ARROW: Automated Repair of Races on Client-Side Web Pages

Weihang Wang[1], Yunhui Zheng[2], Peng Liu[1,2], Lei Xu[3], Xiangyu Zhang[1], Patrick Eugster[1]
[1]Purdue University, USA [2]IBM T.J. Watson Research Center, USA [3]Nanjing University, China
{wang1315, peng74, xyzhang, p}@cs.purdue.edu {zhengyu, liup}@us.ibm.com xlei@nju.edu.cn

ABSTRACT

Modern browsers have a highly concurrent page rendering process in order to be more responsive. However, such a concurrent execution model leads to various race issues. In this paper, we present ARROW, a static technique that can automatically, safely, and cost effectively patch certain race issues on client side pages. It works by statically modeling a web page as a causal graph denoting happens-before relations between page elements, according to the rendering process in browsers. Races are detected by identifying inconsistencies between the graph and the dependence relations intended by the developer. Detected races are fixed by leveraging a constraint solver to add a set of edges with the minimum cost to the causal graph so that it is consistent with the intended dependences. The input page is then transformed to respect the repair edges. ARROW has fixed 151 races from 20 real world commercial web sites.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Automatic repair, constraint solving, race condition

1. INTRODUCTION

Web applications are pervasive, providing the platform for many daily activities such as shopping, social networking, gaming and working. To satisfy increasingly complex functional requirements and provide pleasant user experience, they often include complex logics in client-side pages. According to a survey on 4.2 billion web pages conducted by Google in 2010 [20], on average, each page includes 7.09 external JavaScript (JS) files. The average size of external JS files per page is 57.98KB. A web page takes up to 114.25KB on average, excluding images.

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931052

Client-side web pages contain multiple types of scripts such as HTML, JS, and Stylesheets. In order to be highly responsive, these pages are rendered by modern browsers in a concurrent fashion. When a web page is rendered, some of the included external resources are downloaded, assembled and rendered asynchronously. User input events and internal events indicating the status of various asynchronous tasks fire in a non-deterministic order, due to various factors such as network delay. Some execution orders may be significantly different from the developer's intention, leading to exceptions. We call them web application races. These races are wide-spread and can cause serious problems. According to [19], developers in Mozilla have observed that many web sites used in their regression suite failed non-deterministically due to races. They crashed the JS engine, caused session data loss, or corrupted services such as the Hotmail email service. For example, a race that crashes the JS engine or triggers page reloading in the middle of email composition may cause a loss of the unfinished email. Client side races may lead to permanent data corruption on the server side such as online photos being undesirably deleted.

Existing research efforts on client side races focus on automatically detecting races [28, 19, 21]. Zheng et al. [28] applied static analysis on JS code to detect atomicity violations caused by asynchrony in Ajax. WebRacer [19] is a dynamic race detector that leverages the happens-before relations between common web features to identify races. Its follow-up work EventRacer [21] proposed the concept of race coverage and reduced false warnings caused by ad-hoc synchronizations present in web pages. However, none of these works discusses how to automatically repair races, which is challenging for the following reasons: (1) there is no native support for synchronization in JS so that the concurrency control to fix a race may have to be developed from scratch; (2) adding synchronizations may introduce new bugs such as deadlocks or exceptions if not done properly; (3) one page may have multiple races, their repairs may interfere with each other (e.g. the repair of one race may also fix another race or the repairs of multiple races may cause deadlocks).

In this paper, we propose ARROW[1], a technique that can automatically fix races on client side pages based on static analysis. Given a page, it first statically analyzes the page to detect races. It then transforms the page by re-ordering the elements in the page (e.g. JS code blocks) or adding customized synchronizations. To reason about the complex effects of element reordering and adding synchronization,

[1]ARROW stands for "Automated Repair of Races On client-side Web pages".

and to ensure the repairs for multiple races do not have undesirable interferences, we leverage constraint solving. In particular, we model the page into a causal graph describing the happens-before relations between elements, which is further encoded to constraints. We then infer the dependences between elements intended by the developer from the page source. The solver is queried to detect any inconsistencies between the happens-before relations and the dependences, which are essentially races. The constraints are constructed in such a way that we can further query if a smallest set of causal edges can be added to the graph so that all the races in the same page can be repaired together in a safe fashion. If so, the page is transformed to respect those edges. Since different transformations (e.g. element reordering and adding synchronizations) have different runtime cost, we also encode the estimated cost as part of the constraints such that the repair with the lowest estimated cost can be identified.

