A Framework for Parallelizing Large-Scale, DOM-Interacting Web Experiments

Sarah Chasins
UC Berkeley
schasins@cs.berkeley.edu

Phitchaya Mangpo Phothilimthana
UC Berkeley
mangpo@cs.berkeley.edu

Abstract
Concurrently executing arbitrary JavaScript on many webpages is hard. For free-standing JavaScript — JavaScript that does not interact with a DOM — parallelization is easy. Many of the parallelization approaches that have been applied to other mainstream languages have been applied also to JavaScript. Existing testing frameworks such as Selenium Grid [19] and PhantomJS [16] allow developers to parallelize JavaScript tests that do interact with a DOM. However, these tools frequently limit the user to very constrained programming models. Although it is possible to build testing programs that take arbitrary inputs and yield arbitrary outputs (rather than pass or not pass) on top of these systems, the work required to do so is non-trivial. Controlling inputs — that is, pages — in order to carry out web research on real world pages is even less supported in modern tools. In fact, no existing tool provides same page guarantees. In response to this paucity of flexible and reliable systems, we built a framework designed for running large-scale JavaScript experiments on real webpages. We evaluate the scalability and robustness of our framework. Further, we present the node addressing problem. We evaluate a new node addressing algorithm against preexisting approaches, leveraging our framework’s clean programming model to streamline this large-scale experiment.

1. Introduction
We describe the design and implementation of a framework that efficiently runs JavaScript programs over many webpages in parallel. We envision that users testing multiple approaches to some JavaScript task can provide each candidate solution as a separate program, and then test all approaches over a large corpus of webpages in parallel. Our framework thus facilitates a type of large-scale web experiment for which researchers, until now, have had no appropriate tools.

We further introduce an application of our framework, a large-scale web experiment that relies on the framework’s guarantees. Specifically, we describe the challenge of robust representation of DOM nodes in live pages. The task is to observe a webpage at time \( t_1 \), describe a given node \( n \) from the webpage, then observe the webpage at a later time \( t_2 \), and identify the time-\( t_2 \) node that corresponds to \( n \) based on the time-\( t_1 \) description. A \( t_2 \) node corresponds to a \( t_1 \) node if sending the same event to both nodes has the same effect. We term this the node addressing problem. Several node addressing algorithms have been proposed, but never tested against each other. We test these algorithms side by side on a test set of thousands of nodes by structuring the experiment as a series of stages in our framework.

Finally, we created a DOM evolution simulator for creating synthetic DOM changes from existing webpages. Our simulator takes as input a webpage DOM and produces as output a second webpage DOM, whose structure is similar to the first, but altered by the types of edits that web designers make over time as pages are redesigned. Thus, given a set of DOMs, our simulator can produce a DOM change that approximates the modification a user would see by pulling DOMs from the same URLs at a later point in time.

1.1 Framework
At present, there are few tools for controlled web research. While there is a preponderance of tools targeted at developers testing their own pages, these tools are not easily applied to the more general problem of running arbitrary tests over real world pages. We identified five core desirable properties, properties that are key to making a framework capable of running large-scale experiments on real world webpages. To facilitate large-scale web research, a framework should:

1. Parallelize test execution
2. Allow DOM interaction
3. Run arbitrary JavaScript code
4. Run on live pages from URLs (not only local)
5. Guarantee same input (page) through experiment

No existing tool does all of these. In fact, many do none of the above. Thus, our framework is motivated by the need for a system that offers all the characteristics necessary to carry out controlled, real world web research.

Our system’s central abstraction is the stage. In each stage, the framework takes as input a set of JavaScript algorithms each with \( n \) inputs, and an input table: a set of \( n \)-field records. In each record, the first field represents the webpage on which to run. Both this webpage argument and the remaining \( (n - 1) \) fields are passed as inputs to the JavaScript algorithms. The output table is a set of records. The output records can have an arbitrary number of fields, and there need not be a one-to-one mapping between input and output records.

