(from.name);
    $('#to-airport').val(to.name);
  });
});
</script>
```

During loading, the user’s input may be overwritten, since the fields in lines 172-173 are not initialized until the responses of the two AJAX requests in lines 170-171 have been processed. This may happen after the DOMContentLoaded event, and therefore the policy $P_{\text{init, user}}$ does not suffice to repair the race. To accommodate for this, we define an extension of this policy, $P_{\text{init, user}}^+$, that additionally discards user events until asynchronous initialization has been performed.

Intuitively, $P_{\text{init, user}}^+$ continuously adds $\text{WaitFor}(v)$ (which discards user events until $\text{begin}(v)$ appears in the trace) for every operation $\text{fork}(:, v)$ that matches $qf$, as long an event that has been forked by some other operation matching $qf$ is pending. For example, if $\text{fork}(:, v)$ and $\text{fork}(:, w)$ denote the AJAX requests in lines 170 and 171 respectively, then $\text{WaitFor}(v)$ is added upon $\text{fork}(:, v)$, which discards user events until the callback in lines 170-175 has executed. In addition, $\text{WaitRec}(v)$ is added, which itself adds $\text{WaitFor}(w)$ upon $\text{fork}(:, w)$. The $\text{WaitFor}(v)$ rule discards user events until after the callback in lines 171-173.

The $\text{WaitRec}$ rule recursively adds new rules to approximate when asynchronous initialization is over. This may lead to user events being discarded indefinitely (e.g., in the presence of image sliders that keep changing automatically). Thus, this policy should only be used for pages that always “terminate” (i.e., where the event queue eventually becomes empty if no more user events are made), or $qf$ should be defined such that it excludes operations that are not part of initialization (e.g., by ignoring timer operations).

**Application-Specific Policies**

The application-independent policies can be applied without a deep understanding of the races, and suffice for preventing the majority of the race errors (see Section 10.7). However, sometimes the policies negatively affect web page re-
CHAPTER 10. REPAIRING EVENT RACE ERRORS

Responsiveness (e.g., the user experience of a web page can be degraded when too many user events are interrupted). This motivates application-specific repair policies that reduce disruption. It is straightforward to refine an application-independent policy to specific user events. The manual effort required to design an “optimal” application-specific policy naturally requires understanding the cause of the race.

Specializing an application-independent policy to a concrete web application is straightforward. As an example, recall that the race errors exposed by scenarios A and B can be prevented by enforcing the policy $P_{\text{init, user}}$. However, this may unnecessarily affect clicks to buttons other than “Gallery 1” and “Gallery 2” during page loading. This problem can be alleviated by refining the operation predicate $q_{\text{user}}$ in $P_{\text{init, user}}$ to only match click events on those two buttons.

The interruption of the user is still not minimal, though, since the function loadThumbs (Figure 10.1, lines 131–142) is declared strictly before the DOMContentLoaded event gets dispatched. This can be remedied by, for example, exchanging the policy’s expiration status from PARSED to one that, unlike all of the application-independent policies, relies on the actual program state to return true when loadThumbs is declared in the global scope of $S_r$. With this modification, it not only becomes clear that the policy covers the event races in question; it also minimizes the interruption of the user.

Effectiveness of Repair Policies

To understand if a repair policy $P$ prevents the bad order of an event race, recall that state-of-the-art dynamic race detectors, such as EventRacer [146], report event races as two operations $\sigma$ and $\sigma'$ in a trace $\tau$ where $\text{evt}(\sigma) \parallel \text{evt}(\sigma')$.

Simply checking that the race disappears when running a race detector on the instrumented program that enforces $P$ is too naive, since state-of-the-art race detectors are currently unable to reason about ad-hoc synchronization and will report $(\sigma, \sigma')$ as a false positive. On the other hand, checking that the race becomes covered [146] gives false confidence. Indeed, most races become covered in the instrumented program, since the execution of event handlers is controlled by ad-hoc synchronization in the instrumented program.

To see how a repair policy $P$ prevents the bad order of $(\sigma, \sigma')$, consider that the instrumentation restricts the nondeterminism in the original program, by enforcing an order among certain events in the execution. Assuming that the trace $\tau$ obtained by the race detector is valid according to $P$, it is possible to model the effect of $P$ by defining an augmented happens-before relation $\preceq_P$ as the minimal partial order such that $u \preceq_P v$ if either $u \preceq v$ or $P$ would enforce $u$ to execute before $v$. Using this relation, it is possible to tell if $P$ would prevent the race $(\sigma, \sigma')$ by checking if $\sigma \preceq_P \sigma'$ or $\sigma' \preceq_P \sigma$, giving developers a

\footnote{Intuitively, a race $(\sigma, \sigma')$ is covered by another race $(\delta, \delta')$ if $(\sigma, \sigma')$ is no longer a race when $(\delta, \delta')$ is being treated as synchronization.}
way to automatically repair races that has been reported from dynamic race detectors (for a fixed catalogue of policies). The relation $\leq_P$ can be built for multiple application-independent policies by extending EVENTRACER. It remains open for future work to construct the relation for arbitrary policies.

**Discussion of Limitations and Liveness**

Although it is not a problem for the repair policies we have presented so far, there is a risk for postponing events indefinitely, thereby breaking liveness, when enforcing policies. Generally, we want to prevent some bad ordering $v \cdots u$ by discarding or postponing $v$ until $u$ has been dispatched. To avoid breaking liveness, it must be known by the time $v$ is about to fire that $u$ will inevitably occur later in the execution.

Intuitively, repair policies can only make decisions based on past events and not on future events. Let $F$ be a set of events that are known to happen in the future. Initially, $F$ contains events that always happen during page loading, e.g., `DOMContentLoaded`. During execution, as soon as some event is known to happen in the future (e.g., a timeout is registered or an AJAX request is sent), it is added to $F$. Perhaps surprisingly, $F$ may also contain some user events, since a single user event is typically composed of a sequence of low-level events (e.g., a `keyup` event always follows a `keydown` event). We now define a necessary condition for being able to enforce an order $u \cdots v$: If $v$ comes before $u$, and $u \notin F$, then there is no way to steer away from the bad execution without potentially breaking liveness, since it is unknown if $u$ will ever arrive (safety can be preserved, though, by postponing $v$ until $u$, or indefinitely if $u$ never arrives). Otherwise, if we can define (i) an operation predicate that identifies `begin(v)`, and (ii) a state predicate that becomes false at some point after $u$ has been dispatched, then the desired ordering can be enforced.

We call a repair policy *enforceable* for a program if it does not break liveness in any execution. Conversely, we call a race *repairable* if there exists an enforceable policy that prevents the bad order of that race. The application-independent policies $P_{init,\text{user}}$, $P_{init,\text{system}}$, $P_{async,fifo}$, and $P_{async,\text{user}}$ are enforceable for all programs, and $P_{init,\text{user}}^+$ is enforceable for all programs that “terminate” (see Section 10.5).

There are situations where it is not possible to prevent an ordering $v \cdots u$ by only discarding or postponing events. Consider the following example:

```javascript
177 <script>setTimeout(function () { d = document; }, 0);</script>
178 <script>console.log(d.querySelectorAll('*').length);</script>
```

Here, the callback in line 177 is supposed to execute prior to the script in line 178. If the latter executes first, then the only possible repair is to postpone its execution. However, this will change program behavior, since line 178 counts the number of elements currently in the DOM. We have not seen any such examples in practice, and hypothesize that this situation is rare.
In other cases, although it is technically possible to repair an event race error, the result would have such a negative impact on user experience that we do not consider it. These races involve event handlers that are triggered when the user merely moves the cursor (e.g., `mouseenter`). Using a repair policy, the user can be provided with feedback that the page is not ready. However, for this kind of “indirect” user event (as opposed to mouse clicks and key events), the event handler registration should rather be performed earlier by changing the code.

### 10.6 Implementation

Our implementation, named `EventRaceCommander`, instruments HTML and JavaScript source files of a given web application on-the-fly using MITMProxy [5]. The instrumentation intercepts relevant operations and interacts with the event controller, which is loaded before any application code, such that instrumentation and application code do not race.

The implementation of `EventRaceCommander` is available at https://github.com/cs-au-dk/EventRaceCommander.

**Controlling the execution**

For non-DOM events (e.g., timers, AJAX responses), `EventRaceCommander` replaces each registration of an event handler $h$ with the registration of a new event handler $h'$ that adds $h$ to a queue maintained by the event controller. This involves intercepting calls to a small set of global functions (e.g., `setTimeout`), and instrumenting all property assignments to intercept registrations to, e.g., the `onreadystatechange` property of XMLHttpRequest objects.

For DOM events (e.g., `click`, `load`), the situation is slightly more complicated due to capturing and bubbling. These event delegation mechanisms propagate events from the document root to the target node and back [4]. `EventRaceCommander` handles DOM events as follows. When the page starts loading, event handlers for all DOM event types are registered for the capturing phase of the root element (this ensures that these event handlers are triggered first, since event handlers are triggered in registration order).

When one of these event handlers is invoked with an event $e$ that was not previously postponed, the event controller is notified that $e$ has been emitted. The controller then queries the repair policy for the action $a'$ associated with $e$. If $a' \equiv \text{Dispatch}$, then all event handlers associated with $e$ are triggered, and the controller is notified that $e$ has been dispatched. Otherwise, $a' \in \{\text{Discard}, \text{Postpone}\}$, and the execution of the application’s event handlers and other possible side-effects of the event (e.g., the insertion of a character into a text field) are prevented by calling `stopImmediatePropagation` and `preventDefault` on the event object of $e$. Furthermore, if $a' \equiv \text{Postpone}$, then the process is repeated by re-dispatching $e$ asynchronously.
Intercepting relevant operations

**EventRaceCommander** intercepts *fork*, *begin* and *end* instructions. Operations of type *fork* are intercepted by replacing certain browser API functions and intercepting property assignments. For example, the *send* function on the prototype of *XMLHttpRequest* is replaced by a function that, in addition to sending the request, notifies the event controller that an AJAX response event has been forked.

It is insufficient to monitor events for which the program has an event handler: in order to enforce, e.g., $P_{async, fifo}$, all AJAX response events must be intercepted, even those that have no response event handler. **EventRaceCommander** therefore adds a default event handler for such events.

### 10.7 Evaluation

We aim to answer the following research questions.

**RQ1** How effective is each of the application-independent policies of Section 10.5 at repairing event race errors?

**RQ2** What is the impact of each application-independent repair policy on runtime performance and user experience?

**RQ3** Is it possible to reduce runtime overhead and improve user experience using application-specific policies?

## Experimental Methodology

**Selecting event race errors** We use existing tools, such as, **EventRacer** [146] and **R4** [91], to identify candidate event races in the web applications of the 20 largest companies from the Fortune 500 list. Since front pages of many websites often contain little dynamic behavior, we manually explore the selected sites to find interesting pages.

Following Mutlu et al. [130], we focus on *observable* races that result in errors, such as, uncaught exceptions or visual differences so that we can confirm the effectiveness of our repairs. In order to keep the amount of work manageable, we examine up to 25 candidate races for each website to identify whether they are observable. Altogether this gives us 117 errors that are caused by observable races.

**Selecting application-independent repair policies** We study each observable race in detail to identify which of the application-independent repair policies that can repair the corresponding error.

**Measuring instrumentation overhead** For each website, we create an application-independent policy that repairs all race errors (possibly by combi-
ing multiple application-independent policies), and measure the overhead of that policy. We use the Chrome Debugging Protocol \[3\] to measure: (i) *parsing time* (i.e., time to `DOMContentLoaded`), showing the cost for loading `EVENTRACECOMMANDER` and instrumenting the source, and (ii) *layout time* (i.e., time to last layout event during initialization). In this experiment, we prevent layout events from triggering indefinitely (e.g., due to a slideshow) by stopping recursive timer registrations and intervals so that every web application terminates. We report the mean of 50 repetitions in each case.

**User experience** Parsing time and layout time indirectly reflect the user’s experience: most elements are ready for user interactions after a page has been parsed, and layout time reflects perceived responsiveness. In a few cases where application-independent policies are inadequate because of undesirable impact on the user experience, we attempt to design application-specific versions of application-independent policies that do not exhibit similar problems. For each such case, we attempt to evaluate the impact on user experience by comparing the delays in event processing for application-independent and application-specific policies.

**System details** We run the experiments on Ubuntu 15.10 with an Intel Core i7-3770 CPU and 16 GB RAM.

| Table 10.1 shows the sites and races used to evaluate `EVENTRACECOMMANDER`. The “Race errors” column presents the total number of observable races found in each site. The “Race classification” columns classify these races. Most of the observable races that we found are initialization races, and nearly all of these involve user events, except a race on `att.com` where two dependent scripts are loaded without any ordering, and on `exxon.com`, where an `iframe` `load` event handler is registered late. Late event handler registrations tend to be a recurring problem. We also identify multiple post-initialization races. These typically cause a web application to end up in an inconsistent state. |

---

\[2\] We did not detect any observable races on `berkshirehathaway.com`, `valero.com`, `unitedhealthgroup.com`, and `kroger.com`. Those sites are excluded from the table.
Table 10.1: Observable races, applicability of application-independent repair policies, and instrumentation overhead.

<table>
<thead>
<tr>
<th>Website</th>
<th>Race errors</th>
<th>Race classification</th>
<th>Repair policy</th>
<th>Instrumentation overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>( r_1 \ldots r_{13} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>walmart.com</td>
<td>14</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+609 (1.29x) +247 (1.08x)</td>
</tr>
<tr>
<td>exxon.com</td>
<td>7</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+20 (1.02x) +23 (1.01x)</td>
</tr>
<tr>
<td>chevron.com</td>
<td>8</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+88 (1.12x) +176 (1.13x)</td>
</tr>
<tr>
<td>apple.com</td>
<td>3</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+69 (1.11x) +65 (1.10x)</td>
</tr>
<tr>
<td>gm.com</td>
<td>8</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+60 (1.08x) +60 (1.08x)</td>
</tr>
<tr>
<td>phillips66.com</td>
<td>3</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+87 (1.14x) +31 (1.04x)</td>
</tr>
<tr>
<td>ge.com</td>
<td>10</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+124 (1.08x) +207 (1.13x)</td>
</tr>
<tr>
<td>ford.com</td>
<td>1</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+154 (1.06x) +155 (1.06x)</td>
</tr>
<tr>
<td>cvshealth.com</td>
<td>10</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>-24 (0.98x) -14 (0.99x)</td>
</tr>
<tr>
<td>mckesson.com</td>
<td>2</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+101 (1.08x) +7 (1.00x)</td>
</tr>
<tr>
<td>att.com</td>
<td>12</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+723 (2.12x) +699 (2.10x)</td>
</tr>
<tr>
<td>verizonwireless.com</td>
<td>13</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+360 (1.15x) +266 (1.12)</td>
</tr>
<tr>
<td>amerisourcebergen.com</td>
<td>4</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+13 (1.04x) +12 (1.03x)</td>
</tr>
<tr>
<td>fanniemae.com</td>
<td>5</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+143 (1.26x) +70 (1.10x)</td>
</tr>
<tr>
<td>costco.com</td>
<td>16</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+92 (1.11x) +45 (1.03x)</td>
</tr>
<tr>
<td>hp.com</td>
<td>1</td>
<td>( r_1 )</td>
<td>( r_1 )</td>
<td>+35 (1.01x) +37 (1.01x)</td>
</tr>
<tr>
<td>total</td>
<td>117</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Race classification:**
- **declare/event:** an entity may be used before it is declared, triggering an error (e.g., scenario \( B \)).
- **register/event:** an event handler may be registered late, leading to lost events (e.g., scenario \( A \), lines 160–164).
- **update/event:** a form field may be updated asynchronously, overwriting the user’s input (e.g., lines 167–176).
- **system/user:** a system and user event are unordered, leading to an error or erroneous state (e.g., scenario \( C \)).
Experimental Results

**RQ1** The “Repair policy” columns of Table 10.1 reflect the applicability of the application-independent policies. If, for a given site, an event race $r_i$ appears in the column of repair policy $\mathcal{P}$, then $\mathcal{P}$ repairs the error caused by $r_i$. Otherwise, no application-independent policy prevents $r_i$, and the race appears in the “None” column. In our experiments, all observable races that could not be repaired using application-independent policies involve indirect user inputs (Section 10.5). These races are relatively harmless (e.g., dropdowns that do not unfold when the user hovers a menu item with the cursor during loading). Note that races with the same classification tend to be prevented using the same policies. This is to be expected, since $\mathcal{P}_{\text{init, user}}$, $\mathcal{P}_{\text{init, user}}^+$ and $\mathcal{P}_{\text{init, system}}$ target initialization races, unlike $\mathcal{P}_{\text{async, fifo}}$ and $\mathcal{P}_{\text{async, user}}$.

Although we cannot guarantee that our application-independent policies always suffice, our results suggest that the policies can prevent most event race errors in practice: 94 of the 117 event race errors are repairable using our application-independent policies.

This also suggests that, although EventRaceCommander relies on a light-weight instrumentation, it provides sufficient control of the nondeterminism to prevent the races that occur in practice. Furthermore, the results indicate that our assumption of what “good” schedules are (Section 10.2) agrees with developers’ expectations (otherwise, our policies would enforce erroneous schedules).

