

End-User Record and Replay for the Web

Shaon Barman



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-266

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-266.html>

December 21, 2015

Copyright © 2015, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

End-User Record and Replay for the Web

by

Shaon Kumar Barman

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Rastislav Bodík, Chair
Professor Koushik Sen, Co-chair
Professor Dawn Song
Professor Susan Stone

Fall 2015

End-User Record and Replay for the Web

Copyright 2015
by
Shaon Kumar Barman

Abstract

End-User Record and Replay for the Web

by

Shaon Kumar Barman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Rastislav Bodík, Chair

Professor Koushik Sen, Co-chair

The usefulness of today’s websites is limited by their form and ease of access. Even though the web contains an ever-expanding wealth of information, much of it exists in a form that is not directly useful. How can end-users access the web in a way that meets their needs?

We present record and replay (R+R) as a way to bridge the gap between a website’s functionality and the end-user’s goal. R+R leverages an interface the user knows and is stable — that is, the webpage — in order to automate repetitive tasks. A R+R system observes a user interacting with a website and produces a script which, when executed, repeats the original interaction. End-users can use R+R to automate a sequence of actions and programmers can use these recordings as an API to execute more complicated tasks. Unfortunately, as websites become more complex, R+R becomes increasingly difficult.

The challenge with modern websites is that a single demonstration of the interaction has limited information, making scripts fragile to changes in the website. For past R+R systems, this was less of an issue because of the static nature of websites. But as the web becomes more dynamic, it becomes difficult to produce a robust script that mimics the interactivity of the user and can adapt to changes on the page.

To solve this problem, we developed Ringer, a R+R system for the web. Ringer is built on three key abstractions — actions, triggers, and elements. Ringer takes a user demonstration as input and synthesizes a script that interacts with the page as a user would. To make Ringer scripts robust, we develop novel methods for web R+R. In particular, Ringer uses the following features:

- Inferring triggers automatically which synchronize the script with the state of the webpage
- Monitoring the replay execution to ensure actions faithfully mimic the user
- Identifying elements on the replay-time page using a similarity metric

To evaluate our work, we run Ringer on a suite of real-world benchmarks by replaying interactions on Alexa-ranked websites. We compare Ringer against a current state-of-the-art replay tool and find that Ringer is able to replay all 29 benchmark interactions, compared to only 5 benchmarks for the previous approach. Additionally, our benchmarks show that a replayer needs to synchronize with the state of a webpage in order to replay correctly, motivating Ringer’s use of triggers. We show that our trigger inference algorithm can synthesize sufficient synchronization, while also having the added benefit of speeding up the replay execution. Finally, we show that R+R is useful as a building block for end-user applications by building two such tools using Ringer. One allows end-users to scrape structured data from a website simply through demonstration. The other allows end-users to aggregate real-time data from various websites in the form of live tiles, by specifying the data they want on a website through demonstration.

Contents

Contents	i
List of Code Listings	iii
List of Figures	iv
List of Tables	vi
1 Introduction	1
1.1 Record and Replay	2
1.2 Ringer	5
2 Previous Work in Browser Automation	6
2.1 Applications and Challenges	6
2.2 Audience	8
2.3 Required Skills	9
2.4 Limitations of Current Approaches	10
2.5 Conclusion	12
3 Overview	13
3.1 Properties of Record and Replay	14
3.2 Motivating Example	15
3.3 Approach	17
3.4 Limitations	22
3.5 Conclusion	23
4 Level of Abstraction	24
4.1 User Program	24
4.2 Correctness Requirements	25
4.3 Design Choices	27
4.4 Evaluation	31
4.5 Conclusion	36

5	Trigger Inference	37
5.1	Alternative Methods	37
5.2	Ringer Approach	41
5.3	Problem Formalism	43
5.4	Assigning Triggers	45
5.5	Matching and Identifying HTTP Responses	48
5.6	Evaluation	50
5.7	Conclusion	54
6	Faithful Actions	55
6.1	Challenges with the DOM Event Architecture	55
6.2	Ringer Approach	57
6.3	Record and Replay Algorithm	61
6.4	Evaluation	66
6.5	Conclusion	70
7	Element Identification	71
7.1	Problem Statement	71
7.2	Past Approaches	72
7.3	Ringer Approach	76
7.4	Algorithm	76
7.5	Evaluation	80
7.6	Conclusion	82
8	Applications	83
8.1	API for Developers	84
8.2	Scraping Relational Data by Demonstration	85
8.3	Real-time Updates by Demonstration	88
8.4	Conclusion	90
9	Related Work	91
10	Conclusion	94
	Bibliography	95

List of Code Listings

6.3.1 Record algorithm: Capturing event information	61
6.3.2 Replay algorithm: Outer loop and dispatching events	63
6.3.3 Record algorithm: Capturing event deltas	64
6.3.4 Replay algorithm: Capturing event deltas	64
6.3.5 Replay algorithm: Compensating for deltas	65
6.3.6 Record algorithm: Identifying atomic cascading events	65

List of Figures

1.1.1 Minimized JavaScript code	3
1.1.2 Page changes to Southwest.com across multiple accesses	4
1.1.3 Visual cue on Amazon.com	4
2.1.1 Model of browser	7
3.1.1 Model of browser	14
3.2.1 Screenshot of Amazon.com page	15
3.2.2 User program for Amazon.com scenario	16
3.2.3 Screenshots of Amazon.com page during motivating scenario	17
3.3.1 Comparison of how Ringer interacts with the browser versus a user	18
3.3.2 Ringer program with triggers for Amazon.com	20
3.3.3 Recorded program for Amazon.com	21
3.3.4 Two execution traces from the Amazon.com scenario	21
4.3.1 Screenshots of Southwest.com scenario	27
5.1.1 Sequence of events during Amazon.com scenario	38
5.1.2 Selenium code which waits for silver camera price to load	39
5.1.3 Expressions contained in Selenium’s Expected Conditions library	40
5.2.1 Simplified version of Amazon.com script	42
5.2.2 Two execution traces from the Amazon.com scenario	42
5.4.1 Algorithm to assign triggers to actions	47
5.5.1 Language of HTTP response trigger expressions	48
5.5.2 Algorithm to compute trigger expressions based upon HTTP responses	49
5.6.1 Accuracy vs. speedup on the suite of benchmark websites	53
6.2.1 Screenshots of Southwest.com scenario	58
6.2.2 Event traces comparing atomic events vs. naive replay	59
6.3.1 Trace of events that occur during Southwest.com example	62
7.2.1 Screenshot of Amazon.com page	73
7.2.2 CoScripter recording of user’s interaction with Amazon.com	73

7.2.3 Selenium IDE recording of user's interaction with Amazon.com	74
7.4.1 Algorithm to construct training data used to assign weights to attributes	77
7.4.2 Example element from Southwest.com	78
7.4.3 Algorithm to identify matching element on replay page	79
7.4.4 Algorithm to calculate similarity score	80
8.2.1 Overview of user interactions with WebCombine to scrape data off of Google Scholar	86
8.3.1 Mockup of live tile interface	88
8.3.2 Screenshot of calendar showing when campus pools are open at UC Berkeley	89

List of Tables

2.2.1 Browser automation tools categorized by application and audience	8
2.3.1 Current browser automation tools vs. skills required to use them	10
4.3.1 Comparison of different levels of abstraction	28
4.4.1 Ringer vs. CoScripter on a set of benchmarks	34
4.4.2 Ringer vs. CoScripter on a second set of benchmarks	35
5.1.1 Comparison of the sizes of search spaces of trigger condition expressions	41
5.5.1 Components of a URL used to synthesize a trigger expression	48
5.6.1 Comparison of different trigger synthesis algorithms on a set of benchmarks	52
6.4.1 Ringer vs. variations of action replay algorithm on a set of benchmarks	67
6.4.2 Ringer vs. variations of action replay algorithm on a second set of benchmarks	69
8.1.1 The API for recording and replaying Ringer programs	83
8.1.2 The API for parameterizing Ringer programs	84

Acknowledgments

I'd like to thank my adviser, Ras Bodik, for all his help and guidance. I greatly benefited from his direct mentoring approach and as a result, have become a better researcher. Thanks are also due to all my collaborators, including those I met during my internships at IBM Research and Mozilla. Finally, I'd like to thank my committee for their helpful comments and observations.

To my labmates, it's been great sharing an office space with you guys for the past many years. I thoroughly enjoyed all of our meandering conversations, and I hope our discussions haven't set you back too far in your PhDs. I'd especially like to thank Sarah Chasins, who worked with me on the work in this dissertation for her many great ideas.

I couldn't have made it so many years without my friends, both old friends from Texas and new friends made in the Bay Area. I will cherish all the wonderful moments I have had during my time at Berkeley. I'd especially like to recognize all those I have met at the Hillegass-Parker Co-op, which has been like a second home to me.

Finally, I'd like to thank my family: my parents, my brother Soumen and his wife Sonali. As I grow older, I realize more and more how important they are to my life.

Chapter 1

Introduction

The usefulness of today's web is limited by its form and ease of access. Even though there exists an ever-expanding wealth of information, much of it exists in a form that is not directly useful. Websites have little incentive to make their data completely open. Many large sites intentionally limit access to their user's data to keep switching costs high and prevent competition. And even for the best intentioned websites, a finite amount of resources and developer time limit which features are implemented, making it impossible to satisfy everyone's requirements. Imagine the following scenarios an end-user may face:

- A user wants to copy all his friend's contact info (such as their addresses, phone numbers, etc.) from one social media website to another which syncs with his phone, so he can access this information even if he's without data. Often, the only option is to manually copy and paste this data.
- Or another user who creates a playlist on a streaming music site for her sister's wedding, but realizes that she needs to download these songs since the venue is offline. The website allows her to purchase and download a single song, but not an entire playlist. She is left to purchase each song individually, a painfully tedious task.
- A final scenario is a user who just purchased a camera online. The website, like many retailers, offers price matches if the cost of the camera drops in the next month. He wants to monitor the price, without having to repeatedly visit the product page.

In each of these scenarios, the design decisions made by the website do not match the user's goals, creating tedious and repetitive tasks for the user.

What would it take for a user to automate these scenarios today? For some websites, there exists third-party tools which accomplish the user's task. Multiple applications sync contacts across services, such as transferring contacts between Facebook and Google Contacts. But this requires giving third parties access to private data. Websites also exist which track the price of products online. One site, camelcamelcamel.com, tracks the price of products on Amazon.com

and BestBuy. But such websites only exist for the largest online retailers. Users wanting to automate less popular websites are out of luck.

The other solution today is to become a programmer. Developers can use frameworks such as Selenium[34] or Greasemonkey[13] to write programs which automate their task. But using these technologies requires technical skills outside the domain of the average user. For end-users, they have two options: hope that a developer wrote an application which fits their task or perform the task manually.

Our goal is to enable all users to access websites in a way that's suited for their task. Instead of relying on developers to predict a user's needs, end-users should be able to build custom solutions through an interface they know well, a web browser. As a first step towards this goal we built Ringer, a record and replay (R+R) system for the browser so that all users, even those with no technical background, can easily automate webpages. Basic R+R is a prerequisite for more expressive end-user applications, such as web scrapers built by demonstration or systems to monitor websites. Users can also parameterize their demonstrations, allowing them to replay their interactions on similar but slightly different pages. Record and replay also benefits programmers, since website automation is essential for testing and debugging web applications. This thesis explores Ringer's design and how end-users (and programmers) can use it to automate the web.

1.1 Record and Replay

The idea behind record and replay is simple: a user interacts with a system, in this case a browser, which creates a script. Later on, the system executes this script in order to mimic the original user interactions. Record and replay, when done right, can be a very practical tool, allowing users of all ability levels to automate repetitive tasks without the headaches and difficulties of programming.

Level of Abstraction

To achieve robust replay, it is important to select an appropriate level of abstraction at which to record and replay the interactions. Options range from low-level assembly code to high-level natural language and even visual representations. Low-level abstractions are useful for deterministic R+R, such as those used in debugging and analysis[26, 35], but are fragile to changes that occur on live websites. For example, a page's JavaScript code might be compiled for minimization or obfuscation, such as in Figure 1.1.1. Multiple accesses to such pages contain code which is syntactically different but functionally equivalent. A naive replayer attempting to recreate the execution by executing the same sequence of JavaScript functions would diverge and fail. Similar issues would occur for fresh data on the page.

```
1 qc = function() {
2     var a = sc, c = Wb ? function(d) {
3         return a.call(c.src, c.tb, d)
4     } : function(d) {
5         d = a.call(c.src, c.tb, d);
6         if (!d)
7             return d
8     };
9     return c
10 };
```

Figure 1.1.1. Webpage code (both JavaScript and HTML) is often autogenerated, and therefore changes syntactically between page accesses, even though it remains functionally unchanged. This snippet from google.com shows minimized JavaScript code where all identifiers are shortened.

Like existing R+R tools — CoScripter[21], iMacros[5], Selenium Record and Playback[34] — Ringer works at the level of the user-facing interface, leveraging the fact that web developers have an incentive to keep this interface stable. Even as the underlying HTML and JavaScript code fluctuate, changes in displayed content are restricted by the need to deliver a consistent user experience.

Challenges of Replaying User-level Interactions

Record and replay is an inherently difficult problem due to the limited information contained in a single demonstration. The user’s demonstration represents a *partial* specification of a hidden program that captures the user’s intent. When this hidden program is run, it produces the demonstration, but there may be many such programs which produce the same demonstration. The difficulty in R+R lies in finding a program that generalizes well (*i.e.* behaves correctly with respect to the user’s intentions) when the page changes.

This is especially difficult for the web, where multiple visits to the same web page can lead to drastically different pages. We now present two sources of nondeterminism and their potential pitfalls.

- **Page structure:** Websites often roll out new layouts making it difficult to identify the page’s elements across accesses. One solution to this problem is to identify elements by specifying distinctive features of the element that do not change across executions. But learning such features automatically is difficult. And even small changes on the page, such as embedded advertisements which change between accesses, can lead the replayer to click a different button or type into a different text box than the one the user intended.

Figure 1.1.2 shows a new "Recent Searches" pull-down menu which appears on Southwest.com after the first search. Even this small addition to the page can break naive

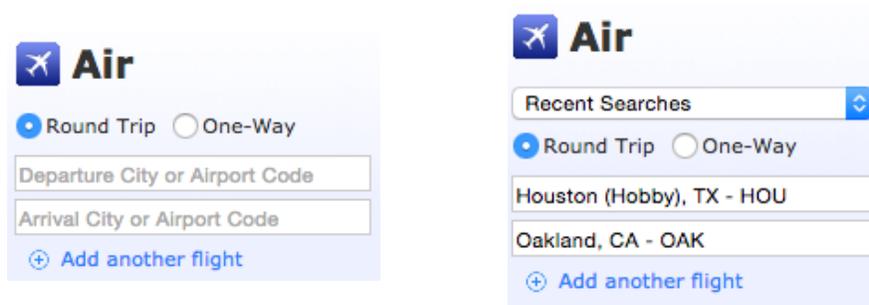


Figure 1.1.2. A new ‘Recent Searches’ pull-down menu is added to Southwest.com after the first search. This extra element makes identifying the departure and arrival text boxes more difficult.



Figure 1.1.3. When a user clicks the silver camera button, the Amazon.com page indicates that the user needs to wait for new information to load by graying out the page.

element identification algorithms, such as an synthesized XPath query which traverses the tree from the root element.

- **Interactivity:** Another difficulty lies in faithfully capturing the interactivity between the user and the web page. Early websites were static, limiting the interactions a user could make to just typing and clicking. However, today’s user expects an interactive experience, leading modern sites to use AJAX and custom JavaScript handlers to respond to user actions. Examples of this include loading data in response to clicks or scrolls and suggestion boxes which update as the user types. Since these updates may be delayed by the network or server, the replayer may need to wait arbitrarily long before replaying a user action. Figure 1.1.3 shows how Amazon.com signals a user to wait by greying out the page. The user must pause while the page loads new information from the server.

While the increasing use of these technologies has made webpages more responsive and interactive, they present substantial challenges for replay tools.

Creating a robust record and replay system for the browser requires a design which can correctly adapt the original demonstration for these sources of nondeterminism. We argue that this requires a synthesis approach. Instead of deterministically constructing a program from a single demonstration, we use synthesis techniques to search a space of candidate programs. We identify two key aspects along which a demonstration is ambiguous: how to choose the element involved in an interaction (element identification) and when an interaction should occur (synchronization).

1.2 Ringer

In his book *Designing for Emotion*[37], Aaron Walters provides a hierarchy of needs for designing interfaces, and argues that the most basic needs of any interface is to be *functional* and *reliable*. Previous tools designed for end-users often failed because they were not reliable. When a failure happened, end-users were expected to go outside their knowledge and comfort zone to fix it. When designing Ringer, our two goals were to create a system that a) any user could use and b) would work reliably without technical expertise.

Our tool, Ringer, gives end-users a functional and reliable way to automate the browser through R+R. For example, lets assume a user wants to get real time updates on the price of an item online. The user demonstrates once how to navigate to the page and the location of the price on the page. Then to update the price, Ringer replays the user demonstration to get the live price information. Unlike previous tools, we focus on robustness of replay, making the recorded script reliable even if the page changes. In addition, such a tool can be used as a building block for more expressive end-user automation systems.

To handle the challenges of record and replay on modern, interactive pages we introduce a new language for Ringer with two novel features. First, we introduce the concept of *triggers* and an algorithm for inferring them. A trigger is an event that must occur before replay can safely continue — for instance, an AJAX response that must complete before a page updates with fresh information. Second, we develop a new approach to element identification. Rather than synthesize a fixed expression during the recording, our element identification algorithm uses a similarity-based approach to find the best matched node at replay time.

Because we see record and replay as a crucial building block for the end-user tools of the future, we developed a simple but powerful API for Ringer. It allows programmers to record an end-user's interactions, treat the recording as a program which can be parameterized, and replay it with new parameters. The application designer can collect and run these programs without understanding any of the details of trigger inference, node addressing or any of the other machinery necessary to handle today's diverse and interactive pages.

Chapter 2

Previous Work in Browser Automation

In this chapter, we compare Ringer against existing browser automation tools. Our tool is not the first system designed to allow automation of websites. Languages, such as Selenium[34] and Sikuli[41], and mashups, such as IFTTT[31], allow audiences with a range of technical abilities to automate websites. But Ringer differs from these tools by targeting end-users, who do not know how to program. To accomplish this goal, Ringer automates websites through user-demonstrations, specifically record and replay (R+R). There have been other browser automation tools which use R+R, such as CoScripter[21] and iMacros[5]. But these tools were designed when the web was static, causing modern webpages to break them. Ringer gives end-users a robust means of record and replay, even in the face of nondeterminism and interactivity found on modern pages. In order to differentiate Ringer, we classify web automation tools according to several properties and show that Ringer fills a previously unmet class of challenges.

We first discuss different applications of web automation and the challenges they present (Section 2.1). We then go over the audiences which use these tools (Section 2.2) and the skills required to use these tools (Section 2.3). We conclude by discussing the limitations and choices current automation tools make and how Ringer differs (Section 2.4).

2.1 Applications and Challenges

We divide current browser automation tools by their intended purpose. Each purpose requires a different type of automation and therefore has a different set of properties.

Before going over these requirements, we go over a simple model of the browser. Figure 2.1.1 illustrates how the user, browser and server interact with each other. At a high level, the interaction starts when the browser requests the webpage from the server (sent as an HTTP request). The user interacts with the webpage by taking in visual cues and raising events on the page, such as clicks and keyboard events. These events are translated by the browser into DOM events, which the page's JavaScript code responds to. The page's JavaScript code can also modify the DOM, which gets translated to visual changes in the page. The page can also

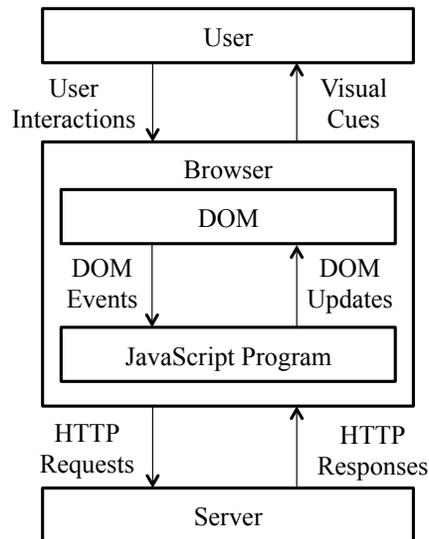


Figure 2.1.1. Model of user interactions with web application.

request new information and send information to and from the server through AJAX requests (which are sent as HTTP requests).

We now go over applications which require browser automation.

- **Testing:** Web applications are difficult to test since they span multiple devices, from server-side databases to client-side JavaScript code. Automation frameworks allow developers to write integration tests for the browser components of a web application. These tests ensure that changes to the server and webpage do not break the desired user interaction by testing the live website. These programs mimic the user by finding desired elements on the page and issuing a sequence of events. But writing these tests is difficult. Developers need to write expressions which precisely identify elements. The program may also need to wait for the page to get into a certain state before proceeding. Often these tests break because of slight changes in the page's structure, even though the page is still functional, requiring the programmer to update the test.
- **Debugging and Analysis:** Debugging a web application can be frustrating, especially when the bug is non-deterministic. For example, the order in which JavaScript callbacks are executed is nondeterministic, leading to data races. Developers can use R+R tools to record failing executions and deterministically replay the execution to investigate the failure. These tools replay the execution by working with cached versions of the webpage instead of going to the server to get a fresh version. This allows the programmer to analyze and replay the client-side webpage independent of any changes which occur to the server.
- **Repetitive tasks:** End-users often have repetitive tasks on a website, such as form filling

or copying data from one website to another. Many tools help a user complete the same sequence of actions multiple times, saving time and increasing productivity. For example, a user could demonstrate how to fill in a form using one row of a spreadsheet, and the tool would automatically fill in the form using the remaining rows. One key requirement is that the tool works with a live version of the server, so that it sees fresh data and the server records side-effects of the interaction.

- **Scraping:** This category is a subset of repetitive tasks, but its prevalence motivates a separate category. These tools allow a user to collect a large amount of data from a website. For example, a user might collect a list of all reviews for a particular restaurant from Yelp. Ideally, this data would be available in a simple format, such as a spreadsheet or a web API. But many websites do not have a publicly available API for accessing data. And often, websites make it difficult to collect data by loading the data on demand or spreading it across multiple pages. A class of tools target this problem, allowing the collection of large amounts of data from a website. Like repetitive tasks, these tools must work with a live version of the server.

2.2 Audience

We further classify tools based upon their target audience, either programmers or end-users. Tools designed for programmers expect their users to know a programming language or to learn a new programming language. These tools may also expect their users to understand browser internals, such as JavaScript, HTML and CSS. End-user tools do not have these requirements, making them accessible to a larger audience.

	Testing	Debugging	Repetitive	Scraping
Developers	Selenium [34] HtmlUnit [14] Sahi [2] Watir[38]	Mugshot [26] Jalangi [35] Timelapse [6]	Selenium [34] iMacros [5] Chickenfoot [4] Sikuli [41] Beautiful Soup [3] Abmash [27]	Scrapy [33] XPath [11] Kimono Labs [17]
End-users	N/A	N/A	Selenium [34] iMacros [5] CoScripter [21] Vegemite [23] IFTTT [31] Intel Mash Maker [9]	OutWit Hub [28]

Table 2.2.1. Browser automation tools categorized by application and audience.

We surveyed current browser automation tools and categorized them based upon application and target audience. Table 2.2.1 presents our results. Since developers are the only ones who face testing and debugging tasks, there exists no tools in those categories for end-users. Several tools are general enough to span multiple categories and we assume all end-user tools can be used by developers and all repetitive task tools can be used for scraping. At first, it may seem that end-users, our target audience, already have many tools for browser automation. But as we describe in the next two sections, many of these tools either have limited uses or require the user to learn a technical skill.

2.3 Required Skills

We now focus on tools designed for scraping and repetitive tasks and categorize them by the skills required to use them. On one end are libraries which allow programmers to manipulate a website's data. These libraries require knowledge of web technologies, including the DOM, JavaScript, HTML, CSS and other languages. On the other end of the spectrum are record and replay applications that do not require any additional skills, relying on observations collected during a user's interaction with a webpage. Table 2.3.1 surveys a number of existing web automation tools, listing which skills each tool requires. We additionally categorize each skill as *developer* or *end-user*, based on whether an end-user could reasonably learn the skill.

Developers Many tools are specifically designed for programmers and require learning new programming languages, or writing and modifying code in order to automate a task. These tools may also require the user to understand the internals of the browser.

- **HTML / DOM tree:** Representation of the webpage as a set of nodes organized into a tree structure. The DOM interface allows elements in the document to be addressed and manipulated programmatically.
- **CSS selectors / XPath:** Languages used to address and select elements in the DOM tree.
- **Tool specific programming language:** Several tools automate the browser with their own domain specific language. User's may be required to learn this language.
- **Writing custom guards:** In order to work on interactive pages, users need to write expressions which indicate that the page is ready to continue.

End-users The other category of tools are focused at end users, specifically those who do not understand the browser internals but are familiar with interacting with a browser. We do not assume end-users are able to program or even modify a program, but can interact with the website and are able to answer questions about their required tasks. End-user tools are simpler

	Skills					
	HTML/DOM	CSS/XPath	Language	Guard	NLP	Visual
Selenium	X	X	X	X		
iMacros	X	X	X	X		
Sikuli			X	X		X
CoScripter					X	X
IFTTT						
Kimono Labs						X
Ringer						

Table 2.3.1. Current browser automation tools require a variety of skills that makes them inaccessible to end-users. Some tools are accessible to end-users, but have limited functionality.

than those for developers, but this simplicity comes at a trade-off with expressivity and the ability to automate complex tasks.