Within a JavaScript algorithm, the user may create multiple subalgorithms. In fact, an algorithm is a sequence of subalgorithms. If a subalgorithm causes a page reload, then the next subalgorithm will be run on the new page. If no page reload occurs, the next subalgorithm will be run on the same page.

Our framework also offers the concept of a session. A session is a sequence of stages during which our framework offers a same page guarantee. In web experiments, the webpage is frequently considered an input, just as much as, for instance, any value passed to a JavaScript function. It is thus crucial that pages stay the same across data points, to ensure fairness and correctness of the experimental procedure. Our framework facilitates experiments that require stable page inputs by offering the session abstraction.

We implement the framework to take advantage of the programming model described above by parallelizing at the level of input rows. In addition, we use a custom page cache to both offer our same page guarantee and to improve framework performance.
1.2 Application

To offer robust record and replay for live web pages, a tool must be able to identify the same node in a web page over time, even if the DOM structure of the page changes between record and replay. A naive xpath, which simply traces the path through the DOM tree to the target node is too fragile; wrapper nodes around a target node or its predecessors make the xpath break, as do sibling nodes added before the target node or any predecessors. Testing an algorithm for this task is difficult, because the test must be repeated on the webpage many times, to determine whether it works in the face of a changing page, and because the sample size must be very large in order to get a large number of (naturalistic) broken paths before the later test. To complete these tests by hand — for many nodes, many times, and many algorithms — is very tedious. To complete them programmatically would still be very time-consuming, using current state-of-the-art tools. To complete them fairly, upholding a same page guarantee where necessary, would require a substantial amount of infrastructure building.

A user can run these tests in our framework with four stages. A first stage takes URLs as input, and produces fragile xpaths for each node on those pages. A second stage, in the same session so that the xpaths do not break, takes the xpaths as input and determines whether each node is reactive, storing the xpaths and reactions of the reactive nodes. A third stage takes the xpaths of the reactive nodes and runs all node addressing algorithms over each node, producing node addresses. This stage must also be part of the same session. A final stage, not a part of the same session, takes the node addresses as input, and runs each node addressing algorithm to identify a corresponding node. It then identifies the reaction of each candidate corresponding node, comparing the reaction with the stage 2 reaction to evaluate each algorithm’s success. Rerunning the final stage at a later time, with the same input, tests the algorithms’ robustness over a longer period. This stage can be repeated as often as desired.

1.3 DOM Change Simulator

We offer five DOM modifiers, each of which makes a different type of edit to DOM structure, any or all of which may be useful for a given application, depending on the needs of the user. They may be combined as desired into a single simulator. For the application described above, which examines the robustness of DOM node addresses in the face of changing DOM structures, we observed the effects of all five types of modification on node addressing algorithms.

1.4 Organization

In Section 2, we detail the framework’s programming model. In Section 3 we discuss the implementation of the framework, and in Section 4 its scalability. Section 5 presents the node addressing application. In Section 6, we discuss our DOM change simulator. Our evaluation of node addressing algorithms appears in Section 7. Section 8 is a discussion of related work, and Section 9 concludes.

2. Framework Programming Model

2.1 Abstractions

Our framework is built around a core set of user-facing abstractions:

- **Session**: sequence of stages for which we offer the same page guarantee
- **Stage**: run of a single (input program, input table) pair
- **Input Program**: set of algorithms
- **Input Table**: each row corresponds to a single run of all program algorithms; the first field is the page on which to run them, while the others represent the arguments to all algorithms

Figure 1: A stage, the central abstraction of our framework.

- **Algorithm**: JavaScript code to run on all input table rows; a sequence of subalgorithms
- **Subalgorithm**: component of an algorithm that takes place on a single page
- **Output Table**: a table with zero or more rows per input table row

During a session, a program can run multiple program-table pairs through our system. We call each program-table pair a stage. During a single session, if the URL in the leftmost column of an input table is the same in multiple rows (even across tables), the page that is loaded for those rows will also be the same. Each stage is defined by its input program and its input table. See Figure 1 for a visual representation of a single stage. A stage’s input program may contain multiple algorithms to run for each row in the input. Each algorithm may also contain subalgorithms, if the algorithm must run over multiple pages. For instance, if the first subalgorithm finds and clicks on a link, the second subalgorithm will run on the page that is loaded by clicking on the link. If clicking on a link does not cause a new page to load, the second subalgorithm will run instead on the original page.