Table 10.1 shows that many race errors can be repaired using more than one application-independent policy. Not surprisingly, many races can be repaired using both $\mathcal{P}_{\text{init, user}}$ and $\mathcal{P}_{\text{init, user}}^+$, but we also find that $\mathcal{P}_{\text{async, fifo}}$ and $\mathcal{P}_{\text{async, user}}$ often repair the same race. This happens when a user triggers an asynchronous event (e.g., an AJAX request) twice. The policy $\mathcal{P}_{\text{async, fifo}}$ avoids such races by enforcing an order among the unordered events, whereas $\mathcal{P}_{\text{async, user}}$ postpones user events while an asynchronous event is pending (thereby ensuring that event handlers and their forked events execute atomically).

**RQ2** The last two columns of Table 10.1 show parsing and layout time. For most sites, the instrumentation overhead is less than 200ms, which we deem to be acceptable. Small websites exhibit larger relative overheads due to the cost of including EventRaceCommander’s 32 KB of JavaScript. The absolute overhead is barely noticeable by a user, though.

Regarding user experience, it is important to interrupt only user events that are involved in races, and only for as long as is necessary to prevent undesirable schedules. Generally, we find that the policies $\mathcal{P}_{\text{init, system}}$ and $\mathcal{P}_{\text{async, fifo}}$ can be enforced obliviously to the user, since they do not involve user events and, in our experiments, do not significantly postpone UI updates. There is often room for improvements over $\mathcal{P}_{\text{init, user}}$, $\mathcal{P}_{\text{async, user}}$, and $\mathcal{P}_{\text{init, user}}^+$, since the operation predicates in these policies are overly general. This is mostly a
problem for $P_{\text{init, user}}^+$ in sites that extensively load code asynchronously (e.g., walmart.com, which uses RequireJS [7]). In such cases, the page appears to be ready significantly before user input is tolerated, and an application-specific policy should be used to target the relevant user events, and minimize the time in which the user is disrupted.

Interestingly, we find that some of the websites, e.g. apple.com, prevent races in ways similar to $P_{\text{async, user}}$, by showing a spinner that takes up most space on the screen, when a user event leads to asynchronous DOM updates.

RQ3 We now briefly report on two event races where application-independent repair policies yield suboptimal results, and discuss how each situation can be remedied using an application-specific policy.

On att.com, event race $r_1$ can cause a TypeError due to two scripts being unordered. Policy $P_{\text{async, fifo}}$ ensures that asynchronous scripts are executed in FIFO order and fixes the error, but unnecessarily imposes an order on 39 scripts. On average, 21 of these scripts are postponed for 292ms. This can be prevented using a specialized policy $P_{\text{async, fifo}}'$, which only postpones the execution of satellite-567046aa64746d0712008241.js. On average, this policy postpones no scripts at all (i.e., in our experiments, the two scripts always load in the desired order).

On walmart.com, a click event handler of a button is registered by an asynchronous script. Until that happens, click events on the button are lost and no dropdown is shown (event race error $r_{13}$). While this problem can be fixed using the application-independent policy $P_{\text{init, user}}^+$, this results in excessive delays for processing a click event. We can avoid such undesirable impact on the user experience by designing an application-specific policy $P_{\text{spec}}$ that postpones click events only until the handler is present. In an experiment, we issue a click immediately when the button is declared, and measure the time until the corresponding event handlers execute. On average, the click event is dispatched 817ms faster when using the policy $P_{\text{spec}}$ instead of $P_{\text{init, user}}^+$.

The application-specific repair policies discussed above are both “optimal” in the sense that they only postpone events that are involved in the races under consideration, for the minimal amount of time required to prevent the undesired orders. We argue that, for these race errors, enforcing repair policies using EventRaceCommander compares well to alternative solutions such as modifying the code to introduce ad-hoc synchronization or explicitly load scripts synchronously.

---

3The global variable s_att is declared in s-code-contents-65778bc202aa3fe01113e6b-6e6d103eda099f4e5.js, and used in satellite-567046aa64746d0712008241.js. The latter may, depending on the event order, crash during an assignment to s_att.events.
CHAPTER 10. REPAIRING EVENT RACE ERRORS

Discussion

Some aspects of our evaluation may affect the generality of the reported results. Most significantly, the selection of websites and event race errors used in our evaluation could be subject to bias. We have attempted to address this concern by evaluating \texttt{EventRaceCommander} on the websites of the 20 largest companies from the Fortune 500 list, similar to previous work on event race detection \cite{91,146}.

The code of the websites used in our evaluation may be subject to change, which may affect reproducibility of our results. Therefore, we spent significant effort using \texttt{mitmproxy} \cite{5} to record the server responses for an interaction with every site under consideration. This enables reproducibility for all front pages. Regrettably, some highly dynamic pages that we consider cannot be replayed, since the URLs of AJAX requests depend on user input, random numbers, timestamps, etc. Still, this is a significant improvement over recent work \cite{78,87,91,131,138,146,176,197}, where the importance of reproducibility has mostly been ignored. The recordings from our study are available with the implementation of \texttt{EventRaceCommander}.

A related concern is that real websites may give rise to unpredictable network delays, which may affect repair policies, such as, $P_{\text{async,fifo}}$. In principle, these delays can become arbitrarily large, so the data from our experiments may not truly reflect the impact on user experience. In our experiments, we avoid large fluctuations by relying on recordings of every website, and by conducting experiments 50 times and reporting average times. To prevent situations where the user is being disrupted for too long, it would be possible to monitor if \texttt{EventRaceCommander} postpones an event for more than a given threshold. In such cases, the event could simply be dispatched, and the users of \texttt{EventRaceCommander} could be notified of the incident, so that the policy can be adjusted.

10.8 Related work

\textbf{Race detection} Ide et al. \cite{87} pointed out that JavaScript programs can have data races despite being single-threaded and non-preemptive. Such races often arise due to asynchronous AJAX communication and HTML parsing. The authors note regarding AJAX races that “the programmer prefers to think of the interaction with the server as a synchronous call”, which is also the foundation for our scheduling policies for such races. Zheng et al. \cite{197} proposed a static analysis for automatically detecting JavaScript races. Due to the dynamic nature of JavaScript, such static analyses are often prohibitively imprecise or unscalable. Inspired by successful techniques developed for multi-threaded programs \cite{56}, \texttt{WebRacer} and \texttt{EventRacer} instead use dynamic analysis and a happens-before relation \cite{138,146}. This significantly improves
precision, however, these tools cannot distinguish harmful from benign races, which has motivated techniques that explore whether races cause observable differences \cite{78, 91, 131}. Still, these techniques tend to report many false positives and also miss harmful races, and it has been observed that the harmful races that are detected are often difficult to fix.

Event race detection algorithms have also been developed for Android, using similar techniques as those targeting JavaScript, but with more sophisticated happens-before relations \cite{31, 81, 118}. Adapting our technique to Android is an interesting opportunity for future work.

**Automated fixing of race errors** The idea of automatically fixing race errors has been studied extensively in a multi-threaded setting, but not much for event-driven applications, in particular JavaScript.

Some techniques patch the program code by inserting, e.g., locks and wait-signal operations, based on reports from race detectors and static analysis \cite{93, 94, 97, 102, 109, 177}. The JavaScript platform provides no explicit synchronization primitives, but our repair policy mechanism can simulate the effect of having wait-signal primitives or atomic groups of event handlers.

Other techniques steer away from nondeterministic errors by postponing selected actions, much like our approach but for multi-threaded programs. The AI technique \cite{193} attempts to stall threads where manifestation of a concurrency bug is about to become deterministic. KIVATI \cite{37} uses static analysis and hardware watchpoints to detect atomicity violations and then dynamically reorders the relevant instructions. The Aviso system \cite{112} learns schedule constraints from successful and failing executions, and then uses these constraints to guide scheduling, much like our policy mechanism and controller. The Loom system \cite{180} uses a notion of execution filters, which resembles our use of application-specific repair policies.

These techniques share the limitation of EventRaceCommander that they cannot fix all race errors while entirely avoiding situations where actions are postponed excessively.

Other approaches include rollback-recovery \cite{195}, replicated execution with different schedules \cite{171}, replication of shared state in critical sections \cite{141, 144}, or require special hardware \cite{113}, which would not be realistic for JavaScript.

**EventHealer** \cite{165}, unlike most of the work mentioned above, considers event-driven programs, but with a different execution model than the one in JavaScript: execution takes place in a main thread, which has lower priority than event handlers, and preemption is possible but can be selectively disabled to protect critical sections. The system uses static analysis to locate event handlers, shared variables, and fragments of code that should be treated as critical sections, which is very different from our setting.

None of the work on automated fixing discussed above targets JavaScript. A position paper by Mutlu et al. \cite{130} proposes a notion of “schedule shepherding” for JavaScript, but does not present any mechanism for how to
actually do it. The recent ARROW tool by Wang et al. \cite{176} is the first to automatically repair races in JavaScript applications. The key difference to EventRaceCommander is that ARROW is based on static analysis, which is notoriously difficult for real-world JavaScript code. Moreover, the main idea in ARROW is to identify inconsistencies between the happens-before and def-use relations, which may miss many race errors, even if more powerful static analysis were developed. ARROW cannot repair any of the errors in the example application in Section 10.2.

10.9 Conclusion

We have presented a general framework for controlling nondeterminism in event-driven applications using specified repair policies, and proposed application-independent policies to prevent nondeterminism that commonly triggers event race errors. The framework is sufficiently general to repair a wide variety of real-world event race errors. Our experimental results show that 94 of 117 event race errors are repairable by our application-independent policies, and that application-specific policies are useful to target specific races, when needed.

For future work, it will be interesting to automate the process of inferring application-specific policies for a given event race, to avoid negative impacts from overly general policies. Such candidate policies should restrict the nondeterminism only as needed to repair a given race, but still be reasonably general, so that they do not only apply to the concrete execution explored by the dynamic race detector.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 647544).
10.A Unreparable Event Races

Some event races from our experiments cannot be circumvented properly by merely restricting the nondeterminism. Due to space limitations, our ICSE 2017 paper only briefly discusses these races, so in the following we provide some additional details. All of the 23 cases from our study are due to the same kind of initialization race, and are relatively harmless. To illustrate the problem, consider that the following button is declared in the HTML: `<a onmouseover="showDropdownMenu()">`, and that it is possible for the user to hover the button before the function `showDropdownMenu` gets declared, thereby exposing an uncaught `ReferenceError`. It may be slightly annoying to the user that the functionality is not working until the web page has been fully loaded, but on the other hand, the error does not leave the application in an inconsistent state, and will respond to the user’s gestures once the web page has loaded. To combat this problem, `EventRaceCommander` can discard the `mouseover` events until `shopDropdownMenu` has been declared. This will prevent the uncaught `ReferenceError` from manifesting, but the dropdown menu is still not going to show up until `showDropdownMenu` has been declared. Alternatively, `EventRaceCommander` can postpone the `mouseover` events until `showDropdownMenu` has been declared. This could potentially lead to counter-intuitive behavior, though, since the user will likely already have moved the cursor to somewhere else by the time the dropdown menu opens. We argue that issues like this one should be fixed by moving the declaration of `showDropdownMenu` before the button, if the developers care to fix the problem.
Chapter 11

Practical Initialization Race Detection for JavaScript Web Applications


Abstract

Event races are a common source of subtle errors in JavaScript web applications. Several automated tools for detecting event races have been developed, but experiments show that their accuracy is generally quite low. We present a new approach that focuses on three categories of event race errors that often appear during the initialization phase of web applications: form-input-overwritten errors, late-event-handler-registration errors, and access-before-definition errors. The approach is based on a dynamic analysis that uses a combination of adverse and approximate execution. Among the strengths of the approach are that it does not require browser modifications, expensive model checking, or static analysis.

In an evaluation on 100 widely used websites, our tool INITRACER reports 1,085 initialization races, while providing informative explanations of their causes and effects. A manual study of 218 of these reports shows that 111 of them lead to uncaught exceptions and at least 47 indicate errors that affect the functionality of the websites.
11.1 Introduction

It is well known that event races are the cause of many errors in JavaScript web applications \[163\]. Such races occur due to nondeterministic ordering of event handlers, for example when program behavior depends on whether a user event appears before or after a script has been loaded. Traditional testing is insufficient for discovering unexpected harmful event orderings, which has motivated the development of a range of powerful techniques and tools to detect event races automatically \[78, 87, 91, 131, 138, 146, 176, 197\]. However, these existing approaches suffer from various limitations, which makes them unsuitable for production use.

For example, the dynamic race detector EventRacer \[146\] reports an overwhelming number of races on typical web applications. Most of those races are benign, and it is difficult to classify each race warning as harmful or benign based on the output of the tool \[78, 91, 131\]. Many races arise due to ad-hoc synchronization that was added by programmers to prevent event race errors, or simply do not affect the application’s functionality. The tool by Mutlu et al. \[131\] attempts to focus on harmful races, specifically those that affect persistent storage, using a combination of dynamic execution and lightweight static analysis. However, with their technique, any error that may disappear by reloading the web page is considered benign, even though the error may damage functionality or user experience. Additionally, even races that may affect persistent storage are often completely harmless. The R\(^4\) tool \[91\], which is based on systematic model checking and a notion of approximate replay, is able to produce witness executions that show the consequences of each race, thereby making it easier to determine the degree of harmfulness.

A general limitation of these tools that are based on dynamic analysis is that they can only find race errors in the parts of the code that have been covered by the given user event sequence. Moreover, some races reported by these tools are technically possible but extremely unlikely in practice. As these techniques rely on dynamic analysis using instrumented browsers, we also find that the available prototype implementations quickly fall behind the rapid evolution of browsers and thereby become incapable of processing modern web applications. Other techniques that do not rely on concretely executing the application code but instead apply purely static analysis generally have low precision, caused by the general difficulties in statically analyzing JavaScript code \[20, 146\]. Related work, including these techniques, is covered in more detail in Sections \[11.6\] and \[11.7\].

In this paper, we take a more pragmatic approach towards automated event race detection. We present a tool named InitRacer that has the following properties: (i) it can detect harmful races with relatively few false positives \[4\] compared to the state-of-the-art alternatives, (ii) it is fast and light-weight by
not requiring expensive model checking or static analysis, (iii) it is platform-independent, so it can be implemented without browser modifications, (iv) it is independent of specific user event sequences unlike the dynamic race detectors mentioned above, and (v) it produces informative error messages to support diagnosing the causes and effects of the races.

We observe that a significant part of the harmful races reported in previous work on event race detection are initialization races that occur during the loading and initialization of the web page and do not involve long sequences of user actions, and identify three types of such races: form input overwritten, late event handler registration, and access before definition. We choose to focus entirely on these three common types of initialization races, specifically those that involve at most one user event since the harmful interleavings of such races are more likely to manifest in practice.

Our approach is inspired by two recently proposed techniques for testing Android apps: Thor [9], which systematically exposes a program to adverse conditions (in our case: events that occur sooner than expected), and AppDoctor [54], which uses a notion of approximate execution to speed up testing by directly invoking event handlers instead of faithfully simulating UI events. Our tool InitRacer analyzes a given web page in three phases. In each phase, the page is loaded in a browser via a proxy server that instruments the HTML and JavaScript code to monitor and control execution. In phase 1, the web page is loaded and executed in observation mode, without any user events. This allows us to collect information about the behavior of the initialization code in conditions where it is unlikely that serious errors occur, since even simple testing during development would have revealed such errors. In phase 2, the web page is reloaded, this time in adverse mode where events are simulated aggressively. Each time an event handler is registered, either by JavaScript code or by HTML code, we eagerly invoke the event handler to mimic a scenario where the event occurs immediately. (Scripts execute non-preemptively, so we naturally wait until the individual scripts have completed before injecting the invocations.) We can thereby detect if, for example, any crashes (i.e., uncaught exceptions) occur when events happen sooner than expected. Finally, in phase 3, the web page is reloaded in validation mode. Since phase 2 uses approximate execution rather than simulating high-level user events faithfully, and it injects all possible events in one execution, it is possible that some of the observed crashes are either impossible due to unforeseen interactions between the inserted event handlers or highly unlikely in practice. In validation mode we therefore inject only the crashing event handlers, one at a time. Section 11.2 gives three examples of different kinds of typical event race errors that can be found with our technique.

an error the developer cares about (note that this determination is subjective). It is benign or harmless if it is a true positive but does not lead to an error the developer cares about.
In summary, our contributions are as follows.

- We present a light-weight technique for finding initialization race errors in JavaScript web applications. The key idea is to monitor execution of the initialization in observation mode, adverse mode, and validation mode, each providing useful information about the possible behavior of the given application.

- We describe three broad categories of initialization race errors that can be detected using our technique: form input overwritten, late event handler registration, and access before definition.

- The technique is implemented in a tool called InitRacer, and an empirical evaluation shows it to be capable of detecting harmful initialization races in real-world websites. On 100 websites from the Fortune 500 companies, it reports 1,085 event races. Compared to other approaches, it is relatively fast and precise, and it produces informative race error messages that explain the causes and effects of the races. A manual study of 218 of the reports shows that 111 of them lead to uncaught exceptions without directly affecting user experience and that at least 47 indicate errors that affect functionality of the websites.

11.2 Motivating Examples

In this section, we discuss three types of initialization race errors that occur commonly, and illustrate them using examples taken from prominent websites.