- **Writing a natural language program:** Natural language programs are sequences of instructions written in English, but can be executed. These languages, such as CoScripter[21], allow end-users to program without having to learn a formal language.
- **Identifying elements on a page visually:** The user interacts with page to select elements. Examples of visual interactions include hovering over highlighted elements and selecting elements by drawing a box.

2.4 Limitations of Current Approaches

We now analyze the trade-offs existing tools have made. While the list of automation tools in Table 2.3.1 is not comprehensive, we believe it is representative of state-of-the-art tools found in industry and academia. We find that current tools either require programming skills but are robust, or require no technical background but have limitations which can hinder their use.

We broadly define four categories of tools. The first gives users an API to manipulate the browser. This API is accessible in a general programming language, allowing users to precisely specify the interaction they want to automate through a program. These tools are not accessible to end-users. The second class allows end-users to specify programs through natural language. While end-users can write these programs, they tend to be fragile and break easily on modern pages. There is also a class of tools which allow end-users to create mashups of different websites,

by using predefined interfaces to those sites. The final class of tools uses demonstrations to create a program which repeats the interaction, this is the approach that Ringer takes.

Web API Tools like Selenium and iMacros are designed for technical users, using an expressive language for automating the browser. But such tools are out of the scope of the average end-user who do not have the time to learn a completely new language. Sikuli makes programming the browser easier by allowing elements to be specified through visual images instead of text expressions like XPath, but using it still requires the ability to program. While these tools are expressive and are especially useful to programmers, their usefulness for end-users is limited.

Natural language program Tools like CoScripter attempt to bridge the gap between end-users and programming by allowing end-users to write their programs in English text. But we found that using natural language limits the types of interactions that can be expressed to a finite set of predefined interactions. CoScripter cannot handle new interaction types, such as clicking an item in an autocomplete menu. CoScripter also cannot handle pages which require the user to wait, making the natural language programs fragile.

End-user tools There do exist tools which require no programming skills. If This Then That (IFTTT) allows end-users to easily create mashups which connect different sites together. But using IFTTT requires a developer to create an interface to a website, limiting its usefulness to popular websites. Kimono Labs is also simple to use, allowing its users to scrape the data off a website by specifying the structure of the data by clicking on the elements within the browser. But its usefulness is limited to web scraping.

Demonstration CoScripter, iMacros and Selenium also come with a recorder, giving users a way to synthesize a script by demonstration. But these scripts are fragile. For example, Selenium produces scripts which contain XPath expressions to identify elements on the page. These expressions work as long as the structure of the page does not change. Once the structure changes, many of these expressions break, requiring the user to either record the script again or to manually write a new XPath expression. Since CoScripter uses natural language, it has difficulties creating descriptions for elements without text. For example, if a button only contains an image, then CoScripter will identify it as “the n th button” button on the page. Automatically synthesizing robust expression identifiers is difficult, but is necessary to produce a robust program.

These tools also cannot infer when the user is waiting. For the scripts synthesized, the replayer only waits until the next element is present on the page. None of the recorders bundled with these tools is able to synthesize a correct program for the Amazon.com example given in Chapter 1.

2.5 Conclusion

We designed Ringer to automate repetitive tasks, relying solely on demonstration. While past tools used similar approaches, they were built when webpages were static and break on today's interactive pages. We find that Ringer differs from the previous state-of-the-art automation tools in several aspects:

- The end user does not need to learn a programming language to use Ringer.
- The user can automate any webpage with Ringer, not just a predefined subset.
- Ringer is more robust to page changes than other comparable tools. We justify this claim by evaluating Ringer and other automation tools on a wide range of websites (Chapter 4).

These features make Ringer accessible to wide audience and give end-users another tool to accomplish their goals.

Chapter 3

Overview

This chapter discusses the challenges of record and replay of the browser and the design of our tool, Ringer. Designing our record and replay tool boiled down to designing a suitable language which facilitates recording. The particular language has to meet two requirements. First, that a working program in this language can be synthesized from user demonstrations. Second, the system can faithfully replay this synthesized program to mimic the user. Note that our goals do not include human readability. We envision a system in which computers, not humans, will write and modify this language. To fulfill these requirements, we designed a language with the following features.

- **Trigger inference:** Because modern webpages are interactive, scripts need to synchronize with the state of the webpage to replay correctly. Ringer's language allows for each statement to have a set of trigger conditions, all of which need to be satisfied before Ringer executes the statement. Ringer infers these triggers automatically by observing successful replay executions.
- **Faithful actions:** The Ringer language guarantees that user actions, such as clicks and key presses, observed during recordings are replayed faithfully. Informally, an action is replayed faithfully if executing the set of statements recorded for the action has the same effect on the page as the original user action. Surprisingly, this guarantee is not upheld by several existing record and replay tools due to the difficulty translating between low level observations and high level language constructs. Ringer solves this problem by observing actions and replaying actions at the same level, as DOM events.
- **Similarity metric to identify elements:** In order to execute a script, Ringer must find elements on the replay-time page which correspond to elements on the record-time page. To deal with changes to the webpage's structure that occur between accesses to the same website, we use a similarity metric, instead of a fixed element expression, so that we can use information from the replay-time page to find a corresponding element.

The rest of the chapter is organized as follows: We first discuss properties of our ideal record and replay system (Section 3.1). We base these properties on the idea of a hidden user program which guides the user through the demonstration. The goal is to synthesize a program from the demonstration that has the same effect as this user program. The rest of the chapter will focus on a running example that we envision an end-user would want to automate (Section 3.2). We then illustrate how our tool, Ringer, handles this scenario, highlighting the challenges with record and replay and how we designed Ringer to meet these challenges (Section 3.3).

3.1 Properties of Record and Replay

User Program

We designed our approach around the concept of a user program. Our informal definition of a user program is a sequence of steps that the user takes when interacting with the website. We represent this program as P . We assume that the user is executing this program during the record phase, allowing our tool to observe how a correct program interacts with the page. As shown in Figure 3.1.1, the user program interacts with a web page by observing visual cues as input, and deciding on how to interact with the page as output. We assume that given a page in state σ_{init} , applying the user program to this state $\langle P, \sigma_{init} \rangle$ can bring the page to a new state σ' such that σ' satisfies some criteria $passing(\sigma')$. Our goal in record and replay is

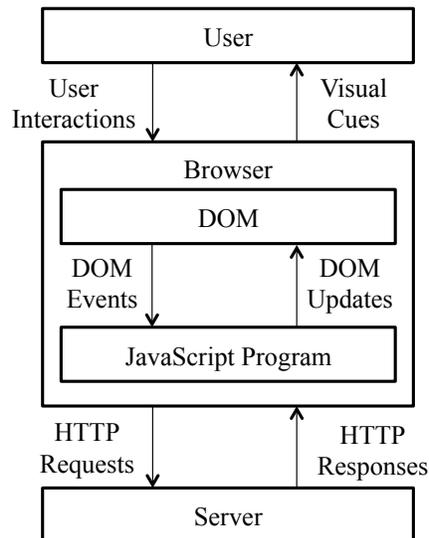


Figure 3.1.1. Model of user interactions with web application. The user interacts with the browser by observing visual cues and sending actions. The browser translates these actions to DOM events, which the page’s JavaScript code can listen to. The page’s JavaScript code can modify the DOM and send requests to the server.

to synthesize a program, P' , that can also bring the state of the page to one that satisfies the criteria *passing*.

For replay, an ideal correctness condition for P' would be: if $\langle P', \sigma_{init} \rangle \rightarrow \sigma'$ then *passing*(σ'). But such a condition is too strong. What if the original user program fails in some instances due to nondeterminism? Or if its impossible to get from σ_{init} to a passing state. Instead, we use the following condition: if $\langle P', \sigma \rangle \rightarrow \sigma''$ then $\langle P, \sigma \rangle \rightarrow \sigma''$. Basically, if our synthesized program causes the page to go to some state, then the user program can also drive the page to this state. If we combine this with the condition that $\langle P, \sigma_{init} \rangle \rightarrow \sigma'$ then *passing*(σ'), we can get our ideal correctness condition.

Assuming that the user program takes an arbitrary form makes the synthesis problem intractable. To limit the scope of the problem, we assume that P does not contain any loops or recursion, *i.e.* that P is a straight line program. While an end-user may not be able to write P in a formal programming language, we assume they can answer simple questions about P 's behavior. Even though the concept of an user program is abstract, it provides a specification for the design of our record and replay tool.

3.2 Motivating Example

We present an example of a task that an end-user might want to automate. The user wants to track the price of a camera on Amazon.com to check if the price goes down. Figure 3.2.2 gives a natural language description of the user program for this scenario. Websites exist that do

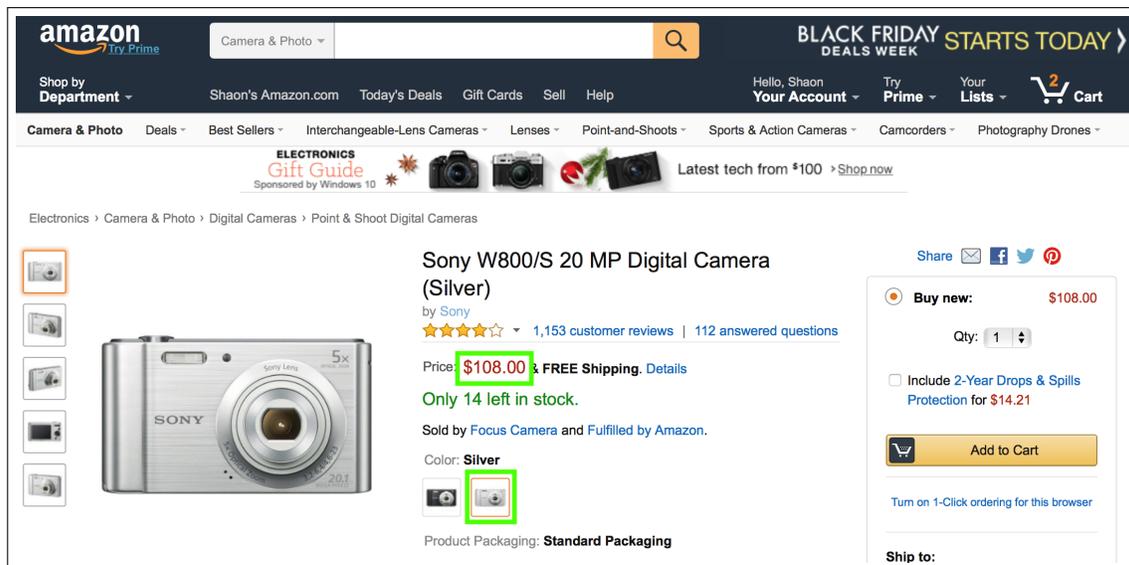


Figure 3.2.1. Screenshot of Amazon.com page for purchasing a camera. Elements involved in the motivating example are highlighted in green.

Go to Amazon.com Search for Sony W800 camera and navigate to page Click the silver color After page updates, scrape title and cost of camera

Figure 3.2.2. User program for Amazon.com scenario.

this for Amazon.com, such as camelcamelcamel.com, but what if the user wants to monitor a different website. Most likely they would be out of luck. We would like to make such price trackers require little to no effort from the user.

Figure 3.2.1 shows a snapshot of the page, with green boxes around the elements involved in the interaction. The user wants to track the price of silver camera, requiring him to click the silver camera button. Once the new price is loaded, the user can scrape the information by selecting the element containing the price. Figure 3.2.3 gives the visual state of the page throughout the interaction. One key point to notice is that after the user clicks the silver button, the page greys out the text on the page to indicate that it is currently loading new information. The price of the silver camera is only available after the screen returns to normal. If the user scrapes the price before this visual cue, they will scrape the price of the black camera which is actually \$20 cheaper than the silver camera. Understanding this visual cue is essential to replay the user's interactions correctly.

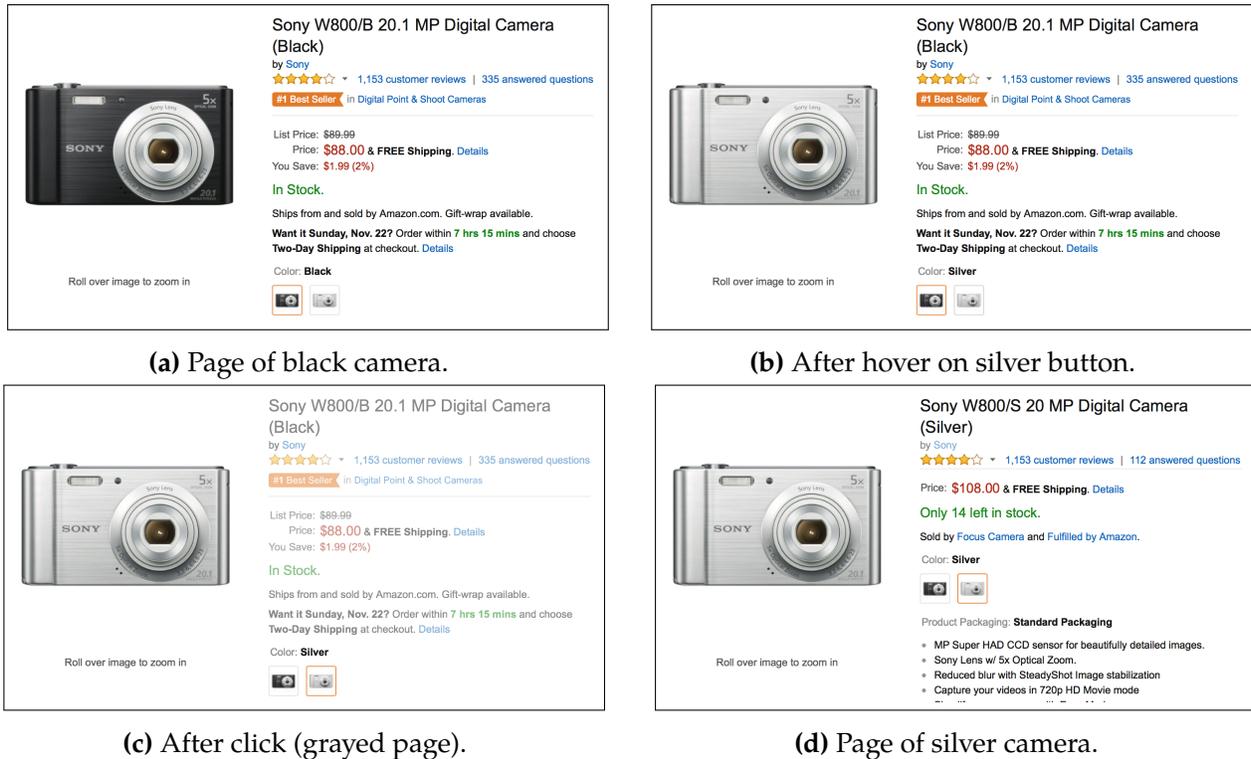


Figure 3.2.3. Sequence of screenshots walking through the user interaction. At first, the page shows information for a black camera (a). Once the user hovers over the silver button, the picture changes but the text remains the same (b). Clicking the button causes the page to grey out (c) until the final state which contains the price of the silver camera (d).

3.3 Approach

We now revisit our original goal of synthesizing a program which can mimic the user program given only demonstrations. To make the synthesis task simple, we assume that the user program can be represented as a sequence of guarded statements, with each statement composed of three components: actions, elements and triggers. Each statement is of the form:

- 1 **if** triggers t_1, t_2, \dots, t_n are satisfied:
- 2 **dispatch** action a on element e

Note that this form, while simple, can express useful interactions such as the Amazon.com example. This is also the form Ringer programs use. But since we only want to synthesize a program that is equivalent to the user program, *i.e.* can get to the same state on the website, we use abstractions for actions, elements and triggers that the user may not understand. While this choice seems counter-intuitive, the end-user does not need to write a program in this language. In fact its quite difficult to directly write a Ringer program. Instead programs are written, refined, and modified through demonstration, playback and a simple developer API. Instead

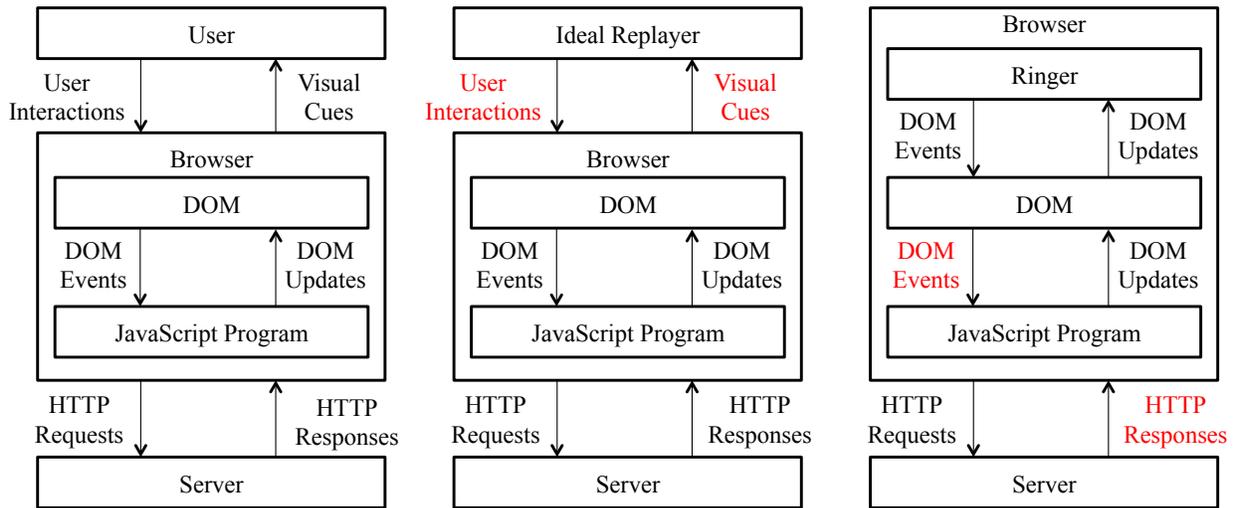


Figure 3.3.1. Comparison of how Ringer interacts with the browser versus an ideal replayer and an end-user. We highlight in red parts that each system uses to monitor and control the webpage. An ideal replayer would work with the browser at the same level as the end-user, but we find this approach to be too difficult. Instead, Ringer directly produces DOM events to mimic user interactions and observes HTTP responses as a proxy for visual cues.

of focusing on readability, we focus Ringer’s design on reliability and ease of synthesis. The challenge though is to pick abstractions that allow this within the constraints imposed by the browser.

Problems with the ‘ideal’ replay abstraction Figure 3.3.1 compares how Ringer interacts with the browser versus an ideal replayer and an end-user. An ideal replayer would work at the same level as the actual user, taking as input the visual appearance of the page and outputting user level actions. But this approach presents two challenges. First, the API exposed by browsers makes it difficult to work at this level and would require modifying browser internals or running separate system process. The other challenge is that working at the visual level requires converting the visual structure into a semantic model. Understanding the significant aspects of the visual appearance is difficult without user input. For these reasons, we decide to use different abstractions for Ringer programs.

Browser Constraints

Our first step when designing Ringer’s language was to understand the constraints imposed by existing browser technologies. Since we target our tool toward end-users, we wanted to work within existing browsers people use. That meant no browser modifications since they are hard to distribute and are fragile to internal browser updates. We also did not want to require

end-users to understand the internals of the browser, such as CSS, HTML, and JavaScript, to use our tool. In the end, we built our tool as a Chrome extension. We now discuss the interfaces Ringer uses to interact with the webpage and user.

Observable events Given our instrumentation, we have limited information about the demonstration. Below we discuss the information we can observe during the record and replay phases.

- **DOM events:** When a user interacts with the page, the browser raises DOM events so that the webpage can react to the interaction. We can listen to these DOM events to observe the user's interactions with the page.
- **HTTP requests:** During the demonstration, the webpage issues requests to interact with the server. These requests can be observed and recorded.
- **DOM changes:** The browser provides interfaces to monitor any changes on the webpage, including the addition, removal or modification of elements on the page. Monitoring these changes to the DOM allows us to understand when and how the visual appearance of the page changes.

Controllable events During replay, we can affect the page in a limited number of ways. We use these interfaces to execute the script, which should mimic the user's interactions. It's also important to note that during replay, Ringer can observe the page as if the user was interacting with the page, allowing for real-time monitoring and updates.

- **DOM events:** We can mimic any DOM event observed during the demonstration by cloning and dispatching it during replay.
- **Modify the DOM:** Another way to affect the page is to add, remove or modify elements on the page.
- **Add scripts to page:** We can also insert custom JavaScript functions into the context of the page.

Synthesized Script

Given the browser constraints and the limited information a demonstration gives, we chose the following abstractions for each component:

- Actions are represented as DOM events. Ringer listens for these events during the recording and saves all information associated with the event. During replay, it raises corresponding DOM events to mimic the user's interaction.

```

1 dispatch action mousedown on element matching {type: 'IMG', ...}
2 dispatch action focus on element matching {type: 'BUTTON', ...}
3 dispatch action mouseup on element matching {type: 'IMG', ...}
4 dispatch action click on element matching {type: 'IMG', ...}
5 if trigger HTTP request matching hostname=='amazon.com' &&
6   path=='gp/twister/ajaxv2' && params.id=='B00CBYMWNQ':
7   dispatch action capture on element matching {type: 'SPAN', ...}

```

Figure 3.3.2. Program synthesized by Ringer for Amazon.com scenario.

- Elements are represented as a set of properties of the record-time element. To find a corresponding element during the replay, we search the set of elements on the page and find an element which ranks the highest according to a similarity function.
- Triggers are represented as conditions which indicate whether the webpage has received a certain HTTP response. If a matching response is seen, then the trigger condition is satisfied. Once all conditions are satisfied, Ringer dispatches the action on the matched element.

Figure 3.3.2 shows part of the program synthesized by Ringer on the Amazon.com example, specifically when the user clicks the silver button and captures the updated price information. No guard is shown for statements which have the trivial true guard. There exists five statements, four DOM events representing the click of the silver button and a single capture event which scrapes the camera’s price information off of the page. The full element expression is not shown, but is composed of features of the element, such as the type, text content, width and other properties. Before the price information is scraped, the program waits for a specific HTTP requests which happens to be the server returning the price of the silver camera. We now go over each phase of record and replay, using the running example to illustrate our approach.

Recording During the recording phase, Ringer listens to all DOM events on the page. For each event that is raised, Ringer records the type of the event plus properties of the event. In addition, Ringer needs to record enough information about the event’s element to find a corresponding element during replay. Figure 3.3.3 shows the program Ringer creates after the recording phase. This program does not contain any triggers, and simply represents the sequence of actions the user made on the page.

Trigger inference In the next stage, Ringer uses a set of passing executions to automatically infer trigger expressions for the recorded actions. To collect this set of passing executions, it replays the program by dispatching the actions according to the timing delays observed during the recording. While this naive form of triggers can break due to server delays causing the script to fail, Ringer can slow down the replay until the script passes. The final result is a set of

```

1 dispatch action mousedown on element matching {type: 'IMG', ...}
2 dispatch action focus on element matching {type: 'BUTTON', ...}
3 dispatch action mouseup on element matching {type: 'IMG', ...}
4 dispatch action click on element matching {type: 'IMG', ...}
5 dispatch action capture on element matching {type: 'SPAN', ...}

```

Figure 3.3.3. During the recording phase, Ringer creates a basic program by listening for events on the webpage. Notice that this program contains no trigger expressions.

action a_1	mousedown	action a_1	mousedown
action a_2	focus	action a_2	focus
action a_3	mouseup	action a_3	mouseup
action a_4	click	action a_4	click
response r_1	http://www.amazon.com/gp/twister/ajaxv2?rid=0CVQ46WY688BD951R2AB&id=B00CBYMWNQ	response r_3	http://www.amazon.com/gp/twister/ajaxv2?rid=1KC215EY0BM8G68N26DQ&id=B00CBYMWNQ
response r_2	http://www.amazon.com/gp/bd/impress.html/ref=pba_sr_0	action a_5	scrape price
action a_5	scrape price	response r_4	http://www.amazon.com/gp/bd/impress.html/ref=pba_sr_0

Figure 3.3.4. These two traces represent two successful executions of the program from Figure 3.3.3. Each trace is a sequence of actions and HTTP responses. Note that the ordering between actions and responses differ between the two traces.

traces, each composed of actions and server requests. Figure 3.3.4 shows two traces from the Amazon.com example.

From these traces, Ringer uses the URL of the response to match responses across traces. After matching responses, Ringer constructs an expression which can distinguish the response from other responses in the executions. The final step attaches these conditional expressions as triggers to actions in the program, ensuring that the program does not continue execution until the page receives the response. The result is the program shown in Figure 3.3.2.

Replay To replay the final program, Ringer executes each statement in sequence. A statement is ready to execute once all trigger conditions are met, *i.e.* the browser receives the matching HTTP responses. At this point, Ringer searches for an element on the page that has the highest similarity score with the element seen during the recording. It then executes the statement by dispatching the action on the corresponding element.

Benefits and Challenges

Working at this level of abstraction allows Ringer to faithfully record and replay a user's interactions with a webpage. We go over each component's abstraction and discuss the benefits of the abstraction, along with challenges created by the choice.

