Ringer: Web Automation by Demonstration

Shaon Barman
University of California, Berkeley (USA)
sbarman@cs.berkeley.edu

Sarah Chasins
University of California, Berkeley (USA)
schasins@cs.berkeley.edu

Rastislav Bodik
University of Washington (USA)
bodik@cs.washington.edu

Sumit Gulwani
Microsoft Research (USA)
sumitg@microsoft.com

Abstract

With increasing amounts of data available on the web and a diverse range of users interested in programmatically accessing that data, web automation must become easier. Automation helps users complete many tedious interactions, such as scraping data, completing forms, or transferring data between websites. However, writing web automation scripts typically requires an expert programmer because the writer must be able to reverse engineer the target webpage. We have built a record and replay tool, Ringer, that makes web automation accessible to non-coders. Ringer takes a user demonstration as input and creates a script that interacts with the page as a user would. This approach makes Ringer scripts more robust to webpage changes because user-facing interfaces remain relatively stable compared to the underlying webpage implementations. We evaluated our approach on benchmarks recorded on real webpages and found that it replayed 4x more benchmarks than a state-of-the-art replay tool.

Categories and Subject Descriptors H.5.3 [*Group and Organization Interfaces*]: Web-based interaction

General Terms Design, Languages

Keywords Record-Replay, Automation, Javascript, Browser

1. Introduction

Programmatic access to user-facing websites serves a range of purposes: news readers reformat news articles for cleaner access (e.g., Readability [3]); end users automate tedious interactions with webpages (e.g., IFTTT [1]); and data scientists scrape data for their studies and journalists for their stories (the investigative news organization ProPublica hires programmers to develop web scrapers [2]). Overall, the interest in web automation is growing: the number of StackOverflow questions on "scraping" grew by 16% in 2014 and 23% in 2015 [34], and commercial scrapers such as Kimono [25] and import.io [22] appeared during the last three years.

Writing good web automation scripts is challenging. Since some websites do not offer APIs for accessing their data, retrieving the data programmatically requires reverse engineering their webpages' DOM tree structures and event handlers. To remain useful, programs must account for future changes to webpages, such as website redesigns or updated content. Additionally, on websites that actively attempt to prevent automated scraping, the programmer must work around obfuscation. For instance, some pages wait to load part of a page's content until the user hovers over or clicks on a relevant component. Further, modern user interfaces, such as autocomplete menus and "infinite scrolling," add to the complexity of automating web access.

1.1 Approach

We describe Ringer, a record and replay system that produces web automation scripts from end-user demonstrations. Ringer is based on the observation that while the internals of a webpage may change, the user-facing interface tends to remain stable in order to maintain a consistent user experience. Reliable automation can thus be achieved by invoking actions at the level of the user-visible interface, rather than by reverse engineering server requests or JavaScript function calls or other low-levels components of the implementation.

Given a user demonstration, Ringer produces a script that completes the user's demonstrated task. The script is a sequence of statements of the form "wait for a condition C, then perform an action a on a webpage node n." Ringer records the sequence of actions in the demonstration. For

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984020

each action, it produces a condition and a node selector that make the script robust to certain types of webpage changes.

1.2 Applications

Because Ringer infers scripts from only a user demonstration, it gives non-programmers access to web automation. Using Ringer, we built an example application (discussed further in Section 8.5) that lets non-coders build a homepage with custom "live information tiles." Users record how to scrape the day's flavors at their local ice cream shop, today's open hours at their pool, or their current commute time; the custom homepage replays those interactions to display up-to-date information.

Ringer benefits expert programmers, too, because record and replay obviates the need for tedious reverse engineering. Programmers can simply record an interaction and then modify the resulting Ringer script or embed it into a larger application. For example, WebCombine [14], a tool built using Ringer, lets a user scrape large datasets from structured websites. The user demonstrates how to scrape one row of the dataset, and WebCombine modifies the resultant Ringer script to scrape all other rows of the dataset.

1.3 Technical Challenges

Replaying recorded actions may appear simple, but fundamental challenges make it difficult to mimic a human user, both spatially (selecting the node on which to apply an action) and temporally (determining the conditions under which the page is ready for the action).

To solve the spatial problem we must define a reliable correspondence between demonstration-time DOM nodes and replay-time nodes. Given a new version of a webpage, we seek the node that a user would select to re-demonstrate the interaction. Existing solutions [6, 27, 36] are fragile because they require a few key node attributes to remain constant. In contrast, our approach selects the replay-time node that maximizes a similarity metric. A longitudinal study of 30 webpages found that after 37 days our approach still identified 83% of nodes, 22 percentage points more than the next best approach.

To solve the temporal problem we must determine when the webpage is ready for the next user action. Modern interactive webpages often use visual cues, like showing a loading bar, to signal that they are waiting for responses from a server. An impatient user (or a naive replay tool) might ignore the cue and use stale data, producing unexpected behaviors. Although these visual cues are intuitive to humans, existing replay techniques are oblivious to them. To address the temporal problem, Ringer replays scripts multiple times to infer triggers — conditions that must be met before each script action can run. We empirically show that these conditions make Ringer programs robust to pages with asynchronous server communication and have the added benefit of being on average more than 2.5x faster than the user's demonstration.

1.4 Results

To evaluate Ringer as a whole, we developed a suite of 34 interaction benchmarks. We compared Ringer to CoScripter, an existing end-user replay tool, and found that CoSripter replayed 6 (18%) benchmarks, while Ringer replayed 25 (74%). We also tested how well Ringer handled page changes by rerunning Ringer scripts over a three-week period. Of the 24 benchmarks that ran initially, 22 (92%) continued to run at the end of the testing period.

To set expectations for how well a replayer can perform, we must acknowledge that sites are free to modify their pages arbitrarily at any time. To counteract the inherent best-effort nature of the replay problem, Ringer uses previously developed techniques [23] to specify invariants, *i.e.*, text that must appear on a page. We use these invariants to increase the confidence that replay has not diverged and as a correctness condition when evaluating Ringer.

This paper makes the following contributions:

- A record and replay approach to webpage automation that mimics a user's interactions. We record scripts that use three constructs: actions, nodes and trigger conditions.
- A node addressing algorithm based on similarity that identifies nodes across multiple webpage accesses.
- An algorithm that generates trigger conditions by observing multiple replay executions.

This paper is organized as follows. Section 2 introduces the challenges of automating websites and the approaches Ringer takes to solve them. Ringer's core language features are presented in Section 3, with the temporal problem discussed further in Section 4 and the spatial problem discussed further in Section 5. Section 6 details Ringer's implementation, and Section 7 details its limitations. In Section 8, we present an evaluation of Ringer on a set of benchmark websites. Finally, Section 9 offers a discussion of related work.

2. Challenges in Web Automation

Using a running example, we describe the inherent challenges of the web automation approaches currently available to programmers (Sections 2.1 and 2.2) and how Ringer addresses these challenges while making automation accessible to end users (Sections 2.3 and 2.4).

2.1 Web Automation Scripts Written by Programmers

Consider the task of searching Amazon for a given camera model, selecting the silver-colored camera from a list of color options, then scraping its price. Perhaps a user wants to automate this process to detect when the price drops below a fixed amount. The interest in such data is strong; Camelcamelcamel [9], which lets end-users track Amazon prices, has more than 180,000 users [21]. Although programmers have built such tools for mainstream retailers like Amazon, users cannot rely on programmers to develop price scraping

(a) Page of black camera. **(b)** Hover on silver button. **(c)** After click (grayed page). **(d)** Page of silver camera. **(e)** Sequence of events.

Figure 1: Amazon price scraping interaction. Each circle in (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 does the page request the silver price from the server and overlays to gray. The response updates the price and removes the gray overlay.

```
1  driver = webdriver.Chrome()
2  driver.get(amazonURL)
3  # Find the silver button
4  button = driver.find_elements_by_xpath('//img[@alt="Silver"]')[0]
5  button.click() # Mimic the user clicking the button
6  # Wait until the product title contains the color name
7  WebDriverWait(driver, 10).until(
8    EC.text_to_be_present_in_element(
9    (By.ID, "productTitle"), "Silver"))
10 price = driver.find_element_by_id("priceblock_ourprice")
11 print price.text # Print out price of item
```

Figure 2: Selenium script to scrape the cost of a silver camera from Amazon.com.

tools for their favorite niche sites, since writing site-specific scrapers for small audiences is not lucrative.