A Form-Input-Overwritten Error

A form-input-overwritten error manifests when JavaScript code initializes the value of a form input field (e.g., a text field, checkbox, or radio button) after the user has already entered a value. Such errors tend to annoy users, because the input they typed is lost and needs to be re-entered.
11.2. MOTIVATING EXAMPLES

Consider Figure 11.1, which shows the relevant fragments of HTML and JavaScript code involved in a real-world form-input-overwritten error that we encountered on http://www.mckesson.com/. In this example, text entered by users in the search field (line 179) may be overwritten by initialization code if page loading is unexpectedly slow. The value of the search field can be modified by the user as soon as the browser has finished rendering it on the screen. However, on line 181, an external script named search.min.js is loaded. This script calls jQuery’s ready function (line 182) to register an event handler for the DOMContentLoaded event, which is invoked when the browser has finished parsing the web page. When that happens, the event handler initializes the value of the search field to the result of getParameterByName("q") (which retrieves the query parameter q from the URL), or the empty string if no query parameter is present (lines 183–184).

We consider the race in Figure 11.1 to be harmful because the schedule where initialization takes place after the user has entered a value leads to a UI state that is different from the one produced by a schedule where these activities happen in the opposite order. Furthermore, the classification of this event race as being harmful is supported by the following observations: (i) the search field becomes visible to the user by the time it gets declared in the HTML (i.e., no CSS prevents it from being displayed, and its type is not "hidden"), (ii) the user can modify the value of the input field (i.e., it is neither read-only nor disabled), and (iii) there is a potentially long delay (caused by loading the external script) between the time when the search field becomes visible and the time when its value is initialized by JavaScript code.

Our tool InitRacer correctly reports that the value of the input field declared at line 79 column 17 in the HTML source code of http://www.mckesson.com/ is overwritten by the empty string. Additionally, InitRacer identifies the JavaScript operation that overwrites the value of the form field by a stack trace, and takes a screenshot of the web page, where the relevant form field has been highlighted. In contrast, EventRacer does not detect the error (in fact, EventRacer does not detect any form-input-overwritten errors). The R^2 tool is based on EventRacer’s trace construction and also fails to detect the error in the example. The tool by Mutlu et al. only considers race errors that affect persistent storage, which is not the case for the error in this example (and for the two following examples).

A Late-Event-Handler-Registration Error

Another type of initialization race occurs when an event is fired before a corresponding event handler is registered. If this happens, there are two possible outcomes. One possibility is that the event is ignored and nothing happens (e.g., the user clicks on a button element before a click event handler
CHAPTER 11. PRACTICAL INITIALIZATION RACE DETECTION

Figure 11.2: A late-event-handler-registration error from apple.com.

is installed). In this case, the race is relatively harmless because the user can simply click on the button again to trigger the desired behavior. However, a more serious problem may occur if the event has a default action that needs to be prevented. For example, if an event handler prevents certain characters from being typed into a text field, and the event handler is registered late, then input validation can be bypassed until the event handler is registered.

The code in Figure 11.2 illustrates such a late-event-handler-registration error on http://apple.com/ where an event handler that prevents the de-
11.2. MOTIVATING EXAMPLES

The web page contains a hyperlink "Search apple.com" (line 186), and clicking on the link causes a dropdown to appear with a search field. Note that the script ac-globalnav.built.js referenced on line 187 is loaded after the search icon has become visible to the user. The “module” with number 194 within this script (lines 198–217) calls the _initializeSearch function (lines 202–208). Looking more carefully at the _initializeSearch function, we can see that, on line 203, it retrieves a reference to the search link, and then registers onSearchOpenClick (lines 209–211) as a click event listener on this element (line 195). Hence, from this point onwards, onSearchOpenClick is invoked when the user clicks on the search link. When execution of this function reaches line 210, it invokes l(E), which causes preventDefault to be called on the event object (line 190). This ensures that the user is not redirected to http://www.apple.com/us/search. The event handler then invokes showSearch to open the dropdown containing a search field (line 210).

At this point, it is clear that the code in Figure 11.2 exhibits a harmful race because program behavior is quite different depending on whether a user clicks on the search link before or after the event handler has been registered. If the user clicks on the search icon before the event handler has been registered, then the user is redirected to http://www.apple.com/us/search (which is intended for browsers that do not support JavaScript) instead of seeing the dropdown search box. InitiRacer correctly identifies this error. It reports that a click event listener, which invokes preventDefault on the event object, is registered too late on the a element at line 146 column 5 in the HTML code. Again, InitiRacer reports a stack trace and a screenshot to aid debugging.

Existing tools for event race detection struggle with this example as well. EventRacer finds the race when manually exploring the website, and clicking on the link. However, it does not find the race in auto-exploration mode, even when trying 10 times. On other websites, EventRacer reports false positives and harmless races because it does not take visibility and long delays into account. Furthermore, EventRacer filters away so-called covered races, and, unlike InitiRacer, it is limited to detecting errors in code that has been executed by a given user event sequence. EventRacer may also continue to report a warning even after the developers have fixed the late event handler registration. Similarly, R^4 does not find the race when using EventRacer’s auto-exploration mode, and it fails to analyze the web page in manual-exploration mode.

An Access-Before-Definition Error

Sometimes event handlers trigger unexpectedly early, before all application code has been loaded fully. When such event handlers attempt to read a

---

3One approach to fix the error is to register the event handler immediately after the element declaration, which would make the harmful interleaving practically impossible.
variable, dereference a property of an object, or invoke a function that has not yet been defined, an access-before-definition error occurs. Figure 11.3 shows an example of such an error that we observed on https://www.aetna.com/.

In this code, a menu item "Individuals & Families" is declared (lines 218–222) before a script named s_code.js (line 223). The script declares a top-level variable named s (line 224) and initializes it to the result of invoking the function s_gi. That function assigns a fresh object to a local variable s (line 228) and then creates a new function that takes one argument and has the same body as the string stored in the c variable from line 227. Here, the key issue of note is that a function taking five parameters is stored in s.tl. After invoking the newly created function with the object stored in s as argument (line 229), s is returned (line 230).

To understand the problem with this web page, consider what happens if a user clicks on the menu item before the s_code.js script has been loaded. In this scenario, the click event handler of the menu item attempts to invoke s.tl (line 220) before the variable s has been declared. This causes the event handler to crash with an uncaught ReferenceError.

There are several ways a programmer could fix this problem. For example, one could move the loading of the script s_code.js before the declaration of the menu items, so that the user cannot click on the menu items before s.tl is defined. However, this has the unfortunate effect of slowing down the rendering of the web page, because the menu items will not appear until the s_code.js script has been loaded. Alternatively, ad-hoc synchronization could be introduced in the click event handler to account for whether s_code.js
has been loaded. In this case, the call can either be skipped, or deferred until \texttt{s.t1} is defined, by setting a timer.

Our \textit{InitRacer} tool detects the error. \textit{EventRacer} fails to detect the problem in auto-exploration mode, but does find it in manual-exploration mode. However, after fixing the problem using ad-hoc synchronization, \textit{EventRacer} \textit{still} reports the same amount of races because the added synchronization code gives rise to an additional (harmless) event race. \textit{R}^4 does not find the error in auto-exploration mode, and fails to analyze the page in manual-exploration mode.

### 11.3 Web Page Loading in Browsers

Before we can explain how our approach works, we briefly review how browsers load and initialize JavaScript web applications and how event race errors may occur.

Given a URL for a web application, the browser fetches and parses the HTML code while building the corresponding DOM structure. JavaScript code that is embedded within HTML code is executed as it is being encountered during parsing, but it is also possible to load external scripts. Event handlers can be registered either as special attributes in HTML (e.g., \texttt{onload}) or by the JavaScript code (e.g., directly by calling the \texttt{addEventListener} function from the DOM API, or indirectly via jQuery’s \texttt{ready} function as in \texttt{Figure 11.1}). HTML parsing is performed in chunks, which allows for user event processing to be interleaved with parsing. All interleavings are not necessarily possible in practice, though. For example, the browser may choose to parse and render an HTML snippet \texttt{<div>…</div><div>…</div>} in one atomic action, which would prevent the user from interacting with the web page in between the rendering of the two \texttt{div} elements. Scripts may also register \textit{timers}, consisting of JavaScript code that is to be executed after a specified delay. The browser additionally allows event handlers to be associated with different stages of initialization, most importantly the \texttt{DOMContentLoaded} event and the \texttt{load} event, which signal that the HTML page has been fully parsed and that the page and all sub-resources have finished loading, respectively.

This entire process is single-threaded, so at each point during initialization the browser is either parsing HTML code or executing a script, and each script runs without preemption. However, the scheduling of HTML parsing and script execution is sensitive to the precise timing of events, so interference may occur between individual scripts and between HTML parsing and scripts that access the same JavaScript objects. As a result, execution is nondeterministic, so testing the initialization by a single execution is generally insufficient to cover all possible behaviors—even if we fix all user input, the browser version, the window size, the machine clock, and other factors that may affect the execution.
Our approach is based on a dynamic analysis in which an execution is modeled by a trace that consists of different kinds of primitive actions. Some of these actions correspond to HTML parsing and others arise from JavaScript execution. Since event handlers execute atomically, we can associate an event identifier (EID) with each JavaScript action, identifying the event that triggered the action. As the size of the chunks read by the HTML parser is browser-dependent, we conservatively model the construction of each HTML element as a separate event. The actions are of different kinds:

**HTML-element-start**\([e, o, i]\) denotes the action of parsing an HTML start tag, where \(e\) is a unique EID and \(o\) is the constructed DOM element (including its attributes). The argument \(i\) specifies extra information about the element: \texttt{VISIBLE} means that the element is currently visible\(^4\) and for form fields \texttt{WRITABLE} means that the field is neither read-only nor disabled.

**focus**\([e, o]\) is the action of invoking the \texttt{focus} method on a DOM object \(o\), where \(e\) is the EID of the current event, or parsing an HTML element \(o\) with attribute \texttt{autofocus}. In either case, \(o\) is focused, meaning that keyboard input is directed to that element.

**register-event-handler**\([e, o, t, h]\) marks the registration of an event handler \(h\), where \(e\) is the EID of the event in which the event handler registration appears, \(t\) denotes the event type (e.g., \texttt{click}), and \(o\) is the DOM or XHR object on which the event handler is registered. This kind of action can appear either due to HTML parsing (e.g., an \texttt{onclick} attribute) or due to JavaScript execution (e.g., invoking \texttt{addEventListener} or setting \texttt{onreadystatechange}).

**dispatch**\([e, h, i]\) represents the beginning of a script being executed, where \(e\) is the EID of the new event and \(h\) is the event handler that is about to execute. The argument \(i\) specifies whether a “long delay” has happened. We have \(i = \texttt{LONG}\) if the event is a timer event with at least 500ms delay\(^5\) an XHR response event, or an external script loading event.

**write-form-field**\([e, o]\) means that a script with EID \(e\) has written to the value of an HTML form field \(o\).

**prevent-default**\([e]\) models invocation of \texttt{preventDefault} on the event object of the event with EID \(e\), which has the effect that the browser’s default event handling (e.g., following a link, in case of a click on an \texttt{a} element) is disabled for that event.

\(^4\)Our implementation uses the \texttt{true-visibility} JavaScript library (https://github.com/UseAllFive/true-visibility) to determine whether the user can interact with the element.

\(^5\)The 500ms threshold is not significant; changing it to 250ms or 1000ms makes practically no difference for the experimental results presented in Section 11.6.
11.4. INITIALIZATION RACE ERRORS

\textbf{crash}[e] indicates an uncaught exception (such exceptions terminate the current event handler).

\textbf{loaded} represents the pseudo-action that the initialization is completed.\footnote{Our implementation triggers \textbf{loaded} 5 seconds after the \texttt{load} event of the window object, which suffices in practice to await completion of XHR and timers.}

Other actions, in particular JavaScript instructions that read or write other object properties, are abstracted away when forming the trace.

In addition to the execution trace, we need the \textit{happens-before} relation $\preceq$ over the EIDs. This relation is easily constructed from the execution, as in previous work \cite{138, 146}. Intuitively, $e_1 \preceq e_2$ means that event $e_1$ must happen before $e_2$, which is the case if, for example, the trace contains HTML-element-start[$e_{1,\_}$] before HTML-element-start[$e_{2,\_}$] or register-event-handler[$e_{1,\_}, h$] before a corresponding dispatch[$e_{2, h, \_}$].

**Example** Loading \url{http://www.mckesson.com/} (see Figure 11.1) may yield the trace $\tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \cdot \tau_5 \cdot \tau_6$ where:

- $\tau_1 = \text{HTML-element-start}[e_1, o_{\text{input}}, \text{VISIBLE}, \text{WRITABLE}]$
- $\tau_2 = \text{HTML-element-start}[e_2, o_{\text{script}}]$
- $\tau_3 = \text{dispatch}[e_3, h_{\text{script}}, \text{LONG}]$
- $\tau_4 = \text{register-event-handler}[e_3, o_{\text{document}}, h_{\text{DOMContentLoaded}}]$
- $\tau_5 = \text{dispatch}[e_4, h_{\text{DOMContentLoaded}}]$
- $\tau_6 = \text{write-form-field}[e_4, o_{\text{input}}]$

We have, in particular, $e_1 \preceq e_3$ and $e_3 \preceq e_4$, and there is a long delay at $\tau_3$, which occurs between $\tau_1$ and $\tau_6$. This information suffices to detect the event race error described in Section 11.2, as we shall see in the following section.

### 11.4 Initialization Race Errors

The three examples presented in Section 11.2 represent different categories of common initialization race errors. We now explain how these categories can be characterized as \textit{trace patterns}, which forms the basis for the design of \texttt{InitRacer}. Such patterns are essentially simple regular expressions over an alphabet of primitive actions.

#### Form-Input-Overwritten

Initialization races that lead to form-input-overwritten errors can be characterized using the following trace pattern:

$$P_1 = \text{HTML-element-start}[e, o, \text{VISIBLE}, \text{WRITABLE}] \cdot \cdots \cdot$$

\begin{itemize}
  \item (write-form-field[$e', o$] | focus[$e', o'$])
\end{itemize}

where $o$ is an \texttt{input} or \texttt{select} element and $o \neq o'$
The pattern $P_1$ matches any trace that contains HTML-element-start[$e, o$, VISIBLE, WRITABLE] and this action is followed eventually by either write-form-field[$e', o$] or focus[$e', o$].

If a trace matches HTML-element-start[$e, o$, VISIBLE, WRITABLE] \ldots write-form-field[$e', o$] then a script may overwrite the value of the form field $o$ after the user has already changed it. Similarly, if a trace matches the pattern HTML-element-start[$e, o$, VISIBLE, WRITABLE] \ldots focus[$e', o'$] where $o \neq o'$ then form field $o$ will lose focus by the time the event with EID $e'$ is dispatched. This is problematic since the user may already be in the middle of modifying the value of the form field $o$. In both scenarios, the form field must be visible and writable.

The requirement that the form field $e$ in $P_1$ must be visible by the time it is declared serves to avoid spurious races where the user cannot possibly interact with $e$ before its value changes, or another element receives focus. In the following example, users can only interact with the search field (line 234) by clicking on the button (line 232) after the event listener (line 238), which toggles the visibility of the search field, has been registered.

```html
232 <button id="search-btn">Search</button>
233 <div id="dropdown-search" style="display: none">
234   <input type="text" id="search" />
235 </div>
236 <script>
237   document.getElementById("search").value = "Enter search terms...";
238   $("#search-btn").click(showSearchDropdown);
239 </script>
```

By that time, however, the value of the search field has already been updated (line 237).

The simple pattern $P_1$ may, however, lead to harmless races being reported: it may be practically impossible for a user to edit form field $o$ between its creation and the write-form-field or focus action, either because the entire HTML fragment is being parsed in one single chunk, or because the actions happen within a few milliseconds. This is the case for the following example:

```html
240 <input id="s" type="text">
241 <script>
242   document.getElementById("s").value = "Enter search terms...";
243 </script>
```

User input to the text field “s” may be overwritten by the script, but only if the edit occurs after line 240 has been parsed but before lines 241, 243 have been processed, which is very unlikely. For this reason, we adjust the pattern slightly:

$$P'_1 = \text{HTML-element-start}[e, o, \text{VISIBLE, WRITABLE}] \ldots \text{dispatch}[e', \_, \text{LONG}] \ldots (\text{write-form-field}[e'', o] \mid \text{focus}[e'', o'])$$

where $o$ is an input or select element, $o \neq o'$, and $e \preceq e' \preceq e''$
Now the pattern only matches a trace if there is a long delay \((\text{dispatch}[e', \_, \text{LONG}])\) between events \(e\) and \(e''\). We use happens-before \((e \preceq e' \preceq e'')\) to ensure that only traces are matched where the long delay is guaranteed to occur between the two events and is not just an effect of nondeterministic scheduling. We compare \(P_1\) and \(P'_1\) empirically in Section 11.6.

This characterization of form-input-overwritten race errors using trace patterns has several advantages over state-of-the-art alternatives, such as EventRacer and \(R^4\): (i) it identifies form-input-overwritten errors even when no user events are performed, (ii) by taking long delays into account, it avoids reporting interleavings that are unlikely to manifest, and (iii) it does not report spurious races even when the happens-before relation is incomplete, which may happen due to incomplete modeling of the DOM API (\(P'_1\) only matches a trace if there is a long delay according to happens-before, whereas existing dynamic race detectors report more (spurious) races when the happens-before relation is incomplete).