2.2 Stage Output

Recall that any given input row may correspond to multiple output rows. This combined with the existence of subalgorithms complicates the design of our programming model. One option is to require that only the final subalgorithm can produce any output. This simplifies the programming model, but for cases in which earlier subalgorithms can access data that the later subalgorithm cannot, this may be undesirable. For instance, consider a test that finds and clicks on a link, and wishes to compare the URLs before and after clicking. To complete this task now, it would need to be split into two stages, with the first stage storing the original URL, while the second clicks the link and stores the second URL. Because this approach simplifies the programming model, and because splitting such tasks across stages is sufficient to make the approach general, this is the design we have adopted.

One plausible improvement is to allow the subalgorithms to pass their output to later subalgorithms, even if they may not directly write to the output table. We believe this modification may indeed be desirable, and if we find applications for which this adjustment would substantially simplify program logic, we may implement it. For now, we find this change unnecessary.

Alternative approaches included allowing all subalgorithms to produce output rows, but requiring that the users JavaScript tests associate each slice of the row with some sort of row id. By using the id, our system would be able to stitch the row slices together correctly after the completion of the final subalgorithm. Similarly, we could require that the number of output rows be the same across subalgorithms, and use the slices ordering to produce the correct full rows.

Another possibility would have been to allow each algorithm to produce only one output row for each input row, but we felt this diminished the expressiveness of the model too substantially.
We faced a similar design challenge in determining the appropriate way to combine distinct algorithms’ outputs. In fact, considered most of the same solutions, except that all algorithms must be allowed to produce output. Ultimately, we determined that all algorithms should be allowed to produce as many rows of output as desired, even if different algorithms produce different numbers of output rows for the same input row. Output rows are stitched together based on the order in which they are returned by the algorithms, as shown in Figure 2. This approach simplifies users’ code for the common cases in which there is only a single algorithm, each algorithm returns only a single row, or each algorithm returns the same number of rows. However, it restricts the model’s expressiveness in cases in which algorithms do not know the order in which other algorithms will return their output rows, but still want to achieve a particular lineup with each other. Although we believe this situation is likely to be rare, it would be simple to address such cases by offering the..
Running the simple title benchmark over the first 500 sites in the Alexa top sites list [5], we measured three kinds of undesirable outcomes: server unreachable errors (Figure 6(a)), timeouts (Figure 6(b)), and wrong outputs (Figure 6(c)). While server unreachability is outside of the web automation tools’ control, the tools are sometimes the cause of timeouts, and always the cause of wrong outputs.

Our results revealed that PhantomJS and Ghost.py did not reliably produce the correct (human-identified) outputs. While PhantomJS provided the correct titles for most pages, it handled redirects poorly, often giving the title associated with a redirect, rather than the one associated with the final destination. Further, although this issue did not appear for the simple JavaScript code that retrieves a document title, we also found that for some JavaScript tests, running the code in PhantomJS failed to produce the same effects that it produced in Selenium, Ghost.py, and normal browsers. Since our framework targets arbitrary JavaScript code, this was unacceptable.

Ghost.py handles redirects correctly, but it times out on a large number of pages, and does not support non-English output. Further, although Ghost.py provides an API for accessing new pages loaded by interacting with a page, this API does not apply when the loading interaction is completed by a JavaScript program. This problem appears to be a known issue that the Ghost.py developers have not yet addressed. Because our framework targets arbitrary JavaScript code, including DOM-interacting code, and because it targets all pages, including non-English pages, this was unacceptable.