- **Actions:** Working at the level of DOM events allows Ringer to minimize the abstraction gap between record and replay, ensuring that the recorded actions are faithful, *i.e.* that they affect the page in the same way as the user's actions. There is one caveat though. The browser treats DOM events raised by the operating system differently than those raised programmatically through JavaScript, which can cause the replay execution to diverge from the demonstration. To compensate for the browser, we develop a runtime system which ensures that replaying DOM events are faithful.
- **Elements:** Fixed element expressions, such as XPath, are brittle to page changes since they encode only one way to access the node and only contain information from the record-time webpage. Instead, a similarity function allows the page the change in arbitrary ways. But the key challenge is to find a similarity function which is fast to compute and robust to the types of changes found on real websites.
- **Triggers:** While visual triggers match the user's intuition, they are challenging to synthesize due to the large number of changes to the website. It's difficult to prune down this space of possible visual cues to find the actual change that the user waits for. By using HTTP responses as trigger conditions, we circumvent this problem. The space of HTTP response triggers is much smaller, allowing Ringer to synthesize expressions with less information. But this requires us to reliably identify a HTTP response across multiple executions, which is challenging.

3.4 Limitations

While our approach is able to automate interactions on many websites, it does have its limitations. One inherent limitation is the inability to replay interactions that require absolute timing. An example of such an interaction is scrolling around a map which has inertia, *i.e.*, the map continues to move based upon how fast the mouse was moving when the user let go. Many online games also depend on the absolute timing between interactions, preventing Ringer from working on them. Ringer assumes that there only exists a relative ordering between user interactions, and that a user interaction can always be delayed without causing the script to fail.

The remaining limitations are specific to our choices for each component.

- **Actions** Some DOM events occur at a very high rate, such as `mousemove`, `mouseover` and `mouseout`. Recording and replaying all of these events can strain computation resources,

making the page unresponsive. Therefore, Ringer does not record these events. An ideal solution would allow Ringer to recognize when these events are important to the interaction, *i.e.*, an event handler on the page is listening for the event, and selectively record the important events.

- **Elements** Our element identification algorithm is inherently best-effort, given that we have no control over how the webpage changes.
- **Triggers** Ringer's triggers are based upon server requests, which we find sufficient in most cases. But, this choice prevents Ringer from handling situations where the webpage requires the user to wait until a client-side computation finishes or for a fixed period of time. For the types of scenarios we focus on, this does not seem to be an issue.

3.5 Conclusion

We built Ringer around the concept of a user program, decomposing a user's interaction into a sequence of actions, elements and triggers. Ringer also uses this form to generate a script from the user's demonstration. We carefully pick abstractions for each of these components which allow us to faithfully mimic the user's interactions during the recording.

Outline of dissertation The rest of the thesis is organized as follows. Chapter 4 compares each abstraction we chose against other possibilities. We also evaluate our choice of abstractions by empirically comparing Ringer against another record and replay tool. Chapters 5, 6, and 7 go over the design of Ringer. Chapter 5 discusses how we synthesize trigger expressions by replaying a naive script multiple times. Chapter 6 discusses our run-time system which ensures that Ringer replays actions faithfully during replay. Chapter 7 discusses how we identify elements across executions using a similarity function. In Chapter 8, we discuss how developers can build end-user applications on top of Ringer. We present two proof-of-concept tools: a web scraper that the user programs solely through demonstration and a live tile interface that scrapes information automatically from a website by replaying a demonstration. We conclude the thesis by discussing related work in Chapter 9 and summarizing our contributions in Chapter 10.

Chapter 4

Level of Abstraction

This chapter discusses different abstraction choices for actions, elements and triggers, exploring how each may be used by a browser record and replay system. Choosing a different abstraction for each component leads to completely different replay systems. For example, if the abstraction chosen to represent actions worked at the machine instruction level, then the system can nondeterministically reproduce the recorded execution. While a different abstraction, say DOM events, may not have this property, but can handle pages whose JavaScript code changes for each execution. A strength of a natural language abstraction is human readability. We explore the trade-offs between different options, motivating the abstractions we chose for Ringer and illustrating the difficulty with record and replay.

This chapter is organized as follows. First, we briefly revisit the idea of a user program which was discussed in Chapter 3 (Section 4.1). We then go over properties of an ideal abstraction, giving us a way of comparing different choices (Section 4.2). The next section uses this framework to compare different abstractions for each component (Section 4.3). Finally, we present an evaluation of abstraction choices by comparing Ringer against CoScripter (Section 4.4). We run each tool on a set of benchmarks involving popular webpages. We show that Ringer succeeds in many cases where CoScripter fails, mainly due to the choice of abstractions it uses for its programs.

4.1 User Program

When a human user interacts with the browser during the recording phase, we assume that the human is driven by an implicit, hidden *program*, in which each statement takes the form "wait for X , then do Y on Z ." The goal of replay is then to mimic this hidden user program. We propose that any replayer that accomplishes this task can be decomposed into a language with the following constructs:

- **Actions:** means by which the replayer affects the application

- **Elements:** components of the application interface on which actions are dispatched
- **Triggers:** expressions which signal when an action should occur

To limit the scope of the problem, we assume the hidden user program is a straight-line sequence of guarded statements, with each statement composed of three components: actions, elements, triggers. Each statement is of the form:

```
1  if triggers t1, t2, ... tn are satisfied:
2  dispatch action a on element e
```

To execute this statement, the system waits until all trigger conditions, t_1, \dots, t_n , are satisfied. It then waits until an element on the page matches element e , at which point it dispatches the action a on the element. But this form does not specify what abstractions to use for the components. One can imagine that an actual user program uses high-level abstractions, such as visual cues for triggers, visual identifiers for elements, and natural language such as "move the mouse and click" or "type the string *foo*" for actions.

But Ringer does not necessarily need to use the same high-level abstractions. In fact, we find that such high-level abstractions are difficult to record from a demonstration.

4.2 Correctness Requirements

We formalize the concepts of elements, actions and triggers in terms of properties they should satisfy. While these properties are not necessary for correct replay, they provide one way of ensuring that replay is successful. Formalizing each of these abstractions allows us to develop a general record and replay algorithm, that is correct as long as the implementation for each component meets certain criteria.

Actions

The key to a good action abstraction is being able to faithfully mimic the user's interactions. One simple way of accomplishing this goal is to minimize the abstraction gap between actions and the browser interfaces used to record and replay them. We observed that previous R+R systems failed because converting between observations and the language of actions was not faithful. Specifically, a user can demonstrate two different interactions, but these systems would record the same action statement, making it impossible to correctly recreate the interaction during replay.

Before giving requirements for faithful actions, we first define the notion of effects. Let $\sigma(\text{page})$ be the state of the webpage at a specific moment. The effect of an user or replayer action is a change in state of the page, $\text{effect}(\text{action}, \text{element}, \text{state}) = \delta(\text{state}, \text{apply}(\text{action}, \text{element}, \text{state}))$. The δ function is domain specific, but informally takes two page states and finds the differences. Note that these differences may also occur on the server, such as when the page submits a form.

When recording a user action, we need to save it into the language of actions. Let *userAction* be a user demonstration of an action. $record(userAction)$ converts the interaction into a sequence of action-element tuples. Assuming that we can find corresponding elements and the pages are in the same state, the effects of replaying this sequence of statements should be equivalent to the effect of the user during the recording. This notion of equality is defined by the webpage domain.

We want to define *record* and *replay* such that for all user actions *a* and elements *e* on any page *p*, the following holds:

$$effect(replay(record(a)), e, \sigma(p)) = effect(a, e, \sigma(p)) \quad (\text{faithful action})$$

Elements

The main constraint for an element expression is the ability to match elements the user interacts with during the recording to elements that exist during the replay. Let each element have a unique identifier, e_i , which is consistent across multiple accesses to the same page. During recording, we need to save information about an element to create an element expression. We do this by calling a function which takes in the element and the state of the page and outputs an expression, $s_i = description(e_i, \sigma(page_{record}))$. During replay, the replayer finds a matching element e'_i by calling a function with the expression and the current state of webpage, $find(s_i, \sigma(page_{replay}))$.

The replay algorithm requires *find* to return strictly one element, meaning that *description* must find a way to distinguish e_i from all other elements. We say that e'_i is a matching element, if it represents the same node as e_i according to some notion of equality. Since the record and replay interfaces are different, this notion of equality is defined by the webpage domain and the intentions of the user.

We want to define *find* and *description* such that for any element e_i on page $page_{record}$ and the matching element e_i on $page_{replay}$, the following holds:

$$find(description(\sigma_e(e_i), \sigma(page_{record})), \sigma(page_{replay})) = e_i \quad (\text{matching element})$$

Triggers

The final piece is to determine when an action should be dispatched. Modern pages change even if the user is not interacting with it, such as updates caused by an AJAX response. We need to detect when the application is in a state to accept the next action, such that the effect of the action is the same as in the recording. We construct trigger expressions which indicate when the page reaches this state. If the trigger expression is true then the script can continue execution. For simplicity, we assume that such a point will happen at some point and once

the page is in this ready state, it will not revert (i.e. it is not possible to wait too long before dispatching an event). A sufficient trigger allows the script to synchronize with the state of the page so that the next action is replayed faithfully.

Let the user demonstrate action a on element e during the recording. Let $action = record(a)$ and $element = description(\sigma_e(e), \sigma(page_{record}))$ which are the text descriptions of the action and element. We want to come up with a trigger expression $trigger$ such that following holds:

$$trigger(\sigma(page_{replay})) \rightarrow effect(replay(action), find(element, \sigma(page_{replay})), \sigma(page_{replay})) = effect(a, e, \sigma(page_{record}))$$

(sufficient trigger)

4.3 Design Choices

We now move onto designing abstractions for a browser record and replay system. We explore the different choices available for each component, analyzing how these choices affect the properties of R+R.

Working Example

We begin by introducing a working example, which will motivate our design choices. The example follows a user as she searches for a flight on Southwest.com. This simple task highlights

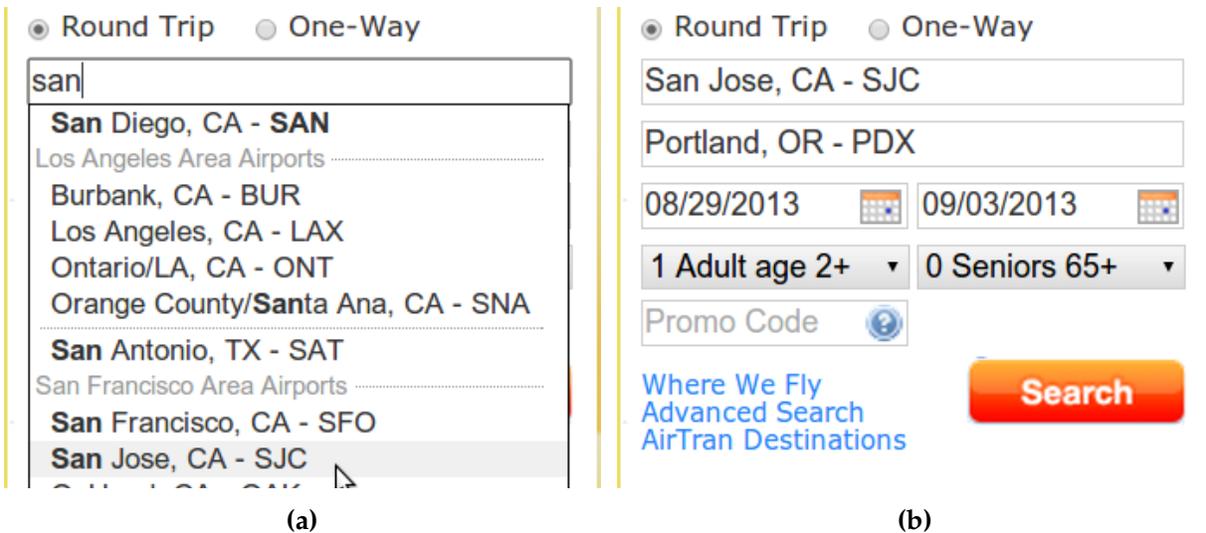


Figure 4.3.1. On Southwest.com, the user interacts with a custom autocomplete widget to select airports. After typing in three letters, the autocomplete menu appears (a). Once the user clicks one of the options, the selected text appears in the text field (b).

	Level of Abstraction	Correctness requirements	Robust to page changes	Generalizable
Actions	Machine Instructions	X		
	JavaScript Function Calls	X		
	DOM Events	X	X	O
	User Level		X	X
Elements	Memory Addresses	X		
	DOM Identifiers	X	X	X
	Visual Identifiers	O	O	X
Triggers	User Wait Times		X	X
	Server Responses	X	O	O
	DOM Changes	X	X	X

Table 4.3.1. We present the levels of abstractions we considered when designing Ringer. The table shows which properties each abstraction satisfies: an ‘X’ indicates that the property is satisfied, an ‘O’ indicates the property is satisfied, but with limitations and a blank space indicates that the property is not satisfied.

many subtleties of the browser which make recordings difficult to replay.

To find a plane ticket on Southwest Airlines’ website, a user enters departure and arrival cities, and the dates of travel. As shown in Figure 4.3.1, an autocomplete menu appears after the user types the first three characters of a city name. To select an airport, the user clicks on one of the suggested options. Next, the user selects the flight dates by typing them in the date fields, or by selecting dates from the calendar widget that appears when a date field is selected.

Widgets like Southwest.com’s autocomplete menu and calendar selector are composed of standard HTML elements, but use JavaScript event handlers to respond to user actions. We expect such custom form widgets to become increasingly common as developers seek more control over the look and feel of their websites.

The user’s actions update an HTML form, which is submitted to the server. The server uses the form values to display flights. Most of the important form values — the dates and number of passengers — are visible to the user. However, when a user clicks on an autocomplete suggestion, the visible textbox’s new content is not the only change. The widget also writes the selected airport’s code to an invisible textbox. This hidden airport code is used by the server, but the visible airport name is not. Custom interactions like this one, with its JavaScript-managed effects on the page state, have frustrated past attempts at live page replay.

Properties of R+R

We go over properties we desire in our record and replay system. Afterwards, we discuss different language abstractions for each component and whether they satisfy our properties. Table 4.3.1 summarizes our findings, showing which properties each abstraction satisfies.

Ability to satisfy correctness requirements Some abstractions are not expressive enough to capture the user’s program, and therefore automatically will not satisfy the requirements outlined in Section 4.2. For other abstractions, it may be difficult to synthesize expressions in that language given the limited APIs the browser offers to observe the demonstration. Some of these limitations may be circumvented by modifying the browser to add hooks, although this is often impossible to do on proprietary software and difficult to maintain. It’s often best to pick abstraction levels that map directly to an available API.

Robustness to page changes A recording made on one page should work on the same page, even if the page changes slightly. But the webpage developer does not need to keep abstractions below the user-facing interface stable. For instance, HTML pages and their accompanying JavaScript code can change on every load, even if the visual appearance of the page is exactly the same. In order to produce robust, reliable scripts, we want to pick stable levels of abstraction, as close to the user’s interface as possible.

Ease of generalization A replay script should work on new data, in order to facilitate end-user programming. For example, if a script recorded on Alice’s profile page will not work on Bob’s profile page, its usefulness is limited. To make scripts generalizable, we should pick levels of abstraction that allow parameterization.

Discussion

Actions Our choice of an action abstraction is shaped by our need for replay that is faithful on unchanged pages and robust to page changes. Thus, we would not be well served by replaying machine instructions or replaying the same sequence of JavaScript function calls using lexical function names. Tools that enforce strict determinism [26, 6] use these levels of abstractions, since they assume the page remains unchanged, but the our goal is to replay live webpages, allowing the page and the underlying JavaScript to change.

This leaves DOM events and user-level actions. User-level actions are essentially a natural language description of actions — *e.g.*, click, select, or enter. This approach makes generalization simpler and because applications aim to give their users stable experiences, it is typically robust to page changes. But the difficulty with user-level actions is that the browser does not expose an interface to observe them. CoScripter observes DOM events but records actions at the user-level. Unfortunately it is not faithful — this conversion abstracts away information that the replayer needs in order to replay the action. For instance, in our working example, CoScripter abstracts the process of typing “san” and clicking on the autocomplete suggestion to the action "Type ‘San Jose, CA - SJC’." Because the Southwest.com page listens for the click event to set a hidden airport code field, CoScripter’s abstraction fails. Replaying the script leads the user to an error message. Bridging the gap between DOM events and natural language prevents CoScripter from faithfully replaying many interactions, even if the page remains unchanged.

Our tool works at the level of DOM events. DOM events can be observed and dispatched through a browser API. During the record phase, Ringer registers to observe all events raised by the user’s interactions. During the replay phase, Ringer mimics the user by dispatching the same sequence of events. Thus, DOM events are faithful. The sequence of DOM events is not human readable, and therefore cannot be directly edited by the user, but we consider this an acceptable trade-off for faithful replay.

Elements We pick from among the different element abstractions based upon whether replay will be robust to changes in page structure and page content. For the same reasons that instruction-level replay is too fragile for actions, memory-address level replay is too fragile to handle changes with elements. Visual identifiers are robust to page changes, since application designers attempt to give users stable experiences, but they may not generalize well. In our working example the “San Jose, CA - SJC” suggestion is our record-time target. But when the time comes to modify the script for a different city, say Detroit, finding the item that looks most like “San Jose, CA - SJC” may lead us astray.

Fortunately, DOM features are closely linked with visual identifiers and are therefore generally stable. Barring substantial changes in webpage design, finding the first item in the autocomplete menu is sufficient to find our “San Jose, CA - SJC” node. If Southwest.com added a destination which changed the order of the items, the replayer might be confused. But DOM identifiers also contain additional information — like position in the DOM tree, text, and so on — that facilitates generalization. We can use these features along with a similarity metric to find a matching element on the replay page. Thus DOM identifiers give us both robust replay on changed pages and the ability to generalize.

Triggers Our choice of trigger abstraction is shaped by another property, not listed in Table 4.3.1. Unlike actions and elements, the replayer cannot observe what cues the user looked for during the demonstration, and therefore must search a space of possible trigger expressions to infer a correct one. The need for an efficient search strategy motivates our choice in trigger abstraction.

One plausible option, which makes trigger inference trivial, is to emulate the user’s timing. Unfortunately, this is not robust. In the Southwest.com example, attempting to click on the autocomplete suggestion before the autocomplete menu appears would lead to unintended results. Consider what would happen if the menu took one second to appear during the recording, but ten seconds during replay. Timing-based triggers are easy to infer, but are not close enough to the user’s real intent to offer sufficient triggers.

DOM changes are much closer to the user’s program. In our working example, a DOM change like “wait for DOM element x to be added” is quite robust to page variation. If the timing before the menu appearance varies wildly, this trigger still works. Unfortunately, it is difficult to infer these expressions in practice. DOM changes are easy to observe, but because

today's pages are interactive, the number of DOM changes is high. Trying to distinguish which changes are relevant requires searching a large space of possibilities, making inference of such expression infeasible.

Fortunately, we can use server responses as a proxy for DOM changes. Often, the page changes because the server sends new data through AJAX requests. If the user only pauses to wait for new data, then listening for these requests serves as a perfect proxy for visual cues. The trigger "wait for server response x to be processed", where server response x contains the autocomplete menu's content, will be robust. Since there are far fewer server responses than DOM changes, we are able to infer sufficient server response triggers through demonstration.

A full system With these abstractions in place, we can build a full system. During recording, our tool observes all DOM events, and the DOM-level features of the nodes on which they are dispatched. We hand off this naive program to our trigger inference algorithm, which replays it to create a set of successful execution traces. From these traces, we infer a set of server response triggers for each action, which we use to construct the final program. During replay, Ringer replays each statement. It first checks if all trigger expressions are satisfied, by comparing each expression against the list of server responses observed so far. Once all expressions are satisfied, the replayer uses the DOM features to identify a node and dispatches the DOM event on that node.

4.4 Evaluation

In this section, we evaluate Ringer's ability to record and replay a wide variety of real websites. We compare against an existing tool, CoScripter, which also offers browser record and replay. These experiments focus on how Ringer's choices of abstractions allow it to successfully replay a user's interactions. We present both a case study and a more empirical evaluation on a set of benchmark interactions.

Southwest.com Case Study

We use the Southwest.com example from Section 4.3 to evaluate Ringer's abstractions against CoScripter's. Like Ringer, CoScripter's only input is a user demonstration. Unlike Ringer, its output is a set of high-level, human-readable instructions. For the Southwest.com interaction, CoScripter produces this script:

- go to 'http://www.southwest.com/'
- enter 'San Jose,CA-SJC' into the 'Enter departure city' textbox
- enter 'Portland,OR-PDX' into the 'Enter arrival city' textbox

- click the first 'Search' button

This script looks correct. However, notice that CoScripter abstracts away the autocomplete interaction. During the demonstration, this interaction involved typing three letters and then clicking an airport name from a menu, which updated the contents of a textbox. CoScripter reduces this to entering text in a textbox, the visible outcome of the user's actions. When CoScripter executes the script, the webpage produces the following response: "Oops! No departure airport selected for the outbound flight. No arrival airport selected for the outbound flight."

The CoScripter script does fill all visible text boxes on the page correctly, but the page also has several hidden inputs. Clicking on an airport causes the page to set these hidden inputs. Typing in the text boxes does not. To faithfully replay this interaction, a tool must trigger the same event handlers that were executed during the record phase.

We attempted to manually write a script in the CoScripter language to execute this task. Unfortunately, even with our deep understanding of why the original script failed, this was not possible. Because CoScripter does not mimic the individual key presses, the event handler that displays the autocomplete menu is never triggered. Without access to the menu, it is impossible to trigger the event handlers that set the hidden form elements. In contrast, Ringer raises all the record-time events at replay-time, correctly replaying the interaction.

This interaction underscores the importance of replaying at the level of DOM events. While CoScripter's abstraction makes scripts more human-readable, we believe this comes at a high cost of expressivity which restricts what interactions are replayable. CoScripter loses information by listening to DOM events, then translating them to higher-level instructions. Any approach that does not raise all observed DOM events can fail, because any DOM event may trigger a callback which is crucial to the correctness of the script.

Empirical Evaluation

To empirically evaluate our approach, we recorded a set of interactions on real world websites using both Ringer and CoScripter. Our benchmark suite includes scenarios from 29 distinct websites. All benchmark websites were taken from Alexa's list of most-visited sites by category[1]. Besides being likely targets for automation, these sites also tend to be complex, making heavy use of AJAX requests, custom event handlers, and the other features that make record and replay difficult. In short, they offer ideal stress tests. We selected from multiple Alexa categories to ensure we tested on a variety of interaction types.

For each interaction, we completed what we perceived to be a core site task, such as buying a product from an online retailer. For pages with custom widgets or complicated user interactions, we made a point of completing those interactions as part of the pages' benchmarks. We checked user-defined invariants to determine whether each replay succeeded or failed. Invariants checked for the presence of a particular string in a particular target node. For instance, for the

Walmart benchmark to succeed, the string "This item has been added to your cart." must appear on the last page of the interaction.

For each benchmark, we recorded the interaction including invariants. We then replayed the interaction, marking the execution successful if the tool dispatched all events and all invariants passed.

Limitations Our evaluation procedure does not stringently test Ringer’s element identification algorithm, since our approach is oblivious to the algorithm that is used to find target nodes. Thus, we clear browser cookies before all recorded or replayed interactions (to avoid page changes that result from multiple visits) and replay immediately after recording (to minimize although not eliminate page changes that result from server-side modifications).

Furthermore, since CoScripter does not automatically synthesize triggers, we do not evaluate Ringer’s trigger abstraction. Instead we default to using the observed timing during the recording as naive type of trigger.

Initial Benchmark Experiments In Section 4.3 we argued that record and replay for live pages should work at the level of DOM events and elements should be saved as DOM features. To evaluate these claims, we ran Ringer and CoScripter on an initial suite of benchmarks. The results appear in Table 4.4.1. We found that Ringer succeeded on 24 of the 25 benchmarks (96%), while CoScripter successfully replayed 4 (16%). While these results were promising, they raised two questions.

- Why did Ringer fail on the single benchmark (Yelp)?
- Why did CoScripter fail so often on these modern pages?

To answer the first question, we debugged Ringer to try to find a root cause of the failure. We found that a focus event was being dispatched twice, causing the browser to execute the event handlers associated with that event twice. After fixing this issue, Ringer was able to replay the Yelp benchmark successfully.

Answering the second question though required more information. We decided to conduct another set of experiments targeted at understanding why CoScripter failed. In this second set of experiments, when a script failed, we examined the CoScripter script and diagnosed a root cause for the failure. Often though multiple causes of failure are identified, in which case we list all.

Second Benchmark Experiments The results of this second set of experiments appear in Table 4.4.2. We found that Ringer succeeded on 26 of the 26 benchmarks (100%), while CoScripter successfully replayed 5 (19%). While many of the scenarios were taken from the first set of experiments, we replaced a few which posed challenges outside of the questions we were trying to answer. One such page, Ebay, constantly required new recordings since auctions only lasted

Site	Description	Ringer	CoScripter
amazon	add camera to the cart	✓	✗
bestbuy	add camera to the cart	✓	✓
bloomberg	get Google stock info	✓	✗
booking	book a hotel in city A	✓	✗
ebay	place bid on a phone	✓	✗
expedia	book flight from A to B	✓	✗
facebook	search, navigate to friend's page	✓	✗
facebook	create event and add friends	✓	✓
facebook	click links to find phone number	✓	✗
gmail	add and remove stars from emails	✓	✗
google	get directions from A to B	✓	✗
google	search for "PLDI 2014"	✓	✓
google	translate "hello" into Hindi	✓	✗
kayak	book flight from A to B	✓	✗
linkedin	find connections from school A	✓	✗
mapquest	get directions from A to B	✓	✗
myfitnesspal	calculate calcs burned swimming	✓	✗
paypal	send money to a relative	✓	✗
southwest	book flight from A to B	✓	✗
tripadvisor	find vacation rentals in city A	✓	✗
twitter	send a tweet	✓	✗
walmart	buy Kit Kats	✓	✓
xe	convert 100 INR to BDT	✓	✗
yelp	find restaurants in city A, filter	✗	✗
youtube	check stats for "Gangnam Style"	✓	✗

Table 4.4.1. We list the results of running Ringer and CoScripter on a set of benchmarks. The table shows Ringer outperforms CoScripter on this set of modern websites, replaying 24 sites correctly vs. 4 sites for CoScripter.