Interestingly, these trace patterns can even detect some race issues that do not involve any JavaScript code, unlike previous race detection tools. In the following example, the form field on line 3 receives focus after `empty.js` has been loaded.

```html
1 <input type="text" />
2 <script src="empty.js"></script>
3 <input type="text" autofocus />
```

Since the scheduling of the actions in this example depends on the network, it is possible that the user has already started editing the form field on line 1 when line 3 is processed.

**Late-Event-Handler-Registration**

As illustrated in Section 11.2, undesirable behavior may occur if an event handler is registered too late, i.e., after an event that was supposed to trigger the event handler has already been dispatched. Trace pattern \(P_2\) is designed to identify exactly such situations:

\[
P_2 = \text{HTML-element-start}[e, o, i] \cdots \text{dispatch}[e', \_, \text{LONG}] \cdots \text{register-event-handler}[e'', o, t, \_]\]

where \(e \preceq e' \preceq e''\) and \(\text{isUserEvent}(t) \Rightarrow \text{VISIBLE} \in i\)

Here, the predicate \(\text{isUserEvent}(t)\) holds if the event type \(t\) is a user event (e.g., `click` or `keydown`).

Note that, similar to \(P'_1\) from the previous section, trace pattern \(P_2\) only reports races where the undesirable interleavings are likely to happen in practice due to a long delay between \(e'\) and \(e''\). The requirement \(\text{isUserEvent}(t) \Rightarrow \text{VISIBLE} \in i\) is important for ruling out infeasible interleavings. Indeed, it is not uncommon for user event handlers to be registered on DOM elements that
are invisible until the page has been loaded, or until a user event has been performed (recall the example on lines 232–238 from Section 11.4 where the search field was initially hidden). In such situations, the user cannot interact with the element until the event handler has been registered, which existing race detectors such as EVENTRACER and R^4 do not account for.

Although trace pattern P_2 suffices for identifying late-event-handler-registration errors, it will often lead to an overwhelming amount of reports, since many modern web pages register hundreds of event handlers during loading, many of which are registered late. Yet, most of these web pages work reasonably well. In the following, we therefore refine P_2 to focus on situations that are more likely to affect the user experience.

Late-event-handler-registrations for system events are generally problematic. Web applications often use the load event for external scripts, iframes, and images, and if the event handler is registered late, the missing event handler execution can only be remedied if the user reloads the page.

For user events, as discussed in Section 11.2, there are situations where the user simply needs to repeat the user event after a late-event-handler-registration error in order to obtain the desired behavior, which is annoying but not a major problem. It is generally more problematic if the event handler calls preventDefault on the event object, since this prevents the browser’s default event handling (recall the example from Section 11.2). We therefore refine the pattern using the prevent-default action to match only traces where the user event handler invokes preventDefault:

\[
P'_2 = \text{HTML-element-start}[e, o, i] \cdots \text{dispatch}[e', _, \text{LONG}] \cdots \text{register-event-handler}[e'', o, h] \cdot \tau
\]

where \( e \preceq e' \preceq e'' \), and isUserEvent\( (t) \Rightarrow (\text{VISIBLE} \in i \land \tau \text{ matches } \text{dispatch}[e''', h, _, \cdots \text{prevent-default}[e''']]) \)

The sub-pattern \( \text{dispatch}[e''', h, _, \cdots \text{prevent-default}[e'''] \) applies if \( t \) is a user event and checks that the event handler \( h \) has invoked preventDefault in the trace. Existing race detectors do not take into account whether late-event-handler-registration races affect the browser’s default event handling.

**Access-Before-Definition**

Access-before-definition errors arise when a variable or object property is read before it has been initialized, as in the example from Section 11.2. In JavaScript, a ReferenceError is thrown if an attempt is made to read a variable that has not been declared, and a TypeError is thrown when dereferencing a property from null or undefined, and when invoking a non-function value. As mentioned in Section 11.3 such exceptions cause the current event handler to abort, which may leave the program in an undesired state. Similar to late-event-handler-registration errors, crashes in system event handlers are generally more problematic than ones in user event handlers.
For simplicity, we classify access-before-definition errors as being harmful only if they lead to uncaught exceptions. Uncatched exceptions may of course appear without relation to initialization races. Such exceptions are likely benign, because they also would manifest during ordinary testing. For this reason, we want to identify each event handler that may crash during initialization but not after initialization.

The following trace pattern \( P_{3a} \) matches a trace \( \tau_a \) if it contains an event handler \( h \) that crashes during initialization, i.e., before the \texttt{loaded} action:

\[
P_{3a} = \text{register-event-handler}[e_a, o_a, t, h] \cdots \text{dispatch}[e'_a, h] \cdots \text{crash}[e''_a] \cdots \text{loaded}
\]

If another trace \( \tau_b \) witnesses that \( h \) may also crash after initialization, then we filter away the crash in \( h \) as likely benign. This situation is captured by the trace pattern \( P_{3b} \):

\[
P_{3b} = \text{register-event-handler}[e_b, o_b, t, h] \cdots \text{loaded} \cdots \text{dispatch}[e'_b, h] \cdots \text{crash}[e''_b]
\]

Section 11.5 explains how to obtain the traces \( \tau_a \) and \( \tau_b \).

Compared to existing race error detectors, this simple approach has several important properties. First, it does not report an overwhelming number of harmless reports in presence of ad-hoc synchronization, unlike EventRacer. Second, similar to \( R^4 \) but unlike EventRacer, it only reports races that actually lead to errors. As we shall see in the next section, we also leverage adverse execution to become independent of specific user event sequences.

11.5 The InitRacer Approach

InitRacer works in three phases that execute the initialization of the given web application in different modes, controlled by instrumentation performed by a proxy server.

Phase 1: Observation Mode Execution

The first phase is dedicated to detecting form-input-overwritten and late-event-handler-registration errors using trace patterns \( P'_1 \) and \( P_2 \). Errors characterized by these patterns can be detected by merely observing the instructions that execute during initialization. Thus, in phase 1 InitRacer simply opens the given web page and collects the trace until the \texttt{loaded} action occurs as described in Section 11.3.

To emit the necessary actions for trace pattern \( P'_1 \), the instrumentation intercepts assignments to the \texttt{value} property of \texttt{input} elements, assignments

\[\text{\footnotesize 7For future work, it may be interesting to also consider the side-effects of code that was not executed because of a crash, similar to the use of \texttt{prevent-default} in Section 11.4.}\]
to the `selectedIndex` property of `select` elements, invocations of the `focus` method of HTML elements, and declarations of HTML elements with the `autofocus` attribute. The relevant actions for \( P_2 \) are generated by intercepting invocations of the `addEventListener` method and assignments to event attributes (e.g., `onclick`, `onreadystatechange`). Event handler attributes in the HTML are ignored; such registrations are never late. `InitRacer` intercepts all property assignments by also dynamically instrumenting code that is passed to `eval` at runtime.

Determining happens-before Patterns \( P'_1 \) and \( P_2 \) both rely on the happens-before relation. This relation is built on-the-fly by monitoring the parsing of HTML elements and the execution of scripts. As explained in Section 11.3, each action takes place in the context of an event. An important step in building the happens-before relation is to wrap each function that is registered as an event handler, in order to record the current event \( e_r \) at the time of the registration. When the wrapper is eventually invoked, it is then possible to insert a happens-before edge between the event in which the handler was registered and the current event \( e_n \), i.e. \( e_r \preceq e_n \).

Note that care needs to be taken when wrapping event handlers because an application may unregister event handlers (using the function `removeEventListener`). If the application passes a reference to the unwrapped event handler function, rather than the wrapped one, the removal fails silently. For this reason, `InitRacer` maintains a map from functions to their wrappers, and intercepts calls to `removeEventListener` to ensure that the correct function is passed.

Form-input-overwrittendetection Trace pattern \( P'_1 \) for form-input-overwritten errors is susceptible to false positives, as illustrated by the following example:

```html
text id="search" value="Default" />
... // assume long delay
<script>

var input = document.getElementById('search');
if (input.value === 'Default') {
 input.value = 'Enter search terms...';
} else { /* avoid overwriting user input */ }
</script>
```

Executing this code produces a trace that matches \( P'_1 \), but the `write-form-field` action (line 9) is guarded by line 8, which checks whether the user has changed the field value. To avoid such situations, we intervene in the observation mode execution as follows: (i) when a form field is declared, `InitRacer` immediately changes its value to a random non-default one, and (ii) when the web page has loaded, `InitRacer` checks if the value of each form field has changed since its declaration.

\[ 8 \text{The implementation uses the } \texttt{falafel} \text{ instrumentation library (https://github.com/substack/node-falafel).} \]
11.5. THE INITRACER APPROACH

Phase 2: Adverse Mode Execution

As discussed, phase 1 provides information for patterns $P'_1$ and $P_2$. The purpose of phase 2 is to collect information needed for the patterns $P'_2$ and $P_{3a}$ (in particular, trace $\tau_a$). Notice that in $P'_2$, the sub-pattern $\text{prevent-default}[e'', h, \_]$

... $\cdot \cdot \cdot \text{prevent-default}[e'']$ will not appear in the trace unless the event handler $h$ has been triggered. Similarly, in $P_{3a}$ and $P_{3b}$, the sub-pattern $\text{dispatch}[e', h]$

... $\cdot \cdot \cdot \text{crash}[e']$ will not be matched unless $h$ has been triggered. In phase 2, INITRACER reloads the given web page in a mode where it systematically simulates events during initialization, in an attempt to reach the relevant prevent-default and crash actions.

Some web pages register hundreds (or even thousands) of event handlers during initialization, so repeatedly reloading the page and injecting a single one of the individual events would not scale well. (Waiting for initialization to finish may take up to 20 seconds, due to the instrumentation needed for the analysis.) We therefore use the idea of adverse execution from Thor [9]: in a single execution, INITRACER simulates all events for which event handlers have been registered. The events are injected eagerly, as soon as possible after the event handlers have been registered. This does not necessarily lead to the “most adversarial” event ordering, but one that developers would not normally observe.

Rather than faithfully simulating users moving the mouse pointer over the screen and interacting with the web page via mouse and keyboard clicks, etc., we use the idea of approximate execution from AppDoctor [84]: to simulate an event INITRACER simply invokes the event handler function directly. This is fast and easy to implement, in particular because it does not require browser modifications. The drawback is that the resulting executions may not be feasible in ordinary execution. For example, we bypass the browser’s event bubbling/capturing mechanism, and we ignore the fact that it is unrealistic to trigger keyup events without preceding keydown events. Phase 3, which we explain later in this section, can filter away some false positives that arise due to artifacts of adverse and approximate execution.

In more detail, phase 2 of INITRACER works as follows. At event handler registrations, INITRACER wraps the event handler function using try-finally so that we can inject code at the exit of the event handler. The code we inject generates fake event objects and invokes the event handlers that have been registered. Each invocation is put into a try-catch block so that we can detect crash events. Note that, because we invoke the event handlers directly, we do not have to worry about the browser’s default actions for the events (e.g., submitting forms, or following links). An alternative approach to inject events would be to use the built-in function dispatchEvent, which simulates events more faithfully, but it does not allow us to control exactly

---

9If one wants to find a broader range of access-before-definition errors beyond uncaught exceptions, it is possible to apply the dynamic analysis from DLint [63].
which event handlers are executed. Certain kinds of events—DOMContentLoaded, load, and unload—play a special role in the lifecycle of the web page, so InitRacer does not inject invocations of event handlers for those events. Our current implementation also does not injectreadystatechange events, since it is difficult to automatically generate a meaningful XHR response; in a future version we will record and reuse the responses from phase 1. Finally, executing event handlers may have undesired side effects, such as form submission, page redirects, alert popups, opening the print dialog, etc., so InitRacer intercepts and disables such effects. For example, form submissions are disabled by monkey-patching the HTMLFormElement.prototype.submit method.

It is important to keep observation mode and adverse mode apart. Using adverse mode execution as basis for detecting form-input-overwritten errors would result in false positives, caused by injection of event handlers with write-form-field actions. As an example, the event handler for a “reset form” button writes to form fields, but that does not imply existence of a form-input-overwritten error. Also, to limit interference of injected events, which could cause spurious matches of $P_2^\prime$, register-event-handler actions that belong to the injected events are omitted from the generated traces.

The existing tools EventRacer and $R^4$ find races only in code that has been executed by a given user event sequence. EventRacer does have an automatic exploration mode, but that is quite limited and only triggers a small fraction of the relevant events. In contrast, InitRacer’s adverse mode execution simulates all events for which event handlers have been registered, to observe their effect when interleaved in the initialization of the web application.

Phase 3: Validation Mode Execution

As explained above, the aggressive injection of event handler invocations in phase 2 may generate traces that are impossible or unlikely in actual execution, which may result in harmless races but also false positives especially for the access-before-definition error detection. For this reason, phase 3 attempts to validate potential initialization errors by reloading the web page again, once for each potential access-before-definition error that was detected in phase 2, and injecting only the single event handler containing the crash action. If this causes the error to disappear, InitRacer by default treats it as a false positive and omits it from its report.

This approach has another benefit: it eliminates (true) errors that only manifest if multiple user events occur during the initialization. Such errors are inevitably less likely to occur in practice, which is why we aim for detecting initialization race errors that involve at most one user event. Note that InitRacer can be adjusted to explore race errors that only manifest when multiple user events are triggered during the initialization. For example, given an error that manifests in adverse execution mode where all event handlers are executed
11.5. THE INITRACER APPROACH

eagerly, delta debugging [190] could be applied to find the minimal set of event
handlers that must be triggered during initialization to detect the error.

In addition to injecting a single event handler during loading, INITRACER
also injects the same event handler after the page has loaded, such that the
trace ($\tau_b$) can be used for trace pattern $P_{3b}$. This serves to identify access-
before-definition errors that only manifest during initialization. The number of
validation mode executions is generally much lower than the total number of
events injected in phase 2, so keeping these two phases separate is important
for performance. This approach is inspired by Thor that performs a similar
validation step to isolate the causes of failures [9].

The validation mechanism requires a way to identify the same event handler
registration across two executions. We employ a pragmatic approach that we
have found to work well: event handler registrations are identified by the name
of the target (e.g., div, img, document), the target’s location in the source code
(if available), the target’s visibility, the event type (e.g., click), and the event
handler’s source code (found by calling toString on the function).

Validation mode execution is not a perfect filter against harmless races and
false positives. It eliminates most interference due to injected event handlers in
adverse mode, but makes no attempt to prevent false positives that may appear
due to approximate execution. This is a pragmatic design choice (implementing
a precise validation mechanism, as AppDoctor’s faithful mode [84], is impossible
without browser modifications).

Error Diagnosis

To support error reproduction and debugging, each report generated by INIT-
RACER concisely shows the relevant actions, including source code and stack
traces, and with screenshots highlighting the involved HTML elements. The
reported issues are grouped according to the three categories and the involved
actions.

Figure 11.4a presents the report that has been generated by INITRACER
for http://www.apple.com/. The bottom of this report shows a screenshot of
the web page, where the UI elements that are involved in a race have been
highlighted. For example, the magnified part of Figure 11.4a highlights that
INITRACER has reported a late-event-handler-registration warning for the
click event type (with ID 6) for the search icon. (The error that causes this
warning is the one described in detail in Section 11.2.) By inspecting the
warning with ID 6 in the table, it can be seen (from column “Name”) that the
problematic event handler is registered on the a element that is declared on line
146 column 5 in the HTML source code, and (from column “Stack trace”) that
the event handler registration is performed in the file ac-globalnav.built.js.
These pieces of information are useful for diagnosing the warning. A natural
first step in debugging is to inspect the expected behavior. Figure 11.4b
shows the screen that results from clicking on the search icon that has been
Figure 11.4: (a) The InitRacer report for \url{http://www.apple.com/}, where the magnified part shows the warning markers placed by InitRacer. (b) The screen resulting from clicking on the search icon after the page has loaded, which is the normal behavior. (c) The screen resulting from clicking on the search icon before the page has fully loaded.
highlighted by InitRacer after the web page has finished loading. In this scenario, the user is presented with a search field in a dropdown. Figure 11.4c, on the other hand, shows the screen that appears when clicking on the search icon before the page (and, in particular, ac-globalnav.built.js) has loaded. In this case, the user is redirected to http://www.apple.com/us/search (see Section 11.2 for details). This behavior can easily be reproduced by simulating a slow network, for example, by enabling throttling (on the “Network” panel) in Chrome DevTools.

11.6 Evaluation

We aim to answer the following research questions through an empirical evaluation:

RQ1 How many form-input-overwritten errors, late-event-handler-registration errors, and access-before-definition errors does InitRacer report on real websites?

RQ2 How fast is InitRacer on real websites?

RQ3 How often do the warnings reported by InitRacer identify actual errors?

RQ4 How does InitRacer compare with EventRacer [146] and R⁴ [91] in terms of usefulness?

Experimental Methodology

The empirical evaluation is based on websites of the 100 largest companies from the Fortune 500 list, similar to evaluations of previous event race tools [11, 91, 146]. To ensure reproducibility of our results we use an intercepting HTTP proxy for recording the server responses observed in interactions with the web pages under consideration. The implementation of InitRacer, recordings of server responses, and all experimental data are available at http://www.brics.dk/initracer/.