ARROW has the following advantages: (a) it is automatic; (b) it is safe, meaning the repairs will not add any new buggy behavior as our analysis is conservative; (c) it delivers cost-effective solutions.

Our main contributions are the following.

- We propose a technique ARROW that can automatically fix certain races on client side pages.
- We develop a technique to detect races statically by identifying inconsistencies between happens-before relations determined by the page rendering order and the dependences extracted from the page source, denoting the developer's intension of the execution order.
- We develop a constraint solving based repair scheme that can ensure safety and achieve low runtime cost.
- We evaluate ARROW on 20 real world websites. It detects and repairs 151 races correctly.

Since ARROW aims to repair races in a page for all possible input scenarios and ensure safety, it is built on static analysis. Note that a dynamic analysis based repair technique may fix the page for one input but introduce undesirable effects (e.g. deadlock) for a diferent input. Static analysis has limitations on handling dynamic web features, such as runtime element insertion. As such, ARROW detects and repairs a subset of races that are amenable to static analysis. More discussion can be found in Section 5. Handling dynamic features [25, 26] is part of our future work.

2. ILLUSTRATIVE EXAMPLES

In this section, we will examine two examples and explain why race conditions happen and how they can be fixed.

2.1 Accessing Unready Objects

Web pages usually include multiple external resources such as JS, CSS, and images. During page loading, web browsers aggressively process resources whenever they become available. Although the loading order usually follows the source code order, due to unpredictable reasons such as browser settings, network conditions and user interactions, it may become different from the source code order. Sometimes, such differences represent race conditions that result in accessing DOM elements or JS objects before they are ready. Accessing unready objects is an important kind of race conditions reported by existing race detection work [19]: three out of four races reported fall into this category. Next we show such an example and illustrate its repair.

Fig. 1(a) presents a page *main.html* that includes *a.html*

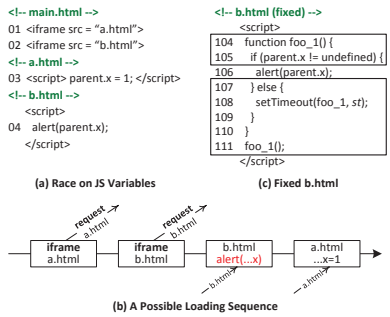

Figure 1: (a) shows a race resulting in accessing an uninitialized variable "x" in line 4. (b) demonstrates the buggy execution. (c) shows the script after repair. The boxed statements are added during repair.

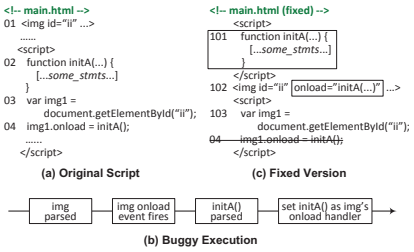

Figure 2: Race condition on an event handler that is triggered only once and the corresponding repair.

and *b.html* using two *iframe* tags. Suggested by the inclusion order, *a.html* is supposed to be processed before *b.html* so that the use of variable x at line 4 will happen after its definition at line 3. However, iframes are loaded asynchronously. As a result, the sequence in Fig. 1(b) can be the actual one, where *b.html* is processed before *a.html*. Therefore, the access to "x" at line 4 refers to an undefined variable and causes exception. Note that such exceptions often trigger page reloading that may cause loss of user's session inputs.