How would one write this price-scraping script? A programmer first studies the Amazon page to reverse engineer it, learning how to programmatically access data on the page and what DOM events must occur to load the data. We show screenshots of this task in Figure 1 and diagram the sequence of events between browser components in Figure 1(e). After producing a first draft of the script, the programmer tests it over the next few days, to adapt the script to the inevitable page changes. The final result resembles the Selenium script shown in Figure 2.

The programmer must first determine how to programmatically access webpage elements (DOM tree nodes), such as the silver camera button, which lacks a node ID. Finding a suitable selector requires examining the DOM tree and identifying features of the target node that both uniquely identify it and remain constant across multiple page accesses over time. For the silver camera image, the programmer notices that the alt field is always “Silver” and uses this insight to identify the element with the XPath expression in line 4.

The programmer must also notice the asynchrony: the script cannot scrape the price immediately after clicking the “Silver” button. Doing so would yield the price of the previously selected black camera. This incorrect outcome occurs because the page responds to the click by requesting the silver camera’s price from the server. The page grays out the content as a visual cue (Fig. 1(c)) to convey that the displayed information is invalid and a request is pending. Surprisingly, the page still lets the user interact with the stale information. For example, a user clicking the “Add to cart” button during this period will add the black camera, after already having clicked on the silver one.

The programmer must devise a trigger condition to identify when the silver camera’s price becomes available. The standard approach of waiting for the relevant DOM node to appear does not work in this case because the node is present throughout. The alternative approach of adding a fixed-time wait may break when the server is especially slow. For the Figure 2 example, the programmer opted to carefully monitor the page changes to determine that the page was ready for scraping when the product title included the word “Silver” (line 7-9). This page-specific condition requires reverse engineering that is likely inaccessible to end users.

2.2 Web Automation Script Failures

In general, one writes web automation scripts for tasks that must be performed many times. Therefore, scripts should be usable for as long as possible. However, today’s webpages are in a constant state of flux, which makes this goal a challenge. They are redesigned often; they undergo A/B testing; and they present breaking news, user-generated content, and other frequently updated content. Many pages are minimized or obfuscated, using new identifiers for their DOM nodes during

each reload. In short, the server-side code, the DOM structure and contents, and the JavaScript code can change at any time.

As an example, consider the Amazon task. The Selenium script used the expression driver.find_element_by_id("price block_ourprice") to find the current product price (line 10). Although this may seem like an intuitive approach — it uses an ID, which should uniquely identify the node — low-level attributes like this are prone to change. During one 60-second interaction with the Amazon page, we logged 1,499 ID modifications and 2,419 class modifications. These changes were caused by the page's JavaScript without the page even reloading! Similar changes can likewise occur during A/B testing or page redesign. These low-level attributes are imperceptible to the user, so a user can continue interacting normally even as they change. Because these attributes are invisible to users, sites have no incentive to keep them stable. Thus, scripts that use these frequently altered attributes often break when they fail to find the correct node.

2.3 Ringer's Approach

Amazon example in Ringer. A Ringer user starts a recording, interacts with the browser as usual, then stops the recording. The experience of recording how to complete a task is exactly the same as the experience of completing the task.

During recording, Ringer logs each action (DOM events such as mouseup, keydown, etc.) that the user makes and the DOM nodes on which those actions are performed. Here is part of the trace generated from the previously described Amazon recording:

```
observed mousedown event on an image node
observed mouseup event on an image node
observed click event on an image node
observed capture event on a span node
```

To turn this trace into a script, Ringer records hundreds of attributes for each node referenced in the trace. At replay time, the script selects the node that retrieves the highest similarity score, based on how many of a node's attributes match the original node's attributes. After the recording stage, the script looks like this:

```
dispatch action mousedown on node matching {type: 'IMG', ...}
dispatch action mouseup on node matching {type: 'IMG', ...}
dispatch action click on node matching {type: 'IMG', ...}
dispatch action capture on node matching {type: 'SPAN', ...}
```

While the hand-written script presented in Sec. 2.1 breaks upon ID changes, Ringer's approach still typically selects the correct node because the ID is only one of the hundreds of features Ringer uses.

This similarity approach succeeds in part because it resembles how a user finds nodes in a webpage. The user does not have a fixed rule in mind for finding the price node but rather looks for a node in more or less the same place as its last position or a node similar to those used in the past. This likeness may be influenced by many features – the price's font size, its position on the page, its proximity to other information. Like a human user, Ringer takes advantage of all these features. Even though we cannot predict the subset ahead of time, some subset of the features typically remain stable since the developer wants to keep the user experience consistent.

After recording, the Ringer program contains actions and reliably identifies nodes, so it can be executed. However, it will not pause between clicking the button and scraping the price, so it will scrape stale data.

To learn when to dispatch an action, Ringer replays the script several times, mimicking the user's timing. Since server response times vary, we typically observe a mix of successful and unsuccessful executions. Whether an execution is successful depends on the user's goals. To automatically detect success, the user must select invariant text that should appear during a successful replay. For the Amazon example, the user would select the title "Sony W800/S 20 MP Digital Camera (Silver)" at the end of the interaction. Ringer saves a trace of each successful replay. Each saved trace includes the relative ordering of actions and server responses during a successful execution.

Ringer then uses these execution traces to infer which server responses must arrive before Ringer can successfully replay an action. The URLs of server requests — and even the number of requests — may vary from run to run, so Ringer uses URL features to identify important server responses across runs. After trigger inference, the final script looks like this:

```
dispatch action mousedown on node matching {type: 'IMG', ...}
dispatch action mouseup on node matching {type: 'IMG', ...}
dispatch action click on node matching {type: 'IMG', ...}
waituntil server response matching hostname=='amazon.com'
    && path=='ajaxv2' && params.id=='bar':
  dispatch action capture on node matching {type: 'SPAN', ...}
```

2.4 Generalization of Scripts

Ultimately, we expect that Ringer will be used as a building block for more expressive end-user programming tools that will adapt Ringer scripts to their needs. We provide an API that lets programmers embed Ringer scripts into other applications and modify the scripts [14]. Our API lets programmers parametrize a script to interact with different nodes, to type a different string, and to open a different URL. For example, a programmer can force the script to choose a certain node by replacing the recorded feature set with a set that uniquely identifies the new node. Programmers building end-user programming tools can use this API to execute many variations on a given interaction.

For a simple example of generalization, consider a user who records how to select an item from a pulldown menu, then clicks a search button. A tool could run this script inside a loop, each time altering the script to select a different node from the menu. A more advanced generalization tool might go further, letting the user identify a relation on a set of webpages, then applying a Ringer script to all rows to scrape large datasets. In fact, WebCombine [14], a relational web

scraper targeted at non-programmers, has used Ringer to do exactly this.

3. Language Design

When a human user interacts with the browser during the recording phase, we assume that user is executing an implicit program, in which each statement takes the form "wait for X, then do Y on Z." The goal of replay is to mimic this intended user program. We propose that to faithfully replay this program, a replayer needs the following language constructs:

- **Actions**: means by which a replayer affects an application
- **Elements**: components of an application interface on which actions are dispatched
- **Triggers**: expressions that control when an action occurs

For pure record and replay, we assume the intended user program is a straight-line sequence of statements. Each statement takes this form:

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

To execute this statement, Ringer waits until all trigger conditions, $t_1, \ldots t_n$, 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.

3.1 Abstractions

This formulation gives us a program structure but does not specify what abstractions to use for the action, element, and trigger constructs. We use the following abstractions:

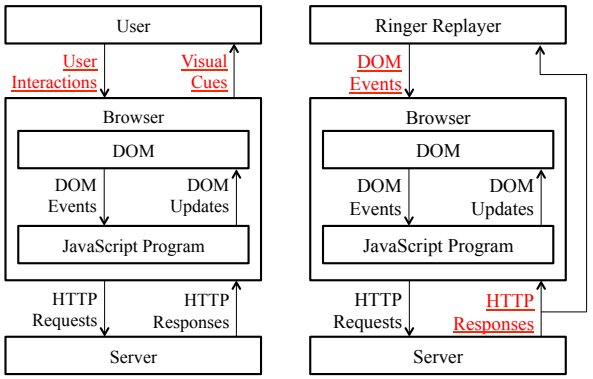
- **Actions**: Ringer records users' interactions as DOM events. DOM events are how scripts on the page listen for and respond to user interactions.
- **Elements**: DOM events are dispatched on DOM nodes, so Ringer must identify a matching DOM node on the replay-time webpage. It attempts to mimic the user through a similarity metric rather than using a fixed expression.
- **Triggers**: It is difficult to distinguish which cues are important. Often, visual cues occur in response to server responses. Therefore, Ringer uses server responses as triggers to approximate visual cues.