Answers to RQ1 and RQ2 are obtained by running InitRacer on each of the web pages and counting the number of reported issues in each category. Given the complexity of the websites under consideration, only a representative subset of these issues is considered to answer RQ3 and RQ4. We manually attempt to reproduce all form-input-overwritten errors. For late-event-handler-registration and access-before-definition errors, we consider all warnings from 10 randomly selected websites. All experiments are run in Google Chrome on Ubuntu 15.10 (Intel Core i7-3770 CPU and 16 GB RAM).
Table 11.1: Results.

<table>
<thead>
<tr>
<th>Website</th>
<th>FIO</th>
<th>LEHR User</th>
<th>Sys. ABD</th>
<th>User</th>
<th>Sys.</th>
</tr>
</thead>
<tbody>
<tr>
<td>21cf.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3m.com</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aa.com</td>
<td>0</td>
<td>19</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>adm.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>aetna.com</td>
<td>0</td>
<td>13</td>
<td>0</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>aig.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>allstate.com</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>amazon.com</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>americanexpress.com</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>anthem.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>apple.com</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>att.com</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>bankofamerica.com</td>
<td>4</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>bestbuy.com</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>cardinalhealth.com</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>chevron.com</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>chsinc.com</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>cigna.com</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cisco.com</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>conocophillips.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>costco.com</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>cvshealth.com</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>deere.com</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>delta.com</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>directv.com</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>disney.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dupont.com</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>energytransfer.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>express-scripts.com</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fedex.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>freddiemac.com</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>ge.com</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>gm.com</td>
<td>0</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>goldmansachs.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>halliburton.com</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>homedepot.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>honeywell.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>hp.com</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 11.1 – continued from previous page

<table>
<thead>
<tr>
<th>Website</th>
<th>FIO</th>
<th>LEHR User</th>
<th>Sys.</th>
<th>ABD User</th>
<th>Sys.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>humana.com</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>11</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>ibm.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ingrammicro.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>intel.com</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>intlfcstone.com</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>jnj.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lockheedmartin.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lowes.com</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>mckesson.com</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>merck.com</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>metlife.com</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>microsoft.com</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>mondelezinternational.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>morganstanley.com</td>
<td>0</td>
<td>51</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>nationwide.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>pepsico.com</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>pfizer.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>phillips66.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>prudential.com</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>statefarm.com</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>tsocorp.com</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>united.com</td>
<td>10</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>unitedhealthgroup.com</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>31</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>ups.com</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>us.coca-cola.com</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>utc.com</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>verizonwireless.com</td>
<td>0</td>
<td>24</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td>walgreens.com</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>walmart.com</td>
<td>0</td>
<td>41</td>
<td>134</td>
<td>1</td>
<td>0</td>
<td>175</td>
</tr>
<tr>
<td>wellsfargo.com</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>wfscorp.com</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>xfinity.com</td>
<td>1</td>
<td>84</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>85</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>24</td>
<td>577</td>
<td>172</td>
<td>308</td>
<td>10</td>
<td>595</td>
</tr>
</tbody>
</table>
RQ1: Number of Reported Warnings

Table 11.1 shows the number of warnings generated by InitRacer for the front pages of the 100 websites, in each of the three categories. The columns in this table show, for each website, the number of form-input-overwritten (FIO) warnings, late-event-handler-registration (LEHR) warnings, and access-before-definition (ABD) warnings. For LEHR and ABD, we further distinguish between user and system event handler registrations (cf. Sections 11.4–11.4). We next discuss the findings for each category.

FIO As can be seen in the ‘total’ row of the table, InitRacer finds a total of 24 FIO warnings due to matches with pattern $P_1'$. Without the check for form field value changes, 2 additional warnings would have been reported. 10 of the 24 warnings are due to write-form-field and 14 are due to focus.

LEHR InitRacer reports a total of 749 LEHR warnings due to matches with pattern $P_2'$. Of which 577 involve user event handlers, and 172 involve system event handlers.

ABD The 100 $\tau_a$-traces collected by InitRacer gives rise to 318 matches with $P_3a$ that can be validated by injecting only a single event handler. Of these, 308 involve user event handlers and 10 involve system event handlers. Without the validation phase, 23 additional warnings (possibly due to interference) would have been reported. Of the 318 validated warnings, 246 user event handlers only crash during initialization (i.e., the corresponding trace $\tau_b$ from validation mode does not match $P_3b$). InitRacer was only able to confirm that 1 of the 10 system event handlers crashes after initialization. The fact that InitRacer succeeds in validating most warnings suggests that adverse mode execution does not introduce too much interference. In fact, on 39 of the 47 websites that have at least one ABD crash, InitRacer is able to validate all crashing user and system event handlers.

RQ2: Performance

In InitRacer’s observation phase and adverse phase, the instrumented web page is loaded just once. Although this is significantly slower compared to loading the original web page (mostly due to instrumentation overhead and...

---

12InitRacer does not report any warnings for 30 of the 100 websites; those have been excluded from the table.

13The less precise race pattern $P_1'$, reports 4 additional write-form-field warnings, suggesting that most issues do involve a long delay. The long delay requirement in $P_1'$ is still useful because, without it, a race detector may report a warning even after the error has been fixed.

14Using the less precise pattern $P_2'$, which does not take into account long delays or whether a registered user event handler calls preventDefault, leads to an overwhelming increase in the number of warnings: 9189 and 585 for user and system event handler registrations, respectively.
11.6. EVALUATION

Taking screenshots) this can be done in 7 to 40 seconds depending on the web page (21 seconds on average).

In the validation phase, INITRACER loads the web page once for each ABD warning. On average, this takes around 1 minute per website with at least one ABD warning (23 minutes for [https://www.deere.com/] which has the most ABD warnings), while it is free for the remaining 53 websites.

RQ3: Qualitative Study

FIO We manually inspected all 10 FIO warnings where the user’s input to a text field is overwritten. For 6 of the warnings we were able to reproduce the errors. 2 of the warnings are spurious due to false positives from the true-visibility JavaScript library that is used by INITRACER, and can be fixed by better visibility checking. We were unable to reproduce the remaining 2 warnings in Google Chrome (apparently due to the browser’s chunk size; in both cases, the form field did not become visible on the screen until after the field’s value had already been updated).

LEHR We manually investigated all 75 LEHR warnings from 10 randomly selected websites. On 8 websites INITRACER reports 27 warnings associated with functionality that is either disabled or malfunctioning during initialization. The most commonly occurring situation is that functionality is disabled during loading. For example, INITRACER finds hyperlinks that fail to open a dialog, a menu, a login form, or redirect the user to a different page, as well as features such as auto-completion that are not enabled during loading. On [http://www.bestbuy.com/] INITRACER detects a scenario where the web page redirects the user when signing up for a newsletter instead of sending an XHR request, as the web page does after loading. As with the example from Section 11.2, the user’s experience is effectively degraded to the one offered to users whose browsers do not execute JavaScript. In another example from [https://www.aetna.com/] a late-event-handler-registration for the submit event causes the user’s search query to be lost upon submitting a form.

The remaining 48 LEHR warnings are spurious. Of these, 11 are false positives from true-visibility and 1 could not be reproduced (possibly due to the browser’s chunk size). Another 5 (all from [https://www.aetna.com/]) are due to click event handlers registered on hyperlinks that invoke preventDefault on the event object and then redirects the user by assigning window.location. These event handlers are superfluous: they re-implement the browser’s mechanism for redirecting the user when a hyperlink is clicked, and one could argue that these should simply be removed. The remaining 31 of the 48 spurious warnings are

\[^{15}\text{Although most false positives originating from true-visibility can be fixed easily, some are non-trivial. For example, a form field from [https://www.bankofamerica.com/] takes up 230x32 pixels on the screen, but is visually indistinguishable from the background until the page has loaded.}\]
due to event handlers that track the user once the page has been loaded. For example, 12 are from [https://www.microsoft.com/](https://www.microsoft.com/) where an event handler is registered that prevents the browser from redirecting the user, such that an XHR request can be sent before the redirection.

**ABD** On 10 randomly selected web pages InitRacer reports 133 crashes, of which we are able to reproduce 125 manually. Of these 125 errors, 14 affect user experience. In an example from [https://www.ups.com/](https://www.ups.com/) an uncaught exception in a click event handler causes the user to get redirected to the splash page when clicking “Change Language”, rather than being presented with a dropdown (as after the page has loaded). The remaining 111 errors are caused by calls to, e.g., analytics libraries that do not get loaded until the end of the initialization. On [https://www.deere.com/](https://www.deere.com/) which has 79 such crashes, the developers are aware of the problem and test if the library has been loaded:

```javascript
if (omniEvents) omniEvents.globalNav(this, 'header:Products');
```

Unfortunately, the test itself leads to a `ReferenceError`.

The 8 warnings that are not reproducible involve interleavings that are extremely unlikely. Each case involves an event handler that crashes by the time it is registered (due to invoking a function that has not yet been declared), but only until the browser has parsed a few more HTML elements.

**Debugging** Although InitRacer reports relatively many LEHR and ABD warnings for some websites, many of those warnings have similar characteristics, and InitRacer’s screenshots and grouping of related issues reduces the debugging effort significantly. For example, the 5 harmless LEHR warnings from [https://www.aetna.com/](https://www.aetna.com/) are due to 5 menu items on the page that share the same event handler. InitRacer groups these races, making it easy to recognize that they are related, and the provided screenshot reveals that each warning is associated with a corresponding menu item, making it trivial to determine that they are in fact all caused by the same problem. Overall, only 11 of the 48 spurious LEHR warnings needed to be investigated in detail.

Furthermore, the three error categories have the desirable property that warnings can easily be tested for reproducibility. For example, a FIO error can be tested for reproducibility by attempting to modify the value of the involved form field before the JavaScript instruction identified by InitRacer updates the form field’s value. For the FIO error from [http://www.mckesson.com/](http://www.mckesson.com/) (Section 11.2), this can be done by postponing the script `search.min.js`. To carry out this evaluation, we used the mitmproxy tool to restrict a URL of our choice to schedules that are unlikely in normal circumstances.

InitRacer also makes it easy to debug the 111 analytics related ABD warnings. For example, the report indicates that the 79 warnings from [https://www.deere.com/](https://www.deere.com/) are all due to a `ReferenceError` on the variable `omniEvents`, and the screenshot clearly indicates which HTML elements are involved.
In summary, the reports generated by InitRacer were sufficiently informative to enable us to quickly identify and reproduce numerous initialization race errors in (often obfuscated) websites that we were not previously familiar with. We generally needed only few minutes to inspect and reproduce a single race, and all 218 races were classified in approximately one day of work.

**RQ4: Comparison to State-of-the-Art**

EventRacer often reports an overwhelming amount of event races. On [https://www.united.com/](https://www.united.com/) alone, EventRacer reports 16,822 races, of which 533 are uncovered. After manually applying EventRacer to 10 initialization errors detected by InitRacer, we found that only 1 of the 10 is uncovered. When applying EventRacer to all 100 websites in our study, we found that it reports more than 1,000 races for 49 of the websites, and more than 10,000 races for 19 of them. Inevitably, most are either harmless or false positives.

The reports generated by EventRacer do not support debugging very well. An EventRacer report is simply a trace of low-level read and write operations, structured in events, with the additional information that two events in the trace are unordered according to the happens-before relation.

R4 systematically explores the possible schedules. An R4 report consists of a specific sequence of steps taken by the browser in order to expose a given error, e.g., an uncaught exception or visual difference. It is very difficult to reproduce errors detected by R4, since there is no means to replay a specific schedule in a real browser and no support is offered for further diagnosis. Each warning reported by InitRacer can be tested for reproducibility in a well-defined way.

Both EventRacer and R4 build on an old version of WebKit, which makes these tools platform-dependent and also leads to problems when analyzing modern web pages. For example, the message “Your browser is not supported” is shown when loading [http://www.ford.com/](http://www.ford.com/) and 13 other of the 100 websites from our study. On [https://www.kroger.com/](https://www.kroger.com/), the web page remains blank even after loading. On [http://www.citigroup.com/](http://www.citigroup.com/) and [http://www.marathonpetroleum.com/](http://www.marathonpetroleum.com/), EventRacer fails to analyze the web page of interest, since its auto-exploration triggers a redirect to another page, and EventRacer, unlike InitRacer, does not disable such undesirable side effects ([Section 11.5](#)). Similarly, [http://www.ge.com/](http://www.ge.com/) keeps reloading in an infinite loop when analyzed using EventRacer. Updating EventRacer and R4 is a nontrivial task; the tools are more than 80,000 commits behind the latest version of WebKit.\footnote{\url{https://github.com/eth-srl/webkit}}

---

\footnote{In an uncovered race, both execution orderings of the corresponding memory accesses are guaranteed to be possible.}
Threats to Validity

It is possible that the websites considered in our evaluation do not provide a representative mix of programming styles and JavaScript feature usage. However, this style of evaluation was also used in previous work on event race tools \cite{11, 91, 146}. A related issue is that the websites under consideration evolve continually. To enable reproducibility, we recorded all websites, and will make \textit{InitRacer} and recorded websites available as an artifact. Full reproducibility is not always possible. For example, if a website exhibits nondeterminism that is unrelated to user events, then executions may differ from ones we observed. To the best of our knowledge such situations do not affect the conclusion of our experiments. Furthermore, while \textit{InitRacer} is platform-independent, it should be noted that the behavior of JavaScript code may vary across different (versions of) browsers, so slightly different results might be expected on different platforms.

11.7 Related Work

It has long been known (see, e.g., Saltzer\cite{149}) that software may exhibit race conditions, i.e., situations where program behavior depends on the nondeterministic ordering of tasks that is not under the control of the programmer. Race conditions are typically considered errors if some but not all orderings result in undesirable program behavior. This problem has been studied in depth for programming languages with shared-memory concurrency (see, e.g., Boyapati and Rinard\cite{32}, Flanagan and Freund\cite{54, 55, 56}, Hammer et al.\cite{75}, Naik et al.\cite{133}, Savage et al.\cite{152}, Voung et al.\cite{174}), but races also appear in languages without concurrency that feature asynchronous or event-driven control flow. The remainder of this section focuses on work involving race conditions in event-driven systems, specifically JavaScript web applications.

Detecting Event Races in JavaScript Web Applications The fact that event race errors occur in JavaScript programs was initially observed by Steen\cite{163} and Ide et al.\cite{87}. Zheng et al.\cite{197} presented the first approach to automatically find such errors, however, it is based on a static analysis that is insufficiently precise to handle real websites. The \textit{WebRacer} tool by Petrov et al.\cite{138} instead uses dynamic analysis, based on a JavaScript-specific happens-before relation. Raychev et al.\cite{146} observed that the number of event races in a JavaScript web application can be overwhelming, which motivated a notion of race coverage. By focusing on uncovered races, their \textit{EventRacer} tool dramatically reduces the number of reported races, but it may hide harmful errors. As discussed, \textit{EventRacer} (as well as its predecessor \textit{WebRacer}) has several limitations that hinder practical use. For example, \textit{EventRacer} reports races regardless of whether they may be harmful. It does not account for “long delays” or visibility of HTML elements (Sections \ref{sect:11.4}}
11.7. RELATED WORK

and 11.4), and it sometimes reports ad-hoc synchronization, which has been inserted into the code to prevent race errors, as likely harmful. Furthermore, incomplete modeling of the happens-before relation, which is inevitable due to the rapid development of the browser APIs, leads to more races being reported (Section 11.4).

Another practical problem with EventRacer is that it builds on top of a version of WebKit that is thousands of commits behind the current release, which, as discussed in Section 11.6 (RQ4), makes it unsuitable for analyzing some modern web pages. InitRacer instead relies on dynamic code instrumentation and is fully platform independent.

EventRacer only detects races in code that has been executed by a given user event sequence, and its auto-exploration only triggers a small fraction of the relevant events, causing it to miss harmful races. For example, it fails to automatically detect the three event race errors in Section 11.2. In contrast, InitRacer’s adverse execution mode explores all registered user event handlers, and its analysis is not limited to races that appear with a specific user event sequence, thus enabling it to find more errors in a single run. Moreover, InitRacer gains precision by identifying long delays in the initialization process and taking HTML element visibility into account.

EventRacer outputs only the trace that contains the races, with no information about how the races may affect the execution, which makes it difficult to diagnose and debug the errors. As discussed in Section 11.5, InitRacer provides detailed diagnostic information to facilitate debugging.

Diagnosing Event Races There is an important difference between races and the errors they may cause. Many races are completely harmless. Despite its attempt to classify races, EventRacer does not have evidence that the two operations involved in a race can in fact be reordered, or that the opposite ordering (if it exists) is harmful. For example, a runtime exception originating from an access-before-definition could be caught by a catch block in the program, rendering the race harmless. In general, any tool for detecting race errors must reason about the effects of the individual races, for example, by establishing that only one of the two possible orderings of a race is “good”. The fact that many races are harmless has motivated the work discussed below on detecting races that lead to actual errors.