To fix this issue, we leverage the *definition-use* (def-use) dependence suggested by the source code order and force the execution to follow an order that respects the dependence, by introducing customized synchronization. In particular, for a variable in a race, we check its availability before the use to make sure it is ready. If not, we wait and try again after some time until it's ready. As shown in Fig. 1(c), the boxed statements are added to enforce the execution order. The access at 106 is wrapped by a new function *foo_1()*. Before accessing variable x, we check if it is ready at line 105. If not, we set up a timer at line 108 and try again after *st* (a predefined number) milliseconds. Note that the timer only fires once and it may be setup again by the handler *foo_1()* if needed. The availability check serves as the customized synchronization and makes sure x is always ready when accessed. Note that since JS does not provide native synchronization support, continuous polling controlled by a timer is a common approach for developers to manually fix race issues in practice [2].

Besides JS variables, race conditions can also happen on HTML DOM elements or JS function objects. We can repair them in a similar way.

2.2 One-Time Handlers

Another common kind of races is related to event handlers. For example, an *onload* event fires only once after a DOM element is loaded. Onload handlers are important as

developers usually use them to initialize the properties of the corresponding DOM object. If an onload event handler is not invoked correctly, the page may become unusable.

By default, the onload handler of a DOM object is empty. Developers can explicitly register a JS function as the new handler. Although it is possible to specify the handler at the place where a DOM object is defined (e.g., *<img onload="foo()">*), developers tend to register event handlers explicitly and separately using JS. This is especially true when programming with third-party libraries. Fig. 2(a) shows an example. At line 1, an image object, *<img>*, is defined. At line 3, the script gets a reference to this object and registers a JS function, *initA()*, as its onload event handler.

One reason behind such a practice is to make page loading faster by parallelizing remote resource downloading and local script interpretation. If a browser handles everything sequentially, it is less efficient since it has to wait until the files requested are retrieved from the remote servers. Therefore, in modern browsers, external resource downloads are usually asynchronous and non-blocking. In this example, when the browser encounters the *<img>* tag, it requests the image file specified by the *src* attribute and starts the asynchronous download. Then it proceeds to the following HTML elements. Once the *<img>* is fully loaded, its onload event fires. Since parsing local script is usually faster than downloading a remote file, in most cases the event handler registration at line 4 can be done before the onload event fires so that the handler can be invoked correctly.

However, the order between the handler registration (line 4) and the onload event firing (i.e. the moment when the *<img>* element at line 1 is fully loaded) is nondeterministic. If the image requested is small or cached locally, it is possible the onload event fires before its handler is registered. If so, the onload event handler may never be invoked. For example, Fig. 2(b) shows a buggy execution sequence. When the onload event of *<img>* fires, function *initA()* has not been registered yet so that it will never have a chance to be executed. Note that unlike events that can be triggered multiple times, an onload event will only fire once. We call an event handler of this kind a *one-time handler*.

Fig. 2(c) shows its repair. Since developers cannot control when the onload event fires, to make sure the event handler is registered and triggered properly, we place the registration to the definition point of the DOM object. In this example, lines 1 and 3 are replaced by line 102. Then, we move the declaration of function *initA()* before the *<img>* tag so that *initA()* is defined before it is used.

The repair in this example is different from the one in Fig. 1(c). In Fig. 1(c), we introduce customized synchronization to enforce the appropriate load order. While the race in this example can also be fixed similarly, we choose to reorder elements in the web page. We change the places where *initA()* is declared and registered. The goal is to avoid additional runtime overhead introduced by synchronization.

Observe that the races are due to inconsistencies between the execution order and the def-use order in both cases. We fix them by transforming the page so the orders become consistent. In this paper, we focus on this kind of race problems.

In practice, similar issues have been observed. For example, according to [4, 5], races are the root causes of malfunctioned user interfaces and incomplete page loading, leading to unusable web pages. Fixing real world web application races is challenging. In particular, there are inter-

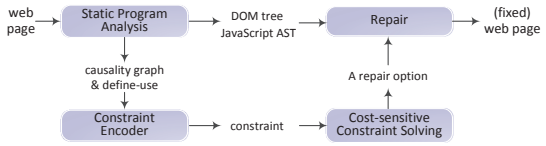

Figure 3: ARROW Overview.

dependencies between DOM objects and JS variables such that the developers have to be very cautious in re-positioning elements and adding synchronizations. For example, moving a piece of JS code forward may break existing def-use dependences and introduce uninitialized variable access errors. Adding synchronizations may introduce deadlocks. Furthermore, a client page often has multiple races. Their repairs may interfere with each other and have consequences that are difficult to reason about. Finally, since there are alternatives in fixing a race, constructing a low-cost overall repair (for multiple races) is a complex optimization problem difficult to address manually.