While it is highly unlikely that a human user applies these exact abstractions, these abstractions directly control the information a human receives from a webpage and are therefore a good fit for our user-imitation approach.

3.2 Ringer System Architecture

With this language in place, we can build the full record and replay system, pictured in Figure 3, to produce scripts in the Ringer language from user demonstrations.

Recording. During recording, Ringer observes all DOM events, and the DOM-level features of all involved nodes.



(a) User **(b)** Ringer

Figure 3: Ringer architecture. The Ringer replayer becomes a replacement user. We highlight and underline the messages used to monitor and control the browser. Ringer directly produces DOM events to mimic user interactions and observes HTTP responses directly.

This trace is the input to our trigger inference algorithm, which uses it to produce 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.

Replay. During replay, Ringer executes each statement. It first checks if all trigger expressions are satisfied by comparing each expression to the list of server responses observed thus far. Once all expressions are satisfied, the replayer uses the DOM features to identify a node and then dispatches the DOM event on that node.

4. Trigger Inference

As a human uses visual cues to synchronize with webpages, Ringer must also synchronize. Without synchronization, an execution could scrape the wrong information, like the Amazon scraping script from Section 2; or, worse, it could silently cause undesired side-effects on a page. To solve this problem, we infer triggers, which pause replay execution until the webpage is ready for the next action.

4.1 Design Rationale

The simplest triggers wait a fixed amount of time, but this approach makes scripts slow and susceptible to network delays. The most user-like option is to look for visual cues to construct expressions like line 4 in Figure 2. However, the high frequency of visual changes makes it difficult to isolate relevant cues. The Amazon example uses a page that contains over 4,000 nodes, and more than 600 DOM changes occur after the user clicks the button, only a few of which are associated with page readiness. The large number of nodes and changes creates a large space of candidate expressions.

Instead of using visual cues, we look to what pages are usually signaling with these cues, *i.e.*, outstanding server requests and responses. The Amazon page contains only

35 such responses, so inferring response-detecting trigger expressions is feasible.

4.2 Example

To infer trigger expressions, Ringer must: (i) align server responses across multiple replays, and (ii) identify trigger-dependent actions. We could find no existing algorithms for this problem and therefore designed a simple but effective one.

We illustrate our approach on a simplified version of the Amazon example. Initially, the user provides a demonstration of the interaction, creating a simple script with two actions:

action a_1	click silver button
action a_2	scrape price

The user must also provide a correctness condition so that Ringer can automatically detect when a replay is successful. The user does this by selecting the title text,"Sony W800/S 20 MP Digital Camera (Silver)," as an invariant, indicating that any successful execution must display this text. Ringer then automatically (without user intervention) replays this script. During these training-time replays, Ringer mimics the user's timing, so we call these *naive* replays. Ringer continues to replay the script until it has observed two successful executions:

action a_1	click silver button
response r_1	`www.amazon.com/ajaxv2?rid=foo&id=bar`
response r_2	`www.amazon.com/impress.html/ref=pba`
action a_2	scrape price

action a_1	click silver button
response r_3	`www.amazon.com/ajaxv2?rid=baz&id=bar`
action a_2	scrape price
response r_4	`www.amazon.com/impress.html/ref=pba`

Aligning server responses. To infer the dependency relation, we must identify common server responses across multiple traces. We use URLs' hostnames and paths for this purpose. For our example, we identify r_1 and r_3 as the same response even though the URLs differ (with different rid parameters) since they share a hostname and path. To increase precision, we incorporate the URL parameters that remain constant, such as the id parameter. Thus, the final expression that identifies the r_1 and r_3 responses is: hostname=='amazon.com' && path=='ajaxv2' && params.id=='bar'.

Identifying trigger-dependent actions. Ringer must also infer which server responses an action requires. Let t_1 and t_2 be the trigger expressions that identify $\{r_1, r_3\}$ and $\{r_2, r_4\}$, respectively. With these traces, Ringer can safely infer that a_2 depends only on t_1. Since the second trace shows a successful execution in which a_2 was dispatched before r_4, it is clear a_2 does not depend on t_2.

Some assignments of trigger expressions to actions cause more synchronization than others. If Ringer observes only the first trace, it cannot eliminate the possibility that a_2 depends on t_2: it would associate a_2 with both t_1 and t_2. This suffices to replay the script, but it causes Ringer to wait for t_2 unnecessarily. In a better program, with less synchronization, a_2 waits only for t_1. Given the second trace, Ringer can infer this better program.

```
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] ← InferTriggers(responses)
11   mapping ← mapping ∪ {action → triggers}
12   used ← used ∪ triggers
13 return mapping
```

Figure 4: Algorithm to associate actions with necessary triggers using a set of successful executions.

4.3 Identifying Trigger-Dependent Actions

We developed our trigger inference algorithm around one key insight: if an action a depends on a response r, then r must appear before a in all successful traces.

Our algorithm seeks to add triggers to a Ringer program so that the script never fails due to an action being replayed early. This correctness condition is met if the trigger expressions added by our algorithm enforce all dependencies between responses and actions. The secondary goal is to minimize the delay caused by the triggers. We should avoid waiting for a server response unless the action must come after the response.

The input to Ringer's trigger inference algorithm is a sequence of actions (a Ringer script without triggers) and a set of successful traces. Each successful trace is a sequence of actions and responses. The output is a mapping from actions to the triggers they require.

For each action, the algorithm identifies the set of server responses that occur before the action in all passing traces. The only proof that no dependency between an action and a response is a successful run in which the action precedes the response. Therefore, we conservatively assign a maximal set of trigger expressions.

The AddTriggers algorithm in Figure 4 iterates over all actions (line 3), identifying the responses that happen before the current action in each execution (lines 6 to 7). It then removes responses that are already associated with previous actions (line 8). We run InferTriggers on the

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

Figure 5: Grammar of response identification expressions.

remaining responses (line 10). INFERTRIGGERS returns a set of trigger expressions, one for each semantically equivalent server response that appeared in all executions. For instance, from our earlier example, we would run INFERTRIGGERS on $[r_1, r_2], [r_3]$, and it would produce t_1, the set containing a trigger expression that corresponds to r_1 and r_2. Finally, we map the current action to the set of trigger expressions (line 11), and the loop continues.

4.4 Aligning server responses

To use server responses as triggers, we must find expressions that identify them across multiple executions. This is difficult because different URLs can represent semantically equivalent responses. For example, a response can use a different session ID parameter for each execution. We also cannot assume that all server responses seen in one execution will occur in another. The goal is to produce an expression flexible enough to identify semantically equivalent responses despite variations of the URL, but restrictive enough to not also identify semantically different server responses that can have similar URLs.

Ringer infers trigger expressions in the grammar shown in Figure 5. We use features of server responses' URLs, including the hostname, path, query parameters, and type of response. A trigger expression evaluates to true if Ringer witnesses a server response that matches each URL feature in the expression. If the trigger expression contains an isAfter(id) clause, then Ringer must receive the response after executing action id.

Our INFERTRIGGERS function takes lists of responses as input. Each list contains responses from a single trace that were not used as triggers for any previous action. For instance, in the Amazon example, the list was $[r_1, r_2], [r_3]$. The output is a set of trigger expressions, one for each semantically equivalent response that occurs in all input lists. In this example, r_1 and r_3 were equivalent.

To produce the output trigger expression, we must first identify which responses are semantically equivalent. Our algorithm makes a (hostname, path, type) tuple for each response and checks whether any response tuple appears in all input lists. If it does, the algorithm produces a trigger expression for this set of responses, including all parameters that the URLs have in common.

If a given trigger expression has already been used to identify a previous response, we add an isAfter clause, indicating that this trigger applies only after a previous event. This ensures that a single response does not satisfy more than one trigger expression.

4.5 Assumptions

Our approach relies on four core assumptions.