3.2.6 Web Driver Selection

Because PhantomJS and Ghost.py failed to provide the reliability and robustness so crucial to our goals, we chose to build our system on top of Selenium.

Selenium’s times, although slow in the sequential version, became competitive with a load balanced implementation. Even more importantly, it produced no incorrect outputs. Because our framework is intended to facilitate web research, it is essential that the results be trustworthy. Ultimately, we preferred seeing more rows without answers to seeing rows with wrong answers. Rows without answers are an unavoidable component of web experiments, servers typically being outside of the experimenters’ control. Thus, we privileged wrong outputs as the deciding factor, and consequently selected Selenium as the provider of our framework’s web driver.

3.3 Caching Proxy Server

Our framework enforces the same-page guarantee though the use of a caching proxy server. The framework directs all HTTP request-response traffic through the caching proxy server as illustrated in Figure 4. All pages for a given session are served from a single cache.
Despite the preponderance of existing caching proxy servers, none proved sufficiently controllable to meet our needs. Squid cache [18, 20] can be configured to ignore some web cache policy parameters, such as ‘no-cache,’ ‘must-revalidate,’ and ‘expiration.’ However, it cannot be configured to ignore others — for instance, ‘Vary.’ As an example, unmodified Squid will never cache responses (with a ‘no-cache’ policy), and redirecting a request for Y to X, until eventually the originally requested X is ready, and the X response is no longer a redirect. At this point, the response for X contains the final page content that our cache should associate with URL X.

In this scenario, if the proxy server caches everything, and there is no mechanism to clear the cache, our system will loop forever. The proxy server will always return the redirects. Our cache addresses this issue by maintaining a redirect table, mapping request URLs to their redirect URLs. Upon receiving a redirect response, the cache checks whether adding the redirect response to the table will create a size-two redirect cycle. If it will, the cache removes the pre-existing redirect entry that causes the cycle. This technique is limited to redirect cycles of size two, but since we have not yet found a redirect cycle of size greater than two, we feel the performance benefits of avoiding full cycle detection justify this limitation.

### 3.4 User Interaction

As discussed in Section 2.3 above, the user controls our framework from a Java program, creating an instance of the framework, then using the session and stage abstractions. She may interleaving processing with processing of her own.

The other inputs are all the files that are passed to the stages: a JavaScript file for each stage, with a function for each subalgorithm; a table for each stage, with the URLs and algorithm inputs.

### 4. Framework Evaluation

To briefly evaluate the scalability of our complete framework, we ran the title extraction benchmark on the first 10,000 sites in the Alexa top sites list [5]. We ran this benchmark on the same machine used in the Section 3.2.4 experiments. We recorded the execution time for every increment of 100 sites. The results appear in Figure 7.

The data reveals that both the competition time and the number of timeouts scale linearly with the size of the input. We conclude that our framework is sufficiently stable, and that its performance meets the needs of the large-scale experiments our framework targets.

### 5. The Node Addressing Problem

As discussed in Section 2.2, web tools — such as record and replay systems — require a node addressing algorithm that keeps nodes addressable even in the face of pages’ changing DOMs. A simple xpath that traces the path from the root to the target node breaks when new wrapper nodes are added, even when new sibling nodes are added for any node along that path. This may occur as the page is redesigned, or simply in response to user actions. For instance,
Figure 7: Scalability test on the simple title extraction benchmark.

Figure 8: A visualization of the DOM representation algorithm testing task, split into stages for our framework.

Figure 9: An illustration of some of the DOM node characteristics that can be used to construct a DOM ‘address.’

5.1 Node Addressing Algorithms

Several solutions have been proposed to the node addressing problem, but they have never been tested in a controlled experiment.

We consider the following 5 plausible node addressing algorithms.

1. **xpath** Records the path from the root of the DOM tree to the target node. To identify corresponding node, follows the same path. If there is no matched node, returns null.

2. **id** Selects the first node whose id matches the recorded id, otherwise returns null.

3. **class** Selects the first node whose class matches the recorded class, otherwise returns null.