Therefore, one key contribution of our work is to model the problem of automatically fixing multiple races in a web page together as a constraint solving problem, and leverage the solver to ensure that (1) our fixes do not break any existing semantics (and hence introduce new bugs); (2) fixes do not interfere in ways leading to any undesirable consequences; (3) and the overall repair plan is cost-effective.

3. DESIGN

In this section, we first present a high-level overview of ARROW. Then, we discuss the design of the individual components in more details in Sections 3.2 - 3.4.

3.1 Overview and Deployment

As shown in Fig. 3, the input of ARROW is a client-side web page and the output is the fixed version of the page. The work-flow of ARROW can be divided into three steps:

In the *first* step, we perform static analysis on the HTML and the JS snippets included in the input page. We construct a *causal graph* that models the happens-before relations between runtime events of page elements (e.g., DOM object creation must happen before the invocations of its event handlers). These relations are determined by the underlying page parsing and execution model of modern browsers, which will be discussed in Section 3.2. We also identify all the *def-use* dependences that describe the definition(s) that a use of variable/object may come from. Def-use relations are different from happens-before orders. They are derived from the source code order in the given page, *reflecting the developer's original expectation of the execution order*. However, the orders suggested by def-use relations are not necessarily respected by the happens-before relations, leading to races. The graph and the def-use relations allow us to reduce the original problem to a partial order reasoning problem. In particular, we detect races by identifying inconsistencies between the causal graph and the def-use relations and we repair races by transforming the causal graph (and hence transforming the page) to respect the def-use relations. In this step, we also produce a DOM tree and JS ASTs that will be used in the repair step. More details can be found in Section 3.2.

In the *second* step, we detect races by identifying inconsistencies between the causal graph and the def-use relations derived from the source code order, as illustrated by the ex-

amples in Section 2. In particular, we model this problem as determining the reachability from an object/variable *definition* to its *use* in the causal graph, leveraging a constraint solver. We encode the directed edges in the causal graph and the orders suggested by def-use relations as constraints. Then, we feed the constraints to a solver and query satisfiability for several purposes:

(*a*) *Race detection.* Assuming the happens-before relations in the graph, we check whether the orders suggested by individual def-use pairs are respected. If not, races are detected.

(*b*) *The existence of a repair.* After races are detected, we check whether the race inducing execution orders can be precluded by introducing additional causal edges. If yes, we say the input page is fixable.

(*c*) *An optimal repair.* The solution produced in the previous query may not be an optimal repair. For example, assume both element reordering and customized synchronizations can fix a race. However, as mentioned before, element reordering has less runtime overhead compared to customized synchronizations. Therefore, reordering is a better repair if it is applicable. On the other hand, reordering is more likely to break existing semantic constraints (e.g. def-use relations) and becomes in-applicable. To find a cost-effective solution, we further associate costs for the two repair options. By asserting a specific cost goal to the constraint solver, starting from a low cost and gradually increasing, we are able to find a repair with the lowest cost.

We explain the details of this step in Section 3.3.

In the *third* step, we transform the page according to the repair, which is essentially a set of additional causal edges in the graph. In particular, elements are reordered or customized synchronizations are introduced according to the edges. The repair will preserve the original looks and functionalities of the page. Section 3.4 explains how to transform the input web page based on the repair solution generated in the previous step.

Deployment. Due to the overhead of constraint solving, ARROW is not suitable for on-the-fly bug repair on the client side. We anticipate the following two possible ways of deploying ARROW.