1. **Slow is safe.** We assume that it is always acceptable to delay an action, but that replaying it too early could cause failures. Since human users may be slow to react, this assumption holds on almost all pages we examined.
2. **Arbitrary waits are rare.** Our approach cannot handle interactions that demand waits of arbitrary time periods (*e.g.*, interactions with JavaScript's setTimeout). Since users are impatient, it is rare for developers to add such waits.
3. **Correctness criteria have no false positives.** Believing a failing trace is successful can lead our algorithm to eliminate required dependencies. Therefore, the correctness criterion must accurately determine whether an execution is successful.
4. **We can collect enough traces.** Because we prioritize correctness, Ringer never eliminates a dependency between a response and an action without witnessing it to be superfluous. If we observe too few executions, Ringer could produce scripts that wait unnecessarily. Even with our conservative approach and two or three successful traces, we can significantly reduce replay execution time (Sec. 8.3), so it is important that we be able to collect enough traces to eliminate potential response-action dependencies. Another danger is that our inferred trigger expressions could overfit a small number of input traces, causing Ringer to ignore semantically equivalent responses. We handle these cases by adding a timeout, so that replay eventually continues even if no server response matches an overfit trigger expression.

5. Node Addressing

Every action observed during a recording is enacted on a node. For a replay script to execute correctly, it must dispatch the action on the corresponding replay-time node.

5.1 Problem Statement

The DOM is a tree of nodes. A node maps attributes to values. At time t_1, we load url u, which yields DOM tree T, and we observe a given node with m attributes $n = \langle a_1 : v_1, ..., a_m : v_m \rangle$ in T. At time t_2, we load u, which yields DOM tree T', and must identify the node $n' \in T'$ that a user would identify as corresponding to $n \in T$.

This problem is difficult because T' can be arbitrarily different from T. The structure of T, the attributes of n, and the attributes of other nodes in T could all change. In the limit, T' could have a wholly different structure than T and not have any common nodes with T. If this happened regularly,

identifying nodes would be almost impossible. However, this would also make replay difficult for human users. The need to offer a consistent user interface suggests that in practice there is often substantial similarity between the DOM trees of the same page loaded at different times.

5.2 Past Approaches

Past record and replay tools and node addressing algorithms – such as iMacros [6], ATA-QV [36], XPath relaxation [15, 17, 26] and CoScripter [27] – solved the problem by selecting at record time what features they would require at replay time. Given T and n, these *node addressing* algorithms construct a function f such that $f(T) = n$. To find a corresponding node on a new page T', they apply f, proposing $f(T')$ as n'. Typically, f uses some combination of the attributes of n. Based on the designer's insights into how webpages change, they select a few important attributes that uniquely identify n in T.

To make these approaches more concrete, we briefly describe two such node addressing algorithms. The iMacros [6] approach records the text of n and an index attribute – the number of nodes before n in T that have the same text. The combination of text and index uniquely identifies n in T and produces at most one node in any other DOM tree.

The ATA-QV algorithm [36] is more complicated. At record time, it finds all nodes with the same text attribute as n. It then compares the subtrees containing these nodes to find text nodes that exist in n's subtree but not in others. It recursively accumulates a list of these disambiguating texts, which let it uniquely identify n in T, although they may not guarantee a unique output on variations of T. Thus, for ATA-QV, the text within a node's subtree serves as the crucial attributes. At replay time, ATA-QV finds the node whose subtree includes the target set.

5.3 Our Approach

Rather than constructing a function $f : T \rightarrow n$ at record time that discards most information about n, our approach builds a function SIMILARITY for measuring the similarity between two nodes. At record time, we save n. At replay time, for each candidate node $n_c \in T$, we run SIMILARITY(n, n_c). The node with the highest score is selected.

Past tools calculate a small set of features at record time and require that they all match during replay. This assumes that all of these features will remain stable. In contrast, our approach requires that only some subset of features match.

The features we use include per-node features: attributes of the node object; getBoundingClientRect features, like width; getComputedStyle features, like font; and portions of the node text. In addition, we also include many features with information about the surrounding subtree: selected XPath and XPath-like expressions and features of parent, child, and sibling nodes. These latter features let Ringer use the context of the original node to find a similar node.

```
SIMILARITY(weights : Map[Attribute, Weight],
  n : Map[Attribute, Value], n_c : Map[Attribute, Value])
1 score : Number ← 0
2 for attribute : Attribute ← n do
3   if n[attribute] == n_c[attribute] do
4     score ← score + weights[attribute]
5 return score
```

Figure 6: Algorithm to calculate the similarity of candidate node n_c to node n.

We constructed three similarity algorithms. All take the same basic form, applying the core SIMILARITY algorithm in Fig. 6. For each attribute a in the set of all attributes, if $n[a] == n_c[a]$, the score is incremented by the attribute's weight. The node with the highest score is the output node.

Each variation of the SIMILARITY algorithm uses different weights. The first algorithm weights all attributes equally. For the second and third algorithms, we produced weights using machine learning, one using linear regression and one using SVM with a linear kernel (details in Appendix A). Surprisingly, we found that the algorithm that weighted all attributes equally achieved the best performance. This result indicates that past changes to a website are not good predictors of future changes and supports our claim that using a fixed set of features is fragile.

5.4 Benefits of Our Approach

The key benefit of our approach is an increased robustness to page changes over time. As detailed in Sec. 5.2, past approaches rely on each feature in a small subset of features to remain stable over time. The chosen features are essentially a heuristic, guided by the designers' instincts about how pages change over time. Unfortunately, even when designers choose a good subset that has been consistent in the past, past performance does not guarantee that they will remain consistent in the future.

Let us consider how past techniques handle the Amazon task. The user's goal is to scrape the price as it changes over time. How do the iMacros and ATA-QV node addressing techniques identify the price node? Both techniques first filter for nodes with the original node's text, which is the price observed during recording – the stale data! If the price has changed, there is no such node, and the tools fail.

In contrast, our similarity approach loses only one matching attribute when the price text changes. reducing the similarity score by only one. Other attributes – from background color to font family, border thickness to node height – still match the original node. So even as the price changes, our similarity-based approach finds the correct price node.

6. Implementation as a Chrome Extension

Ringer is implemented in JavaScript, which offers portability, and is distributed as a stand-alone Chrome extension,

which offers easy deployment. We made our code publically available at `https://github.com/sbarman/webscript`.

Recording and replaying actions. Ringer records actions by recording DOM events. To intercept and log each event, our Chrome extension content script adds Ringer's event listeners to all important DOM events before the recorded page loads. (By default, we exclude high-frequency `mousemove`, `mouseover`, and `mouseout` events although these could be enabled as desired.) Our listeners are called during the capture phase so that they are executed before other listeners; any listener could suppress the execution of later listeners, which would hide events from the recorder.

At replay time, we recreate the observed events and raise them with the dispatchEvent function, carefully handling two subtleties. First, the browser distinguishes between events dispatched natively (*i.e.*, via user interactions) from those dispatched from a JavaScript program, including from an extension. The former events are trusted and are allowed to cause side-effects; some effects of the latter are ignored. For example, clicking a checkbox node `n` with a mouse sets `n.checked` to true. In contrast, dispatching the `click` event to `n` from JavaScript has no effect on `n.checked`.

Second, the replayer must avoid breaking an implicit invariant established by JavaScript's single-threaded and non-preemptive semantics. In particular, the browser dispatches some consecutive events atomically, preventing non-related events from executing their handlers within the atomic block. One such group of events is `keydown`, `keypress`, `textInput` and `input`. We found that at least one website (Southwest.com) relied on this behavior: An early version of our replayer allowed a scheduled function to be executed in the midst of an atomic block. As a result, the autocomplete menu triggered by the typing did not use the last character the user typed.

To handle these challenges, we developed a runtime system that actively corrects any inconsistencies between when an event is recorded and when it is replayed. Details of this system are beyond the scope of this paper but can be found in [11].

Observing triggers. Ringer uses the Chrome webRequest API to receive notifications of all requests to the server as well as all server responses. During recording, we use these notifications to collect the set of candidate triggers. During replay, we use these notifications to identify when triggers have fired. Ringer does not control when the browser sends or receives a request. Instead, during replay, Ringer delays dispatching events to control the order of actions relative to server notifications; this is the trigger mechanism described in Section 4.

Identifying nodes. Ringer's event listeners also record information about the event's target node and relevant adjacent nodes. Attributes of these nodes are used in Ringer's node addressing algorithm. The recorded information includes a mix of XPath expressions, CSS selectors, text, coordinates, traversals of the DOM tree, and many other features.

7. Limitations

We now review the limitations of Ringer's components.

Actions. Some DOM events – for example, mousemoves, mouseovers, and mouseouts – occur at a very high rate. Because JavaScript is single-threaded, the same thread that records and replays each event must also processes webpage interactions, so recording the very large number of high-frequency events can make pages slow to respond. Therefore, Ringer does not record these events unless high-frequency mode is explicitly turned on by the user. For most pages we encountered, these events were unnecessary.