4. **iMacros** Collects the list of nodes with the same node type and inner text as the target node. Records the target node’s position in this list. To identify the corresponding node, constructs the list from the new page, selects the item at the target node’s original index. If no such node, returns null.

5. **Ringer** Collects the xpath as described above, the id, the class, and the text. To identify the corresponding node, it uses six strategies; the original xpath, xpath suffixes, common variations on the xpath, the class, the id, the text. Each strategy votes for up to one node. Returns the most voted-for node, or null if no nodes receive votes.

The first three, xpath, id, and class each use only a single characteristic to identify the corresponding node. They rely on that characteristic to identify exactly one node. The second two, iMacros...
5.2 Testing Node Addressing Algorithms

Our framework offers a means of testing node addressing algorithms in a fair setting. For testing in our framework, the experiment is split into four stages, as depicted in Figure 8. The first stage traverses the DOM tree, recording an xpath for each node. The second stage, with an input row for each xpath, clicks on the node at the xpath. Recall that finding a node on one page and then again after a reloading is extremely difficult, that this problem is in fact the entire motivation for this application. Even reloading after a few seconds, the original xpath may break. It is thus crucial that stages 1 and 2 occur in the same session, seeing the same instance of each page during both stages. Stage 2 also compares the pre-click URL with the post-click URL, producing an output row only if the URL has changed. That is, each output row corresponds to a reactive node. If clicking on the node has no effect on the state, it will be impossible to check (in stage 4) whether the stage 4 node corresponds to the stage 2 node. Since, for this experiment, we take the URL as a proxy for the state, the URL must change in order for a node to be considered reactive. Stage 3 takes the xpaths of all reactive nodes as input, and runs each node addressing algorithm on each node, producing the node addresses for each node as output.

The fourth stage takes the node addresses as input. It uses the addresses and the corresponding node addressing algorithms to find the appropriate node on the new page, and then click on it. It produces the new post-click URL as output. The new post-click URL can be compared with the stage 2 post-click URL to determine whether each algorithm successfully identified the corresponding node. Figure 11 offers a pictorial representation of the stage 4 algorithm, showing the different node addressing approaches as different algorithms, and the clicking and URL inspecting functions as different subalgorithms.

Thus, this experiment can be cleanly divided into four stages in our framework. Note that the experiment relies on all five of the crucial characteristics identified in Section 1.1:

1. Parallelize test execution — a thorough test demands running this task on many nodes, in order to reveal a sufficiently large number of naturalistically broken addresses to distinguish between approaches, which makes parallel execution highly desirable.

2. Allow DOM interaction — the test must be able to click on the algorithm-identified nodes.

3. Run arbitrary JavaScript code — the algorithms cannot be limited to, for instance, returning pass or fail as their outputs.

4. Run on live pages from URLs (not only local) — the test should run on the real top Alexa sites, not DOMs built up locally.

5. Guarantee same input (page) through experiment — stages 1 through 3 require a same page guarantee.

In Table 1, we show content descriptions for the input and output files of each stage.

6. DOM Change Simulator

Testing node addressing algorithms requires a suite of DOMs, each with multiple versions. Alternatively, testing can work over DOMs with naturalistic changes introduced to simulate DOM changes over time. To explore the latter approach, we created a small suite of DOM edits that can be combined to create various DOM workload simulations.

We implemented 4 types of DOM edits:

- **Wrapper: wrap first-level divs**
  
  Wrap every div node that is a direct child of the body node with a center node.

- **Method 1**
  
  1. Get node, method 1, click.
  
  2. Allow DOM interaction — the test must be able to click on the algorithm-identified nodes.

- **Method 2**
  
  1. Get node, method 2, click.
  
  2. Allow DOM interaction — the test must be able to click on the algorithm-identified nodes.

- **Method 3**
  
  1. Get node, method 3, click.
  
  2. Allow DOM interaction — the test must be able to click on the algorithm-identified nodes.

- **Method 4**
  
  1. Get node, method 4, click.
  
  2. Allow DOM interaction — the test must be able to click on the algorithm-identified nodes.

Ringer, are in use by real web tools. The iMacros algorithm comes from the approach used by the iMacros [1] web scripting tool. The Ringer approach is also in use by a real tool, Ringer [3], which is a web record and replay system. We are particularly interested in the Ringer approach, which is being developed for the Ringer tool by Shaon Barman and one of the authors of this paper.