First, during in-house testing and debugging, ARROW can aid developers in fixing races. Note ARROW only fixes races in a client page, which may be dynamically generated from a server script. ARROW currently does not fix server scripts. Instead, it will provide the fixed client pages to the developers who will integrate the fixes to the server scripts.

Second, ARROW may be used to protect against races during production runs when used with page caches. When a page is downloaded, ARROW performs analysis and repair in the background and then replaces the cached page with the fixed version such that it can prevent future races.

3.2 Causal Graph Construction and Def-Use Relation Identification

In step one, ARROW constructs a *causal graph* to model the happens-before orders of events based on the standard page parsing and execution order of modern browsers.

3.2.1 Web Page Rendering Process

While modern browsers are highly concurrent, they still strictly follow certain orders during page rendering. Understanding these orders is critical to causal graph construction.

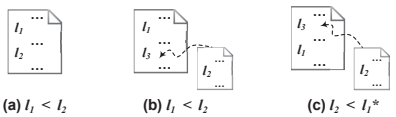

Figure 4: Location Precedence. '*' in (c) means $l_2 < l_1$ holds if l_2 is not included by an iframe or an async script.

Given a page, the browser parses the page in the source code order. Once a DOM object is parsed, it is created and its onload event fires. For DOM objects whose creation relies on remote resources (e.g., an image with a remote URL), the browser requests the resource in a non-blocking mode and proceeds to render the remainder of the page. Some browsers will not create the DOM object until the remote resource is successfully retrieved. The browser also parses and executes JS code snippets following their location order in the page. Statements within a pair of *<script>* and *</script>* are parsed and executed atomically. In other words, they cannot interleave with others. The page may explicitly declare asynchronous artifacts such as iframes and asynchronous scripts, whose executions are non-deterministic and decoupled from their parsing and creation. Event handler executions are mostly non-deterministic, depending on the event triggering time, except onload events.

From the page rendering process discussed above, we can observe that the source code location order of elements is very important. We model it through a relation called *script location precedence*. Given two different source locations l_1 and l_2 in a web page or external resources included, we define *location precede* (denoted as "$<$") as follows.

(1) l_1 and l_2 are in a same file. As shown in Fig. 4(a), if l_1 appears before l_2, $l_1 < l_2$.

(2) l_1 and l_2 are in different files. If there exists l_3 such that l_1 and l_3 are in the same file, and l_2 is *transitively* included at l_3, there are two cases as follows:

- If $l_1 < l_3$ (Fig. 4(b)), $l_1 < l_2$. In this case, before the browser sees the tag at l_3 and goes into the inclusion chain, the elements at l_1 must have been processed.
- If $l_3 < l_1$ and l_2 is *not* included by an iframe or an asynchronous script (Fig. 4(c)), $l_2 < l_1$. Otherwise, $l_1 \not< l_2 \wedge l_2 \not< l_1$. In this scenario, the browser sees the tag at l_3 before l_1. The order between l_1 and l_2 depends on whether the inclusion chain is processed sequentially: If so, the browser must process l_2 before l_1. Otherwise, when l_2 is included in an iframe or an asynchronous script, l_2 will be handled asynchronously so that the order between them is nondeterministic.

3.2.2 Causal Graph

Since our repairs need to be safe (i.e. not introducing any new bugs but rather just precluding problematic execution orders in the original page), we make use of conservative static analysis in graph construction.

The *causal graph* of a web page is a directed graph $\mathbb{CG} = (\mathbb{N}, \mathbb{E} \cup \mathbb{W})$. $\mathbb{N}$ is a set of nodes that represent runtime events during page parsing and rendering such as DOM creations and event handler invocations. $\mathbb{E}$ and $\mathbb{W}$ denote two kinds of happens-before edges between nodes.

Nodes. We model the following five kinds of nodes.

(N1) We use $\mathcal{D}$ to denote the set of HTML elements. If $d \in \mathcal{D}$ is declared and created at location l, $create(d, l) \in \mathbb{N}$.

(N2) Let o be a global JS object, such as a global variable or a function, declared at l, $create(o, l) \in \mathbb{N}$.