Site	Description	Ringer	CoScripter	Element	Interaction	Other
allrecipes	find recipe and scale it	✓	×	×	×	
amazon	find price of camera	✓	×		×	
best buy	find price of camera	✓	×		×	
bloomberg	find price of Google stock	✓	×		×	
drugs	find side effects of Tylenol	✓	✓			
facebook	find friends phone number	✓	×	×	×	
gmail	compose and send email	✓	×	×		
goodreads	find books related to Infinite Jest	✓	✓			
google	search for "Berkeley"	✓	×		×	
google maps	find estimated time of drive	✓	×		×	
google translate	translate "hello" into French	✓	×		×	
hotels	book a hotel	✓	×			×
kayak	book a flight	✓	×		×	
mapquest	find estimated time of drive	✓	×		×	
myfitnesspal	calculate calcs burned swimming	✓	×			×
opentable	make a reservation	✓	×		×	
paypal	transfer funds to friend	✓	×	×		
southwest	book a flight	✓	×		×	×
tripadvisor	book a hotel	✓	×		×	
target	buy Kit Kats	✓	✓			
thesaurus	find synonyms of "good"	✓	×	×		
walmart	buy Kit Kats	✓	×		×	
xe	convert 100 INR to BDT	✓	×		×	
yahoo finance	find price of Google stock	✓	✓			
yelp	find restaurants in Berkeley	✓	×		×	
youtube	find stats for video	✓	✓			

Table 4.4.2. We list the results of running Ringer and CoScripter on a second set of benchmarks. Ringer outperforms CoScripter, replaying 26 sites correctly vs. 5 sites for CoScripter. For every benchmark that CoScripter failed to replay, we diagnosed a root cause for that failure as shown in columns **Element**, **Interaction**, and **Other**. **Element** indicates that replay failed due a misidentified element. **Interaction** indicates that replay failed due to a user interaction which was not recorded properly. **Other** indicates a different issue with the replay.

a few days. Other pages required replaying an extremely long sequence of events, such as all mousemove events, in order to succeed.

The benchmarks reveal two common CoScripter failures. First, some scripts fail because CoScripter cannot find target elements on replay-time pages (shown as **Element** in table). We do not consider this a fundamental flaw, and believe it could be fixed with some careful engineering.

The second and most foundational problem is that CoScripter abstracts away some of the interactions observed during the recording (shown as **Interaction**), making CoScripter incapable of correctly replaying these interactions. One example of this is on the Google search page. In that scenario, the user types a partial string "berk" and then clicks on one of the options from an autosuggest menu. This entire interaction is recorded as "enter 'berk' into the 'Search' textbox." But without the click, the query is not executed and the replay fails. CoScripter might improve performance by adding new instructions to capture common webpage idioms, such AJAX-updated pulldown menus. However, as web practices continue to evolve, such a scheme would be difficult to maintain, and would probably never be able to capture all types of interactions.

As for the three benchmarks which failed for other reasons, two failed because CoScripter did not faithfully mimic a user's key press. When a user types a string on a webpage, the page dispatches a sequence of DOM events, but CoScripter only updates the text without dispatching these DOM events. The last benchmark failed due to CoScripter not waiting for the page to fully load.

4.5 Conclusion

In this chapter, we go over the different components which make up Ringer's record and replay algorithm, and possible abstractions for each component. Choosing suitable abstractions is key to creating a R+R system which faithfully mimics the user's interactions. We compare our choice of abstractions against CoScripter, showing that our choices allow us to replay significantly more modern webpages in our benchmark suite.

Chapter 5

Trigger Inference

This chapter discusses how Ringer synchronizes with a webpage, similar to how the end-user synchronizes their interactions with the visual state of the page. As seen in the Amazon.com example from Chapter 3, without this synchronization an execution may get into a undesired state, unable to proceed or worse, silently interacting with a page in an undesired manner. To solve this problem, we synthesize a set of triggers for each action, which pause the replay execution until the webpage is ready for the next action. Ringer synthesizes these trigger expressions by observing several successful executions of the script, and then uses these executions to correlate possible trigger expressions with an action.

While we would ideally mimic the end-user and synchronize on the visual state of the page, the triggers we learn are in form of "*Have we seen a HTTP GET/POST response with URL x?*". By learning these triggers, the scripts Ringer synthesizes become more robust to nondeterminism in timing, such as server delays. Related to this robustness is the ability to speed-up user interactions. Knowing when the page is ready for the next action prevents us from waiting unnecessarily, allowing Ringer to replay minute-long recorded interactions in seconds.

This chapter is organized as follows. We first go over alternative methods of synchronizing a script with the state of a web page (Section 5.1) in order to give context for our own approach (Section 5.2). We then formalize the problem of synthesizing and assigning trigger expressions to actions (Section 5.3). We discuss the general algorithm of assigning trigger expressions (Section 5.4) and then discuss how these trigger expressions are constructed from HTTP responses (Section 5.5). We conclude this chapter by evaluating our approach on a set of real-world websites, showing that triggers are crucial to successful record and replay (Section 5.6).

5.1 Alternative Methods

To motivate our approach, we first discuss existing methods of webpage synchronization. We use the Amazon.com scenario from Chapter 3 to illustrate how we differ from these approaches.

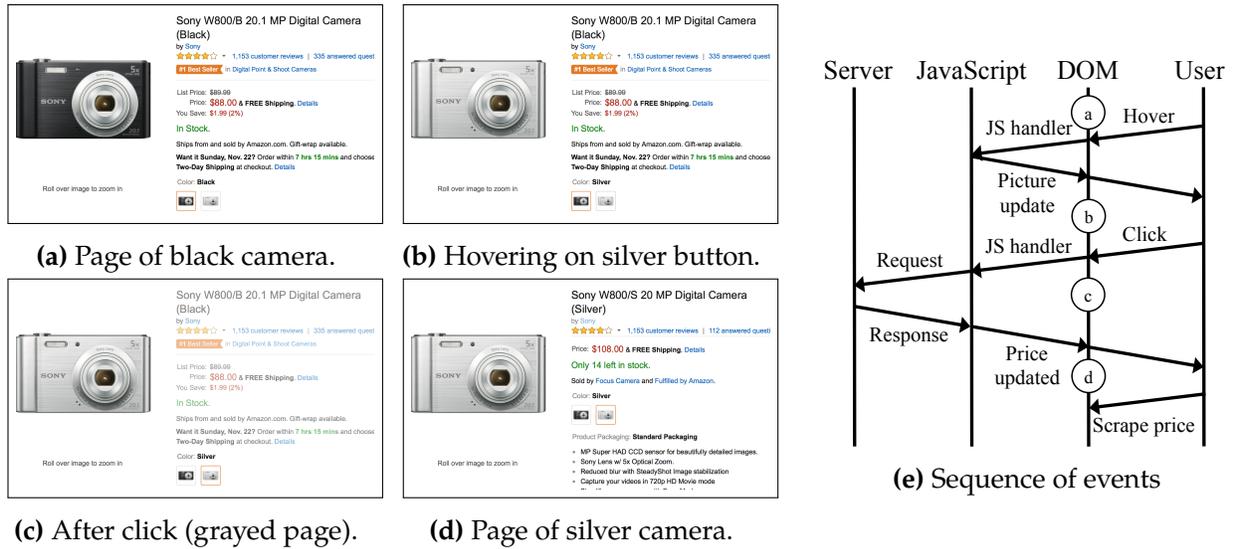


Figure 5.1.1. We illustrate the synchronization in the Amazon.com scenario. Each circle in Subfigure (e) corresponds to a webpage state shown in (a-d). Note that hovering over the silver option instantly displays the silver camera picture but not its price. Only after the click, the page requests the silver price from the server, graying out the page. The response updates the price and removes the gray overlay.

Running Example

We now go over the Amazon.com example, illustrating the need for trigger expressions in the scripts Ringer synthesizes. In this example, the user wants to scrape the cost of a silver Sony W800 camera. She does this by first navigating to the black version of the camera and clicking the silver button. Figure 5.1.1 shows the sequence of pages the user sees before and after the click. Subfigure (e) shows sequence of events that occurs during the example. When the user clicks the silver button, the page asynchronously loads the new price information through an AJAX request. In order to scrape the correct information, the user must wait until this AJAX request completes. Otherwise, she will scrape the cost of the black version.

In order to successfully replay the user interaction, Ringer must also wait for the page to load the new price information before attempting to scrape the price. To deal with this problem, we synthesize trigger expressions. These expressions observe the replay execution and evaluate to true only if the page is in a state to receive the next action.

For this example, the user understands that the page is ready once the title contains "silver" and the page is not grey. Ideally, Ringer would mimic the user and synthesize an expression that picked up on the visual cue. While Ringer could evaluate such expressions by observing changes in the DOM, the large search space of possible visual cues makes it difficult to automatically synthesize such expressions. Instead, we use HTTP requests as a proxy for visual cues. Before explaining our approach, we use the Amazon.com example to illustrate the pitfalls of past approaches.

```
1 # Save color name of the button that is clicked
2 color_button = driver.find_elements_by_xpath(...)
3 color_name = color_button.get_attribute('alt')

5 # Wait until the product title contains the color name
6 WebDriverWait(driver, 10).until(EC.text_to_be_present_in_element(
7     (By.ID, "productTitle"), color_name))
```

Figure 5.1.2. Selenium code which waits for silver camera price to load.

Manually Writing Triggers

Currently, users writing web scripts must manually write these trigger conditions. A common way of writing such conditions is to observe the page over multiple demonstrations and identify text which indicates readiness for next action. We give an example of these types of conditions for Selenium[34], a widely used testing framework that can automate user interactions with a website. Figure 5.1.2 gives a Selenium code snippet containing a trigger expression which is sufficient to correctly replay the Amazon.com example. The snippet must pause the script until the page loads the updated product information, using the `text_to_be_present_in_element` function from Selenium’s Expected Conditions library to detect when the product’s text changes. To write this condition, a programmer must first understand that the color’s name exists in the button’s `alt` property. Then, he must manually write XPath expressions that identify the button and title by examining the DOM tree. Expecting end-users to write these conditions is unreasonable. Most end-users have no knowledge of HTML or the DOM, nor do they understand the mechanisms of the page. Instead, any record and replay system needs to synthesize trigger expressions automatically.

Synthesizing Visual Triggers

Ideally by observing a user demonstration, Ringer could synthesize satisfactory trigger expressions that detect visual changes, such as an image appearing. But this approach has problems: the search space of possible expressions is too large and creating an expression that identifies a DOM node across executions is difficult.

We now argue why the space of expressions is too large. Let n be the number of nodes in the DOM tree. If we assume all trigger expressions only involve one node, such as “*Wait for node e ’s text to contain foo* ”, then the space of expressions is $n * c$ where c is the number of expression types. For the Amazon.com example, there are over 4000 nodes on the webpage and 17 types of Selenium conditions (listed in Figure 5.1.3), naively producing a space of 68000 expressions. But a single node is often not sufficient to capture the visual cue. If the trigger expression requires two nodes, such as if one node’s text matches another as in Figure 5.1.2,

alert is present
 element located selection state to be
 element located to be selected
 element selection state to be
 element to be clickable
 element to be selected
 frame to be available and switch to it
 invisibility of element located
 presence of all elements located
 presence of element located
 staleness of
 text to be present in element
 text to be present in element value
 title contains
 title is
 visibility of
 visibility of element located

Figure 5.1.3. Expressions contained in Selenium’s Expected Conditions library

then this space is $n^2 * c$, which is 272 million expressions. Such a large space of expressions is difficult to prune, leading to spurious triggers that do not actually correspond to the state of the page. Furthermore, monitoring this space of expressions is computationally infeasible.

One way to reduce the search space would be to use the changes of the DOM tree as triggers, eliminating any trigger expressions with unchanged elements. The browser exposes a mutation events API, which tracks changes such as adding and removing DOM nodes or modifying properties of a DOM node. For the Amazon.com example, there are only 600 mutation events observed when the user clicks the button. But this approach has limited expressivity, for example it cannot express the condition in Figure 5.1.2 where one node contains the text of another. In addition, it requires Ringer to find a robust expression to identify a DOM node across multiple executions. In general, node identification is non-trivial since there are many possible expressions which could identify the node, but only some which are robust, *i.e.*, correctly identifies the node across all executions.

Because of the large number of nodes on modern webpages and the difficulty in identifying nodes across executions, trigger conditions which involve DOM nodes are difficult to synthesize. Instead, we use server responses as a proxy for visual cues. We assume that if a user needs to wait for a visual cue, then the cause of that visual change is a response from a single server request. Ringer observes when these responses complete and uses them as triggers. This approach allows us to sidestep the challenge of synthesizing an expression which detects a visual change. For the Amazon.com example, there are only 35 such server responses.

Type	Size	# for Amazon.com
One-node Selenium	$n * c$	68,000
Two-node Selenium	$n^2 * c$	272,000,000
Mutation Events	m	600
HTTP responses	r	35

Table 5.1.1. Comparison of the sizes of search spaces of trigger condition expressions. n is the number of nodes on the page, c represents the number of condition expression types, m is the number of mutation events and r is the number of HTTP responses.

But using HTTP responses as triggers has its own challenges. Specifically, Ringer needs to robustly identify server responses across executions and infer dependencies between responses and user actions. We also acknowledge that this approach may fail if there exists waits induced purely by client-side JavaScript, such as a timed countdown. We have not come across many such pages in practice, although this observation may be skewed by the types of tasks we target.

5.2 Ringer Approach

Before going into Ringer’s approach, we informally discuss the problem of synthesizing trigger expressions from server responses. From the recording, we get a sequence of actions that have a fixed total order. We assume each action has a dependency on zero or more server responses. These responses occur in every execution, but not necessarily between the same actions. We say an action is *dependent* on a response if dispatching that action before the response arrives causes the script to fail. Let (a, r) be in the dependency relation if action a is dependent on response r . If (a, r) exists in the relation, then (a', r) also exists in the relation for all actions a' that occur after a . This relation is unknown to Ringer. A dependency is satisfied if response r completes before action a is dispatched and a trace satisfies the dependency relation if all dependencies are satisfied. The goal is to synthesize a set of trigger expressions that ensure that all replay executions satisfy the hidden dependency relation.

We now give a high level overview of Ringer’s approach to synthesizing these expressions. The first step is to replay the script multiple times, creating a set of totally ordered traces composed of actions and responses. If an execution is successful, then we know that the execution’s trace satisfies the hidden dependency relation. Given a set of passing traces, we infer an optimal dependency relation and synthesize a set of trigger expressions.

Example We illustrate our approach on a simplified version of the Amazon.com example. Let the original script be a sequence of two actions (Figure 5.2.1). We can naively replay this script by following the observed timing during the recording. While mimicking the user’s timing is slow and can be fragile to network delays, it is often sufficient to satisfy all dependencies

that exist in the script. In this case, we replay the script to get two traces of passing executions (Figure 5.2.2).

action a_1	click silver button
action a_2	scrape price

Figure 5.2.1. Simplified version of Amazon.com script.

action a_1	click silver button	action a_1	click silver button
response r_1	http://www.amazon.com/gp/twister/ajaxv2?rid=0CVQ46WY688BD951R2AB&id=B00CBYMWNQ	response r_3	http://www.amazon.com/gp/twister/ajaxv2?rid=1KC215EY0BM8G68N26DQ&id=B00CBYMWNQ
response r_2	http://www.amazon.com/gp/bd/impress.html/ref=pba_sr_0	action a_2	scrape price
action a_2	scrape price	response r_4	http://www.amazon.com/gp/bd/impress.html/ref=pba_sr_0

Figure 5.2.2. These two traces represent two successful executions of the program from Figure 5.2.1. Each trace is a sequence of actions and HTTP responses. Note that the URLs of responses r_1 and r_3 have different rid values.

Identifying responses To infer the dependency relation, we must identify common HTTP responses across multiple traces by giving each an unique expression. These expressions need to have both high recall and high precision with regards to the true matching of HTTP responses. This is important since these expressions will be used as trigger expressions to signal if Ringer should execute an action. If an expression has low precision, then an action may be prematurely dispatched leading the script to fail. If an expression has low recall, then Ringer may wait unnecessarily long, possibly deadlocking.

In our approach, we error on the side of high precision, and use timeouts to handle cases of low recall. For our example, we identify r_1 and r_3 as the same response even though the URLs are different (the rid parameter is different). They match since both the URL hostname and URL path are the same. To increase precision, we also use URL parameters that remain constant in all traces to identify the response. In this case, the id parameter is consistent across executions but the rid parameter is not. The final expression to identify this response is then `hostname=='amazon.com' && path=='gp/twister/ajaxv2' && params.id=='B00CBYMWNQ'`.

Assigning triggers Another challenge Ringer faces is inferring which responses an action depends upon. Let t_1 and t_2 be the trigger expressions which refer to $\{r_1, r_3\}$ and $\{r_2, r_4\}$ respectively. Let $\{a_i \rightarrow \{t_m, \dots\}\}$ represent the program where action a_i waits on trigger

expressions in $\{t_m, \dots\}$. Given only the first trace, Ringer can only safely infer the program $\{a_2 \rightarrow \{t_1, t_2\}\}$ even though second trace shows that a_2 cannot depend on t_2 (since a_2 comes before r_4).

Some assignments of trigger expressions to actions will cause greater synchronization than others. Assume the only actual dependency for the Amazon.com example is a_2 depends on t_1 . The program $\{a_2 \rightarrow \{t_1, t_2\}\}$ is sufficient to replay the script successfully, but causes Ringer to wait for t_2 unnecessarily. A better program, *i.e.* one which has less synchronization, is $\{a_2 \rightarrow \{t_1\}\}$. Given the second trace, Ringer can infer this less synchronized program.

Because Ringer has limited control during replay, it can only delay when an action is dispatched, not when a HTTP response is received. This limits which traces may be observed, leading to programs with unnecessary synchronization. Because our approach is conservative, we wait until possibly false dependencies are fulfilled to ensure that we do not break an actual dependency. In practice, we found that even with this unnecessary synchronization, we are able to significantly speed up replay executions while preserving correctness.

Our approach can be summarized as follows: we attempt to synthesize a set of trigger expressions for each action such that:

- the expressions accurately identify the correct HTTP responses across executions (expressions have high precision and high recall)
- the trigger expressions are satisfied only when the page is in a state which can accept the next action (triggers are sufficient)
- the page is in a state which can accept the next action only when all trigger expressions are satisfied (triggers are necessary)

To satisfy these goals, we infer trigger expressions from a set of totally ordered traces created by successfully replaying the script. The algorithm works by synthesizing a set of trigger expressions for each action by:

- Finding all responses which occur before the action and are not matched with a dependency to a previous action
- Synthesizing a set of trigger expressions such that each expression has a matching HTTP response across all traces
- Adding these trigger expressions to the action

5.3 Problem Formalism

We assume we have an original trace o which is a sequence of events, $e_0, \dots, e_{m+n-1} \in E$. An event is either an action $a_0, \dots, a_{m-1} \in A$ or a trigger $t_0, \dots, t_{n-1} \in T$ (for the sake of brevity, an

action represents an action-element tuple). There exists a partial order, $C \subseteq E \times E$ between the events in t . The partial order represents dependencies between events. We represent $(x, y) \in C$ as $x \leq y$. Part of this partial order is known, namely that there exists a total order for all actions and that there is no relationship between triggers. We also assume that the greatest element in C is a_{m-1} . Actions can be causally dependent on triggers, but no trigger is causally dependent on an action. The exact relationship between triggers and actions is not known. We summarize the constraints on C below:

$$a_0 < a_1 < \dots < a_{m-1} \quad (5.3.1)$$

$$\forall t_i, t_j \in T. t_i \neq t_j \rightarrow t_i \not\leq t_j \quad (5.3.2)$$

$$\forall e \in E. e \leq a_{m-1} \quad (5.3.3)$$

$$\forall t \in T, a \in A. t \perp a \rightarrow t < a \quad (5.3.4)$$

Let the *action-rank* of an action be one plus the number of actions less than it. The action a *action-covers* the trigger t if $t < a$ and no other action a_i exists such that $t < a_i < a$. Every trigger t has a unique action cover, $acover(t) = a$. Finally, let $delay(t, a)$ be the difference in action-rank between $acover(t)$ and a .

$$arank(a_i) = 1 + |\{a \in A | a < a_i\}| \quad (5.3.5)$$

$$acover(t) = a \text{ such that } t < a \wedge \nexists a_i. t < a_i < a \quad (5.3.6)$$

$$delay(t, a) = arank(a) - arank(acover(t)) \quad (5.3.7)$$

Lattice of Partial Orders The space of all possible partial orders C which meet our criteria forms a lattice, ordered by the subset relation on elements in C . The bottom element in the lattice only pairs triggers with the final action. The top element pairs all triggers with all actions.

$$\perp = \{(e, e) | e \in E\} \cup \{(a_i, a_j) | i < j\} \cup \{(t_i, a_{m-1}) | 0 \leq i < n\} \quad (5.3.8)$$

$$\top = \{(e, e) | e \in E\} \cup \{(a_i, a_j) | i < j\} \cup \{(t_i, a_j) | 0 \leq i < n, 0 \leq j < m\} \quad (5.3.9)$$

Each element in the lattice represents a possible program. We can convert a partial order C to a mapping between an action and a set of triggers, which can be directly transformed into a Ringer program. An action must wait for all triggers mapped to it before being executed. The lattice ranks partial orders based upon the amount of synchronization in the corresponding program. The bottom element requires the least, while the top element requires the most.

$$toMapping(C) = \bigcup_{i=0}^{m-1} a_i \rightarrow \{t | t <_C a \wedge \nexists a_j. t <_C a_j <_C a_i\} \quad (5.3.10)$$

To create a mapping, we associate each action with the set of triggers that it action covers. This mapping partitions the triggers across the actions in the trace. We can also convert a mapping m to a partial order.

$$\text{toPartialOrder}(m) = (\{(e, e) | e \in E\} \cup \{(a_i, a_j) | i < j\} \cup \{(t, a) | t \in m(a)\})^+ \quad (5.3.11)$$

To create a partial order, we first create the relations defined by the problems constraints. Then we add each pair (t, a) which relate triggers to actions in m . To make this relation a partial order, we find the transitive closure.

Objectives We want to infer a mapping between user actions and a set of trigger expressions, $m \in A \times \mathcal{P}(T)$ such that that two constraints are met: triggers are **sufficient** and triggers **minimize delay**.

The first constraint ensures that the hidden dependencies between triggers and actions are met, ensuring that replay is successful. Let C represent the hidden true partial order and M be $\text{toPartialOrder}(m)$. m is consistent if:

$$C \subseteq M$$

Sufficiency ensures that the mapping we infer satisfies all actual dependencies required to correctly replay the script, *i.e.*, that the replayer does not dispatch an action too early. But sufficiency by itself does not lead to the best replay execution, as the mapping which maps the first action to all triggers, *i.e.* \top , is always sufficient in our model but may wait unnecessarily long.

Our second constraint attempts to remedy unnecessary waits by minimizing the delay induced by the triggers. Let $m' = \{(t, a) | t \in m(a)\}$. We define the delay of m to be:

$$\sum_{(t,a) \in m'} \text{delay}(t, a)$$

Intuitively, we unnecessarily delay the replay execution whenever we prematurely wait for a response. Therefore, we want to wait for a response only if the next action requires that response. The challenge in minimizing these delays is that we have limited information to observe the hidden dependencies. Since we cannot enforce that an action is dispatched before a specific server response is handled, we may not be able to observe all orderings between actions and responses.

5.4 Assigning Triggers

To add triggers to actions, we developed an algorithm to discover which user actions depend on which responses. Our algorithm is conservative, *i.e.* if an action needs to wait for a particular

HTTP response and we find a correct expression to identify that response, then the set of triggers synthesized is sufficient. But because we have limited control over the browser, we may not produce all possible passing traces and therefore infer extra dependencies causing unnecessary delay.

Points to note We go over insights about the problem that we use in our algorithm's design.

- A trivial solution to this problem would take one trace, and let each action depend on those HTTP responses which immediately preceded it in that trace. While the program produced by this algorithm satisfy the sufficiency criteria, we can use all traces to find a better program that has less delay.
- If a dependency (t, a) exists, then t will appear before a in all passing traces.
- Its difficult to get any information from a failing execution. The cause of a failure may have nothing to do with a trigger, such as a misidentified element. Even if we assume that a trigger is at fault, a failing run tells us that some dependency was not met, but not which dependency. Isolating this dependency is difficult given our limited control over the browser.

We now go over Ringer's algorithm to assign trigger expressions to actions.

Algorithm The input to the algorithm is the original sequence of actions and a set of passing traces, each a sequence of actions and responses. The output is a mapping between actions and a set of trigger expressions.

For every action, the algorithm finds a set of trigger expressions that match responses seen before that action in all passing traces. The only sure way to know that there does not exist a dependency between an action and a response is to observe that the action happens before the response during a passing execution. Therefore we conservatively assign a maximal set of these trigger expressions to the action.