Elements. The similarity-based node addressing approach is inherently best-effort. We obtain no theoretical guarantees but find in practice that the approach is sufficient.

Triggers. The Ringer approach was designed for interactions that satisfy the trigger assumptions (see Section 4.5). Ringer fails when these do not hold. For instance, Ringer is not intended to target interactions that require precise absolute times between actions, such as most games. As a concrete example, consider Google Maps, which calculates "inertia" during map movements to continue scrolling even after the user releases the click. It does so by using the time between mouse movement actions. Since Ringer does not reproduce exact times between actions, it cannot replay this interaction.

Another possible source of failures is client-side delay, such as delays from animations, timeouts, or the use of browser-local storage. Since these delays do not occur because of server requests, Ringer's trigger algorithm will not infer them. In practice, we have not observed websites that fail because of client-side delays.

8. Evaluation

We start with an end-to-end comparison of Ringer and CoScripter, another by-demonstration replayer for end-user programming. We show that our design decisions make Ringer more reliable on modern websites (Section 8.1). We also examine how well Ringer handles page changes by executing Ringer scripts repeatedly over a period of three weeks (Section 8.2). We evaluate the components of replay in isolation, showing that triggers are indeed necessary for correct replay (Section 8.3). We evaluate node identification in isolation from the full Ringer system, showing that it outperforms existing algorithms in robustly handling page changes over time (Section 8.4).

Benchmarks. We designed a benchmark suite consisting of one web navigation scenario for each of 34 sites taken from Alexa's list of most-visited sites[4]. These websites tend to be complex, making heavy use of AJAX requests, custom event handlers, and other features that challenge replay.

Each interaction completes what we perceive to be a core site task, such as buying a product from an online retailer. Each benchmark comes with user-specified invariant text used to automatically detect whether the execution has been successful. For instance, the Walmart benchmark succeeds if the final page includes the string "This item has been added to your cart."

The websites, interactions, and invariants were selected by one of the authors. The author selected familiar websites in order to choose a task and an invariant for each benchmark. We excluded from consideration any tasks that lacked checkable correctness criteria. While the benchmarks were chosen without knowing if Ringer would work on them, there was the possibility of selection bias due to the author's knowledge of Ringer's limitations.

8.1 Comparison to CoScripter

CoScripter[27] is a web record and replay tool built to automate business processes. We chose to compare against CoScripter because it is the only existing web replay tool that can be used by non-programmers. While other replay tools let users draft scripts by providing recordings, the drafts are too fragile to use directly. Users must understand code and the browser well enough to fix the scripts produced by those replayers.

Unlike Ringer, CoScripter's scripts are human readable. This feature comes at a cost: CoScripter's actions are limited to a set of predefined operations, such as clicking an element or typing text. This approach, successful when webpages were less interactive, fails on modern interactions, *e.g.*, the use of an autocomplete menu, where the page reacts to a combination of mouse and keyboard events to implement custom behavior; successful replay requires faithfully reproducing these fine-grain events. Our evaluation confirmed that using a small set of high-level actions limited the interactions that can be replayed, and that faithfully mimicking all fine-grained user actions was more successful.

Procedure. To test Ringer, we recorded each benchmark interaction and selected invariant text. We then replayed the script 5 times, using the invariant text to determine whether the replay succeeded. If all 5 replays succeeded, we considered Ringer's replay successful on the benchmark. We tested CoScripter manually, so each script was replayed only once. Thus for CoScripter, we marked a single correct replay as a success. We performed all replay executions within one day of the recording.

Results. We found that Ringer succeeded on 25 of the 34 benchmarks (74%), while CoScripter successfully replayed 6 benchmarks (18%). In Table 1, columns **Ringer** and **CoScripter** present the results of our experiment.

Of the 9 benchmarks on which Ringer failed, 5 are complete failures for which Ringer never produced a successful replay. The other 4 are partial failures; Ringer replayed the script successfully at least once. We provide a sample of the reasons why Ringer replay failed:

- **ebay**: The script reached an item for which bidding had already ended, preventing a bid from being placed.
- **g-docs**: Ringer could not identify the correct node.
- **paypal**: The recorded script did not include the login interaction. (For most pages, the need to record login is rare, but PayPal logs users out of their accounts very quickly.)
- **webmd**: The site used a Flash interface, which Ringer does not support.

Our post-mortem analysis of CoScripter's results revealed two common failures. First, some scripts failed because CoScripter did not identify the correct target elements on replay-time pages (column **Element** in Table 1). We do not consider this a fundamental flaw as it may be fixed by substituting our node identification algorithm. The second limitation is fundamental and results from the decision to use a small number of predefined human-readable actions. The lack of fine-grain actions makes CoScripter incapable of correctly replaying some interactions (column **Interaction**). For instance, on the Google search page, the user typed a partial string "SPLA" and then clicked on an option from an autosuggest menu. CoScripter recorded this interaction as "enter 'SPLA' into the 'Search' textbox." Without the click, the query is not executed, which causes the replay to fail. While CoScripter could add new actions for more webpage idioms, such as AJAX-updated pulldown menus, such a scheme is unlikely to keep pace with the rapid evolution of web design.

Of the benchmarks marked **Other**, two failed because CoScripter did not faithfully mimic user key presses. When a user types a string, the webpage dispatches a sequence of DOM events for each key press. However, CoScripter only updated the node's text attribute, without dispatching individual key presses. Skipping keypress event handlers caused the executions to diverge. The final benchmark failed because CoScripter did not wait for a page to fully load.

8.2 Performance Over Time

We evaluated how well Ringer scripts handle page changes by running the scripts repeatedly over a period of three weeks.

Procedure. We used the same set of benchmarks that we used for the comparison to CoScripter. We repeatedly replayed each script to determine if the script remained successful at each point. We ran each benchmark once every one to four days, over the three week period.

Results. We found that of the original 24 passing scripts, only 3 experienced failures on more than a single testing date. The other 21 scripts were generally successful, with 15 experiencing no failures and 6 experiencing only one failure

Site	Description	# Events	Time (s)	Ringer	CoScripter	Element	Interaction	Other
allrecipes	find recipe and scale it	130	22	✓	×	×	×	
amazon	find price of silver camera	76	22	✓	×		×	
best buy	find price of silver camera	58	13	✓	×		×	
bloomberg	find cost of a company's stock	27	5	✓	×		×	
booking	book a hotel room	75	21	✓	×		×	
drugs	find side effects of Tylenol	38	13	✓	✓			
ebay	place bid on item	67	36	×	×	×		
facebook	find friend's phone number	47	9	3/5	×	×	×	
g-docs	add text into new document	15	6	×	×		×	
g-maps	estimate time of drive	121	35	✓	×		×	
g-translate	translate 'hello' into French	48	12	✓	×		×	
gmail	compose and send email	159	22	✓	×	×		
goodreads	find related books	88	27	✓	✓			
google	search for a phrase	29	4	✓	×		×	
hotels	book a hotel room	75	17	✓	×			×
howstuffworks	scrape off common misconceptions	32	11	✓	✓			
kayak	book a flight	74	19	✓	×		×	
linkedin	view connections sorted by last name	12	16	✓	×	×		
mapquest	estimate time of drive	106	15	✓	×		×	
myfitnesspal	calculate caloriess burned	102	23	✓	×			×
opentable	make a reservation	69	24	4/5	×		×	
paypal	tranfser funds to friend	150	30	×	×	×		
southwest	book a flight	93	15	✓	×		×	×
target	buy Kit Kats	70	10	✓	✓			
thesaurus	find synonyms of "good"	39	9	✓	×	×		
tripadvisor	book a hotel room	28	5	×	×		×	
twitter	send a direct message	133	36	✓	×	×	×	
walmart	buy Kit Kats	51	19	3/5	×		×	
webmd	use symptom checker	34	17	×	×		×	
xe	convert currency	51	24	✓	×		×	
yahoo	find cost of a companies stock	30	3	2/5	✓			
yelp	find and filter restaurants in area	57	28	✓	×		×	
youtube	find statistics for video	97	19	✓	✓			
zillow	find cost of a house	90	19	✓	×		×	

Table 1: We present the results of running our benchmark suite. **# Events** gives the number of DOM events recorded, and **Time** gives the duration in seconds. **Ringer** shows how well Ringer performs immediately after the recording. **CoScripter** shows how well CoScripter performs immediately after recording. A ✓ indicates all replay runs succeeded, and an × indicates all runs failed. A partial failure is denoted by "# successes / # runs." Ringer outperforms CoScripter, replaying 25 interactions correctly vs. 6 interactions. For every benchmark that CoScripter failed to replay, we diagnose a root cause for that failure, shown in columns **Element**, **Interaction**, and **Other**. **Element** indicates that CoScripter misidentified an element. **Interaction** indicates that replay failed due to an improperly recorded user interaction.