(N3) Let s be a top level statement (i.e. a statement not in any function or composite statement such as conditional or loop) parsed and executed at location l, $exec(s, l) \in \mathbb{N}$. Note that although all statements in a JS code block execute atomically, we create nodes for individual top level statements so that we can separate a code block to multiple smaller blocks and reorder them. These nodes are different from the nodes in *(N2)* as a function may be declared but not invoked. We do not represent lower level statements as reordering them or adding synchronizations to them is very difficult and also unnecessary in most cases. We can ensure orders of low level statements by enforcing orders of their corresponding top level statements.

(N4) Let f be a function registered as the handler of an event e of object $d \in \mathcal{D}$, the invocation of the handler $handler(f, d, e) \in \mathbb{N}$. For instance, $handler(f, d, onload)$ denotes the invocation of the onload handler of d.

(N5) Let a be an asynchronous external JS element declared at l (e.g. *<script async src=...>*). Its declaration $create(a, l) \in \mathbb{N}$ and the asynchronous execution/interpretation $interpr(a) \in \mathbb{N}$.

During page loading, the browser parses and renders/executes HTML and JS objects sequentially according to the location order. The invocations of onload event handlers of DOM objects are hence also sequential. We call nodes involved in sequential processing *sequential nodes*. They include the nodes tagged with location label l (e.g., *N1*, *N2*, *N3*). In contrast, other nodes represent executions of asynchronous scripts or callbacks triggered by user or timer events. We call them *asynchronous nodes*. A special case is that the invocation of an iframe's onload event handler is treated as an asynchronous node.

Edges. $\mathbb{E}$ is the set of directed edges indicating irreversible happens-before relations between nodes. We classify happens-before relations into two kinds: *reversible* and *irreversible*.

Irreversible edges. Edges cannot be transformed during repair such as the order between DOM object creation events because mutating such edges may lead to undesirable visual differences in page rendering. In contrast, some happens-before relations such as the order between a DOM object creation and a JS object creation may be reversed.

For any two nodes n_1 and n_2, $(n_1, n_2) \in \mathbb{E}$ if any of the following conditions holds.

(E1) $n_1 = create(d_1, l_1)$, $n_2 = create(d_2, l_2)$, where $d_1, d_2 \in \mathcal{D}$, $l_1 < l_2$, $\nexists\ create(d_3, l_3) \in \mathbb{N}$ such that $d_3 \in \mathcal{D}$ and $l_1 < l_3 < l_2$. It means the orders between static DOM elements are irreversible so that they will be preserved in the repair.

(E2) $n_1 = create(a, l)$, $n_2 = interpr(a)$, where a is an asynchronous external JS element.

(E3) $n_1 = create(d, l)$, $n_2 = handler(f, d, e)$. The creation of DOM element d happens before the invocation of a handler of event e of the element.

(E4) $n_1 = handler(f_1, d_1, onload)$, $n_2 = handler(f_2, d_2, onload)$, d_2 encloses d_1.

(E5) $n_1 = exec(ajaxS(d), l)$, $n_2 = handler(f,\ d,\ ajaxR)$. If a JS statement $ajaxS(d)$ at l creates an ajax object d, registers f as its response handler and sends the request, $ajaxS(d)$ executes before f.

(E6) $n_1 = exec(setTimeOut \mid setInterval,\ l)$, $n_2 = handler(f,\ ,\ timer)$. Function *setTimeOut/setInterval* registers f as a timer event handler. f can be invoked after its registration at l.

(E7) $n_1 = create(iframe(x), l_1)$, $n_2 = create(d_2, l_2)$ with $d_2 \in \mathcal{D}$ the first HTML element in the iframe page x.

(E8) $n_1 = create(d, l_1) \mid create(o, l_1) \mid exec(s_1, l_1) \mid create(a, l_1)$, $n_2 = exec(s_2, l_2)$, where $l_1 < l_2$ and s_2 denotes a method call that evaluates a string as JS (e.g., *eval()*). Function *eval()* may generate some script on the fly that uses a variable defined before l_2 so we do not want to reposition anything before l_2 to after l_2. To handle this, we prevent any reposition between a node preceding an *eval()* node and the *eval()* node, by introducing these conservative irreversible edges. Note that this does not prevent reordering of elements before the *eval()*.