ADDTRIGGERS in Figure 5.4.1 assigns the described triggers. The algorithm iterates over all user actions (line 3), finding responses which happened before the action across all executions, but have not been mapped to a previous action. It does so by first computing all responses that happen before the action (lines 6 to 7) and then removes triggers which are already assigned (line 8). This set of responses is then fed to SYNTHTRIGGERS which produces a maximal set of trigger expressions given the unmatched responses (line 10). The triggers are then assigned to the current action (line 11) and loop continues.

Lemma 1. *ADDTRIGGERS produces a mapping which is sufficient.*

```

ADDTRIGGERS(actions : List[Action], runs : Set[List[Event]])
1  mapping : Map[Action, Set[Trigger]] ← {}
2  used : Set[Trigger] ← {}
3  for action : Action ← actions do
4    responses : Set[List[Event]] ← {}
5    for run : List[Event] ← runs do
6      prefix : List[Event] ← run.slice(0, run.indexOf(action))
7      all : List[Event] ← prefix.filter(isResponse)
8      unmatched : List[Event] ← REMOVEMATCHED(all, used)
9      responses ← responses ∪ {unmatched}
10     triggers : Set[Trigger] ← SYNTHTRIGGERS(responses)
11     mapping ← mapping ∪ {action → triggers}
12     used ← used ∪ triggers
13  return mapping

```

Figure 5.4.1. Algorithm to add triggers to actions based upon a set of passing executions.

Proof sketch We present a proof by contradiction. Assume that some dependency (t, a) is not met by the mapping produced by ADDTRIGGERS. Since (t, a) is a dependency, each trace given to ADDTRIGGERS must satisfy this dependency. Therefore, when *action* is set to *a* in the for loop on line 3, *t* will exist in *all* for all iterations of the inner for loop. If *t* is not in *used*, then it will be a part of the triggers returned by SYNTHTRIGGERS and mapped to *a*. If *t* is in *used*, then *t* is already mapped to an earlier action. Either outcome is a contradiction, since the mapping returned by ADDTRIGGERS satisfies the dependency (t, a) .

Lemma 2. ADDTRIGGERS produces a mapping with minimal delay with respect to the set of passing traces.

Proof sketch We present a proof by contradiction. Assume that there exists a mapping m' with less delay than the mapping m return by ADDTRIGGERS. We also assume that m' is sufficient with respect to the hidden true partial order. That means we can reduce the $delay(t, a)$ for some

t, a such that $t \in m(a)$. The delay can be reduced by mapping a higher index action a' to t . But the loop in line 3 maps an action to the set of triggers that have not been mapped a previous action and happened before that action in all successful runs. This means that t appears before a in all traces. We can construct a partial order p such that $(t, a) \in p$ which is consistent with all traces. Because p may be the hidden true partial order and m' is not sufficient with respect to p , we reach a contradiction.

5.5 Matching and Identifying HTTP Responses

In order to use HTTP responses as triggers, we must synthesize an expression that can robustly identify the response. But identifying server responses across multiple executions is difficult since different URLs may actually represent the same semantic data. For example, a server response may use different session ids in the URL or use a proxy server instead of the original. The challenge is to identify a response, while not confusing it with other responses witnessed during the script's execution.

We developed a technique to identify responses based primarily upon the URL of the response. Figure 5.5.1 shows a grammar of our synthesized trigger expressions. The features of the HTTP response that we use are the hostname of the URL, the path of the URL, the type of response (either GET or POST) and query parameters. Table 5.5.1 shows the breakdown of a sample URL into these components.

Before dispatching an action during replay, Ringer waits until all trigger expressions associated with the action evaluate to true. A trigger expression evaluates to true if Ringer witnesses a server response that matches the hostname, path and type. This response must also match all

```

trigger := hostname && path && type (&& params)* (&& order)?
hostname := hostname == string
path := path == string
type := type == (GET | POST)
params : params.string == string
order := isAfter(id)

```

Figure 5.5.1. Language of HTTP response trigger expressions.

hostname	path	params
amazon.com	gp/twister/ajaxv2	rid: 0CVQ46WY688BD951R2AB id: B00CBYMWNQ

Table 5.5.1. Components of a URL used to synthesize a trigger expression. The URL is r_1 from Table 5.2.2, `http://www.amazon.com/gp/twister/ajaxv2?rid=0CVQ46WY688BD951R2AB&id=B00CBYMWNQ`.

```

SYNTHTRIGGERS(respsSets : Set[List[Response]])
1  baseSets : Set[List[Tuple[String, String, String]]] ← {}
2  for resps : List[Response] ← respsSets do
3    base : List[Tuple[String, String, String]] ← resps.map(r → (r.host, r.path, r.type))
4    baseSets ← baseSets ∪ {base}
5    common : Set[Tuple[String, String, String]] ←  $\bigcap_{b \in \text{baseSets}} b$ 
6    exprs : List[Trigger] ← {}
7    for base : Tuple[String, String, String] ← common do
8      paramSets : Set[Map[String, String]] ← {}
9      for resps : List[Response] ← respsSets do
10     match : List[Response] ← resps.filter(r : Response → isMatch(base, r))
11     params : Map[String, String] ← match.first().params
12     paramSets ← paramSets ∪ {params}
13     commonParams : Map[String, String] ←  $\bigcap_{p \in \text{paramSets}} p$ 
14     expr : Trigger ← CREATEEXPR(base, commonParams)
15     exprs ← exprs ∪ {expr}
16 return exprs

```

Figure 5.5.2. Algorithm to compute trigger expressions based upon HTTP responses.

query parameters added to the expression. If the trigger expression contains an `isAfter(id)` clause, then Ringer must receive the response after dispatching action `id`, otherwise the response will not satisfy the trigger expression.

Algorithm We now got over the algorithm to synthesize trigger expressions from responses. The algorithm takes as input a set which contains lists of responses. Each list corresponds to unmatched responses from a single trace. The output is a set of trigger expressions, which match responses from the input.

CREATEEXPR in Figure 5.5.2 presents pseudocode of our algorithm to create trigger expres-

sions. The key idea of the trigger synthesis algorithm is to use a response's hostname, path and type as a first attempt of matching responses across traces. The algorithm takes as input a set where each element corresponds to the list of unmatched HTTP responses from a single trace. It then maps each response to a tuple of hostname, path, and type (line 3) and checks if there exists a common tuple amongst all runs (line 5). For each common tuple, the algorithm will form a trigger expression. To increase the precision of the expression, we find the set of common query parameters (line 9 to 13).

Two parts of the actual expression synthesis algorithm are omitted for clarity.

- **Ordering with actions:** If the trigger expression has been used to identify a previous response, we add a `isAfter` clause to the trigger expression stating that the response must occur after the action which used the previous trigger. This ensures that a single response does satisfy two trigger expressions.
- **Multiple responses with matching hostname, path and type:** On line 11, the algorithm chooses the first response from the list of matching responses, which may not be optimal if there more than one response. In this situation we choose the responses which maximizes the number of common parameters.

The algorithm assigns a unique expression to each HTTP response that can be matched across executions. But one issue is that these expressions may be overfit to the limited set of traces observed, leading Ringer to not recognize the correct HTTP response during replay. To deal with these cases (low recall), we add a timeout for each trigger which causes replay to continue even if it cannot match an HTTP response for the trigger.

5.6 Evaluation

In this section, we present an evaluation of Ringer's trigger inference on a wide variety of real websites. We evaluate the trigger inference algorithm by running it on a suite of benchmark interactions and compare the results against more naive trigger algorithms.

Procedure

To empirically evaluate our approach, we recorded a set of interactions on real websites using Ringer. Our benchmark suite includes 21 scenarios. We found all benchmark websites on Alexa's list of most-visited sites by category[1]. Besides being likely targets for user automation, these sites also tend to be complex, making heavy use of AJAX requests, custom event handlers, and the other features that make record and replay difficult. In short, they offer an ideal stress test. We selected from multiple Alexa categories to ensure we tested on a variety of interaction types.

For each website, we completed what we perceived to be a core site task, such as buying a product. For pages with custom widgets or complicated user interactions, we made a point of completing those interactions as part of the pages' benchmarks. We did however avoid interactions that required non-click mouse events, such as mouseover, mouseenter and mousemove. While Ringer can replay such events, the large number of such events during a typical interaction make such benchmarks impractical.

To determine whether each replay succeeded or failed, we manually defined invariants for each benchmark. Invariants checked for the presence of a particular string in a particular target element during the execution. For instance, for the Walmart benchmark to succeed, the string "This item has been added to your cart" must appear on the last page of the interaction.

Our evaluation procedure is not meant to stringently test Ringer's target identification algorithm, since our approach to triggers is oblivious to the algorithm that is used to find target elements. Thus, we clear browser cookies before all recorded or replayed interactions (to avoid page changes that result from multiple visits) and replay immediately after recording (to minimize although not eliminate page changes that result from server-side modifications).

For each benchmark, we recorded the interaction, including invariants. We then replayed the interaction, marking the execution successful if all invariants passed.

Results

Table 5.6.1 summarizes our results. For each benchmark, we created four versions of the script. For the **user-timing** version, Ringer waits the same amount of time that the user did before dispatching an event. The **no-wait** version dispatches an event as soon as possible, only pausing when a sufficient element cannot be found on the page. The **2-run trigger** and **3-run trigger** versions each represent versions which used HTTP response triggers synthesized from 2 and 3 traces respectively.

We ran each version ten times, recording whether the execution succeeded and the running time of the execution. If all ten runs succeeded, then an 'X' is placed in the corresponding cell in Table 5.6.1.

Figure 5.6.1 shows graphically how each of these versions performs against each other. The x-axis gives the percentage of runs which succeeded while the y-axis shows speedup compared to the original version. The **no-wait** version (square) gives a theoretical ceiling on speedup. The **user-timing** version (circle) shows that the original script should succeed if given enough time. An ideal version of the script would have perfect correctness and maximum speedup, placing it on the top-right of the graph.

For 9 out of 21 benchmarks, trigger inference was not needed. This is not to say no synchronization was needed, since we identified many of these benchmarks as requiring a trigger. But often the naive trigger of waiting for the action's element to appear is sufficient.

For 10 out of 21 benchmarks, triggers were necessary and we found that our trigger inference algorithm produced a version that was faster than the original version and succeeded more

Name	Description	User identified trigger	user-timing	no-wait	2-run trigger	3-run trigger
allrecipes	Find recipe and scale it (chicken soup)	-	X			X
drugs	Find side effects (Tylenol)	autocomplete menu	X	X	X	X
gmail	Compose and send email	autocomplete menu		X	X	X
goodreads	Find related books (Infinite Jest)	-	X	X	X	X
google	Search for phrase (Berkeley)	wait for results	X	X	X	X
mapquest	Find length of commute (Berkeley to Stanford)	wait for directions to load	X	X	X	X
thesaurus	Find related words ("good")	-	X		X	X
yahoo	Find graph of stock price (Google)	autocomplete menu	X	X	X	X
zillow	Find cost of house (310 Oakland)	wait for page to load	X	X	X	X
amazon	Find price of camera	wait for new price	X			
facebook	Find friends phone number (Vikram)	wait for page to load	X		X	X
g-translate	Translate word to another language (hello in French)	wait for translation	X		X	
hotels	Book a hotel (SFO)	autocomplete menu	X		X	X
paypal	Transfer funds to friend	waiting to load confirmation	X		X	X
myfitnesspal	See how many calories burned during activity (swimming)	wait for exercise to load	X		X	X
target	Buy item (kit kats)	wait for store availability	X		X	X
walmart	Buy item (kit kats)	autocomplete menu	X		X	X
xe	Convert money	-			X	X
yelp	Find businesses in area (Berkeley)	wait for results to load	X		X	X
opentable	Make reservation (Gather)	autocomplete menu		X		
southwest	Book a flight (SFO to HOU)	-	X	X		X

Table 5.6.1. We list the results of running Ringer using different trigger inference algorithms. **User identified triggers** lists any visual cues that the user waited for. The first grouping shows benchmarks which did not require extra synchronization, while the second grouping of benchmarks does require synchronization (since the **no-wait** version fails). For each benchmark, we created four versions: replay with absolute user timing (**user-timing**), replay with no delays between actions (**no-wait**), infer triggers using two traces (**2-run triggers**) and infer triggers using three traces (**3-run triggers**). Each version was executed 10 times with X indicating that all 10 executions were successful.

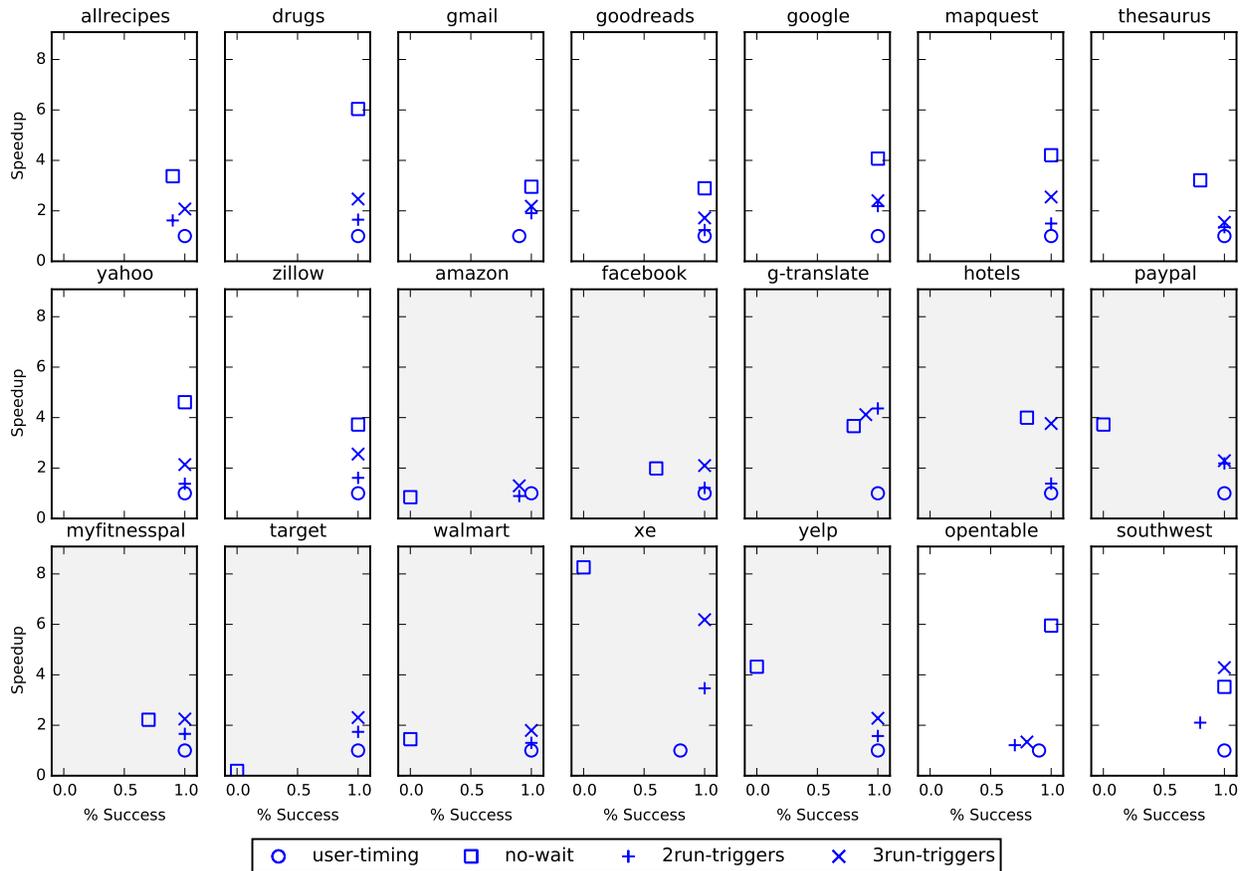


Figure 5.6.1. Accuracy vs. speedup on the suite of benchmark websites. Benchmarks with shaded backgrounds are ones which require synchronization.

often than the **no-wait** version.

For two benchmarks, the **no-wait** version passed all executions while the trigger versions failed at least one execution. We believe these websites break our assumption that the user can always wait longer without breaking the script.

Understanding Synchronization We now analyze a subset of the ten benchmarks which require triggers to illustrate how modern websites use synchronization and why naive synchronization techniques are fragile.

- **paypal:** If a replayer dispatches a click early, then it identifies the wrong element and clicks the wrong link, leading the replay to navigate to an unknown page.
- **google translate:** If a replayer does not wait, the text of the translation is scraped before it can be updated, thereby capturing the translation of a different word.

- **yelp**: Without enough delay, a replayer clicks one of the filter options while a previous filter is being applied. Additionally, a replayer can scrape off the list of restaurants before the page has a chance to fully apply a filter.
- **amazon**: After clicking a different version of an item, Ringer can scrape off the price of of the previous version of the time if it does not wait for the page to update.
- **target**: If a replayer does not wait for the page to fully load, then the page freezes after clicking on the button. This is most likely due to a JavaScript program not being fully loaded
- **facebook**: The wrong link will be clicked if a replayer does not wait for the page to finish navigation.

5.7 Conclusion

Triggers are essential to replay modern webpages correctly. Without some type of synchronization, the script can execute actions prematurely, causing the replay execution to diverge from the recording. But unlike actions and elements, understanding what signals the user uses is not obvious. In fact, the space of visual cues is large, making the inference of visual triggers challenging. Instead, we use triggers based upon HTTP responses. We show that these triggers are sufficient by running an experiment on a large number of real-world websites.

Chapter 6

Faithful Actions

This chapter discusses how Ringer ensures that it replays actions faithfully. Informally, an action is faithfully replayed if it has the same effect on the page as the corresponding user action observed during the recording (we provide a more formal definition in Section 4.2). To achieve this goal, actions are both recorded and replayed at the same level, as DOM events.

But even this strategy has difficulties which arise due to the way the browser handles DOM events raised through JavaScript. In particular, the browser distinguishes between DOM events raised by the user interacting with the page versus those raised programmatically through JavaScript. The former are distinguished as *trusted events*. When the browser handles a trusted event, it calls a predefined, *i.e.* page-independent, default action. The execution of this default action may cause side-effects which do not occur for non-trusted events.

Since Ringer replays DOM events through JavaScript, which are treated as non-trusted events, it needs a way to faithfully simulate the side-effects caused by the default actions. To do so, we implemented a runtime system which monitors the replay execution, detects when side-effects do not occur and compensates for the non-trusted events.

This chapter is organized as follows. We first go over background material (Section 6.1), illustrating the challenges of replaying at the level of DOM events. Next we discuss Ringer's approach at a high level (Section 6.2) and the record and replay algorithm (Section 6.3). We then evaluate our approach on a set of real websites by selectively disabling features (Section 6.4), demonstrating that all parts of our runtime system are necessary to ensure that Ringer replays actions faithfully.

6.1 Challenges with the DOM Event Architecture

The DOM event architecture is quite complicated. To ensure standard behavior across different browsers, the W3C wrote a natural language specification for its behavior[36]. We go over snippets from this specification to explain the challenges we encountered when designing Ringer's actions.

Default actions and trusted events One challenge of replaying DOM events is mimicking default actions. According to the specification,

Events are typically dispatched by the implementation as a result of a user action, in response to the completion of a task, or to signal progress during asynchronous activity (such as a network request). Some events can be used to control the behavior that the implementation MAY take next (or undo an action that the implementation already took). Events in this category are said to be cancelable and the behavior they cancel is called their default action.

Default actions occur in response to a DOM event and affect the state of the page. For example, if a click occurs on a input node, then the checked attribute of that node is set to true. Other events may have more complicated default actions. The default action for a click may need to setup a state machine in order highlight text or drag an image, depending on the current selected element. On the surface, it would seem that default actions should not be difficult to replay, since they occur in response to DOM events. But not all DOM events are treated equally.

Events that are generated by the user agent, either as a result of user interaction, or as a direct result of changes to the DOM, are trusted by the user agent with privileges that are not afforded to events generated by script through the `DocumentEvent.createEvent("Event")` method, modified using the `Event.initEvent()` method, or dispatched via the `EventTarget.dispatchEvent()` method.

Most untrusted events SHOULD NOT trigger default actions, with the exception of the click event.

In fact, events dispatched programmatically are treated differently from those generated from user actions. They are marked as not trusted and therefore do not execute default actions. This presents a challenge to Ringer, which replays DOM events through JavaScript. If the default actions of these events are not triggered, then the replayed page will diverge from the recording and the original actions are not faithful.

To account for default actions which are not executed during the replay, Ringer monitors changes on the elements involved in the demonstration and records how they change. Ringer assumes a default action is the root cause of these changes. If Ringer fails to see the change during the replay execution, it simulates the side-effects of the default action by applying the change to the element.

Synchronous and asynchronous events The other challenge is preserving the order and grouping of DOM events during replay. According to the specification,

Events MAY be dispatched either synchronously or asynchronously.

Events which are synchronous ("sync events") **MUST** be treated as if they are in a virtual queue in a first-in-first-out model, ordered by sequence of temporal occurrence with respect to other events, to changes in the DOM, and to user interaction. Each event in this virtual queue **MUST** be delayed until the previous event has completed its propagation behavior, or been canceled. Some sync events are driven by a specific device or process, such as mouse button events. These events are governed by the event order algorithms defined for that set of events, and a user agent **MUST** dispatch these events in the defined order.

Events which are asynchronous ("async events") **MAY** be dispatched as the results of the action are completed, with no relation to other events, to other changes in the DOM, nor to user interaction.

During the demonstration phase, we can listen for and record the sequence of DOM events dispatched on the page. But some of these DOM events are dispatched as a sequence specified by an event order algorithm. For example, when a user clicks on an element, it triggers a sequence of mousedown, mouseup, click events. These sequences present two challenges. First is that the browser executes some of these sequences atomically, preventing unrelated code from executing during the sequence. If Ringer does not detect these atomic sequences, then page-level code which assumes these atomic sequences will misbehave. The second challenge is that dispatching some of the events in the sequence will cause the browser to dispatch other events automatically. A naive replay strategy may dispatch an event twice, once automatically as a response to a previous event and once by the replayer itself. If the replayer does not handle this case, then a side-effect may occur twice causing the replay-time page to misbehave.

6.2 Ringer Approach

Before going into Ringer's approach to actions, we going over an example which illustrates the challenges and our solutions. The example is the same Southwest.com scenario presented in Chapter 4.

Running Example

The example follows a user as she searches for a flight on Southwest.com. To find a plane ticket on Southwest Airlines' website, a user enters departure and arrival cities, and the dates of travel. As shown in Figure 6.2.1, an autocomplete menu appears after the user types the first three characters of a city name. To select an airport, the user clicks on one of the suggested options. We will focus on this part of the interaction and why its difficult to replay.

As the user types into the text box, an event sequence of five DOM events is produced: keydown, keypress, textInput, input and keyup. The first challenge is that the textInput event has a

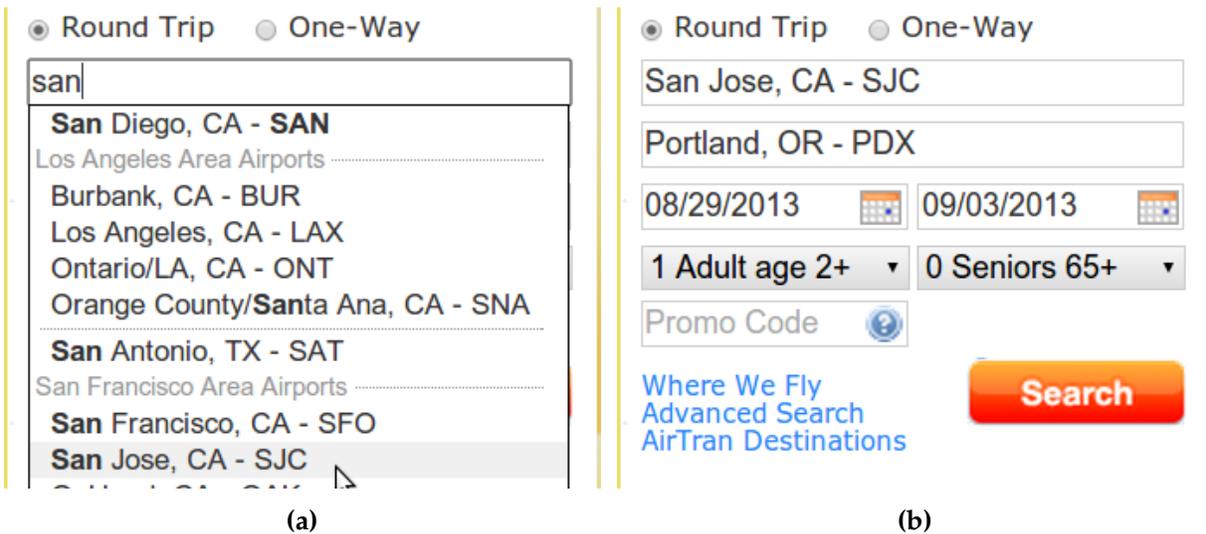


Figure 6.2.1. On Southwest.com, the user interacts with a custom autocomplete widget to select airports. After typing in three letters, the autocomplete menu appears (a). If the replayer does not mimic the default handler or dispatch event sequences atomically, then the pull down menu will not appear causing the script to break. Alternatively, a faithful replay of actions does enable the pull down menu allowing the replayer to repeat the user demonstration (b).

default action which updates the text box during the demonstration by appending the letter that the user typed. Since the replayed event is not trusted, a naive replay strategy would not update the text box leaving it blank. To compensate for untrusted events, Ringer snapshots each element before and after the browser dispatches a DOM event on it. Using these snapshots, Ringer records a delta which captures how the default action modifies the element. In this case, Ringer records that the letter 's' is appended to the text content. During replay, Ringer sees that the browser does not perform the default action, and therefore simulates the effects of the action by appending 's' onto the text of the replay-time text box.