	Day																					
	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
allrecipes	✓			✓		✓	✓		✓			✓	✓	✓	✓				✓			✓
bloomberg	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
drugs	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
facebook	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
g-maps	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
g-translate	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
hotels	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
howstuffworks	✓				✓			✓	✓			✓	✓	✓	✓				✓			✓
linkedin	✓				✓			✓	✓			✓		✓	✓				✓			
mapquest	✓				✓			✓	✓			✓		✓	✓				✓			
southwest	✓				✓			✓	✓			✓		✓	✓				✓			
tripadvisor	✓					✓		✓			✓	✓		✓	✓		✓		✓			
twitter	✓					✓		✓			✓		✓	✓	✓		✓		✓		✓	
yelp	✓					✓		✓				✓	✓	✓	✓				✓		✓	
youtube	✓					✓		✓				✓	✓	✓	✓				✓		✓	
goodreads	✓				✓			✓	✓			✓	✓	✓	✓				4/5			✓
google	✓				✓			✓	✓			✓	✓	✓	✓				3/5			✓
opentable	✓				✓			✓	✓			✓		✓	1/5				✓			
zillow	✓					✓		✓				✓	4/5	4/5	✓				✓		✓	
thesaurus	✓				✓	✓		✓			✓	3/5	✓	4/5			4/5		✓			
target	✓				✓			✓	✓		✓	×	✓	✓			✓		✓			
best buy	✓			✓				×	✓			✓	✓	✓	✓				✓			✓
yahoo	✓					✓		✓			✓		✓	✓	✓				✓		×	
xe	✓					✓		✓			4/5		4/5	✓	×				×		×	
kayak	3/5				3/5			4/5	4/5			✓		4/5	✓				2/5			
myfitnesspal	3/5				✓			2/5	4/5			4/5		2/5	✓				3/5			
booking	×				×			×	×			×	×	×	✓				✓			✓
ebay	×				×			×	3/5			×	×	×	×				×			×
amazon	×			×				×	×			×	×	×	×				×			×
g-docs	×				×			×	×			×	×	×	×				×			×
gmail	×				×			×	×			×	×	×	×				×			×
paypal	×				×			×	×			×		×	×				×			
walmart	×					×		×			×		×	×	×				×		×	
webmd	×					×		×			×		×	×	×				×		×	
Passing	24	24	24	24	25	25	25	23	24	24	23	22	23	22	24	24	24	24	22	22	21	23

Table 2: We present the results of running our benchmark suite over a period of three weeks. Each column heading indicates the number of days since the initial recording. Text in a cell indicates that the benchmark was run that day. A ✓ indicates that all replay runs succeeded and a × indicates all runs failed. A partial failure is denoted by "# successes / # runs". Benchmarks are grouped by their status at the start of testing and whether their status changed over the 3 week period. The last line counts the number of passing benchmarks, treating a benchmark as passing if the latest run of the benchmark passed. The first segment of the table includes all scripts that produced perfect success rates at every run. The second segment of the table includes scripts which produced perfect results at the start, but experienced any kind of failure during the test period. The third and fourth segments include all scripts that failed to produce 5/5 correct replays at the start – that is, these are the scripts that would be considered failures in the comparison against CoScripter section. The third segment includes scripts whose performance varied at all during the course of the test period. The fourth includes scripts that consistently produced 0/5 correct replays throughout the test period.

over the three week period. Table 2 presents the results of our experiment.

We used Day 3 as the initial baseline, since our testing framework experienced failures during Days 1 and 2.

Note that since we conducted the CoScripter experiment and the longitudinal study months apart, the sites changed in the intervening period. Therefore, we do not expect the first day performance to correspond exactly to the first day performance in the CoScripter experiment. Indeed, 9 benchmarks produced different results, even though the rates of success are similar at 25/34 and 24/34.

Of the 24 scripts that initially succeeded in 5/5 runs, 9 experienced some type of failure during the testing period. For 7 scripts, these failures were temporary. We did not diagnose all failures but did determine that some (including the unusual **target** failure) were caused by rate limiting. Only 2 scripts experienced failures that lasted until the end of the testing period.

Of the 7 scripts that failed temporarily, only 2 experienced complete failures, meaning only 2 produced 0/5 successful replays on any day. The remaining 5 experienced partial failures, meaning that the script continued to replay successfully at least once (and often more) each day.

Overall, with 20 out of 24 scripts producing at least one successful replay on every single test date, the results indicate that Ringer scripts are fairly robust to page changes over a three week time period.

8.3 Trigger Inference

We evaluated whether interactions require synchronization; whether Ringer learns sufficient triggers to mimic users' interpretation of visual cues; and how much speedup we obtain by replaying actions as soon as triggers are satisfied.

Procedure. We used a subset of benchmarks from the CoScripter experiment. Each benchmark is executed in four Ringer configurations: the **user-timing** configuration waited as long as the user waited during the demonstration. The **no-wait** version dispatched events as soon as possible, only pausing when the target node has not yet been created by the webpage program. The **2run-trigger** and **3run-trigger** versions used triggers inferred from two and three traces, respectively. We ran each configuration 10 times.

Results. In Figure 7, the x-axis gives the percentage of runs that succeeded, and the y-axis shows speedup compared to **user-timing**. The ideal replayer would pair perfect correctness with maximum speedup, placing it at the right and closest to the top.

The **no-wait** version gives a theoretical ceiling on speedup. Overall, the **3run-trigger** versions of each benchmark had an average speedup of 2.6x while the **no-wait** version had an average speedup of 3.6x.

The **user-timing** version shows that the original script generally succeeds if the user was sufficiently slow and the server was sufficiently fast.

For 9 out of 21 benchmarks, trigger inference was not needed since the **no-wait** version succeeded at least 90% of the time. This is not to say no synchronization was needed, but often waiting for the target node 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 more successful than the **no-wait** version.

For two benchmarks, all of the **no-wait** executions succeeded while the trigger versions succeeded less than 90% of the time. Since we automate our tests, it is difficult to diagnose these failures; possible reasons include bad trigger conditions, misidentified elements, or the server being down.

Understanding synchronization. To understand why the **no-wait** synchronization is fragile, we examined three benchmarks where Ringer succeeded and **no-wait** frequently failed. **paypal**: If a click was dispatched prematurely, replay misidentified an element, clicking the wrong link and navigating to an unknown page. **yelp**: Without a sufficient delay, replay clicked a restaurant filter option while a previous filter was being applied, causing the page to ignore the second filter. **target**: If replay did not wait for the page to fully load, the page froze after a button click, most likely due to a partially loaded JavaScript program.

8.4 Node Identification

Our final experiment tested our node identification algorithm in isolation from Ringer. We compared our similarity-based algorithm to the state of the art and found that our approach outperformed existing node addressing 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 identify the corresponding node, we would have solved the original problem. Consider a webpage with a list of blog posts; each post starts at the top and is shifted down as new entries appear. Let our record-time target node n be 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 choose which node they intended to select.