The $R^4$ tool by Jensen et al. [91] uses a stateless model checking approach to analyze the entire state space of a nondeterministic web application, relative to a given user event sequence. In addition, it uses a notion of approximate replay to investigate the effect of each race. $R^4$ explicitly filters away races involving late registration of event handlers and classifies harmfulness of each detected race according to its effect on, e.g., the HTML DOM and uncaught exceptions. In comparison, InitRacer specifically targets the three different types of initialization races described in Section 11.4.

The technique by Mutlu et al. [131] is designed for detecting races that
affect persistent storage. In their view, any error that may disappear by reloading the web page is considered benign, even though the error may damage functionality or user experience, which makes their technique unsuitable for detecting initialization race errors.

The WAVE tool by Hong et al. [78] aims to investigate the effect of each race by executing alternative schedulings of the same user event sequence, like $R^4$ but without using model checking or approximate replay techniques. Previous work has shown that the approach taken by WAVE often results in an overwhelming amount of false positives [91].

The RClassify tool by Zhang and Wang [192] aims to determine whether a given race is harmful. It uses a replay-based method that forces the execution of a pair of racing events in two different orders and assesses the impact on the program state by comparing the values stored in, for example, the DOM and the JavaScript variables. Similar to InitRacer, RClassify works using instrumentation and is platform independent. However, unlike InitRacer, RClassify requires as input a set of races produced by a separate race detection tool; the experiments reported by Zhang and Wang were based on EventRacer to supply this initial set of races.

In contrast to these event race classification techniques, InitRacer entirely avoids the need for explicitly identifying racing memory accesses and is capable of detecting initialization race errors, with high speed and accuracy, using relatively simple instrumentation techniques.

Automated Repair of Event Races Experience thus far has been that event races are extremely common, and that many event races have similar characteristics and occur for similar reasons. Beyond detecting races and determining their harmfulness, recent work has focused on automatic repair of web applications, in ARROW [176] by reordering script fragments, and in EventRaceCommander [11] by controlling how event handlers are scheduled for execution. Such techniques are complementary to InitRacer. For example, we believe it is possible to define repair policies for EventRaceCommander that are tailored to the different categories of initialization races errors that are targeted by InitRacer.

11.8 Conclusion

We have presented a simple but effective technique for detecting initialization race errors in JavaScript web applications, and its implementation in a tool called InitRacer. Our technique matches a small number of patterns against the trace of actions performed by a web application, using a three-phase approach to observe actions in different execution modes. Unlike previous techniques, InitRacer is based on dynamic code instrumentation and is therefore platform-independent. Furthermore, InitRacer produces informative error messages to support diagnosing the causes and effects of the races.
In an evaluation on 100 real-world websites, INITRACER reports 1085 initialization races, while providing informative explanations of their causes and effects. A manual study of 218 of these reports shows that 111 of them lead to uncaught exceptions, although without directly affecting user experience, and at least 47 indicate errors that affect the functionality of the websites.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 647544), and, in part, by NSF grant CCF-1715153.
Chapter 12

Practical AJAX Race Detection for JavaScript Web Applications

By Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. To be submitted.

Abstract

Asynchronous client-server communication is a common source of errors in JavaScript web applications. Such errors are difficult to detect using ordinary testing because of the nondeterministic scheduling of AJAX events. Existing automated event race detectors are not much help, because they are generally too imprecise or too inefficient to be practically useful. To address this problem, we present a new approach based on a light-weight combination of dynamic analysis and controlled execution that directly targets identification of harmful AJAX event races.

We experimentally demonstrate using our implementation, AjaxRacer, that this approach is capable of automatically detecting harmful AJAX event races in many websites, and producing informative error messages that support diagnosis and debugging. Among 20 widely used web pages that use AJAX, AjaxRacer discovers harmful AJAX races in 12 of them, with a total of 72 error reports, and with very few false positives.

12.1 Introduction

Millions of JavaScript web applications use AJAX for client-server communication. AJAX is today a commonly used term that describes uses of the XMLHttpRequest (abbreviated XHR) API that all modern browsers support.
This API enables JavaScript programs running in browsers to send HTTP requests to a server based on a user’s input, and receive responses that are then used to update the UI. A prominent example is the autocomplete feature at google.com where suggestions are provided as soon as the user starts entering a search term.

To ensure a smooth user experience, AJAX communication is usually asynchronous, meaning that a user can continue interacting with the page and that more JavaScript code can be executed between the time that the HTTP request is sent and the time when the response is received. Additionally, it is possible to perform multiple AJAX interactions simultaneously. This may cause nondeterminism when the same sequence of user events leads to different behaviors depending on the order in which the AJAX responses and other events are being processed. Often, this leads to errors that are missed by ordinary testing when programmers are insufficiently aware of the many possible interleavings of events. For the end user, the consequences of such errors typically range from minor functionality glitches to misleading inconsistencies in the UI.

The fact that the JavaScript execution model is susceptible to so-called event races is well known [87, 163]. Many techniques have been developed to detect and prevent event race errors and their harmful consequences [11, 12, 78, 91, 131, 138, 176, 176, 197]. However, none of those existing techniques are capable of detecting harmful event race errors that involve AJAX with sufficient precision and performance to be practically useful. In particular, EventRacer [146] reports too many benign races and may miss harmful ones [12, R4] [91] relies on stateless model checking, which does not scale well and requires complex browser instrumentation, and InitRacer [12] only detects event races that appear during the initialization of a web application, not those that involve AJAX later in the execution.

The goal of our work is to provide an automated event race detector that is practical for event races that involve AJAX. Specifically, such a tool should be able to detect AJAX event races that have observable effects, without reporting a large number of spurious or harmless races. Also, it should not require browser instrumentation, which is platform-specific and difficult to maintain as browsers evolve, and it should be fast and produce informative error messages that facilitate debugging.

We present an approach that meets these requirements, inspired by the ideas of adverse execution used by InitRacer [12] and controlled execution used by EventRaceCommander [11]. Our approach is based on the key observation that JavaScript developers typically test their code using networks and servers that are fast and reliable, so in their tests AJAX is effectively synchronous, meaning that the AJAX request-response pairs are essentially atomic, without other events occurring in between. This observation allows us to establish a notion of “expected” event schedules as those where an AJAX response event handler $e_{resp}$ executes immediately after the event handler $e_{req}$.
that sent the request. In contrast, any schedule where another event handler $e$ is scheduled between $e_{req}$ and $e_{resp}$ can be regarded as less likely to be exercised during ordinary testing. An AJAX event race occurs if the effects of $e$ conflict with the effects of $e_{resp}$. The idea of adverse execution is to systematically expose a program to adverse conditions and compare the result with the normal behavior. In our case, schedules where AJAX is processed synchronously define the normal expected behavior, and adverse conditions are situations where the network or server is slow or unreliable allowing other events to interfere.

Our approach consists of two phases. The first phase dynamically monitors an execution of a web application, with the purpose of identifying (1) user event handlers that have conflicting AJAX response event handlers, and (2) information about which event handlers may be reordered. This initial execution may be driven by a human user, an automated testing tool, or a pre-existing test script, similarly to other dynamic race detectors. For each user event handler $u$ that has been observed, an event graph $G_u$ is generated that captures relevant information about the events that have been triggered either directly or indirectly by $u$. For example, clicking on a button may create a timer event that leads to an AJAX request that, in turn, triggers an AJAX response event, which finally updates the UI. The second phase uses these event graphs to plan a series of tests. Each test simulates two event schedules, one where AJAX is synchronous and one that simulates adverse conditions as discussed above, and automatically compares screenshots of the resulting web pages. Observable differences are reported along with detailed information about the event schedules that gave rise to them. To control the scheduling of event handlers when executing the tests, we use an event controller mechanism inspired by EVENTRACECOMMANDER in which nondeterminism is restricted by selectively postponing the execution of event handlers.

We evaluate AjaxRacer using 20 web pages from 12 large and widely used web applications. The results show that the approach is effective in detecting AJAX races in real settings. AjaxRacer generates 152 tests, of which 69 reveal harmful races among 12 of the web pages, and only three reports are false positives. We additionally demonstrate the usefulness of AjaxRacer’s comprehensive web-based reports for understanding the detected AJAX races and diagnosing their root causes.

In summary, this paper makes the following contributions:

- We define a notion of event graphs that captures relevant information about effects and orderings of event handlers, relative to a given initial execution.
- We present a two-phased approach for automatically detecting harmful AJAX event races in JavaScript web applications. The first phase performs a dynamic analysis for computing event graphs; the second phase executes the generated tests under different event schedules and determines if observably different results appear.
13 function fetchJSONFromURL(url, callback) {
14     var xhr = new XMLHttpRequest();
15     xhr.open('GET', url, true);
16     xhr.onreadystatechange = function () {
17         if (xhr.readyState == XMLHttpRequest.DONE && xhr.status == 200) {
18             var obj = JSON.stringify(xhr.responseText);
19             return callback(obj);
20         }
21     };
22     xhr.send(null);
23 }

Figure 12.1: AJAX example that demonstrates how a web application can fetch a JSON object from a server.

• We describe the open-source tool AjaxRacer\(^1\) which implements the approach.

• We present experimental results showing AjaxRacer to be effective at detecting AJAX races in real-world web applications, that it reports few false positives, and that it provides insightful explanations that are helpful to developers.

12.2 Background on AJAX

AJAX (Asynchronous JavaScript and XML) is a technology that enables web applications to exchange data asynchronously with a server without imposing page reloads, which enables rich and responsive client-side web applications. Figure 12.1 illustrates how a web application can retrieve a JSON object asynchronously from a server using the XMLHttpRequest (XHR) API\(^2\). To send an XHR request, a web application first needs to construct an XHR object (line 14) and initialize the object by calling the open method with the relevant HTTP method and URL (line 15). The open method takes as optional arguments a boolean that specifies if the request should be asynchronous (defaults to true) and credentials for authentication purposes. When the XHR object has been initialized, the AJAX request can be sent by calling the send method, optionally with data for the body of the request (line 22).

Each XHR object goes through several phases during the lifecycle of the corresponding request. The current state of an XHR object can be accessed

\(^1\)To be submitted for artifact evaluation.
\(^2\) See https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. The new Fetch API (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API) and WebSockets (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) provide related functionality. In this paper, we focus on XHR, which is currently the most widely used AJAX API, but the alternatives may be interesting for future work.
at any time by reading its `readyState` property. This state indicates (among others) if the request has been sent, if the headers and status code have been received from the server, or if the entire response has been received. Each time the state of an XHR object changes, a so-called `readystatechange` event is triggered. Web applications can react to these events by registering an event handler for this event type, as in line 16. The event handler in lines 16–21 explicitly checks that the response has been fully received before it accesses the body of the AJAX response in line 18. XHR involves several other kinds of events, in addition to `readystatechange` events. These include a `load` event when the resource has been loaded, a `timeout` event if the response takes too long, and an `error` event if, for example, the request is blocked by the browser’s same-origin policy.

To circumvent the same-origin policy of XHR, many websites instead implement AJAX using JSONP. To get data from a server with that approach, the client code dynamically creates a `script` element with the URL of a script, which is executed when it has been retrieved from the server. For this reason we also need to take dynamically loaded scripts into account.

We distinguish between `user` events (mouse click events, keyboard events, etc.) and `system` events (most importantly, AJAX response events and timer events). After the web page has been loaded and initialized, every system event is triggered either directly or indirectly by a user event. Each such system event can thus be associated uniquely with a user event; we say that the system event is derived from that user event.

AJAX is one of the key ingredients of modern web applications. However, it also introduces complexities in the execution of web applications. In particular, there are no guarantees regarding the exact timing and order of arrival of AJAX requests at the server, nor of the corresponding AJAX response events at the client. The user controls the ordering of user events, but the execution of system events is to some extent nondeterministic. Borrowing terminology from concurrency in multi-threaded settings, a `schedule` fixes the nondeterministic choices relative to a given sequence of user events. As a consequence of this nondeterminism, event race errors may occur in production web applications when the order of events in the execution differs from the ones observed during testing.

### 12.3 Motivating Example

Figure 12.2 shows a snippet of HTML and JavaScript code from `www.chevron-withtechron.com/findastation.aspx`. This web page allows the user to search for gas stations in a given area, and to filter these gas stations based on various criteria. For example, the user can search for gas stations that have a car wash by clicking on the “Car Wash Locations” button defined in line 25, which causes the JavaScript function `addRemoveFilter` in line 33 to execute. This function
updates the set of filters that have been selected by the user (line 34), clears the contents of the HTML element that presents the list of gas stations to the user (line 35), and finally invokes the function searchLocationsNearByJSON (line 36) to retrieve the list of gas stations from the server according to the search query provided by the user (lines 39–41). By the time the server response arrives, an AJAX response event fires, causing the event handler parseStationData (line 43) to execute. This function constructs a snippet of HTML for each gas station in the server response (lines 46–49), and then updates the UI using these snippets (line 50).

The example web page exhibits an AJAX event race when the user selects more than one criterion. Consider what happens when the user clicks on
12.3. MOTIVATING EXAMPLE

(a) Correct schedules.

(b) Erroneous schedules.

Figure 12.3: Possible interleavings in the motivating example.
“Car Wash Locations” button. If the AJAX responses arrive out of order, the markers on the map are inconsistent with the selected filters. The technique we describe in the following sections finds both these errors.

For an event race error detection technique to be practical, it is not sufficient for it to detect errors and produce useful error messages; it is also important that it does not report too many false positives. Predictive event race error detectors like EventRacer generally report many races that are infeasible or harmless [12, 131, 192]. This is particularly problematic when web application programmers carefully use ad-hoc synchronization to avoid race errors. For this reason, our technique is designed so that it only reports event race errors that can be witnessed by concrete schedules that exhibit visible differences in the browser.

12.4 The AjaxRacer Technique

Our technique comprises two phases. Phase 1 generates an event graph that can be used to identify pairs of user events that are likely to be involved in an observable AJAX event race. Phase 2 examines, for each such pair of events, whether or not an observable AJAX event race actually exists.

Phase 1: Generating Event Graphs

Phase 1 is seeded by a sequence of user events, similar to other dynamic race detectors [91, 131, 146]. This sequence can be obtained by a single manual execution of the web application, or using an automated crawler [21, 124]. AjaxRacer loads the (instrumented) web page in the browser and waits until it has been fully initialized (meaning that the HTML has been parsed, its scripts have been executed, and there are no pending system events; see Section 12.5 for details). It then triggers the user events in the sequence one by one, in each step awaiting a quiescent state where no system events are pending, until the next user event is triggered. With such a controlled execution, it is easy to determine from which user event each system event is derived, and we reduce the risk of interference.  

As an example, if we did not wait between the user events but triggered them without any delay, an unfinished XHR interaction initiated by one user event might be aborted by an XHR interaction initiated by another user event.

For each user event $u$, AjaxRacer generates a trace $\tau_u$ by monitoring the execution of $u$ and its derived system events. A trace is a sequence of operations of the following kinds:

- $\text{fork} [v, w, k]$ models the fact that an event $v$ creates a new system event $w$ of kind $k$ to be dispatched later. For example, $\text{fork} [v, w, \text{XHR load}]$ means that $v$ performs an XHR request and $w$ is the associated XHR
load event, and \texttt{fork}[v, w, \text{timeout}] means that \( v \) sets a timer using \texttt{setTimeout} and \( w \) is the associated timeout event.

- \texttt{join}[v, w] specifies that event \( w \) cannot occur before event \( v \). Every single XHR request creates several XHR \texttt{readystatechange} events and an XHR \texttt{load} event, and we use \texttt{join} to model the ordering constraints on those events.

- \texttt{mutate-dom}[v, x, y, w, h] models that event \( v \) has modified the HTML DOM, where the parameters \( x, y, w, h \) specify the position and size of the affected bounding box on the screen.\footnote{\textcolor{red}{Other effects, for example involving web storage or cookies \cite{131}, can be modeled as variants of this operation.}}

Compared to the notion of event actions in \texttt{EventRacer} \cite{146}, the key differences are that (1) we generate one trace per user event rather than one global trace, and (2) we use a different model of memory accesses where we consider the effects of HTML DOM write operations on the pixels on the screen instead of low-level read/write operations.

From each trace \( \tau_u \), \texttt{AjaxRacer} now generates an event graph \( G_u \). An event graph is a directed graph \( G_u = (N, E, \ell) \) where each node \( v \in N \) is an event, which is either \( u \) itself or an event derived from \( u \), and where the edges \( E \) represent constraints on the event order.\footnote{\textcolor{red}{Notice that the event graph captures a happens-before relation in the style of Petrov et al. \cite{138}: \( v \preceq w \) if there is a path from \( v \) to \( w \).}} Each operation \texttt{fork}[v, w, k] in \( \tau_u \) gives rise to a labeled edge \( v \xrightarrow{k} w \in E \), and each operation \texttt{join}[v, w] in \( \tau_u \) gives rise to an unlabeled edge \( v \rightarrow w \in E \). The component \( \ell \) labels each node with a set of bounding boxes according to the HTML DOM modifications: for each operation \texttt{mutate-dom}[v, x, y, w, h], the bounding box \((x, y, w, h)\) is included in \( \ell(v) \). The event graph thus describes the HTML DOM modifications made by the user event \( u \) and all its derived system events. We will refer to the user event \( u \) as the (unique) root of \( G_u \).