(E9) $n_1 = exec(s_1, l_1)$, $n_2 = create(d, l_2) \mid create(o, l_2) \mid exec(s_2, l_2) \mid create(a, l_2)$, where $l_1 < l_2$ and s_1 evaluates a string as JS. This rule is symmetric to (E8), preventing moving things after l_1 to before l_1.

Note that *(E5)* and *(E6)* introduce causal edges between event handler registration and invocation for Ajax responses and timer events, whereas such causality does not exist for regular events such as onload and onclick. The reason is that regular events fire even without the explicit registration of handlers but Ajax responses and timer events do not.

Reversible Edges. A separate relation set ($\mathbb{W}$) is defined to denote reversible edges. In particular, $\mathbb{W}$ is a set of ordered pairs of *sequential* nodes in $\mathbb{N}$ satisfying one of the conditions.

(W1) $n_1 = create(d, l_1) \mid create(o, l_1) \mid create(a, l_1)$, $n_2 = exec(s, l_2)$. If $l_1 < l_2$ and there is not another node with label l_3 s.t. $l_1 < l_3$ and $l_3 < l_2$, $(n_1, n_2) \in \mathbb{W}$. If $l_2 < l_1$ and there is not a node l_3 s.t. $l_2 < l_3$ and $l_3 < l_1$, $(n_2, n_1) \in \mathbb{W}$.

(W2) $n_1 = exec(s_1, l_1)$, $n_2 = exec(s_2, l_2)$. If $l_1 < l_2$ and there is not a node l_3 s.t. $l_1 < l_3$ and $l_3 < l_2$, $(n_1, n_2) \in \mathbb{W}$.

Recall we only create nodes for top level statements *(N3)* so that the location order is also the control flow order in *(W2)*. An edge between a script statement execution node with any other node is a reversible edge, denoting that the script statement may be repositioned during repair without causing visual differences. However, whether a reversible edge can be really reversed is also determined by the def-use relations which we will discuss next.

Handling Dynamic Features. Rules *(E8)* and *(E9)* allow us handle *eval()* conservatively. HTML pages can be modified at runtime by their own JS code. To handle this, ARROW currently requires the developer to record the set of possible pages and apply the technique to individual pages. As part of the deployment procedure (Section 3.1), the developer may need to integrate the fixes.

3.2.3 Def-Use Relation Identification

The *def-use* relation ($\mathcal{P}_{du}$) is a set of node pairs in $\mathbb{CG}$. Particularly, $(n_1, n_2)^x \in \mathcal{P}_{du}$ means a JS global object/variable or a DOM element x is defined in n_1 and used in n_2. Note that if n_1 and n_2 denote top level composite statements (e.g. conditionals or function calls), definitions/uses *inside* n_1/n_2 will introduce edges between n_1 and n_2. Def-use pairs are identified following the location order in the web page source. The essence is that *the source order reflects the original semantics intended by the developer*, which may not be respected by the process order in the browser, causing races. The definition is presented as follows.

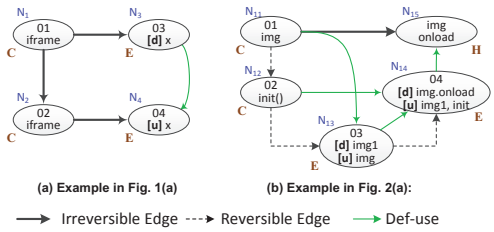

Figure 5: The causal graph and the def-use pairs of the web pages shown in Fig. 1(a) and Fig. 2(a). "[d]" and "[u]" mean objects defined and used in this node respectively. Labels "C", "E" and "H" beside nodes denote the node types: object creations, JS executions and event handler invocations, respectively.