Another issue is that programmatically dispatching certain events triggers the browser to dispatch other events, *i.e.* causing a cascade of events. On Southwest.com, when Ringer replays the `textInput` event, the browser automatically dispatches an `input` event. If Ringer replayed the `input` event again, then the `input` callbacks would execute twice. On Southwest.com, executing these callbacks twice is benign, but its simple to construct a scenario which is not. Ringer handles these cascading events by monitoring the replay execution and matching the replay-time events with the recorded script. If Ringer observes that the browser already replayed an event, then it simply skips its and moves onto the next unmatched event.

The last challenge is subtle, but important for correctness. In particular, it deals with the order in which events and their callbacks are executed. During the user demonstration, the browser dispatches the `keydown`, `keypress`, `textInput` and `input` events atomically. Since JavaScript is single-threaded, this means that the browser executes all callbacks associated with these events

keydown	Execute callbacks	keydown	Execute callbacks
	Callback adds set Timeout		Callback adds set Timeout
keypress	Execute callbacks	setTimeout	Call code for autocomplete menu (on 'sa')
textInput	Execute callbacks	keypress	Execute callbacks
	Default action adds letter to the text box ('san')	textInput	Execute callbacks
input	Execute callbacks		Default action adds letter to the text box ('san')
setTimeout	Call code for autocomplete menu (on 'san')	input	Execute callbacks
keyup	Execute callbacks	keyup	Execute callbacks

(a) Atomic

(b) Naive

Figure 6.2.2. Each time a user types a letter into the textbox shown in Figure 6.2.1a, it triggers five DOM events. The browser dispatches four of these events, `keydown`, `keypress`, `textInput` and `input`, atomically, as shown in (a). A naive replayer though allows arbitrary interleavings of unrelated callbacks (b), leading to the situation where the function which adds the autocomplete menu executes before the last character is appended to the text box.

before executing any other callbacks, such as ones associated with a `setTimeout` or other events. For many pages, whether this sequence is played atomically or not does not affect the outcome of the replay. But for the Southwest.com scenario, there exists observable side-effects if replay is done naively. Specifically, during a naive replay, the page does not register the last character typed by the user. If the user originally typed 'san', then during the replay, the page would only register 'sa'. This causes the replay execution to fail since the autocomplete menu only appears after three characters.

Figure 6.2.2 shows the ordering of callbacks during the demonstration. The key point is the ordering between the default action which adds a letter to the text box and the timeout which executes the code for the autocomplete menu. During a correct execution (a), the letter is added before the timeout. In a naive replay (b), the browser can interleave the timeout between the event sequence, leading the page to register $n - 1$ characters even though the user typed n characters. Ringer prevents this behavior by monitoring which actions are atomic and ensuring that these actions are replayed atomically during replay.

Challenges and Approach

We now go over how Ringer handles actions and the challenges they present.

Observing and controlling actions Before the page loads at record time, Ringer adds event listeners for all important DOM events. These listeners are attached for the bubble phase of the event cycle to ensure that Ringer's callback executes before all other callbacks. Since callbacks

have the power to prevent all later callbacks from running, our callback must run first in order to ensure it will be executed. Ringer's callbacks record each event into a trace of events. At replay time, Ringer clones the observed events and uses the DOM's `dispatchEvent` function to raise them.

Default handlers We considered many possible static approaches to imitating the effects of default handlers on DOM state. Handler effects can be quite complex; the effects of a keypress on the text of a text box depend on the location of the cursor, whether any text is selected, what type of the key (such as the delete key, an arrow key, a character), etc. Also, a page's JavaScript event handlers can cancel the default handlers, so even a complete JavaScript version of all default handlers would produce the wrong effects when the default handler should not run. In short, the difficulty of predicting the effects in any given context leads us to prefer a dynamic approach.

Thus, along with information about the event, Ringer must record information about changes to the target node since the last event, changes that can be attributed to the default handlers. Ringer compares snapshots from before and after each event. The snapshot is limited to only the target element, rather than the entirety of the DOM, on the assumption that default handlers affect only the target node. Ringer identifies any properties of the last target node that have changed between two snapshots, and records their new values. For instance, if the event is a click on an option in a select menu, we would find that the `value` and `selectedIndex` had changed. Ringer then runs the same analysis during replay. We manually set the values of any properties that fail to update in the same way during replay as they did during recording.

Replaying cascading events exactly once Because each record-time event must be replayed exactly once, Ringer must check whether an event has already been sent, before recreating and dispatching the event itself. This prevents executions in which the browser's default handler or a page's JavaScript code raises an event at replay time, only to have the tool raise the event a second time.

Replaying cascading events atomically During normal browser execution and thus during record, cascading events are executed atomically. Recall that JavaScript is a single threaded, non-preemptive language. Two events e_1 and e_2 are executed atomically if all callbacks listening for e_1 are executed before all callbacks listening for e_2 , without no unrelated callbacks executed between them.

During record, Ringer must identify sets of events that are executed atomically, so that it can run them atomically during replay. We accomplish this by setting a zero timeout during the callback for each event. This is the JavaScript convention of queuing up a function. As soon as any atomic block is completed, the JavaScript interpreter draws from this queue. By saving

the length of the trace when the queued up function runs, we can identify exactly which events run before the function that corresponds to each event.

6.3 Record and Replay Algorithm

We present pseudocode for our record and replay algorithms. We abstract away code which deals with synthesizing element and trigger expressions. These pieces are orthogonal to replaying actions and are discussed in other chapters.

Our discussion also leaves out the process of mapping record-time pages to replay-time pages, proceeding as though each interaction occurs on a single page, with only a single frame. Our implementation adds a layer that handles this mapping, which makes the algorithm applicable to interactions on multiple pages and pages with multiple frames. In general, this mapping is non-trivial since URLs often contain session-specific text that will vary across executions.

Core Algorithm

During the record phase, Ringer creates a trace of the events that are dispatched on the page, as shown in Listing 6.3.1. For example, Figure 6.3.1 is a small slice of the trace that Ringer records during the Southwest.com autocomplete interaction.

To collect such traces, Ringer must be able to detect when events are raised. Before the page loads at record time, Ringer adds event listeners for all important events. These listeners are

```
1  var trace = []
2  function record(e) {
3    addDeltaInfo(e.target);
4    var r = {}
5    for (var prop in e) {
6      var type = typeof(e[prop])
7      if (type == 'number' or 'boolean' or 'string')
8        r[prop] = e[prop]
9      else if (prop == 'target' || prop == 'relatedTarget')
10         r[prop] = saveTarget(e[prop])
11    }
12    trace.push(r)
13    addAtomicEventInfo();
14  }
15  var evts = [ 'focus', 'click', ... ]
16  for (var i = 0; i < evts.length; ++i)
17    document.addEventListener(evts[i], record, true)
```

Listing 6.3.1. Record algorithm: Capturing event information

Type	Target	Notes
mousedown	<input type="text" value="Departure City or Airport Code"/>	User clicks the departure city text box
focus	<input type="text" value="Departure City or Airport Code"/>	
mouseup	<input type="text" value="Departure City or Airport Code"/>	
click	<input type="text" value="Departure City or Airport Code"/>	
keydown	<input type="text" value="Departure City or Airport Code"/>	User types 's' into the text box
keypress	<input type="text" value="Departure City or Airport Code"/>	
textInput	<input type="text" value="s"/>	
input	<input type="text" value="s"/>	
keyup	<input type="text" value="s"/>	
...	<input type="text" value="san jose"/>	Repeat for each letter in 'an jose'
mouseover	<input type="text" value="San Jose, CA - SJC"/>	User selects from pull-down menu
mousedown	<input type="text" value="San Jose, CA - SJC"/>	
blur	<input type="text" value="San Jose, CA - SJC"/>	
focus	<input type="text" value="Arrival City or Airport Code"/>	
mouseup	<input type="text" value="Arrival City or Airport Code"/>	
keydown	<input type="text" value="Arrival City or Airport Code"/>	User begins process to select arrival city
keypress	<input type="text" value="Arrival City or Airport Code"/>	
textInput	<input type="text" value="pl"/>	
input	<input type="text" value="pl"/>	

Figure 6.3.1. This trace shows a selection of the events raised when a user chooses a destination using Southwest.com's custom widget.

attached for the bubble phase of the event cycle to ensure that Ringer's callback is executed before all other callbacks. Since event callbacks have the power to prevent all later callbacks from running, our callback must run first in order to ensure it will be executed.

For each event in the trace, Ringer must store enough information to recreate the event. When an event is dispatched, the browser passes the event object to the callbacks. As shown in Listing 6.3.1, lines 7 and 8, Ringer's callback retrieves and saves details from the event object, such as the event type and a timestamp.

During replay, Ringer uses the information recorded about each event to reconstruct the event object, as shown on lines 18 and 19 of Listing 6.3.2. Each event is dispatched using the DOM's `dispatchEvent` function, as shown on line 20.

Listings 6.3.1 and 6.3.2 represent the core of our algorithm, the process of recording event information and the process of recreating corresponding events from that information. In what follows, we ask when this elegant approach can be expected to succeed, and introduce the other components of our algorithm, which we use to expand to the class of replayable interactions.

```

1  var index = 0, trace = getTrace();
2  function replay() {
3    if (ready()) { simulateAtomicEvents() }

5    setTimeout(function() {
6      replay()
7    }, getWaitTime())
8  }
9  function simulateAtomicEvents() {
10   var end = trace[index].endIndex
11   for (; index <= end; ++index) {
12     var rec = trace[index]
13     if (checkReplayed(record)) { continue }

15     var target = getTarget(record.target)
16     if (!target) { return }

18     var e = document.createEvent(...)
19     e.initEvent(rec.type, rec.bubbles, ...)
20     target.dispatchEvent(e)
21   } }

```

Listing 6.3.2. Replay algorithm: Outer loop and dispatching events

Elements and Triggers

We abstract away parts of the R+R algorithm that deal with elements and triggers. Lines 9 and 10 of Listing 6.3.1 contain a call to the `saveTarget` function, which records information about the element. During replay, the `getTarget` function of line 15 in Listing 6.3.2 must find a corresponding node. In Listing 6.3.2, the functions `getWaitTime` and `ready` stand in for trigger expressions which decide when the replayer should dispatch an event. We discuss both these concepts in further detail in Chapter 5 (triggers) and Chapter 7 (elements).

Default Actions

To implement our dynamic approach to handle default handlers, when recording information about the event, Ringer must also record information about changes to the target node since the last event. The changes can be attributed to the default handlers. We call each property change a *delta*, and identify them with the `addDeltaInfo` function, the last unexplained line of Listing 6.3.1.

The `addDeltaInfo` function compares snapshots from before and after the event. We limit the snapshot to only the target element, rather than the entirety of the DOM, on the assumption that default handlers affect only the target node. Ringer identifies any properties of the last target

```
1  var lastSnapshot = null, lastTarget = null

3  function addDeltaInfo(target) {
4    var record = trace.last()
5    if (lastSnapshot)
6      record.delta = compare(lastSnapshot, getSnapshot(lastTarget))

8    lastSnapshot = getSnapshot(target)
9  }
```

Listing 6.3.3. Record algorithm: Capturing event deltas

```
1  var recordDelta = null
2  function replayRecord(e) {
3    var r = {}
4    ... // save event info
5    replayTrace.push(r)

7    var replayDelta = getDelta(e.target)
8    if (recordDelta)
9      updateDom(recordDelta, replayDelta)

11   recordDelta = getMatchingEvent(r).delta
12 }

14 for (var i = 0; i < evts.length; ++i)
15   document.addEventListener(evts[i], replayRecord, true)
```

Listing 6.3.4. Replay algorithm: Capturing event deltas

node that have changed between two snapshots, and records their new values. For instance, if the event is a click on an option in a select menu, compare would find that the value and selectedIndex properties had changed.

To ensure that default handlers have the correct effects at replay time, we must track replay-time deltas. Listing 6.3.4 shows how we attach event listeners and calculate deltas, exactly as we did at record-time. In fact it is crucial that the delta be calculated at the same point in the event's processing.

At replay-time, line 9 of Listing 6.3.4 calls an updateDom function in Listing 6.3.5, which compares the record-time and replay-time delta sets. Deltas are changes to the properties of the target, so two deltas match if they change the same property. If Ringer finds a delta that occurred during record but not during replay, updateDom executes it, setting the property to the record-time, post-event value.

```

1 function updateDom(recDelta, replayDelta, target) {
2   for (delta in recDelta)
3     if (delta not in replayDelta)
4       apply(delta, target)
5
6   for (delta in replayDelta)
7     if (delta not in recDelta)
8       revert(delta, target)

```

Listing 6.3.5. Replay algorithm: Compensating for deltas

```

1 function updateAtomicEvent() {
2   var record = trace.last();
3   setTimeout(function() {
4     record.endIndex = trace.length - 1;
5   }, 0) }

```

Listing 6.3.6. Record algorithm: Identifying atomic cascading events

Replaying Cascading Events Exactly Once

Because each record-time event must be replayed exactly once, line 13 of Listing 6.3.2 first checks whether an event has already been sent, before recreating and dispatching the event itself. This prevents executions in which the browser’s default handler or a page’s JavaScript raises an event at replay time, only to have the tool raise the event a second time.

As described above, the same event recording code is run during replay as is run during record. Thus, the tool can maintain a trace of replay-time events, as shown in Listing 6.3.4, creating a `replayTrace`. To check if the browser already replayed an event, `checkReplayed` traverses the `replayTrace`, attempting to find a matching event. If `checkReplayed` succeeds in finding a matching event, Listing 6.3.2 shows that the tool continues to the next event.

Replaying Events Atomically

During record, we must identify sets of events that are executed atomically, so that we can ensure they are run atomically during replay. We accomplish this by setting a zero timeout during the callback for each event. This is the idiomatic JavaScript way of queuing up a function to run as soon as the current computation completes. As soon as the atomic block completes, the JavaScript interpreter will execute this callback. By saving the length of the trace when the queued up function runs, we can identify exactly which DOM events run before the timeout function.

Note that the `simulateEvents` function defined in Listing 6.3.2 replays not one, but a sequence of

events. The `simulateEvents` function dispatches all the events in this sequence before relinquishing control. Because JavaScript is single threaded and non-preemptive, this ensures that any queued functions will be correctly interleaved with replay-time events.

6.4 Evaluation

In this section, we present an evaluation of Ringer’s approach for actions on a wide variety of real websites. We disable each runtime feature in turn, and check whether the modified tool can successfully replay each benchmark. We find that while not all websites require all features, at least one of the websites requires each feature of the runtime system, validating our design decisions.

Empirical Evaluation

Our evaluation procedure is similar to that of Section 4.4. To empirically evaluate our approach, we recorded a set of interactions with real world websites using Ringer. Our benchmark suite includes scenarios from 29 distinct websites. All benchmark websites were taken from Alexa’s list of most-visited sites by category[1]. Besides being likely targets for user scripts, these sites also tend to be complex, making heavy use of AJAX requests, custom event handlers, and the other features that make record and replay difficult. In short, they offer ideal stress tests. We selected from multiple Alexa categories to ensure we tested on a variety of interaction types.

For each interaction, we completed what we perceived to be a core site task, such as buying a product. For pages with custom widgets or complicated user interactions, we made a point of completing those interactions as part of the pages’ benchmarks.

We checked user-defined invariants to determine whether each replay succeeded or failed. Invariants checked for the presence of a particular string in a particular target node. For instance, for the Walmart benchmark to succeed, the string "This item has been added to your cart." must appear on the last page of the interaction.

For each benchmark, we recorded the interaction, including invariants. We then replayed the interaction, marking the execution successful if the tool dispatched all events and all invariants passed.

Limitations Our evaluation procedure does not stringently test Ringer’s target identification algorithm, since our approach is oblivious to the algorithm that is used to find target nodes. Thus, we clear browser cookies before all recorded or replayed interactions (to avoid page changes that result from multiple visits) and replay immediately after recording (to minimize although not eliminate page changes that result from server-side modifications).

Site	Description	All	Compensation	Cascading	Atomic
amazon	add camera to the cart	✓	✗	✗	✓
bestbuy	add camera to the cart	✓	✓	✓	✓
bloomberg	get Google stock info	✓	✗	✓	✓
booking	book a hotel in city A	✓	✗	✗	✗
ebay	place bid on a phone	✓	✓	✓	✓
expedia	book flight from A to B	✓	✓	✓	✓
facebook	search, navigate to friend's page	✓	✓	✗	✓
facebook	create event and add friends	✓	✓	✗	✓
facebook	click links to find phone number	✓	✓	✓	✓
gmail	add and remove stars from emails	✓	✓	✗	✓
google	get directions from A to B	✓	✓	✓	✓
google	search for "PLDI 2014"	✓	✓	✓	✓
google	translate "hello" into Hindi	✓	✓	✓	✓
kayak	book flight from A to B	✓	✓	✓	✓
linkedin	find connections from school A	✓	✓	✓	✓
mapquest	get directions from A to B	✓	✓	✓	✓
myfitnesspal	calculate calcs burned swimming	✓	✗	✓	✓
paypal	send money to a relative	✓	✗	✓	✓
southwest	book flight from A to B	✓	✓	✓	✗
tripadvisor	find vacation rentals in city A	✓	✗	✓	✓
twitter	send a tweet	✓	✓	✗	✓
walmart	buy Kit Kats	✓	✓	✓	✓
xe	convert 100 INR to BDT	✓	✓	✗	✓
yelp	find restaurants in city A, filter	✗	✗	✗	✗
youtube	check stats for "Gangnam Style"	✓	✓	✗	✓

Table 6.4.1. We list the results of running Ringer with variations of the action replay algorithm on a set of benchmarks. **All** shows results for the unmodified tool. **Compensation**, **Cascading**, and **Atomic** show results when the following features are individually turned off: DOM state compensations, cascading event detection, and cascading event atomicity. The table shows that each feature is necessary for Ringer to replay DOM events faithfully.

Discussion

Table 6.4.1 gives the results of our evaluation. We now discuss the results of each feature of the action algorithm.

DOM Updates When the effects of default handlers are not mimicked, 6 benchmarks break (24%), as shown in the column labeled **Compensation**. For most of these benchmarks, the browser failed to update a textbox in response to a keypress event, or added a character at the wrong index. In another case, a click is dispatched to an item in a dropdown menu, but the browser does not mark that item as selected.

Replaying cascading events exactly once When cascading events are not run exactly once, 8 benchmarks break (32%), as shown in the column labeled **Cascading**. Many benchmarks break because the tool attempts to replay an event after the browser navigates away from the target node's page. For example, clicking a submit button causes click and submit events. During replay, the tool dispatches the click event, causing the browser to submit the form and load a new page. On the new page, no appropriate target node exists for submit, so Ringer deadlocks. In addition, we found that executing an event's callback twice (by dispatching the event twice) can cause the script to fail. On one example, incorrectly repeating a cascading event twice causes a pulldown menu to not appear.

Replaying Cascading Events Atomically When cascading event atomicity is not enforced, only 2 benchmarks break (8%), as shown in the column labeled **Atomic**. These two benchmarks have custom pulldown menus which are updated as a user types. On the Southwest benchmark, the menu is updated by a function that is executed after a set of cascading events. The function is queued up by the keypress event handler, but retrieves text from the textbox that is updated by the later textInput event.

Second Benchmark Experiments

We ran the same set of experiments on a second set of benchmark scenarios. Results of these experiments are shown in Table 6.4.2. We did not have an explicit reason to run these additional experiments, but ran them since they required little extra effort when running the second set of experiments in Section 4.4.

We obtain similar conclusions from these benchmarks. Many benchmarks did not require any of our runtime features, but at least one benchmark required each of the features. One point to note is that of the 17 shared scenarios between the first and second set of benchmarks, 9 required a different set of run time features. This is one indication that the implementation of a website is constantly changing, motivating abstractions that work at the same level as the user.

Site	Description	All	Compensation	Cascading	Atomic
allrecipes	find recipe and scale it	✓	✗	✓	✓
amazon	find price of camera	✓	✓	✓	✓
best buy	find price of camera	✓	✓	✓	✓
bloomberg	find price of Google stock	✓	✓	✓	✓
drugs	find side effects of Tylenol	✓	✓	✓	✓
facebook	find friends phone number	✓	✓	✗	✓
gmail	compose and send email	✓	✓	✓	✓
goodreads	find books related to Infinite Jest	✓	✓	✓	✓
google	search for "Berkeley"	✓	✓	✓	✓
google maps	find estimated time of drive	✓	✓	✓	✓
google translate	translate "hello" into French	✓	✓	✓	✓
hotels	book a hotel	✓	✓	✓	✓
kayak	book a flight	✓	✓	✓	✗
mapquest	find estimated time of drive	✓	✓	✓	✓
myfitnesspal	calculate calcs burned swimming	✓	✗	✓	✓
opentable	make a reservation	✓	✓	✓	✓
paypal	transfer funds to friend	✓	✗	✗	✓
southwest	book a flight	✓	✓	✗	✓
tripadvisor	book a hotel	✓	✓	✓	✓
target	buy Kit Kats	✓	✓	✓	✓
thesaurus	find synonyms of "good"	✓	✓	✓	✓
walmart	buy Kit Kats	✓	✓	✓	✓
xe	convert 100 INR to BDT	✓	✓	✓	✓
yahoo finance	find price of Google stock	✓	✓	✓	✓
yelp	find restaurants in Berkeley	✓	✗	✗	✓
youtube	find stats for video	✓	✓	✗	✗

Table 6.4.2. We list the results of running Ringer with variations of the action replay algorithm on a second set of benchmarks. **All** shows results for the unmodified tool. **Compensation**, **Cascading**, and **Atomic** show results when the following features are individually turned off: DOM state compensations, cascading event detection, and cascading event atomicity. The table shows that each feature is necessary for Ringer to replay DOM events faithfully.

6.5 Conclusion

DOM events are an ideal abstraction for actions since they can be both observed and mimicked, but they also have their challenges. Most of these challenges stem from the browser treating DOM events raised through JavaScript differently from those raised natively through the user's actions. To compensate for these differences, we developed a runtime system which monitors the executions and updates the page appropriately. We show through an empirical study that our system is sufficient to replay complicated webpages and additionally show that all parts of the runtime system are necessary to replay modern webpages.

Chapter 7

Element Identification¹

Identifying elements across multiple executions is essential to successfully replaying Ringer scripts. During the recording, the user performs each action on a particular element, such as clicking a submit button. During replay, we need to find a matching element to dispatch the action upon. Choosing an incorrect element, such as the cancel button, can cause the script to fail. Even worse though, choosing an incorrect element can cause unpredictable side-effects on the server-side state of the webpage, such as clicking a delete button.

The challenge with finding the correct element is that the webpage may arbitrarily change between executions, making it difficult to synthesize an expression which robustly identifies the element. Our approach to handle these changes is to use a similarity metric that compares elements on the replay-time page with the element that the user interacted with during the recording. This approach allows the page to change arbitrarily, while still choosing elements which have a high likelihood of being the correct element.

This chapter is organized as follows. We go over problem of element identification in Section 7.1 and discuss past approaches in Section 7.2. We then discuss how our approach fundamentally differs from these methods in Section 7.3. We present our algorithm for the similarity metric in Section 7.4 and evaluate it in Section 7.6.

7.1 Problem Statement

The browser represents the webpage as a document with a tree structure, called the DOM tree, with nodes in the tree representing elements in the document. Each node has many properties which affect how the browser renders the node on the page. These properties range from semantic information such as text content, to style properties such as background images or text color. There are some properties, such as id or class, which do not affect the visual appearance, but are used by the page's JavaScript program to identify the element. The webpage

¹This chapter is based upon work mainly done by Sarah Chasins. We include this chapter for completeness.

can programmatically interact with these nodes through its JavaScript program, observing properties of the nodes and modifying them.

During the demonstration, we can observe which elements the user performs each action upon. At this time, we can also record properties of the element, such as the text content or id, and the document tree, such as the path between the element and the root of the tree. During replay, we use these recorded values to find an element which matches the original element so that we can replay the action on the webpage.

More formally, at record time t_1 , we load a URL u which yields a DOM tree T . Let the user select a given node in T with m attributes $n = \langle a_1, \dots, a_m \rangle$. At a later time t_2 , we load u again yielding DOM tree T' . The goal is to identify the node $n' \in T'$ that a user would identify as corresponding to $n \in T$.

This problem is difficult because T' may be arbitrarily different from T . For example, the page may push a new layout causing the structure of T change, or n may display constantly changing information, causing its properties to change. Even if n remains constant, the site may modify or alter nodes around it, such as when the page displays a different advertisement. In the limit, T' could have a completely different structure than T with no node that shares any features with n .

If drastic changes happened regularly on real webpages, robust record and replay would be near impossible. However, these changes would also make it difficult for human users to use the website, since they would need to carefully reexamine the website for each visit. We assume that the need to offer a consistent user interface motivates website developers to keep the user-facing interface stable. Therefore, in practice there is often substantial similarity between the DOM trees of the same page accessed multiple times.

7.2 Past Approaches

We now go over past approaches of identifying elements. We use the scenario from Chapter 3 as a running example. In the example, a user wants to scrape the price of a silver camera off of Amazon.com. To do so, he navigates to the camera page on Amazon.com (Figure 7.2.1), clicks the silver camera button and waits for the price information to load. He scrapes the price once the page loads the new information.