\textbf{Example}  Figure 12.4 shows a simplified version of the event graph for a click event on the “Car Wash Locations” button from Section 12.3. The root is the click event itself. Since the \texttt{addRemoveFilter} function clears the contents of the HTML element with ID \#stationResult (line 35), the node label contains its bounding box \((x = 280, y = 1132, w = 1024, h = 334)\). The bottom-most node represents the XHR \texttt{load} event, whose event handler updates the same HTML element, as indicated by the node label. \(\Box\)

Our approach targets a scenario in which web application programmers have tested their code using fast servers and networks, and with plenty of time between each user event. In such situations, if a user event \( u_1 \) is followed by a user event \( u_2 \), it is to be expected that all events derived from \( u_1 \) appear before \( u_2 \) and all of its derived events. It is less likely that the programmers have
encountered executions in which some of the events derived from \( u_2 \) appear before some of the events derived from \( u_1 \). Such executions are exactly what AjaxRacer aims to explore.

For that purpose, we now define a suitable notion of event conflicts. Let \( u_1 \) and \( u_2 \) be user events with event graphs \( G_{u_1} = (N_1, E_1, \ell_1) \) and \( G_{u_2} = (N_2, E_2, \ell_2) \), respectively. The two user events \( u_1 \) and \( u_2 \) are potentially AJAX conflicting if there exists an event \( v_1 \in N_1 \) and an event \( v_2 \in N_2 \) such that

1. \( u_1 \) and \( v_1 \) are separated by an AJAX event, meaning that \( G_{u_1} \) has a path from \( u_1 \) to \( v_1 \) containing an edge \( k \rightarrow \) where the label \( k \) is XHR load or script load, and

2. a bounding box in \( \ell(v_1) \) overlaps with one in \( \ell(v_2) \).

The intuition of the first condition is that \( u_1 \) triggers an XHR request or loads an external script, which subsequently leads to an event \( v_1 \), and the second condition checks whether \( v_1 \) may interfere with events derived from \( u_2 \).

We say potentially conflicting, because the criterion does not guarantee that \( u_1 \) and \( u_2 \) are simultaneously enabled. For example, \( u_1 \) and \( u_2 \) may be click events on two different buttons, where the button for \( u_2 \) is created by \( u_1 \) or one of its derived events. Also, the event handlers may behave differently depending on the schedule, due to, e.g., ad-hoc synchronization. Phase 2, described in Section 12.4, examines whether potential conflicts are realizable.

**Example** As mentioned in Section 12.3, a “Car Wash Locations” button click event not only conflicts with a “Diesel Locations” button click event, but also with itself. The event graph for a click on “Car Wash Locations”, as shown in Figure 12.4, indeed satisfies the conditions for this event to potentially AJAX conflict with itself: there is a path from \( v_1 \) to \( v_5 \) containing an XHR load event,
12.4. THE AJAXRACER TECHNIQUE

Algorithm 1: Planning AJAX race tests.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{foreach} \((u_i, u_j)\) \textbf{where} \(i, j \in 1, \ldots, n\) \textbf{do}
\State \quad \textbf{if} \(u_i\) and \(u_j\) are potentially AJAX conflicting \textbf{then}
\State \quad \quad test \((u_i, u_j)\)
\State \textbf{end}
\State \textbf{end}
\end{algorithmic}
\end{algorithm}

and the bounding box of \(v_1\) overlaps with that of \(v_5\) (in fact, they are identical in this case). This tells us that it may be worthwhile in Phase 2 to test a user event sequence containing two clicks on “Car Wash Locations”, with a schedule where the events derived from the second click appear before those derived from the first click.

Phase 2: Testing Potential Conflicts

From Phase 1, we have a sequence of user events \(u_1, \ldots, u_n\), each described by an event graph, and we know for each pair of user events whether or not they are potentially AJAX conflicting. In principle, AJAXRacer could simply output the resulting pairs of events as warnings to the user, which would be reminiscent of how predictive race detectors work [131, 140]. However, to avoid many false positives and produce more informative error messages, Phase 2 attempts to provoke actual observable race errors, similar to other techniques [12, 78, 91, 192], but using a mechanism specifically designed for AJAX event races.

We perform a set of tests according to Algorithm 1. For each pair of user events \((u_i, u_j)\), one test is created if the two events are potentially AJAX conflicting. Note that we consider all ordered pairs of user events from \(u_1, \ldots, u_n\), including those where \(i = j\), which is relevant for the previously mentioned example involving multiple clicks on “Car Wash Locations”. In practice, relatively few of the event pairs are potentially AJAX conflicting, so the total number of tests performed is usually low (see Section 12.6).

Algorithm 2 shows how each test is performed. Lines 1-6 simulate a user event sequence where \(u_i\) and \(u_j\) are performed after the web page has been loaded, using a schedule where all system events derived from \(u_i\) appear before those derived from \(u_j\), as if AJAX communication were synchronous. Next, lines 7-13 simulate the same two user events, but this time using an “adverse” schedule where the AJAX events derived from \(u_i\) are postponed until after all the events derived from \(u_j\) have appeared. After each run, we take a screenshot of the browser contents, and an error is reported if the two screenshots are not identical (line 14).

When attempting to trigger an event (lines 2, 4, 8, and 9), the test aborts without emitting any error message if the event is not enabled because the
Algorithm 2: Executing an AJAX race test.

// execute $u_i$ and $u_j$ in ‘synchronous’ mode
1 reload the web page
2 trigger $u_i$
3 wait until the events in $G_{u_i}$ have been executed
4 trigger $u_j$
5 wait until the events in $G_{u_j}$ have been executed
6 $s_1 =$ screenshot

// execute $u_i$ and $u_j$ in ‘adverse’ mode
7 reload the web page
8 trigger $u_i$, and postpone all its derived AJAX events
9 trigger $u_j$
10 wait until the events in $G_{u_j}$ have been executed
11 allow the events derived from $u_i$ to execute
12 wait until the events in $G_{u_i}$ have been executed
13 $s_2 =$ screenshot

// decide outcome
14 if $s_1 \neq s_2$ then emit error message

associated DOM element does not exist or is not visible. This can happen because other events that appear in the Phase 1 execution but not in the Phase 2 executions may have changed the system state, however this is rarely a problem in practice (see Section 12.6). One pattern is quite common, though: In many web pages, an HTML element (e.g., a menu item) only becomes visible after clicking or hovering over another HTML element. For this reason, we allow the user of AjaxRacer to group such low-level events in the initial event sequence into “macro events” \[^{[47]}\], so that AjaxRacer can trigger them together, which increases the chance of the events being enabled.

Each time the web page is reloaded (lines [1] and [7]), we wait until it is fully initialized, as in Phase 1. Waiting for derived events to be executed (lines [3], [5], [10] and [12]) is also implemented by waiting until the web page becomes idle. In this way, we do not risk waiting for derived events that were observed in Phase 1 but do not occur in this execution, which is reminiscent of the concept of approximate replay in R\[^{[9]}\] \[^{[91]}\]. Postponing events (line [8]) and allowing them to execute (line [11]) is implemented using an approach inspired by EventRaceCommander \[^{[1]}\].

**Example** Continuing the example, an initial user event sequence that contains a single click event $u$ on “Car Wash Locations” and a single click event $v$ on “Diesel Locations” suffices to find both errors described earlier. One test being performed is for the event pair $(u, v)$. This test first executes $u$ followed by $v$ in “synchronous” mode, and then in “adverse” mode. The resulting screenshots are different, so an error is reported. Another test is performed for
the event pair \((u, u)\), and again an error is reported because the screenshots differ between the synchronous and the adverse schedules.

An important difference between \textsc{AjaxRacer} and other event race detectors like \textsc{EventRacer} \cite{li2013}, \textsc{R}\textsuperscript{4} \cite{li2014}, and \textsc{WAVE} \cite{zhao2016} is that \textsc{AjaxRacer} not only explores different schedules for the system events but also user event sequences that are different from the seed execution. This allows \textsc{AjaxRacer} to detect errors that are missed by the other techniques.

Consider for example a web page with two buttons, \(A\) and \(B\). Clicking the \(A\) button triggers an XHR request where the XHR load event adds contents to an HTML element, and clicking the \(B\) button clears the contents of the HTML element. In this case, a race error appears if the \(B\) button is clicked after the \(A\) button is clicked but before the XHR load event occurs. If the initial sequence of user events consists of a click on \(A\) followed by a click on \(B\), then \textsc{EventRacer}, \textsc{R}\textsuperscript{4}, \textsc{WAVE}, and \textsc{AjaxRacer} will all find the error. However, if the initial event sequence consists of a click on \(B\) followed by a click on \(A\), then \textsc{EventRacer}, \textsc{R}\textsuperscript{4}, and \textsc{WAVE} do not find the error (because they treat user events as being happens-before ordered), but \textsc{AjaxRacer} does find it.

As another example, consider a web page with a single button \(C\) where clicking on \(C\) triggers an XHR request, and the XHR load event handler writes the server response data into the HTML DOM. A user event sequence that contains a single \(C\) click event may cover all the JavaScript code, but it is not enough for \textsc{EventRacer}, \textsc{R}\textsuperscript{4}, or \textsc{WAVE} to expose the race error that occurs if \(C\) is clicked twice and the responses arrive out of order. In contrast, \textsc{AjaxRacer} can find the error, even with a single occurrence of the \(C\) click event in the initial event sequence.

### 12.5 Implementation

\textsc{AjaxRacer} is implemented as a command-line JavaScript application that takes as input a URL and a user event sequence to analyze, and is available at \url{https://github.com/cs-au-dk/ajaxracer}.

The implementation uses a proxy server, \textsc{mitmproxy} \footnote{https://mitmproxy.org/} to dynamically instrument HTML and JavaScript source files as they are fetched by the browser. The instrumentation wraps all property assignments and DOM API functions that involve event handlers and modifications of the HTML DOM, so that we can intercept the relevant operations at runtime. Dynamically generated code is instrumented by wrapping the built-in functions \texttt{eval} and \texttt{Function}.

When the proxy is running, \textsc{AjaxRacer} uses the end-to-end testing framework \textsc{Protractor} \footnote{http://www.protractortest.org/} to load the given URL in Google Chrome via the
proxy server, trigger a given sequence of user events (or macro events, as discussed in Section 12.4), store results from the execution, and optionally take a screenshot of the resulting state. These steps are carried out once for Phase 1 and twice for each test that has been planned in Phase 2 (recall Algorithm 2). The screenshots that are captured for each test are compared using the LooksSame library. AjaxRacer ignores a difference at a pixel \((x, y)\), if the adverse mode and synchronous mode executions already differed at \((x, y)\) when the web page finished loading. This mechanism helps to prevent false positives in situations where a server returns slightly different HTML each time. In addition to classifying the two screenshots as identical or not, AjaxRacer also uses the LooksSame library to generate an image where the differences (if any) are highlighted, which is useful for further debugging.

The instrumentation of the web application code allows AjaxRacer to generate a trace for each user event. It also makes it possible to determine when the web application has finished loading (by waiting for the set of pending events to become empty, as explained in Section 12.4), and when the web application becomes idle after a user event has been triggered and processed.

Some web applications never finish loading, in the sense that they continuously react to timer events (e.g., to implement a slideshow that automatically changes every few seconds). AjaxRacer deals with such situations by deleting timer events with a delay above a given threshold, and by stopping a chain of timer events if the length of the chain reaches some threshold. We have not found cases where this breaks the main functionality of the web application. Because we wait until the web application is entirely idle, the user event handlers triggered by AjaxRacer cannot interleave with code that has been spawned during the loading of the web application. This helps prevent false positives from the screenshot comparison. For example, in presence of a slideshow, the screenshots taken by AjaxRacer would otherwise depend on the exact timing, and be unsuitable for use as an oracle. GIF animations are another source of nondeterministic results. To combat this issue, AjaxRacer uses its proxy to intercept the loading of GIF images and remove animations.

12.6 Evaluation

To assess the effectiveness of our approach, we conducted three experiments to answer the following research questions:

RQ1 (Effectiveness) Does AjaxRacer report AJAX event race errors in real-world web applications? How often do AjaxRacer’s warnings identify real errors?

RQ2 (Race characteristics) Do the detected AJAX races exhibit interesting patterns?

\(^7\)https://www.npmjs.com/package/looks-same
### Table 12.1: Summary of results.

<table>
<thead>
<tr>
<th>Company Name</th>
<th>Web page</th>
<th>Tests</th>
<th>Avg. runtime (s)</th>
<th>False positives</th>
<th>Phase 1</th>
<th>Phase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Amerisource Bergen</td>
<td>Job Openings</td>
<td>25 22</td>
<td>32 64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Apple</td>
<td>Accessibility</td>
<td>4 0</td>
<td>13 34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Buy MacBook</td>
<td>4 2</td>
<td>30 66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>Customize</td>
<td>4 2</td>
<td>17 41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>Search Jobs</td>
<td>4 0</td>
<td>19 48</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>Search Support</td>
<td>4 4</td>
<td>17 39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Bank of America</td>
<td>Search Locations</td>
<td>9 2</td>
<td>31 54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. Berkshire Hathaway</td>
<td>Search Listings</td>
<td>12 2</td>
<td>76 136</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Chevron</td>
<td>Find a Station</td>
<td>30 21</td>
<td>33 54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10. Citigroup</td>
<td>News</td>
<td>4 2</td>
<td>13 34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11. Exxon Mobil</td>
<td>Job Locations</td>
<td>4 2</td>
<td>14 36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12. Fannie Mae</td>
<td>Search</td>
<td>4 0</td>
<td>13 32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13. Grainger</td>
<td>Home</td>
<td>4 0</td>
<td>35 78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14. McKesson</td>
<td>Home</td>
<td>4 0</td>
<td>38 86</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.</td>
<td>Blog Archive</td>
<td>4 2</td>
<td>28 64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16.</td>
<td>Event Calendar</td>
<td>4 2</td>
<td>22 51</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17.</td>
<td>Press Releases</td>
<td>16 9</td>
<td>25 55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18. Verizon</td>
<td>Search Locations</td>
<td>4 0</td>
<td>29 68</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19. Wells Fargo</td>
<td>Home</td>
<td>4 0</td>
<td>16 41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20.</td>
<td>Search</td>
<td>4 0</td>
<td>17 41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>152 72 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>7.6 3.6 0.15</td>
<td>25.9 56.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RQ3 (Usefulness)** Do the generated reports provide informative explanations of the causes and effects of each AJAX event race?

**RQ4 (Performance)** Is AjaxRacer’s performance acceptable?

**RQ5 (Comparison with state-of-the-art)** How effective is AjaxRacer compared to other tools, most importantly EventRacer?

### Experimental Methodology

To answer the research questions, we consider randomly selected web pages from a subset of the companies from the Fortune 500 list[^9]. We manually identified web pages that use AJAX by browsing the company web sites using the Chrome browser, while enabling the “Log XMLHttpRequests” feature and the “Network” panel from the Chrome DevTools[^10] which makes it easy to

[^10]: [https://developer.chrome.com/devtools](https://developer.chrome.com/devtools)
CHAPTER 12. PRACTICAL AJAX RACE DETECTION

recognize when an XHR message is being exchanged or an external script is being loaded dynamically. We ignored requests that send analytics data. With this approach, we obtained 20 web pages from 12 different companies, as shown in the “Company” columns of Table 12.1.

For each of the web pages, we manually create a short user event sequence that exercises some of the dynamic behavior on the web page. Each user event sequence consists of two to nine user events and has been made without any knowledge of the JavaScript code on the web page or the client-server communication. We then carry out the following experiments.

**Experiment 1.** We run AjaxRacer on each subject application using the given manual event sequence. To answer RQ1, we inspect the AJAX event race errors that it reports, and manually check whether each of them can be reproduced. To answer RQ2, we present patterns that we observe in the reported AJAX races. We answer RQ3 by reporting on our experiences during this study with the asynchronous code and the generated reports.

**Experiment 2.** To assess the performance of AjaxRacer and answer RQ4, we measure the time needed by AjaxRacer’s two phases. For Phase 1, we separately report the time spent on loading the web page and on generating traces for the user events. For Phase 2, we separately measure the time spent on test planning (Algorithm 1) and test execution (Algorithm 2). We repeat the experiments three times and report average and worst-case running times.

**Experiment 3.** We run EventRacer on the subject applications using the manually created user event sequences, and answer RQ5 by investigating the results. EventRacer also detects races during the loading of a web page. To estimate how many races arise from the execution of the user event sequence, we analyze the results of EventRacer when no user events are triggered. We report average numbers across three three runs. Regrettably, we could not compare to RClassify [192], as it was not available to us.

The experiments were conducted on an Ubuntu 15.10 desktop machine with an Intel Core i7-3770 CPU and 16 GB RAM. All experimental data is available at [http://ajaxracer.casadev.cs.au.dk/](http://ajaxracer.casadev.cs.au.dk/)

**Results and Discussion**

In this section, we present the results of our experiments, summarized in Table 12.1 and elaborate on more interesting findings, while addressing RQ1–RQ5.