Figure 10: A visualization of the types of changes that are made to DOMs as they are redesigned or obfuscated.

Figure 11: A visualization of the stage 4 input program, showing the first two node addressing algorithms. For each algorithm, the first subalgorithm uses the node address to identify the corresponding node according to the given node addressing algorithm. It then clicks. The second subalgorithm checks the URL of the resultant page.

Figure 12: A visualization of the types of changes that are made to DOMs as they are redesigned or obfuscated.
Insert: insert many nodes

For every div node, insert as the div’s first child a new node with the same tag as the div’s original first child.

Type: span to p

Convert every span node into a p node.

Move: become sibling’s child

Move every object whose next sibling is a div object such that it becomes that sibling’s first child.

Text: modify node text

Add a letter to the inner text of each node.

Figure 12 gives a pictorial representation of the effects of these edits.

7. Node Addressing Evaluation

For the purposes of this work, we consider two correctness conditions. The first, a conservative estimate of success rate, considers a result correct if the stage 4 post-click URL is the same as the stage 2 post-click URL. This produces some false negatives. Consider clicking the ‘Random Article’ link on wikipedia.org or the top story on a news site. Clicking on the correct node will lead to different urls during different runs. However, whenever this correctness condition is met, the node addressing algorithm has definitely succeeded. Thus, this condition offers a lower bound on robustness.

The second correctness condition may overapproximate success rates, considering a result successful if the stage 4 post-click URL is different from the stage 4 pre-click URL. This approach correctly handles the ‘Random Article’ and top story cases described above, but may also produce false positives. This approach allows an algorithm to click on any URL-changing node and still succeed, regardless of whether it is the corresponding node. However, if an algorithm fails by this criterion, it has definitively failed. If no URL effect is produced, the identified node was not the corresponding node. Thus, this condition offers an upper bound on robustness.

7.1 Robustness Over Time

To evaluate node addressing algorithms’ robustness over time, we compared their immediate performance with their performance after a delay. Specifically, we ran them once on the same day that the node addresses were recorded — day 0 — and once almost a week later on day 6.

We ran our experiment on the top 15 sites in the Alexa top sites list, executing the four stages as described in Section 5, running stage 4 on both day 0 and day 6. The 15 sites yielded 11,573 nodes, 1106 of which were reactive.

The algorithms’ success rates on the day 0 and day 6 runs appear in Figure 13. The lighter colored bars represent the success rate using the first correctness criteria — that is, the lower bound. The darker colored portions of the bars represent the additional
percentage points gained by using the second correctness criterion, the upper bound on the true success rate.

As Figure [13] makes evident, the id and class approaches largely unsuccessful. In contrast, xpath, iMacros, and Ringer all prove reasonably robust. Ringer definitively outperforms iMacros on day 0, with the lower bound on Ringer’s success rate being higher than the upper bound on iMacros’. While Ringer’s success region overlaps with xpath’s on day 0, both its upper and lower bounds are higher than xpath’s respective upper and lower bounds. The success region for xpath in day 0 does overlap with iMacros’, but barely — xpath comes quite close to definitively outperforming iMacros.

All three of the successful approaches see their performance degrade with the passage of time. On day 6, Ringer’s upper bound is still greater than all other approaches’ upper bounds, and its lower bound is still greater than all other approaches’ lower bounds; however, note that it no longer definitely dominates the iMacros approach, there being some overlap in their success regions. Nevertheless, since the lower bound on Ringer’s day 6 performance is in fact higher than the lower bound on iMacros’ day 0 performance, we consider Ringer’s approach largely successful.