Sloppy programming

Previous approaches asked users to write *sloppy programs* which could then be executed. CoScripter[21], an end-user record and replay tool built to automate business processes, allows users to write a high-level natural language description which it then executes. But such programs are fragile and ambiguous, making them unreliable. In fact, we attempted to write a sloppy program in CoScripter but failed to produce a robust program. Our first attempt used

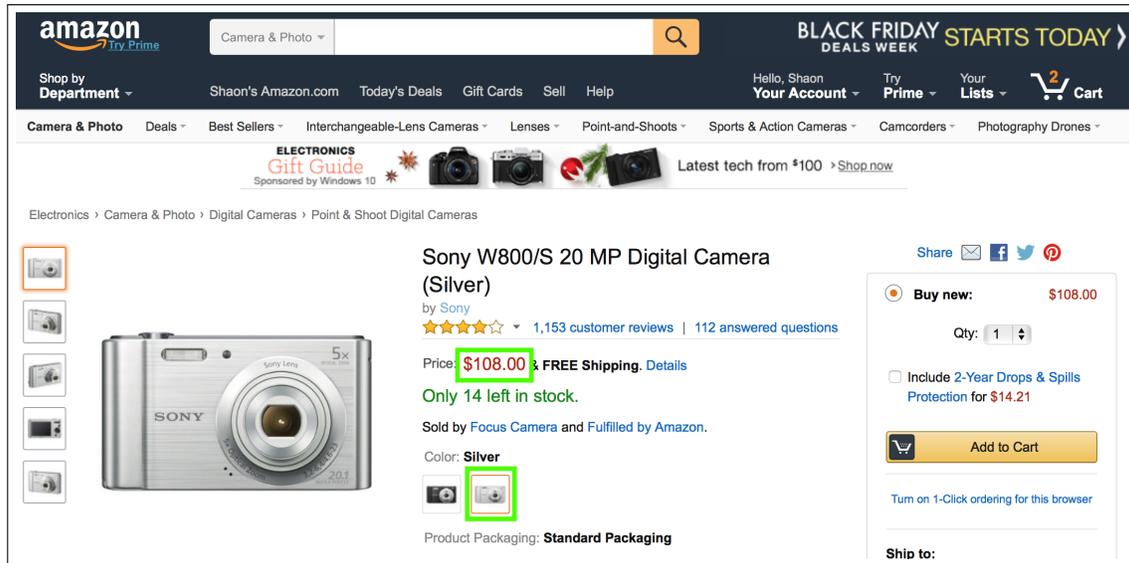


Figure 7.2.1. Screenshot of Amazon.com page for purchasing a camera. Elements involved in the motivating example are highlighted in green.

```

go to "http://www.amazon.com/"
enter "sony w800" into the "Go" textbox
click the "Go" button
click the "Sony W800/B 20.1 MP Digital Camera (Black)" link
click the "$108.00" button
clip the x"//*[@id='priceblock_ourprice']"
clip the x"//*[@id='title']"
  
```

Figure 7.2.2. CoScripter recording of user's interaction with Amazon.com

CoScripter's record feature which creates a sloppy program from a demonstration, producing the script in Figure 7.2.2.

The program is almost correct, except for clicking the silver button. Instead of synthesizing "click the silver button", CoScripter generates "click the second "\$179.00" button", a statement which will break once the price changes. The primary reason for this issue is that CoScripter cannot bridge the semantic gap from the button which contains an image of a silver button to "silver button". In fact, we could not write any expression which robustly identified this button.

The difficulty writing a natural language expression for this button arises because there is no textual marker located near the button in the DOM tree, unless you hover over the button. In general, the user may not be able to precisely define their behavior in natural language. Such sloppy programming methods are bound to have unresolved ambiguities due to the large abstraction gap between an end-user's understanding of the page, and the actual implementation of the page (*i.e.* the HTML and DOM tree).

```
1 def setUp(self):
2     self.driver = webdriver.Firefox()
3     self.driver.implicitly_wait(30)
4     self.base_url = "http://www.amazon.com/"
5     self.verificationErrors = []
6     self.accept_next_alert = True

8 def test_amazon(self):
9     driver = self.driver
10    driver.get(self.base_url + "/ref=nav_logo")
11    driver.find_element_by_id("twotabsearchtextbox").clear()
12    driver.find_element_by_id("twotabsearchtextbox").send_keys("sony w800")
13    driver.find_element_by_css_selector("input.nav-submit-input").click()
14    driver.find_element_by_xpath(
15        "//li[@id='result_0']/div/div/div/div[2]/div/a/h2").click()
16    driver.find_element_by_id("a-autoid-10-announce").click()
17    title = driver.find_element_by_id("productTitle").text
18    cost = driver.find_element_by_id("priceblock_ourprice").text
```

Figure 7.2.3. Selenium IDE recording of user’s interaction with Amazon.com

Trying to synthesize a direct representation of how the user views the element (high level language) by only observing the page through the DOM API (low level events) is difficult due to the large abstraction gap. Instead, our approach views the user’s program as an unknown specification. We can observe a concrete trace of the specification but not the program itself. We use the specification to synthesize a correct program that uses a similarity metric, versus a fixed feature, that takes advantage of the large number of properties of each element.

Selenium

When designing Ringer, our first thought was to build upon an existing technology such as Selenium[34], which is widely used by developers to automate web pages. But we quickly realized such an approach would not be feasible. Specifically, writing a Selenium script requires concrete expressions to name DOM elements. But a single user demonstration provides limited information, creating ambiguity and making it extremely difficult if not impossible to synthesize robust expressions.

Selenium IDE Developers can use Selenium IDE to synthesize Selenium scripts by demonstrating the interaction they want to automate. Figure 7.2.3 shows the program synthesized by Selenium IDE for our running example. Unlike CoScripter, there is no ambiguity when

identifying elements on the page. At first, this script seems to fit our criteria, but we find that it is fragile and not suitable for end-users.

Each XPath expression represents an element on the page and is synthesized by Selenium IDE during the demonstration. While XPath expressions can precisely identify any element, these expressions are fragile. Slight modifications, such as a new div being added, can cause no element to be found. For example, adding a new div wrapper in the Amazon.com page will not visually impact the page, but will cause this script to fail. For developers, for whom Selenium is designed for, fixing these expressions or writing more robust expressions is feasible. But for end-users, manually fixing these expressions can be daunting. An ideal solution would have a precise way of identifying an element that matches the user's intuition and is robust to changes on the page.

Other Tools

To give a more complete view of current approaches, we briefly describe two other approaches for element identification.

iMacros First, we cover the target identification algorithm used by iMacros[5], a popular web automation tool which also uses record and replay. At a high level, iMacros records the text of an element n and the number of other nodes which precede n that have the same text attribute. This number becomes the position attribute. The combination of text and position uniquely identifies n in T . The text and position also identifies a single node in the replay page, which iMacros assumes is the intended element.

ATA-QV The ATA-QV algorithm [40] is more complicated. During the demonstration, it creates a label for the target element n . The label is usually the text content of the element, and if the element has no text it defaults to using the type of the node, *e.g.*, div, input, h1, h2, button, etc. If this label is unique on the page, then the algorithm finishes. If not, then algorithm distinguishes the element from the set of all matching elements with the same label by creating a set of anchors. An anchor is an element whose label occurs in the subtree rooted at n but does not occur in the subtrees rooted at the matching elements. The algorithm recursively accumulates a list of anchors, so that that it can uniquely identify n on T . At replay time, ATA-QV goes through the anchors in the reverse order, finding smaller subtrees until it finds the desired element. For ATA-QV, it uses text content around the element as the crucial attribute to distinguish the node.

7.3 Ringer Approach

Past approaches treated the element identification problem as a synthesis problem. The webpage T and the element n constitute a specification of the desired element. Given T and n , the algorithm synthesizes a function f such that $f(T) = n$. To find a corresponding node n' on a new page T' , the algorithm applies f , proposing that $f(T')$ is n' . As we have seen with CoScripter, Selenium, iMacros and ATA-QV, f uses some combination of the attributes of n , such as the position in the tree and text content. These attributes are chosen based upon the designer's insights into how webpages change. However, the problem with choosing a fixed set of properties is that it makes assumptions about how the webpage may evolve over time.

Rather than synthesize a function $f : T \rightarrow n$ during the demonstration that discards most of the information about n , our approach is to build a similarity function s for measuring how similar two nodes are. At record time, we save all attributes of the target element n . At replay time, for each node $n_i \in T'$, we run $similarity(n_i, n)$ and select the node with the highest score. In contrast to past approaches, a similarity function can use all features of the record-time element to find a matching element during replay. The similarity function can also handle changes in a continuous manner, meaning that it will always find some element. In contrast, other approaches, such as XPath, can either find a match or not find any element, there is no middle ground.

We constructed several similarity functions. All take the same basic form, which is weighted sum of matching attributes. For each attribute a , if $n.a = n_i.a$, the score is incremented by the weight. We consider the element with the highest score the best match. The naive similarity function weighs all features equally. We also used a training set of real websites to automatically learn a set of weights. One function used regression analysis to come up with the weights while another used a support vector machine (SVM).

7.4 Algorithm

Constructing the Training Data

To get the training data set, we first selected the thirty most popular sites, according to Alexa [1]. For each page, we recorded all attributes of all elements on the page. We also programmatically clicked on each element, and observed whether the page navigated to a new URL and if so, recorded the new URL. With such a large dataset, it was infeasible for us to manually label which elements are equivalent between accesses to the same URL. Instead, we approximate this label by observing whether or not clicking the element causes the page to navigate to the same URL. If so, then we mark the elements as equivalent. We do admit that this skews our similarity heuristic towards only those elements which responds to clicks, such as buttons or links.

We collected the data twice, a week apart. The initial run represents elements during the record phase. The final run represents elements during the replay phase. For each element

```

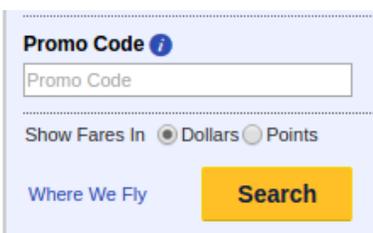
attributes[List[String]] = [...]
getTrainingData(initialNodes : Set[DOMNode], finalNodes : Set[DOMNode])
1  positive[Set[List[Boolean]] ← {}
2  negative[Set[List[Boolean]] ← {}
3  for n1 : DOMNode ← initialNodes do
4    for n2 : DOMNode ← finalNodes do
5      v : List[Boolean] = attributes.map(a → n1[a] == n2[a])
6      if n1.navigateURL == n2.navigateURL then
7        positive ← positive ∪ {v}
8      else
9        negative ← negative ∪ {v}
10 return (positive, negative)

```

Figure 7.4.1. Algorithm to construct training data used to assign weights to attributes.

which caused a page navigation in the initial run, we check whether any element in the final run caused a navigation to the same URL. If so, we mark the pair of elements as a matching pair. We construct a boolean feature vector for each pair of elements. If the two elements have the same value for a feature, then the corresponding index in the feature vector is true. We mark the vector as a positive example if the elements are a matching pair.

Figure 7.4.1 presents pseudocode of our algorithm to construct the training data. For each page, we construct all pairs of elements such that the first element e_1 is taken from the initial run and the second element e_2 is taken from the final run (lines 3- 4). For each pair, we construct a boolean vector v which represents the matching attributes between the two elements (line 5). If attribute i matches, *i.e.* $e_1[i] = e_2[i]$, then $v[i] = 1$, else $v[i] = 0$. If the pair of elements is a matching pair, then we label v as a positive example. Otherwise, it is a negative example (lines 6-9). Using this data set, we learn a weighting of attributes that maximizes the number of correctly labeled examples.



Property	Value
Path from root	HTML/BODY[1]/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
Text content	"Search"
Id label	"submitButton"
Class labels	"swa-button", "swa-button_primary"

Figure 7.4.2. Example element from Southwest.com. To start searching for the flights, the user must complete the form and click the "Search" button. The table lists properties of the "Search" button, which Ringer uses to collect an initial set of possible target elements on the replay-time page.

Finding Possible Elements

Since there may be thousands of elements on a page, it's infeasible to calculate the similarity score for every element. Instead, we do a first pass on the DOM tree to collect a set of possible elements. We then rank these elements using the similarity function, returning the element with the highest score. During the first pass, we find a set of candidate elements by running a set of XPath queries based upon properties of the original node. We go over these queries, illustrating how they work on an example element from the Southwest.com scenario from Chapter 4. The element is the "Search" button shown in Figure 7.4.2.

- **XPath suffix:** We start with the original XPath expression which identifies the element from the root node in the page. We then take progressively smaller suffixes of this expression until it finds at least one element. This method collects elements when the page relocates an entire subtree.

```
//HTML/BODY[1]/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
//BODY[1]/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
//DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
```

...

- **XPath search:** Like before, we start with an XPath expression identifying the element from the root node. This time, we either remove or replace parts of the expression with wildcards. This method collects elements when the page changes the container for the element.

```

*/BODY[1]/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
BODY[1]/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
HTML/*/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
HTML/DIV[2]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
HTML/BODY[1]/*/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
HTML/BODY[1]/DIV[1]/DIV[4]/DIV[2]/FORM[1]/DIV[10]/BUTTON[1]
...

```

- **Text content:** We find all elements which have the same text content as the original element.

```
//*[text()='Search']
```

- **Class and id names:** We find all elements which either share a class name or id name as the original element. Unlike the other queries which are done using XPath, we implement this search in JQuery[16].

```

#submitButton
.swa-button
.swa-button_primary

```

Identifying the Element

With all the pieces in place, we go over the actual steps Ringer takes during replay to find a matching element. The pseudocode of the element identification algorithm is shown in Figure 7.4.3.

```

getTarget(targetInfo : Dictionary) : DOMNode
1 possibleTargets : Set[DOMNode] ← {}
2 for f : Function[Dictionary, Set[DOMNode]] ← queryFunctions do
3 possibleTargets ← possibleTargets ∪ f(targetInfo)
4 rankedTargets : List[DOMNode] ← possibleTargets.sort(x, y →
5 similarity(x, targetInfo) – similarity(y, targetInfo)
6 )
7 return rankedTargets[0]

```

Figure 7.4.3. Algorithm to identify matching element on replay page.

```

similarity(x : DOMNode, targetInfo : Dictionary) : Integer
1 sum ← 0
2 for a : String ← targetInfo.keys() do
3   if x[a] == targetInfo[a] then
4     sum ← sum + weight[a]
5 return sum

```

Figure 7.4.4. Algorithm to calculate similarity score.

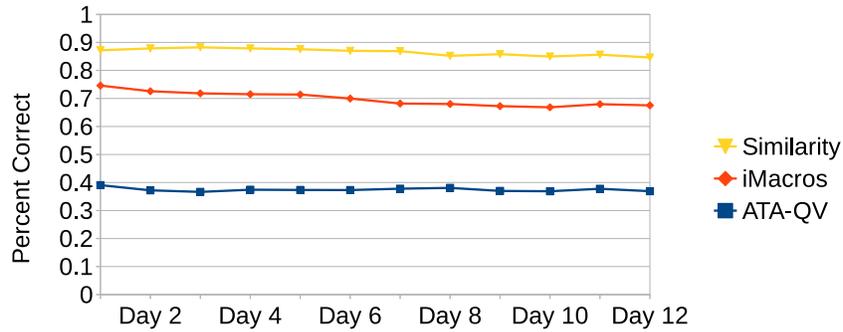
The algorithm first searches the webpage for a set of possible elements, using the different search strategies discussed in the previous section (lines 2-3). It then sorts the possible elements (lines 4-6) using the similarity function described previously (Figure 7.4.4). Notice that *weight* on line 4 in Figure 7.4.4 can be modified to give specific properties more weight. We finally return the element with the highest similarity score (line 7).

7.5 Evaluation

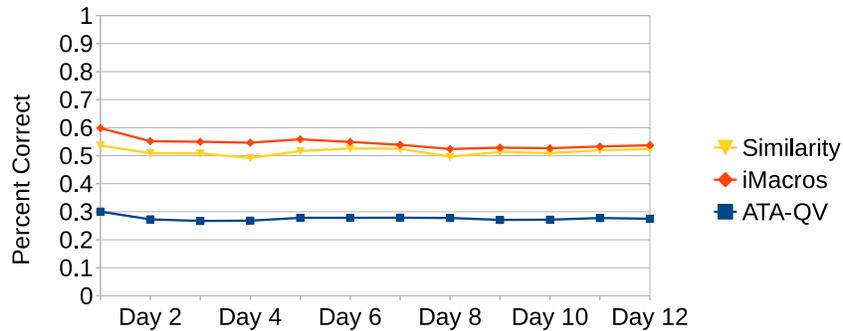
We test our element identification algorithm in isolation from Ringer. We compare our similarity-based algorithm against the state of the art and find that our approach outperforms existing element identification approaches.

Defining the ground truth, *i.e.* whether a replay-time node corresponds to a record-time node, cannot be perfectly automated. After all, if we could automatically and perfectly identify the corresponding node, we would have solved the element identification problem. Consider a webpage with a list of blog posts; each post starts at the top of the list and is shifted down as new entries appear. Say our record-time target node n is the post at index 0, with title t . Say at replay-time, the post with title t appears at index 2. What node corresponds to n ? The post at index 0 or the post with title t ? Either could be correct, depending on the script that uses it. Only the human user can definitively answer what node he would select if he looked at the new version of the page.

Unfortunately, involving humans vastly reduces the number of nodes on which we can test. To increase the scale of our experiment, we define two automatic measures of node correspondence. Both are based on clicking a node, and subsequently checking if the click caused a page navigation. First, a lower bound: a replay-time node n' corresponds to a record-time node n if clicking on n' causes a navigation to the same page (the same URL), as clicking on n . Second, an upper bound: a replay-time node n' corresponds to a record-time node n if



(a) Upper bound on element identification performance.



(b) Lower bound on element identification performance.

Figure 7.5.1. Performance of state-of-the-art element identification algorithms against our similarity approach.

clicking on both nodes causes navigation, possibly to different pages. The upper bound will handle cases like the blog described above. Clicking on the top post may be the right action, but we should expect it to lead to a different URL. Of course, with these bounds we can only test the element identification algorithms on nodes that cause page navigations. We consider this to be a reasonable tradeoff, as it allows us to test on a much larger set of nodes than would be possible through human labeling.

Procedure

We use our upper and lower bounds to test our **similarity** algorithm against the **iMacros** [5] and **ATA-QV** [40] algorithms on a dataset of 7,750 clickable DOM nodes from the 50 most popular websites, according to Alexa rankings [1]. On Day 0, for each webpage, we programmatically click on every node and record the destination URL. We keep only nodes that cause navigation. Once per day, we test whether each algorithm can find each node.

Results

Figure 7.5.1 shows the percentage of nodes on which each algorithm succeeds each day, using both metrics. Using the upper bound metric, **similarity** consistently outperforms the other approaches. Already on the first day, **similarity** exhibits about 1.17x the success rate of **iMacros** and about 2.23x the success rate of **ATA-QV**. By the final day, **similarity** exhibits about 1.25x the success rate of **iMacros** and about 2.30x the success rate of **ATA-QV**.

In contrast, **iMacros** initially outperforms **similarity** according to the lower bound metric. This occurs because **iMacros**, which requires its output node to have the same text as the original target node, favors text over role or location. In contrast, **similarity**, since it has more features related to location than text, favors location over text. Recall the blog page described above. **iMacros** selects the post with title t even if it is now at index 2, while **similarity** selects the post at index 0, even though it does not have title t . Either of these may be the correct, but the former is much more likely to lead to the same URL as the original node. The lower bound, which asks for post-click URLs to be exactly the same, rewards text-sensitive approaches more than location-sensitive approaches in these ambiguous cases, and thus rewards **iMacros**. Note however that **iMacros**' slight initial advantage over our approach quickly fades.

Similarity weights. The **similarity** approach in this experiment uses uniform weights. In additional experiments, we tested this variation against the SVM and regression versions, and found that the uniform weights performed best. Already on Day 1, the upper bound of the uniform weights version was 94% compared to 53% for the regression-trained weights and 86% for the SVM-trained weights. The machine learning variations failed because they placed too much emphasis on a small number of features. Like prior element identification approaches, these algorithms relied on the assumption that future webpage changes would be like the changes observed in the past. In today's ever-evolving web, this is not a reasonable expectation. Thus uniform weights, an approach which can adapt to the largest variety of changes, produces the best performance.

7.6 Conclusion

Being able to identify elements involved in the demonstration is crucial to robust record and replay. Past approaches use a fixed strategy, looking at a few key attributes such as the text content or location in the DOM tree. We instead use a similarity function to compare the desired element against those found on the replay page. Our similarity function differs from past approaches in that it's based upon empirical analysis of real websites and uses all attributes of an element to determine whether it matches the recorded element.

Chapter 8

Applications

In this chapter, we show the usefulness of record and replay as a building block by going over two end-user tools built using Ringer. R+R by itself has limited applications, since the replayer only repeats the exact same sequence of actions. But if we can generalize a recording, then we can apply R+R to a much wider range of problems. For example, imagine that a user demonstrated how to scrape the cost of a train ticket for a particular day. If we generalize the recording, then the user can automatically scrape the cost of a ticket for any day of the month. Generalization also allows application developers to use R+R to develop end-user applications which can automate tasks through demonstration.

This chapter is organized as follows. We go over the interface we expose to developers in Section 8.1. This API gives end-user application developers a simple way to interact with Ringer, abstracting away the intricacies of browser record and replay. We then go over two end-user applications built using Ringer. The first is a system which allows end-users to build a relational scraper by demonstration (Section 8.2). The other application allows the user to build a page of live tiles, where each tile displays up-to-date information scraped by demonstration from a website (Section 8.3). These two applications show how R+R can enable end-users to accomplish tasks which previously required programming ability.

Function	Return Type	Description
startRec()	-	starts a new recording
stopRec()	Program	ends current recording
replay(p:Program)	Program	replays the program p

Table 8.1.1. The API for recording and replaying Ringer programs.

Function	Description
Program.parameterizeXPath(name:ID, origXPath:XPath)	replace <i>origXPath</i> with the parameter <i>name</i>
Program.useXPath(name:ID, newXPath:XPath)	supplies <i>newXPath</i> as the argument for parameter <i>name</i>
Program.parameterizeTypedString(name:ID, origString:String)	replace <i>origString</i> with the parameter <i>name</i>
Program.useTypedString(name:ID, string:String)	supplies <i>string</i> as the argument for parameter <i>name</i>
Program.parameterizeFrame(name: ID, origFrame: FrameID)	replace <i>origFrame</i> with the parameter <i>name</i>
Program.useFrame(name:ID, frame: FrameID)	supplies <i>frame</i> as the argument for parameter <i>name</i>

Table 8.1.2. The API for parameterizing Ringer programs.

8.1 API for Developers

We provide a metaprogramming interface to Ringer which allows developers to manipulate Ringer scripts as if they were programs. Table 8.1.1 gives the basic interface which the host application uses to record and replay user demonstrations. This interface by itself is not very useful since the host application cannot modify the recorded script. Table 8.1.2 gives the interface which the host application uses to modify the recorded script. The interface works by parameterizing the recorded scripts in three dimensions: elements, typed strings and frames.

- **Elements:** Often, the host application will want to interact with a different element from those in the initial demonstration, such as when the replayer should click a different button or select a different item in a pull-down menu. In these situations, we use XPath as a way of identifying which element from the original demonstration should be parameterized. When replaying the parameterized script, the host application can fill in this parameter by specifying an element on the replay-time page.
- **Typed strings:** Another common parameter is to change the text a user types. Once the host applications parameterizes a text string, Ringer looks for a sequence of keyboard events that correspond to the sequence. The host application can then specify a new string, which Ringer uses to generate a replacement sequence of keyboard events.
- **Frames:** Finally, the host application can modify which page the script is replayed upon by parameterizing the frame. A frame is a part of a webpage which displays content independently of the container. Each page has a top-level page and may contain multiple iframes. When the replayer executes an action, it must first find a frame for that action and

then chooses an element within that frame. By specifying a frame, the host application can force the replayer to use an existing webpage instead of opening a new page.

The Ringer API abstracts away the details of altering event objects, adding and removing event objects, and managing Ringer's frame mapping. From the programmer's perspective, she merely replaces the values from the original demonstration with parameters and fills in those parameters with arguments.

8.2 Scraping Relational Data by Demonstration

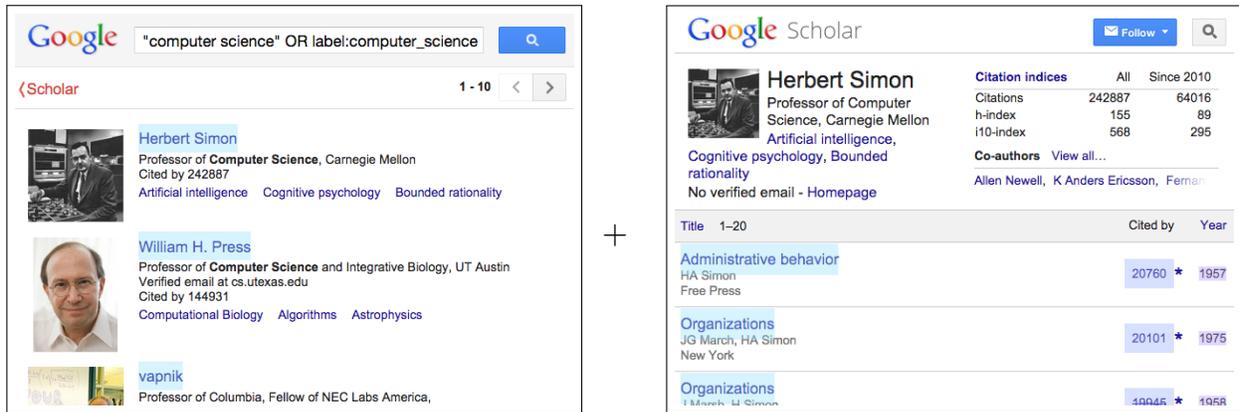
Many websites display structured data that a user wants. Scraping this data is simple if all the data exists on a single page, but becomes substantially harder when spread across multiple pages or exists behind a form. Even for these more difficult situations, a user should be able to scrape this data through demonstration, without needing to write a script. To implement such a tool, we can decompose the problem into two parts: finding logical lists on a website and demonstrating an interaction to perform for each list item. Chasins implemented such a tool, WebCombine, which allows non-coders to scrape structured data off of websites [7], using Ringer to record and replay interactions on the list items. We go over an example scenario for WebCombine which highlights the use of Ringer as a building block for end-user applications.