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 clicking on both nodes causes navigation, possibly to different pages. The upper bound handles cases like the blog described above. Clicking on the top post may be the

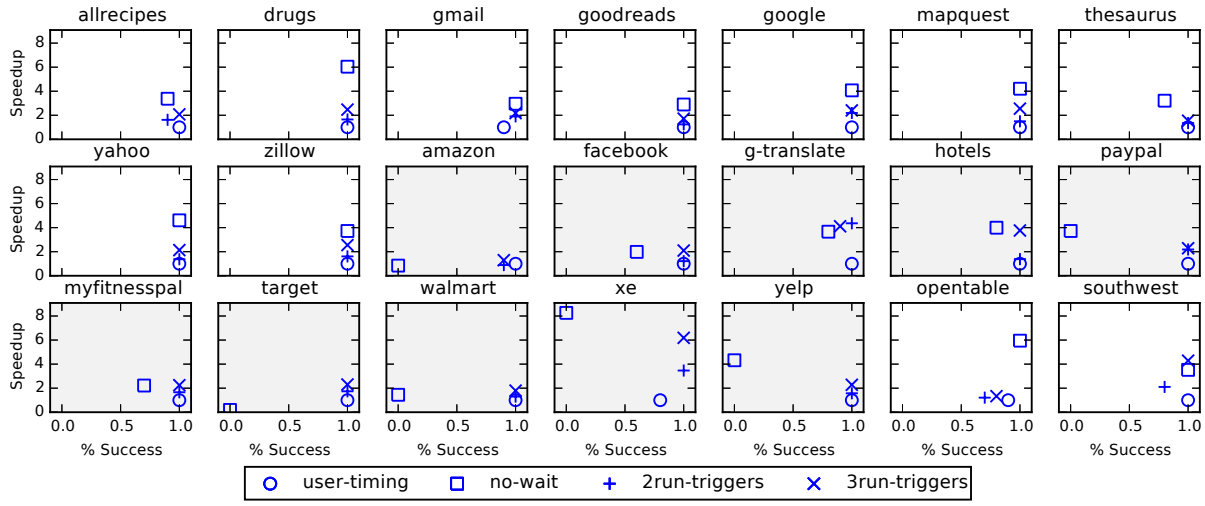


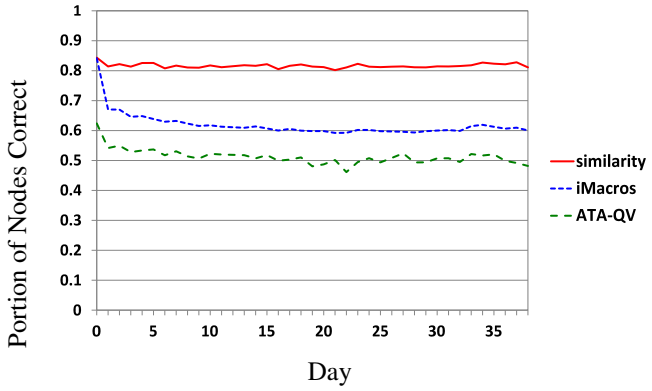
Figure 7: Success rate v. speedup on a set of benchmarks. Benchmarks with shaded backgrounds require synchronization.

right action, but we should expect it to lead to a different URL. Of course, with these bounds we can only test the node addressing 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.

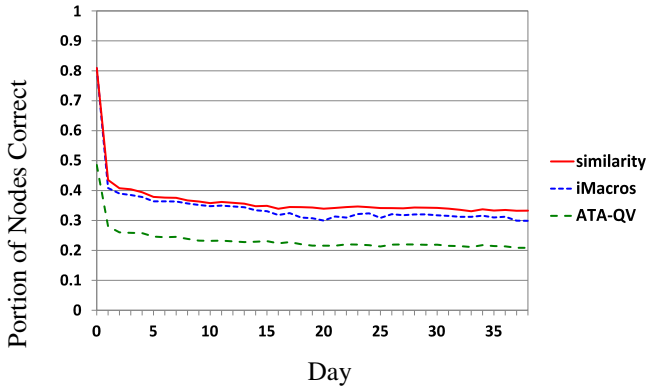
Procedure. We used our upper and lower bounds to test our **similarity** algorithm against the **iMacros** [6] and **ATA-QV** [36] algorithms on a dataset of all 5,928 clickable DOM nodes from the 30 most popular websites, according to Alexa rankings [4]. On Day 0, for each webpage, we programatically clicked on every node and recorded the destination URL. We kept only nodes that caused navigation. Once per day, we tested whether each algorithm could find each node.

Results. Figure 8 shows the percentage of nodes on which each algorithm succeeded each day, using both metrics. Using the upper bound metric, **similarity** consistently outperformed the other approaches. On day zero, performance is comparable across approaches, with **similarity** slightly outperforming **iMacros** and **iMacros** outperforming **ATA-QV**. After the relative consistency of the day 0 performance, **similarity** begins to substantially outperform **iMacros**. Already on the first day, **similarity** exhibited 1.21x the success rate of **iMacros** and 1.50x the success rate of **ATA-QV**. By the one month (Day 31) mark, **similarity** exhibited 1.35x the success rate of **iMacros** and 1.60x the success rate of **ATA-QV**. In absolute terms, **similarity** succeeded on 81.4% of the clickable nodes, while **iMacros** succeeded on only 60.2%.

The performances of the **similarity** and **iMacros** approaches are much more similar when judged with the lower bound metric, although **similarity** produces a better lower bound than **iMacros** on every date. On the first day, **similarity** produces a success rate of 1.06x **iMacros**' success rate



(a) Upper bound on node addressing performance.



(b) Lower bound on node addressing performance.

Figure 8: Performance of state-of-the-art node addressing algorithms against our similarity approach.

and 1.55x **ATA-QV**'s success rate. By the one month mark, **similarity** produces a success rate of 1.08x **iMacros**' success rate and 1.58x **ATA-QV**'s success rate. Although these 8% and 58% improvements are not as large as the 35% and 60% improvements suggested by the more permissive acceptance criterion, we are satisfied that by either metric, **similarity** represents an improvement over the alternative methods.

Similarity weights. The **similarity** approach in this experiment was the SIMILARITY algorithm with uniform weights. In additional experiments, we tested this variation against the SVM and regression versions, and found that the uniform weights performed best. This result occurs even though we gave the machine learning versions the advantage of being tested on the same websites that were used to train them. The training and testing data were from the same sites, but different time periods. 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 node addressing 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 that can adapt to the largest variety of changes, produces the best performance.

8.5 Ringer as a Building Block

Ringer has already been used to construct several tools that treat replay as a subroutine. For instance, the relational web scraping tool WebCombine [14] uses Ringer to infer scripts that are then parameterized into functions and invoked to collect complex online datasets, *e.g.* a 3 million row database of all papers published by the top 10,000 authors in Computer Science. WebCombine was also used to collect the StackOverflow statistics cited in Section 1.

We have used Ringer to build a tool for authoring custom homepages with "live information tiles." Non-programmers demonstrate how to scrape the information they want to see – when the bus arrives, whether a package has been delivered – and the homepage runs Ringer scripts to populate the tiles with the most recent information. Another application resembles Greasemonkey [10], allowing users to modify webpages by running small Ringer scripts.

9. Related Work

Web automation. Many previous tools allow users to automate the browser through demonstration. The CoScripter [27] line of research focuses on making web scripting accessible to all. This line also includes Vegemite for creating mashups [29], and ActionShot [28] for representing a history of browser interactions. CoScripter works at a higher level of abstraction, making the assumption that its small language of actions represents all possible interactions. But this language cannot faithfully capture all interactions on modern pages, causing it to break in unexpected ways.

Other tools offer record and replay systems as a way to obtain a rough draft of the target script. This simplifies the script writing process but not to remove the programming component altogether. iMacros [6] is one such commercial tool. The browser testing framework Selenium [8] is another. Its Selenium IDE component offers a record and playback functionality. However, both iMacros and Selenium produce scripts that a programmer must edit. Both of these tools are targeted at experienced programmers, whose alternative is to write such scripts from scratch.

Another class of tools aims at streamlining the process of writing scripts. Chickenfoot [12] lets users combine high-level commands and pattern-matching to identify elements. Sikuli [37] uses screenshots to identify GUI components. Beautiful Soup [5] is a Python library for interacting with webpages, and libraries such as OXPath [19, 20, 24] and Scrapy [7] are aimed specifically at scraping. While these tools do simplify webpage automation relative to writing scripts in low-level languages, they still demand that the user be able to program and understand how the browser works.

Mashups [16, 35] allow end-users to connect content from different sources to create new services. For instance, IFTTT [1] creates simple conditional mashups that follow the pattern "If this, then that." Unfortunately these mashup tools access data through APIs, and thus depend on programmers to produce those APIs.

Deterministic replay of websites. Another class of projects deterministically recreates webpage executions. Mugshot [31] is a record and replay tool aimed at web developers, allowing them to recreate buggy executions for debugging purposes. 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. Timelapse [13] also implements deterministic replay, but works at a lower level, using techniques from virtual machine literature. Jalangi [33] performs replay for JavaScript code, in order to run dynamic analyses. Like Mugshot, it must instrument the Javascript code and relies on the JavaScript structure being the same across executions. These record and replay approaches work on cached versions of a webpage and therefore cannot be used to automate interactions with the live page.