Overall, we conclude that both iMacros and Ringer appear to degrade more slowly over time than xpath. However, since Ringer’s original performance solidly dominates iMacros’, we conclude that the Ringer approach is the more robust.

7.2 Synthetic Benchmark

We evaluated the node addressing algorithms using our synthetic DOM evolution simulators on 4 sites: amazon.com, wordpress.com, bing.com, and ask.com. Each run consists of a single session with 9 stages. The first 3 stages correspond to stages 1-3 in the node addressing task, described in Section 5. The last 6 stages are variations on stage 4, node retrieval. The node retrieval stage runs with 6 different types of DOM modifications: none (original baseline), wrapper, insert, type, move, and text.

This benchmark identified 63 reactive nodes out of 773 total DOM nodes. Figure [14] shows the percentage of nodes that each algorithm successfully identifies. The results for id and class are excluded, since they identified none of the nodes correctly. The 100% success rate of the xpath algorithm when no DOM changes are present confirms that our framework provides the same page guarantee within the same session. Note that iMacros could not correctly identify all nodes even when no changes had been applied to the DOMs, because node type, text, and pos were not enough information.

As evidenced by Figure [14] xpath performs poorly on wrapper, insert, and move. These edits altered the xpaths to most DOM nodes, substantially degrading xpath’s effectiveness. It performed well on type because most nodes were not of type span, and it could correctly identify all nodes when only the text content was changed. Ringer consistently performed better than xpath. On insert and move, it benefited greatly from considering the xpath suffixes and common variations on the xpath. However, Ringer performed worse than iMacros on insert and move, since Ringer gives xpath so much weight. iMacro’s success rates were relatively consistent across different types of DOM changes, with the exception of text. As expected, iMacros could not identify any node correctly when text content was modified.

8. Related Work

8.1 Frameworks

A large number of tools have been developed for the purpose of running JavaScript tests. Almost all are targeted towards web developers who want to test their own pages, or even only their own JavaScript. Below we cover the main subcategories of this class of tools.

Jasmine [9] is one of the most prominent system for running JavaScript unit tests. While its ease of use makes it a good platform for small-scale experiments, it lacks many of the characteristics we desire for large-scale, general purpose web research. First, its parallelization mechanism is quite limited. Second, it uses a restrictive programming model, tailored to offer pass/fail responses for each test. Third, it only runs on locally constructed DOMs — it cannot be fed a URL for its tests.

These limitations characterize a large portion of the JavaScript testing space, which very heavily tailors platforms to web developers with limited experimental needs. This category of tool includes projects such as QUnit [17], Mocha [18], and YUI Test [19]. In fact, QUnit, Mocha, and YUI Test do not offer even the limited parallelization that Jasmine provides.

Some tools, like Vows [11], offer parallelization, but are aimed only at testing JavaScript. These typically run on Node, which eliminates any DOM-interactive code from their domains, and naturally any URL-loaded code.

Finally we consider the three web automation tools on which we built our framework implementations: Ghost.py [10], PhantomJS [16], and Selenium [4]. None of these is explicitly a testing framework. Rather, they are generic web automation tools. Clearly it is possible to build up a testing framework on top of any of them, as this paper has shown. However, we found that the amount of infrastructure and insight necessary to do so was far from trivial. Ghost.py and PhantomJS have built-in parallelization. There is a variation on Selenium, Selenium Grid [19], that offers parallelization. We note however that it is tailored for users who want to run the same tests on multiple browsers and on multiple operating systems, rather than for users with large-scale experiments. In fact, we found the Selenium Grid approach sufficiently unwieldy for our needs that we chose to implement our own parallelization layer.

Ghost.py, PhantomJS, and Selenium do all offer DOM interaction, the ability to run arbitrary JavaScript code, and the ability to load pages from URLs, all characteristics that made them accept-
able candidates for serving as our framework’s web driver. How-
never, in the case of Ghost.py and PhantomJS, we found the ability to
run arbitrary code was sometimes hindered by the tools’ limited
robustness.