Example Assume a user wants to scrape information off of Google Scholar, which contains information about academic publications. In particular, they want to scrape off the citation counts of papers written by the most cited authors in Computer Science. Google Scholar contains this information, but not in an accessible format. To do this, an end-user must 1) scrape a list of authors, 2) for each author, follow a link to the author's page, and 3) from the author's page, scrape a list of papers.

Walkthrough of User Interaction

To scrape this data with WebCombine, a user alternates between demonstrating lists and recording interactions. WebCombine interprets a list demonstration as the introduction of a for loop, and a recorded interaction as a procedure in the loop body. Each new loop is nested in the one before.

Specify list of authors The user starts his WebCombine script with a list. Figure 8.2.1a displays a screenshot of the browser window. WebCombine's control panel is not shown. In Figure 8.2.1a, the user has used the control panel to start list finding mode, then clicked on the name "Herbert Simon" in the webpage. Nodes in WebCombine's current hypothesized list are highlighted. In this case, WebCombine has correctly hypothesized that the user wants to scrape all authors,



(a) Author list page

(b) One author's paper list page



Herbert Simon	Administrative ...	20760	1957
Herbert Simon	Organizations	20101	1975
Herbert Simon	Organizations	19945	1958
Herbert Simon	The Sciences ...	17865	1981
...
William H. Press	Numerical ...	116562	1992
William H. Press	Numerical ...	113482	1992
William H. Press	Numerical ...	113289	1986
William H. Press	Numerical ...	112755	2007
...
vapnik	Statistical ...	55113	1998
vapnik	The Nature ...	54856	1995
vapnik	Support-vector ...	16111	1995
vapnik	A training ...	6254	1992
...

(c) WebCombine output

Figure 8.2.1. Overview of how a user interacts with WebCombine to scrape data off of Google Scholar. The user first identifies the list of authors by clicking on author names. Next the user demonstrates how to access the citations page for one author (by clicking on the link). On the resultant page, the user identifies the list of papers. WebCombine uses these demonstrations to automatically scrape the website, producing a single table of papers with each paper's author, title, citation count, and year.

so he need not provide more sample list elements. To indicate how to reach the next page of authors, the user will click a “Next Button” option on the control panel, then click on the arrow button in the upper right corner of the webpage. With this, the user will have completely specified how to scrape the list of authors.

Demonstrate how to get to author’s page The user’s next step is to enter recording mode and record himself clicking on “Herbert Simon” in the browser window. WebCombine uses Ringer to record the user’s actions. This loads the author’s publications page, pictured in 8.2.1b. Once that page has opened, the user may end the recording.

Demonstrate list of papers In Figure 8.2.1b, the user has started a new list. He has clicked on the title, year, and citations of the first paper. The list finder has correctly hypothesized that the user wishes to collect the list of Herbert Simon’s papers, including all clicked attributes. If he next clicked on the paper’s authors, an additional column would be added to the output.

Scrape papers Running WebCombine after these demonstrations produces the output displayed in Figure 8.2.1c.

Using the Ringer API

From a sequence of alternating lists and recordings — `list_1`, `recording_1`, `list_2`, ... — WebCombine creates a program of the following form:

```

1 recording_1 = recording_1.parameterizeXPath(first_row[0].XPath)
2 recording_2 = recording_2.parameterizeXPath(first_row[1].XPath)
3 ...
4
5 for item_1 in find(list_1):
6     id_1 = table1.insert(item_1.text)
7     replay(recording_1.useXPath(item_1.XPath))
8     for item_2 in find(list_2):
9         id_2 = table_2.insert(item_2.text, foreign_key=id_1)
10        replay(recording_2.useXPath(item_2.XPath))
11        ...

```

For each for loop in the scraping program, WebCombine maintains a relational table. For this example, WebCombine would have one authors table, with a single column of author names, and one paper table, with columns for title, year, and citations. Each record generated by an inner loop is linked by a foreign key to the most recent record in the immediate surrounding loop. Thus, each paper record scraped from Herbert Simon’s page is linked to the Herbert Simon author record. Because such databases may be unfamiliar to end users, we present the data to the user as a single table by doing a natural join over all tables, as shown in Figure 8.2.1c.

Note that the user only demonstrated the process on the first element of the outer loop, Herbert Simon. If the user had chosen to record an interaction to repeat on each paper, he would also demonstrate on only the first paper. Essentially, the user records how he collects the first row of data, leaving the tool to automatically collect the rest.

WebCombine benefits from Ringer’s ability to record and replay user demonstrations. Essentially, WebCombine uses Ringer to navigate between nested lists, even if those lists exists across multiple pages or are dynamically loaded onto the page. By using the Ringer API, WebCombine does not have to deal with the intricacies of browser R+R, and instead can focus on finding and scraping lists.

8.3 Real-time Updates by Demonstration

Imagine a user wants his browser home screen to display the daily special at his local pizza shop, his local ice cream shop, and when the local pool is open. The problem is that these local establishments may not have resources to offer more than a basic web site, with no API to the information.

For some widely used services, there may already already exist a way to see real-time updates. IFTTT allows developers to provide APIs for websites, so that end-users can create simple mashups[31]. However, many websites will not have enough demand for a developer

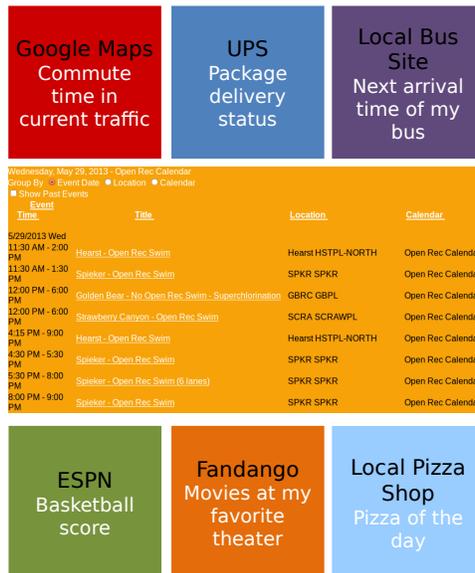


Figure 8.3.1. We present a mockup of the live tile interface. Each tile represents a different demonstration. When the user clicks a tile, it enlarges to display up-to-date information scraped during the last execution of the recording.

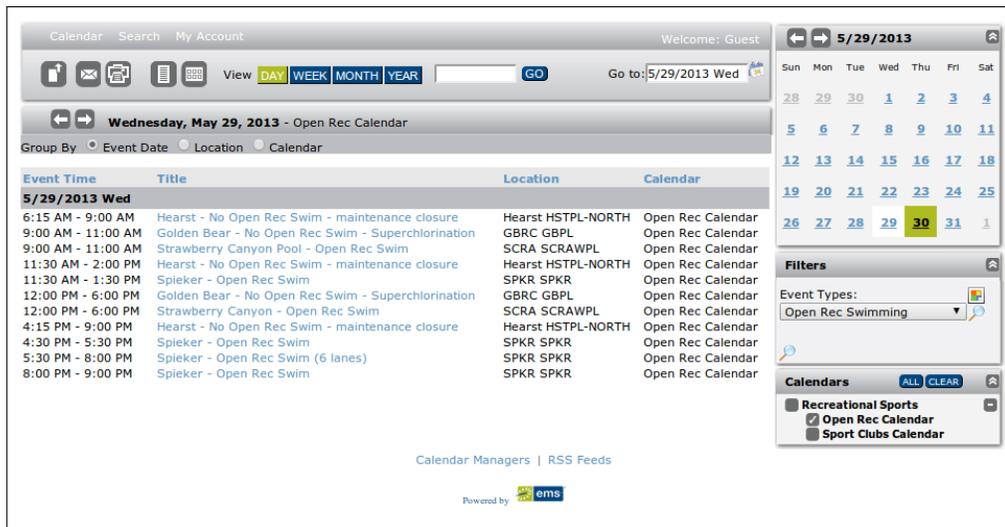


Figure 8.3.2. This screenshot shows the online calendar for recreational facilities at UC Berkeley. To find out which pools are open today, the user must click the "Day" button and then select "Open Rec Swimming" as the event type.

to build such hooks, such as a given user's local pool, local restaurants, local school. To get data off such pages, the user needs to play a role — perhaps by demonstrating how to scrape the data. We built an application which allows end-users to build such custom webpages by demonstration.

The user interacts with our application solely through demonstration. First, the user demonstrates the sequence of actions required to load the data. Once the data exists on the page, the user can select it directly on the page through an interface which outlines elements as the user hovers over it. With these two interactions, the application is able to continuously scrape the desired data and present it on the user's home screen as a live tile.

We illustrate the steps an end-user takes to use our application. Figure 8.3.2 shows the online calendar for recreational facilities at UC Berkeley. When the page first loads, it shows the hours of all facilities. To filter out irrelevant information, the user needs to click the "Day" button and select "Open Rec Swimming" as the event type. The user then selects text which contains the schedule by hovering over it. Once he finishes the demonstration, the user adds a tile to their homepage, shown in Figure 8.3.1.

Each time the application wants fresh data, it uses Ringer to replay the user demonstration and then scrapes the current text. Even though this application is just a thin wrapper around the Ringer API, it enables end-users who cannot program to create a fully customized homepage which contains precisely the information they want.

8.4 Conclusion

We see record and replay as a building block for more expressive end-user applications. Using Ringer's simple API, we built two applications. The first allows users to scrape a website solely through demonstration. The second allows end-users to create custom webpages which consolidate information from other sites. Creating these two end-user applications with Ringer was greatly simplified, since we did not have to deal with the difficulties of R+R in the browser.

Chapter 9

Related Work

We go over previous work that relates to the topics covered in this thesis. Ringer builds upon previous work done in many fields, from record and replay to end-user programming.

Web Automation

Browser Record and Replay

Many previous tools allow users to automate the browser through demonstration. Like our work, the CoScripter line of research focuses on making scripting accessible to truly non-technical users via record and replay. Besides CoScripter[21] itself, this line includes the CoTester project for testing web programs[24], Vegemite for creating mashups[23], and ActionShot[22] for representing a history of browser interactions. CoScripter allows end-users to create natural-language programs, using only a demonstration as input. The key difference between CoScripter and Ringer lies in the assumptions about webpage behavior on which the two tools rely. CoScripter works at a higher level of abstraction, making the assumption that its small language of instructions can represent all possible interactions. Because of this, CoScripter's language cannot faithfully capture all interactions, causing it to break in unexpected ways.

Other tools offer record and replay as an aide to scripting, providing users a rough draft of the target script which they can modify and update. These tools are intended to streamline the process of programming a script, but not to remove the programming component altogether. iMacros[5] is one such commercial tool. Selenium[34], a browser automation framework targeted at web application testing, is another. Selenium IDE offers a record and playback functionality. However, both iMacros and Selenium produce scripts that a programmer must edit before they can be replayed. Both tools target experienced programmers, who can generally write the scripts from scratch. For all but the simplest interactions, these tools' recording systems do not produce functional replay scripts. Robofox[18] is yet another record and replay tool. It differs from past tools allowing end-users to insert assertions, which can detect script failures

Another branch of tools targets end-users by focusing on a limited class of scripts. For instance, the Smart Bookmarks [15] offers an extension to standard Internet bookmarking, allowing users to provide an argument to a form on the bookmarked page. Creating a Smart Bookmark for google.com with the argument "foo" would load the Google results page for the string "foo." Platypus[30] focuses on another small slice of possible script uses, allowing users to alter the way a page is displayed, for instance to hide ads or change text style. While these tools are usable without any programming skill, they do not give users the means to automate arbitrary tasks.

Scripting Languages and Libraries

Another class of tools offers APIs to streamline the process of web script writing. Chickenfoot[4] lets users combine high-level commands and pattern-matching to identify components of a webpage. Abmash[27] is another such library, allowing users to control the webpage through high-level commands. Sikuli[41] allows script writers to use visual screenshots to identify GUI components. Beautiful Soup[3] is a Python library which allows programs to interact with a web page. Libraries also exist specifically for webscraping, such as XPath[11, 12, 19] and Scrapy[33]. While these approaches do simplify webpage automation, they still demand that the user be able to program and understand how the browser works.

End-user Programming

End-user programming has been a long-held dream, giving non-technical users the ability to automate a computer. Early work[8] showed promise, but did not take off for various reasons[20]. There has been limited success in end-user tools for the web though. Mashups[9, 39] allow end-users to connect content from different sources to create new services. IFTTT is one such mashup service, which creates simple conditional mashups that follow the pattern "If this, then that." But these mashups all rely on a API to the original data or service, which requires a programmer.

Deterministic Record and Replay

Another class of projects explored web record and replay not to process ever-changing live webpages, but with the goal of holding them perfectly stable. Mugshot[26] is a record and replay tool aimed at web developers, used to debug buggy executions of web applications. It records all sources of non-determinism at record-time and prevents their divergence at replay-time by using a proxy server and instrumenting the webpages. Because Mugshot specifically targets — and in fact enforces — non-liveness, it can rely on the structure of the JavaScript code and the contents of the page to be the same across executions. Timelapse[6] also implements deterministic record

and replay, but works at a much lower level, using techniques from virtual machine literature. Jalangi[35] performs record and replay for JavaScript code, in order to run dynamic analyses. Like Mugshot, it must instrument the Javascript code to accomplish replay, and relies on the JavaScript structure being the same across executions. JSBench[32] takes the same approach, instrumenting the JavaScript code to create a straight-line, deterministic benchmark from a user interaction. These record and replay approaches successfully allow programmers to review and analyze web applications. However, they are not directly useful to end-users who want to automate the live webpage.

Trigger Inference

The trigger inference problem is closely related to race detection on webpages [29]. In fact, what we want to discover is when there is a race between executing some code on the webpage and a user interaction. But not all races on the webpage are harmful, and we believe actually most are benign. Therefore, instead of trying to find all races, we empirically identify races which cause the script the break. Another related field is inference of partial orders [10, 25]. This work tries to infer a hidden partial order based upon linear extensions, *i.e.*, a trace, of the hidden partial order. Unlike past work, we want to find a over-approximation of the partial order and therefore use a more conservative approach.

Chapter 10

Conclusion

This thesis explores the design of a record and replay system for the browser, focusing on how to faithfully mimic a user's interactions. As the web grows, end-users will need to handle more and more tasks on web, many of which are tedious in nature. Our system, Ringer, aims to help the end-user by turning their demonstration into a script, which then can be replayed. Ringer script's take the form of a sequence of statements, with each statement composed of an action, element and trigger. We believe this form is key to faithful replay, allowing Ringer to infer each component from demonstrations. For each component, we design an abstraction and an algorithm such that Ringer can synthesize a program from a demonstration which faithfully mimics the user's interactions. Our use of triggers allows Ringer scripts to work on interactive pages, unlike many of the past approaches. We show the benefits of our approach through an empirical study, showing that Ringer can replay interactions on even the most complicated modern websites.

But Ringer is just one step toward bridging the gap between an end-users goals and the interfaces provided to them by a website. To get closer to this goal, we envision more expressive end-user tools which use record and replay as a building block. These tools would ask user's to demonstrate their tasks, but would then modify the script. The programming-by-demonstration web scraper is one such tool. Future directions could also query the user to create even more robust scripts, or even observe a user performing a repetitive task and automatically continue it.

Bibliography

- [1] *Alexa Top 500 Global Sites*. <http://www.alexa.com/topsites>. July 2013.
- [2] *Automation Testing Tool for Web Applications*. <http://sahipro.com/>. 2015. (Visited on 11/30/2015).
- [3] *Beautiful Soup: We called him Tortoise because he taught us*. <http://www.crummy.com/software/BeautifulSoup/>. July 2013.
- [4] Michael Bolin et al. "Automation and customization of rendered web pages". In: *Proceedings of the 18th annual ACM symposium on User interface software and technology*. UIST '05. Seattle, WA, USA: ACM, 2005, pp. 163–172. DOI: 10.1145/1095034.1095062. URL: <http://doi.acm.org/10.1145/1095034.1095062>.
- [5] *Browser Scripting, Data Extraction and Web Testing by iMacros*. <http://www.iopus.com/imacros/>. July 2013.
- [6] Brian Burg et al. "Interactive Record/Replay for Web Application Debugging". In: *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*. UIST '13. St. Andrews, Scotland, United Kingdom: ACM, 2013, pp. 473–484. ISBN: 978-1-4503-2268-3. DOI: 10.1145/2501988.2502050. URL: <http://doi.acm.org/10.1145/2501988.2502050>.
- [7] Sarah Chasins et al. "Browser Record and Replay As a Building Block for End-User Web Automation Tools". In: *Proceedings of the 24th International Conference on World Wide Web Companion*. WWW '15 Companion. Florence, Italy: International World Wide Web Conferences Steering Committee, 2015, pp. 179–182. ISBN: 978-1-4503-3473-0. DOI: 10.1145/2740908.2742849. URL: <http://dx.doi.org/10.1145/2740908.2742849>.
- [8] Allen Cypher. *What What I Do: Programming by Demonstration*. 1993. URL: <http://acypher.com/wwid/WWIDToC.html> (visited on 05/08/2015).
- [9] Rob Ennals et al. "Intel Mash Maker: Join the Web". In: *SIGMOD Rec.* 36.4 (Dec. 2007), pp. 27–33. ISSN: 0163-5808. DOI: 10.1145/1361348.1361355. URL: <http://doi.acm.org/10.1145/1361348.1361355>.
- [10] Proceso L. Fernandez et al. *Reconstructing Partial Orders from Linear Extensions*. 2006.

- [11] Tim Furche et al. "XPath: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web". In: *The VLDB Journal* 22.1 (Feb. 2013), pp. 47–72. ISSN: 1066-8888. DOI: 10.1007/s00778-012-0286-6. URL: <http://dx.doi.org/10.1007/s00778-012-0286-6>.
- [12] Giovanni Grasso, Tim Furche, and Christian Schallhart. "Effective Web Scraping with XPath". In: *Proceedings of the 22Nd International Conference on World Wide Web Companion. WWW '13 Companion*. Rio de Janeiro, Brazil: International World Wide Web Conferences Steering Committee, 2013, pp. 23–26. ISBN: 978-1-4503-2038-2. URL: <http://dl.acm.org/citation.cfm?id=2487788.2487796>.
- [13] *Greasespot*. <http://www.greasespot.net/>. July 2015.
- [14] *HTMLUnit*. <http://htmlunit.sourceforge.net/>. 2015. (Visited on 11/30/2015).
- [15] Darris Hupp and Robert C. Miller. "Smart bookmarks: automatic retroactive macro recording on the web". In: *Proceedings of the 20th annual ACM symposium on User interface software and technology. UIST '07*. Newport, Rhode Island, USA: ACM, 2007, pp. 81–90. DOI: 10.1145/1294211.1294226. URL: <http://doi.acm.org/10.1145/1294211.1294226>.
- [16] *jQuery: write less, do more*. <https://jquery.com/>. 2015. (Visited on 07/22/2015).
- [17] *Kimono: Turn websites into structured APIs from your browser in seconds*. URL: <https://www.kimonolabs.com> (visited on 05/08/2015).
- [18] Andhy Koesnandar et al. "Using Assertions to Help End-user Programmers Create Dependable Web Macros". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT '08/FSE-16*. Atlanta, Georgia: ACM, 2008, pp. 124–134. ISBN: 978-1-59593-995-1. DOI: 10.1145/1453101.1453119. URL: <http://doi.acm.org/10.1145/1453101.1453119>.
- [19] Jochen Kranzdorf et al. "Visual XPath: Robust Wrapping by Example". In: *Proceedings of the 21st International Conference Companion on World Wide Web. WWW '12 Companion*. Lyon, France: ACM, 2012, pp. 369–372. ISBN: 978-1-4503-1230-1. DOI: 10.1145/2187980.2188051. URL: <http://doi.acm.org/10.1145/2187980.2188051>.
- [20] Tessa Lau. "Why PBD systems fail: Lessons learned for usable AI". In: *CHI 2008 Workshop on Usable AI*. Florence, IT, Apr. 2008.
- [21] Gilly Leshed et al. "CoScripter: automating & sharing how-to knowledge in the enterprise". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. CHI '08*. Florence, Italy: ACM, 2008, pp. 1719–1728. DOI: 10.1145/1357054.1357323. URL: <http://doi.acm.org/10.1145/1357054.1357323>.

- [22] Ian Li et al. "Here's What I Did: Sharing and Reusing Web Activity with ActionShot". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: ACM, 2010, pp. 723–732. DOI: 10.1145/1753326.1753432. URL: <http://doi.acm.org/10.1145/1753326.1753432>.
- [23] James Lin et al. "End-user programming of mashups with vegemite". In: *Proceedings of the 14th international conference on Intelligent user interfaces*. IUI '09. Sanibel Island, Florida, USA: ACM, 2009, pp. 97–106. DOI: 10.1145/1502650.1502667. URL: <http://doi.acm.org/10.1145/1502650.1502667>.
- [24] Jalal Mahmud and Tessa Lau. "Lowering the barriers to website testing with CoTester". In: *Proceedings of the 15th international conference on Intelligent user interfaces*. IUI '10. Hong Kong, China: ACM, 2010, pp. 169–178. DOI: 10.1145/1719970.1719994. URL: <http://doi.acm.org/10.1145/1719970.1719994>.
- [25] Heikki Mannila and Christopher Meek. "Global Partial Orders from Sequential Data". In: *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '00. Boston, Massachusetts, USA: ACM, 2000, pp. 161–168. ISBN: 1-58113-233-6. DOI: 10.1145/347090.347122. URL: <http://doi.acm.org/10.1145/347090.347122>.
- [26] James Mickens, Jeremy Elson, and Jon Howell. "Mugshot: deterministic capture and replay for Javascript applications". In: *Proceedings of the 7th USENIX conference on Networked systems design and implementation*. NSDI'10. San Jose, California: USENIX Association, 2010, pp. 11–11. URL: <http://dl.acm.org/citation.cfm?id=1855711.1855722>.
- [27] Alper Ortac, Martin Monperrus, and Mira Mezini. "Abmash: mashing up legacy Web applications by automated imitation of human actions". In: *Software: Practice and Experience* 45.5 (2015), pp. 581–612. ISSN: 1097-024X. DOI: 10.1002/spe.2249. URL: <http://dx.doi.org/10.1002/spe.2249>.
- [28] *Outwit Hub - Your Own Web Collection Engine*. <http://www.outwit.com/products/hub/>. July 2013.
- [29] Boris Petrov et al. "Race Detection for Web Applications". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 251–262. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254095. URL: <http://doi.acm.org/10.1145/2254064.2254095>.
- [30] *Platypus*. <http://platypus.mozdev.org/>. Nov. 2013.
- [31] *Put the internet to work for you.* - IFTTT. URL: <https://ifttt.com> (visited on 05/08/2015).

- [32] Gregor Richards et al. "Automated Construction of JavaScript Benchmarks". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 677–694. ISBN: 978-1-4503-0940-0. DOI: 10.1145/2048066.2048119. URL: <http://doi.acm.org/10.1145/2048066.2048119>.
- [33] *Scrapy*. <http://scrapy.org/>. July 2013.
- [34] *Selenium-Web Browser Automation*. <http://seleniumhq.org/>. July 2013.
- [35] Koushik Sen et al. "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 488–498.
- [36] W3C. *UI Events (formerly DOM Level 3 Events)*. <http://www.w3.org/TR/DOM-Level-3-Events/>. 2015. (Visited on 07/22/2015).
- [37] Aaron Walter. *Designing for Emotion*. 1st ed. A Book Apart, Oct. 2011. ISBN: 978-1-937557-00-3.
- [38] *Watir.com | Web Application Testing in Ruby*. <http://watir.com/>. 2015. (Visited on 11/30/2015).
- [39] Jeffrey Wong and Jason I. Hong. "Making Mashups with Marmite: Towards End-user Programming for the Web". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. San Jose, California, USA: ACM, 2007, pp. 1435–1444. ISBN: 978-1-59593-593-9. DOI: 10.1145/1240624.1240842. URL: <http://doi.acm.org/10.1145/1240624.1240842>.
- [40] Rahulkrishna Yandrapally et al. "Robust Test Automation Using Contextual Clues". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISTA 2014. San Jose, CA, USA: ACM, 2014, pp. 304–314. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610390. URL: <http://doi.acm.org/10.1145/2610384.2610390>.
- [41] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: using GUI screenshots for search and automation". In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. UIST '09. Victoria, BC, Canada: ACM, 2009, pp. 183–192. DOI: 10.1145/1622176.1622213. URL: <http://doi.acm.org/10.1145/1622176.1622213>.