Effectiveness (RQ1)

After Phase 1, AjaxRacer created a total of 152 tests for the web pages in Table 11.1, which follows from column “Tests”. Of the 152 tests, four proved to be infeasible (i.e., one of the user events in these tests was not enabled by the time it was scheduled to be executed). The number of test failures is reported in column “Failures”. Each failure reveals a situation where adverse mode execution of a pair of user events leads to a state that is observably different from the corresponding synchronous mode execution. In total, 72 tests failed. The page from Amerisource Bergen held the highest number of failing tests with 22 failures (row 1). After manually inspecting the results, we found that only three of the 72 test failures were false positives (column “False positives”). This is a significantly smaller false positive rate than that of existing predictive race detectors such as EventRacer [146]. In particular, each of the 80 succeeding test cases indicates a situation where EventRacer would report a race warning, but where the race is not observable, because ad-hoc synchronization prevents the harmful effects, or the two events from the race commute (i.e., the events have the same effects, irrespective of their arrival order).

Overall, our results show that AjaxRacer is capable of detecting observable AJAX races in real-world web applications with only few spurious warnings.

The fact that the web applications of some of the largest companies in the United States suffer from observable AJAX races demonstrates that this is a widespread problem. Left undetected, they may render the application in an inconsistent state (as we give examples of later in this section). As such, they can frustrate end users and negatively impact their experience. AjaxRacer unveils such situations semi-automatically, with relatively few tests per web page. In summary, AjaxRacer generated an average of eight test cases per web page, of which half exposed an observable AJAX race.

False positives. As mentioned above, we observed only three spurious warnings among the 72 failing tests. A single false positive arose for Chevron (row 9) because live traffic, which was changing during the execution of the tests (row 9), was being shown on a map. AjaxRacer also reported two false positives for a web page from Berkshire Hathaway (row 8), where the user can search for real estate listings. One test was failing because the screenshot from synchronous mode showed “35,537 Results,” whereas the one from adverse mode showed “35,536 Results.” Presumably, a listing was removed from the website during the execution of the test. The other false positive from Berkshire Hathaway was similar. We confirmed this behavior by rerunning the tests, which lead to successful executions. If AjaxRacer did not ignore pixels that were already different by the time web page had been loaded (Section 12.5), then 9 additional false positives would have been reported (for rows 9, 13, and 19). Generally, there may be other sources of nondeterminism in the
application’s UI, such as multimedia resources and third-party entities (e.g.,
videos and advertisements). Using AjaxRacer, we were able to detect these
inconsistencies in a glance, without any need for manual analysis of the code,
with the help of the generated reports.

Observable AJAX races are neither the only type of races that exist in web
applications, nor did we attempt to reveal all such races within each application.
However, the prevalence of observably harmful AJAX races in our subject
applications indicates the need for systematic analysis of race-prone code. We
used AjaxRacer’s reports to gain further insight into the behavior of such
code, by manually examining successful tests. As expected, a group of tests
succeeded because they lead to the same DOM state, regardless of the ordering
of the AJAX events. More interestingly, we encountered another group of
successful tests that did not show symptoms of AJAX races, contrary to our
initial assumptions (e.g., rows 2, 5, and 8). After a thorough examination of the
source code of these applications, we found that the developers had deployed
means of remedying the AJAX races. Their strategies not only strengthened
our motivation regarding the problematic nature of such races in practice,
but also provided insights on common practices for preventing AJAX races
(discussed more thoroughly in Section 12.6). We also observed AJAX race
ersors that lead to a series of uncaught exceptions (row 18). AjaxRacer
conservatively classified these errors as benign, since they had no observable
effects on the screen. Overall, we encountered no cases where the user event
sequence lead to an observable AJAX race that was missed by AjaxRacer.

Only four tests among all 152 were deemed infeasible during Phase 2 of
AjaxRacer. These tests all belonged to the Press Releases web page of
McKesson (row 17). A “next” button on the page was removed from the UI
when the number of results did not exceed one page. Therefore, a click event
could not be issued on this button if an event that filters the press releases
had already been triggered (e.g., an event that clicks on the “November 2017”
button).

Race Characteristics (RQ2)
We observed that most of the detected AJAX races fit in one of the following
categories.

Dataset Queries. Many applications present some data from a database to
the user through a list or a table (e.g., web shops). With the overwhelming
amount of information available to users, many modern web applications
provide means of filtering the displayed dataset, based on users’ needs. This
is the case for the web pages in rows 1, 6, 8, 10, and 17, 18 of Table 12.1
However, when users send multiple queries and the corresponding responses
arrive asynchronously, race conditions arise. Such races may cause the displayed
data to be inconsistent with the user’s query.
12.6. EVALUATION

(a) Adverse mode. (b) Autogenerated diff.

Figure 12.5: Inconsistent state when customizing a MacBook.

Interactive Maps. Many web applications display interactive maps that are used for various purposes, e.g., specifying the locations of retail stores (row 9) or available job postings (row 11). Triggered by user queries, the information overlaid on the maps is updated using AJAX. Conflicts between the user events, discovered by AjaxRacer, lead to incorrect data on the maps of these applications.

Autocompletion. Autocompletion is a feature for generating textual suggestions as soon as a user starts typing in a text field. Such suggestions are often updated asynchronously, which can cause AJAX races that lead to incorrect recommendations. Rows 12–14 and 19–20 correspond to autocompletion features.

Usefulness (RQ3)

Manual analysis of the dynamic, asynchronous, and event-driven behavior of JavaScript applications is a challenging endeavor. To assist developers with understanding AJAX races and locating their root causes, AjaxRacer creates a comprehensive web-based report as the final step. From the report, developers can view the event graphs corresponding to the user events that have been triggered during Phase 1, and see which user events are potentially AJAX conflicting. The report also allows developers to navigate the test results, examine the screenshots that have been taken at different steps during the adverse and synchronous mode executions, and compare the final screenshots.

Example (Apple) On a web page from Apple (row 4), users can customize a MacBook before purchasing it. When the user selects one of the available processors, the UI is updated asynchronously, in response to an AJAX request that fetches the model’s information from a web service. AjaxRacer automatically found an AJAX race error on this web page. One of the generated tests is for a user event sequence that clicks on the button for the 1.3GHz processor, and then on the 1.4GHz processor button. Figure 12.5a shows the undesirable outcome that results from executing this scenario in adverse mode. As it follows from the screenshot, the UI showed that the price would increase by $150 if the user selected the 1.4GHz processor, although it was already chosen.
(indicated by the blue border). In the screenshot from synchronous mode, a price difference was only shown for the processors that were not selected. The total price of the model was also incorrect in adverse mode. It reflected the price of the previously selected configuration ($1,549.00), rather than that of the 1.4GHz model. Figure 12.5b shows the diff image that was automatically generated by AjaxRacer. From this image, it was effortless to recognize the consequences of the race, which could otherwise be labor intensive.

Further analysis of all the generated reports revealed that AJAX event races were common in most subject applications. We examined each report to locate and understand the underlying mechanisms that enabled the races.

Example (Chevron) Recall the motivating example from Chevron, discussed in Section 12.3. Figure 12.6 shows the two screenshots that were captured from a test that clicks twice on the “Car Wash Locations” button. The map that results from adverse mode (Figure 12.6b) did not show any gas stations without a car wash, although it should (the “Car Wash Locations” button is a toggle switch, initially turned off). When the same user event sequence was executed in synchronous mode, the web application correctly showed all the gas stations in the given area, regardless of whether they had a car wash.

Overall, we found that AjaxRacer’s reports provided informative explanations of the causes and effects of each AJAX race.

Performance (RQ4)

The average execution time of AjaxRacer is shown in the rightmost columns of Table 12.1. In Phase 1, on average, AjaxRacer spent 18 seconds waiting for the web application to load, and eight seconds executing the input user event sequence, while monitoring the execution to build a trace. In the worst case, Phase 1 took 78 seconds (row 8). The test planning (Algorithm 1) took
0.2 seconds in the worst case, including the time required for constructing the event graphs from the traces. In Phase 2, AjaxRacer executed Algorithm 2 for each planned test. On average, this took 56 seconds per test— with 36 seconds spent waiting for the web application to load, and two seconds spent on generating the report. The average running time of Phase 2 sequentially was approximately seven minutes, with the worst case of 27 minutes. However, all tests could easily be executed in parallel. Column “Phase 2” depicts the average duration of test completion, obtained by parallel execution of the tests. The required time for executing a test in the worst case was just over four minutes (row 8). These results demonstrate that the overall performance of AjaxRacer is acceptable for practical use.

Comparison with state-of-the-art (RQ5)

When running EventRacer on the subject applications, we found that it reports an overwhelming number of races. As an example, we applied EventRacer to the web application of Berkshire Hathaway (row 8) with a user event sequence that searches for real estate listings, by clicking on the buttons “4+ Beds” and “12+ Beds” (a subsequence of the one given to AjaxRacer). On average, across three runs, EventRacer reported 103,166 races on 37,697 memory locations. 741 of the races were uncovered. The reports contain no information about the effects of the races. When no user events were triggered, EventRacer reported 45,161 races on 28,956 memory locations. This time, 368 of the races were uncovered. Thus, somewhat surprisingly, the two user events approximately caused the number of reported races to double. This shows that EventRacer would still report an overwhelming number of races, even if it had a mechanism for ignoring races that manifest during the loading of web pages. Inevitably, the majority of these races are harmless. Even after manual investigation of the web page, we were unable to detect any observable races. We did find examples of ad-hoc synchronization in the web page, as described below.

Common Development Practices

We encountered several practices in the subject applications, which prevented AJAX race errors from manifesting. A simple solution is to avoid the use of AJAX altogether, by reloading the entire web page upon a user event. Although offering a less smooth user experience, this approach was still widespread in practice. Among the applications that utilized AJAX, it was common practice to circumvent AJAX races by disabling UI elements while waiting for a pending response. For example, many applications render a dialog showing

\[\text{Intuitively, a race is uncovered if it is guaranteed that no ad-hoc synchronization prevents the two events of the race from being reordered (assuming the happens-before relation is complete).}\]
CHAPTER 12. PRACTICAL AJAX RACE DETECTION

a spinner when an AJAX request is sent, until the corresponding response arrives, in a manner that prevents the user from interacting with the web page. While generally offering a better user experience, this approach reduces the responsiveness of the application. Another group of applications used ad-hoc synchronization in a way that did not prevent the user from interacting with the page. For example, on McKesson, an autocompletion feature was implemented in a way that ignored all AJAX response events except the one corresponding to the last request, as documented in the code:

```javascript
success: function (data) {
  // make sure it's the latest request
  if (__global_counter[container.index] ===
      requestcounter[container.index]) {
    ... o.render(container, data, query); ...
```

The following code from Berkshire Hathaway illustrates one of the more sophisticated remedies we found, in terms of the logic and the quality of the user experience.\[1\]

```javascript
var jqXHRs = {}; $(checkbox).change(submit); function submit() {
  if (jqXHRs.search) {
    jqXHRs.search.abort();
  }
 (jqXHRs.search) {
    jqXHRs.search = $.ajax(...);
    xhr.onreadystatechange = jQuery.noop;
    xhr.abort();
    $$
```

When the user clicks on a button labeled “2+ Beds”, the function `submit` executes (lines 60–65). This function contacts a web service and updates the search results (line 64). If an AJAX request is already active, the function cancels it by calling the function in lines 70–74, from jQuery 1.7.2. This is done by replacing the `readystatechange` event handler with the empty function in line 66 and calling the native method `abort` of the XHR object.

These countermeasures are helpful in their scope and prevent many AJAX race errors in practice. The mere existence of such treatments indicate that AJAX races are real problems, and that professional developers make an effort to prevent them.

\[1\]http://www.mckesson.com/js/min/adobe.target.targetcomplete.min.js
\[2\]The code has been simplified for presentation. It originates from the scripts f345a312-25b7-4242-8165-d6d6f8834fa and 91ff98c9-a847-4d7b-8bf8-ef59c697c8ba from http://www.bhhsneprime.com/jscss/23.0.1474/js/
12.7. RELATED WORK

Threats to Validity

We addressed the external threats of representativeness of our subjects and
generality of the investigated scenarios by testing executable sequences of
events within widely-used pages of large companies. An internal threat arises
from selection of pages, particularly triggering AJAX races as targeted in
the scope of this work, a subset of all potential races. To mitigate this bias,
we devised scenarios for analysis prior to experiment, similar to exploratory
testing. Inspection of races and their severity was performed manually, and was
thus labor-intensive, and prone to examiners’ bias and errors. We alleviated
this bias by having two of the authors carefully examining the code and the
reports independently.

12.7 Related Work

It has long been known that JavaScript applications may experience nonde-
terministic failures depending on the order in which event handlers execute.
Steen [163] observed situations where web applications that rely on setTimeout
to modify a page’s DOM representation fail in mysterious ways when browsers
parse web pages too quickly or too slowly. Ide et al. [87] point out that these
problems can be viewed as a type of race condition, similar to data races in
programming languages with concurrency (see, e.g., [32, 54, 57]). One scenario
discussed by Ide et al. involves erroneous UI updates that occur when AJAX
requests are processed out of order, similar to scenarios we consider. The throttling
feature in Google’s Chrome Developer Tools [95] can be viewed as a
poor man’s race detector: by simulating various network conditions, situations
can be identified where event race errors cause nondeterministic failures.

Zheng et al. [197] present an approach based on static analysis for automat-
ically detecting bugs in web applications where an asynchronous event handler
writes to a global variable $v$, and a user event handler reads $v$. In such cases,
serious errors (e.g., deleting the wrong file on a server) may occur if other
event handlers are interleaved that also write to $v$. Some of the asynchronous
scenarios studied in our work have similar characteristics.

Petrov et al. [138] define a happens-before relation for commonly used
HTML and JavaScript features and a model of logical memory locations on
which web applications operate. These concepts form the basis of Web-
Racer, a dynamic race detector. Raychev et al. [146] propose a notion of
race coverage to eliminate false positives that are due to synchronization
deliberately introduced by programmers (ad-hoc synchronization). Intuitively,
a race $a$ covers a race $b$ iff treating $a$ as synchronization eliminates $b$ as a race.
Nevertheless, predictive race detectors such as WebRacer and EventRacer
have been found to report an overwhelming number of races, the majority of
which are harmless or benign.
Several projects focus on classifying event races as harmful or harmless. Mutlu et al. [130, 131] apply a dataflow analysis to a trace in order to detect situations where executing racing event handlers under different schedules results in different values being written to persistent storage (cookies and local and session storage). WAVE [78] and R4 [91] explore executions that can be obtained by reordering events in a sequence of events observed in some initial execution. These tools classify a race as harmful if reordering a pair of conflicting events results in a different DOM, heap state, or uncaught exception. RCAlSSify [192] classifies a race reported by EVENTRACER as harmful or harmless by generating two executions in which the racing events are executed in both orders and determining if the resulting program states differ in important fields of the DOM, heap, or environment variables. Our work differs from these existing approaches by focusing specifically on AJAX races, by providing detailed explanations for reported issues, and by not relying on the modification of a JavaScript engine.

INITRACER [12] detects race errors that commonly arise during page initialization (form-input-overwritten, late event-handler registration, and access-before-definition errors) using adverse and approximate execution. AJAXRACER follows a similar instrumentation-based implementation technique as INITRACER and provides similar, detailed explanations. However, unlike INITRACER, we focus on detecting AJAX-related races that occur after page initialization.

Several projects focus on repairing event race errors. ARROW [176] performs a static analysis to determine happens-before relationships between page elements and record these in a causal graph. Races are detected by identifying inconsistencies between the causal graph and def-use relationships inferred from source code order, and prevented by adding causal edges that preclude undesired execution orders. EVENTRACETOOL [11] is an instrumentation-based tool for repairing event race errors that match patterns that reflect undesirable interleavings (e.g., AJAX requests that are processed out of order). EVENTRACETOOL avoids these errors by dropping or postponing events so that no undesirable patterns can occur. We use the same mechanism to implement adverse execution.

Several projects focus on detecting event races for other programming languages, including Android [31, 82, 85, 118] and C/C++ [150]. While these works are directly inspired by the work on detecting event races in JavaScript applications [146], applications written in these languages do not rely on AJAX, so the techniques explored in our work do not apply there.

Brutschy et al. [33] show how a generalization of the notion of conflict-serializability can be used to detect race errors in applications that use eventually-consistent data stores.
12.8 Conclusion

We have presented a technique for detecting AJAX event race errors in JavaScript web applications, and described its implementation, AJAXRACER. Our technique uses a combination of light-weight dynamic analysis and controlled execution, and identifies pairs of user events that are potentially AJAX conflicting. For each pair, it generates a test that is expected to fail only if the corresponding AJAX race has observable effects on the screen. Unlike previous techniques, AJAXRACER has been designed specifically to detect AJAX races. As a result, AJAXRACER can detect observable AJAX races in real-world web applications with very few false positives.

In an evaluation on 20 widely used web pages, AJAXRACER detects errors in 12 of them. In total, AJAXRACER generates 152 tests of which 69 reveal AJAX race errors and only three are false positives. We additionally report on the usefulness of AJAXRACER’s comprehensive web-based reports, from which it was easy to locate the root cause and effects of AJAX races, although we had no prior experience with the web pages. In summary, our results show that AJAX race errors are commonplace in web applications and that AJAXRACER is an effective tool for detecting them.
Bibliography


[100] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. ACM Transactions on Programming Languages and Systems, 32(2), 2010. 115


International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 505–519, 2015. 64


on *Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005. 25


[167] TechCrunch. Users Have Low Tolerance For Buggy Apps – Only 16% Will Try A Failing App More Than Twice. [https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/](https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/), accessed on 2017/12/13. 20