All three of Ghost.py, PhantomJS, and Selenium — like all the
projects described here — lack any support for same page guar-
antees. Ultimately, most tools, being targeted towards developers,
are targeted towards users who know their test pages will stay the
same, or know how they will change. This makes them generally
unsuitable for broader web research.

8.2 Node Addressing

There are several existing web tools that require robust node ad-
addressing algorithms. Therefore, despite the paucity of literature on
the node addressing problem, several solutions have been put into
practice.

We have already described two such solutions, iMacros and
Ringer. The iMacros [1] approach uses node type and node text
to identify a list of nodes, and then uses the target node’s position
in that list as an address. The Ringer [2] approach uses an xpath,
several variations on the xpath, the id, the class, and the text in a
voting scheme to identify corresponding nodes.

CoScripter, another web tool which offers some record and
replay functionality, takes an iMacros-style approach [12]. Where
possible, it associates a target node with text, whether it be the
text within the node, or text that precedes it — as when a textbox
appears to the right of a label. When no related text is available
— for instance, if the node is a search button that displays a
magnifying glass icon rather than the word ‘search’ — CoScripter
falls back on the position approach. ActionShot [13] also uses
CoScripter’s technique.

Some tools take even more fragile approaches, relying on pro-
grammers to adjust the node addresses by hand as appropriate. For
instance, Selenium IDE offers a record and replay option that de-
scribes nodes by id when available, describes links by their inner
text contents, and backs off to an approach that describes nodes by
the combination of node type and parent node type.

Other tools take wholly different approaches. For instance,
Chickenfoot [7] allows users to access and interact with nodes via
high level commands and pattern-matching. Sikuli [21] takes a vi-
sual approach, using screenshots and image recognition to identify
nodes.

8.3 DOM Workloads

To our knowledge, there have been no previous tools for DOM
evolution simulation. To this point, researchers conducting web ex-
periments who need realistically changing DOM workloads appear
to have had two main options. First, they could collect their own
workloads by pulling DOMs at the desired intervals. If a researcher
needed control over the amount of change, this was the only op-
tion. Alternatively, researchers could use DOMs pulled by down by
archiving operations such as the WayBack Machine [2]. However,
this option gives researchers no control over the intervals between
collection points, and there is no guarantee that a researcher’s sites
of interest will have been archived.

9. Conclusion

Existing tools for running JavaScript tests in parallel are limited.
They offer limited parallelization, limited DOM interaction, lim-
lited programming models, often run only on locally constructed
DOMs, and never provide same page guarantees. Our framework
offers a system with none of these limitations. The node address-
ing application revealed that our framework can be used to cleanly
structure large and complicated web experiments. We found that
our new node addressing approach is more effective than a prior
state-of-the-art algorithm. Finally, we introduced a DOM evolution
simulator for generating DOM workloads with realistically modi-
fied structures. We obtained interesting insights into node address-
ing algorithms through their use on our synthetic workloads. Ulti-
mately, we believe that large-scale web research is an increasingly
important area, with exciting problems to solve and many crucial
insights to uncover. Increased tool support should accelerate the
pace of discovery in this burgeoning young field of study.

References

[1] Browser scripting, data extraction and web testing by imacros.
   http://www.iopus.com/imacros/


   October 2013.


[7] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and
   Robert C. Miller. Automation and customization of rendered web
   pages. UIST ’05.

   univ-paris-diderot.fr/~jch/software/polipo/

   October 2013.

   2013.

   December 2013.

   Coscripter: automating & sharing how-to knowledge in the enterprise.
   CHI ’08.

   Here’s what i did: Sharing and reusing web activity with actionshot.
   CHI ’10.

    docs/test/

    http://visionmedia.github.io/mocha

    December 2013.


    squid-cache.org/ May 2013.


    JOURNAL ON SELECTED AREAS IN COMMUNICATION, 16:345–

gui screenshots for search and automation. UIST ’09.