Trigger inference. The trigger inference problem is closely related to race detection on webpages [32]. The aim of trigger inference is to discover races between webpage code execution and user actions. However not all races on the webpage are harmful; we believe most are benign. Therefore, we empirically identify races that cause the script to break. Another related field is inference of partial orders [18, 30]. This work infers hidden partial orders based on linear extensions, *i.e.*, traces, of the hidden partial orders. Unlike past work, we aim to find an over-approximation of the partial order and therefore use a more conservative approach.

10. Conclusion

As more data appears on the web, users need better ways to programmatically access it. Our system, Ringer, aims to help end users by turning demonstrations into replay scripts. But Ringer is just one step towards bridging the gap between a user's goals and the interfaces provided by a website. To achieve this goal, we envision more expressive applications which use record and replay as a building block. Web scraping by demonstration is one such tool, but more tools are needed to help users, especially non-programmers, increase their access to data on the web.

A. ML Approaches to Node Addressing

In addition to the version of our similarity-based node addressing algorithm that uses uniform weights, we tested two variations that used learned weights. We trained the weights for one with linear regression and the other with SVM.

We collected the training data from the 30 top sites according to Alexa rankings [4]. For each node in each page, we recorded all node attributes and the URL reached by clicking on it. Usually this was the start URL, but in cases where a page navigation occurs, the post-click URL gives us a way to check if two nodes are equivalent. We collected this same information twice, a week apart. For each node n_i in the initial run, we checked whether the post-click URL differed from the pre-click URL. If yes, we checked whether any node n_f in the final run shared the post-click URL. If yes, we collected the set of all nodes N_f from the same page as n_f. For each node in N_f, we calculated which attributes matched the attributes of n_i. This vector of Boolean values is the feature vector for our dataset. The vector for n_f (the node with the matching post-click URL) is the positive example labeled with 1; all other nodes in N_f are negative examples and are labeled with 0. Thus, each node may appear in the dataset multiple times, for different target nodes n_i in the initial run, but the vector is different each time, since the vector reflects the attributes that match the current n_i.

As an example, a node $n_{f1} = \langle a_1, b_2 \rangle$ that corresponds to $n_{i1} = \langle a_1, b_1 \rangle$ produces feature vector $\langle 1, 0 \rangle$ and label 1 for node n_{i1}; however for node $n_{i2} = \langle a_2, b_2 \rangle$, the n_{f1} feature vector is $\langle 0, 0 \rangle$ and the label is 0. Both of these (feature vector, label) pairs would be in the dataset.

We used linear regression on this dataset to produce a set of weights for the linear regression SIMILARITY algorithm. The same steps, but with a SVM using a linear kernel, produce the weights used in the SVM node addressing algorithm.

References

[1] IFTTT - make your work flow. URL `https://ifttt.com/`.

[2] The propublica nerd blog - propublica. URL `https://www.propublica.org/nerds`.

[3] A free web & mobile app for reading comfortably - readability. URL `https://www.readability.com/`.

[4] Alexa top 500 global sites, July 2013. URL `http://www.alexa.com/topsites`.

[5] Beautiful soup: We called him tortoise because he taught us. `http://www.crummy.com/software/BeautifulSoup/`, July 2013.

[6] Browser scripting, data extraction and web testing by imacros. `http://www.iopus.com/imacros/`, July 2013.

[7] Scrapy. `http://scrapy.org/`, July 2013.

[8] Selenium-web browser automation. `http://seleniumhq.org/`, July 2013.

[9] Amazon price tracker, Dec. 2015. URL `http://camelcamelcamel.com/`.

[10] Greasemonkey :: Add-ons for firefox, Nov. 2015. URL `https://addons.mozilla.org/en-us/firefox/addon/greasemonkey/`.

[11] S. Barman. *End-User Record and Replay for the Web*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2015. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-266.html`.

[12] M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 163–172, New York, NY, USA, 2005. ACM. doi: `10.1145/1095034.1095062`. URL `http://doi.acm.org/10.1145/1095034.1095062`.

[13] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 473–484, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2268-3. doi: `10.1145/2501988.2502050`. URL `http://doi.acm.org/10.1145/2501988.2502050`.

[14] S. Chasins, S. Barman, R. Bodik, and S. Gulwani. 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, pages 179–182, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-3473-0. doi: `10.1145/2740908.2742849`. URL `http://dx.doi.org/10.1145/2740908.2742849`.

[15] N. Dalvi, P. Bohannon, and F. Sha. Robust web extraction: An approach based on a probabilistic tree-edit model. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 335–348, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: `10.1145/1559845.1559882`. URL `http://doi.acm.org/10.1145/1559845.1559882`.

[16] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi. Intel mash maker: Join the web. *SIGMOD Rec.*, 36(4):27–33, Dec. 2007. ISSN 0163-5808. doi: `10.1145/1361348.1361355`. URL `http://doi.acm.org/10.1145/1361348.1361355`.

[17] f. dfgdfg, S. Flesca, and F. Furfaro. Xpath query relaxation through rewriting rules. *IEEE Transactions on Knowledge*

and Data Engineering, 23(10):1583–1600, Oct 2011. ISSN 1041-4347. doi: 10.1109/TKDE.2010.203.

[18] P. L. Fernandez, L. S. Heath, N. Ramakrishnan, and J. P. C. Vergara. Reconstructing partial orders from linear extensions, 2006.

[19] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. Sellers. Oxpath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal*, 22(1):47–72, Feb. 2013. ISSN 1066-8888. doi: 10.1007/s00778-012-0286-6. URL http://dx.doi.org/10.1007/s00778-012-0286-6.

[20] G. Grasso, T. Furche, and C. Schallhart. Effective web scraping with oxpath. In *Proceedings of the 22Nd International Conference on World Wide Web Companion*, WWW '13 Companion, pages 23–26, Republic and Canton of Geneva, Switzerland, 2013. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-2038-2. URL http://dl.acm.org/citation.cfm?id=2487788.2487796.

[21] R. Hutton. Amazon discount tracker camelcamelcamel tips users to deals, December 2013. URL http://www.bloomberg.com/bw/articles/2013-12-05/amazon-discount-tracker-camelcamelcamel-tips-users-to-deals.

[22] Import.io. Import.io | web data platform & free web scraping tool, Mar. 2016. URL https://www.import.io/.

[23] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, C. Scaffidi, and K. T. Stolee. 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, pages 124–134, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: 10.1145/1453101.1453119. URL http://doi.acm.org/10.1145/1453101.1453119.

[24] J. Kranzdorf, A. Sellers, G. Grasso, C. Schallhart, and T. Furche. Visual oxpath: Robust wrapping by example. In *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, pages 369–372, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1230-1. doi: 10.1145/2187980.2188051. URL http://doi.acm.org/10.1145/2187980.2188051.

[25] K. Labs. Kimono: Turn websites into structured APIs from your browser in seconds, Mar. 2016. URL https://www.kimonolabs.com.

[26] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Reducing web test cases aging by means of robust xpath locators. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 449–454, Nov 2014. doi: 10.1109/ISSREW.2014.17.

[27] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM. doi: 10.1145/1357054.1357323. URL http://doi.acm.org/10.1145/1357054.1357323.

[28] I. Li, J. Nichols, T. Lau, C. Drews, and A. Cypher. 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, pages 723–732, New York, NY, USA, 2010. ACM. doi: 10.1145/1753326.1753432. URL http://doi.acm.org/10.1145/1753326.1753432.

[29] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*, IUI '09, pages 97–106, New York, NY, USA, 2009. ACM. doi: 10.1145/1502650.1502667. URL http://doi.acm.org/10.1145/1502650.1502667.

[30] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 161–168, New York, NY, USA, 2000. ACM. ISBN 1-58113-233-6. doi: 10.1145/347090.347122. URL http://doi.acm.org/10.1145/347090.347122.

[31] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855711.1855722.

[32] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254095. URL http://doi.acm.org/10.1145/2254064.2254095.

[33] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. 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, pages 488–498, New York, NY, USA, 2013. ACM.

[34] stackoverflow.com. Posts containing 'scraping' - stack overflow, July 2016. URL http://stackoverflow.com/search?q=scraping.

[35] J. Wong and J. 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, pages 1435–1444, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240842. URL http://doi.acm.org/10.1145/1240624.1240842.

[36] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 304–314, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2645-2. doi: 10.1145/2610384.2610390. URL http://doi.acm.org/10.1145/2610384.2610390.

[37] T. Yeh, T.-H. Chang, and R. 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, pages 183–192, New York, NY, USA, 2009. ACM. doi: 10.1145/1622176.1622213. URL http://doi.acm.org/10.1145/1622176.1622213.