- Let n_2 be a sequential node that uses a global DOM/JS object x. Based on the source location order from the beginning to n_2, for any definition of x in n_1 that can reach n_2, $(n_1, n_2)^x \in \mathcal{P}_{du}$.
- Let n_2 be an asynchronous node that uses x. If a node n_1 defines x, $(n_1, n_2)^x \in \mathcal{P}_{du}$.

Example. The graphs in Fig. 5 show the casual graphs with both reversible and irreversible edges, and the def-use pairs for the two examples in Section 2. In particular, Fig. 5(a) is for the first example in Fig. 1(a). Fig. 5(b) models Fig. 2(a). In Fig. 5(a), the causal edges are established by Rules (E1) and (E7). The def-use relation is straightforward. In Fig. 5(b) the causal edges are introduced by Rules (E3), (W1) and (W2).

Our def-use analysis is mostly standard, very similar to that in WALA [1]. The analysis distinguishes *must* def-use pairs (i.e., the use variable is a must alias of the definition variable and there is a path between the definition and the use) from *may* def-use pairs (i.e., the use variable is a may alias of the definition variable and there is a path between the definition and the use). We use must-pairs in race detection to avoid false positives. We use may-pairs in repair to ensure safety. More discussion is in Section 5.

3.3 Repair Generation: A Constraint Solving Based Approach

In this paper, we consider fixing races that manifest themselves as inconsistencies between the page rendering order and the def-use pairs. As mentioned at the end of Section 2, the challenges of fixing these races lie in avoiding breaking any existing semantic constraints, reasoning about the interferences between individual fixes, and achieving cost effectiveness. We hence leverage constraint solving to construct a universal repair that fix all races together. Here a repair is essentially a set of new edges in the causal graph.

The overarching design is to encode causal graph edges including reversible and irreversible ones into relations. We also encode def-use pairs as a different relation. We then query the solver if def-use pairs can be inferred from the edge relations. This is equivalent to performing graph reachability analysis. If not, races are detected. We then further query the solver if a smallest set of weighted edges can be added such that the def-use pairs can be inferred in the mean time the added weight/cost is minimal.

Edge Encoding. We define a function $hasEdge(\rhd)$ to encode edges including those in both $\mathbb{E}$ and $\mathbb{W}$.

$$\rhd : Node \times Node \rightarrow Bool$$

The relation is populated by the edges from the causal graph. As suggested by the following theorem, $\rhd$ is irreflexive and asymmetric.

$$\begin{array}{ll} \forall n \in \mathbb{N}, & \neg(n \rhd n) \\ \forall n_1, n_2 \in \mathbb{N}, & (n_1 \rhd n_2) \implies \neg(n_2 \rhd n_1) \end{array} \quad [1]$$

It is not transitive either. From $n_1 \rhd n_2$ and $n_2 \rhd n_3$, we cannot infer $n_1 \rhd n_3$.

Inferring Happens-Before from *hasEdge* Relation. Next, we introduce the $happensBefore(\prec)$ relation as follows.

$$\begin{array}{l} \prec : Node \times Node \rightarrow Bool \\ \forall n_1, n_2 \in \mathbb{N}, n_1 \prec n_2 \quad := \quad n_1 \rhd n_2 \\ \qquad\qquad\qquad\qquad\quad \mid \ \exists n_3, n_1 \prec n_3 \wedge n_3 \rhd n_2 \end{array} \quad [2]$$

If there is a path between two nodes, there is a happens-before relation between them.

The relation has similar properties as the *hasEdge* relation, except that it is transitive, as suggested by the following theorem.

$$\begin{array}{ll} \forall n \in \mathbb{N}, & \neg(n \prec n) \\ \forall n_1, n_2 \in \mathbb{N}, & (n_1 \prec n_2) \implies \neg(n_2 \prec n_1) \\ \forall n_1, n_2, n_3 \in \mathbb{N}, & (n_1 \prec n_2 \wedge n_2 \prec n_3) \implies (n_1 \prec n_3) \end{array} \quad [3]$$

Note that the theorem dictates the causal graph is acyclic.

Def-Use Pair Encoding. A def-use pair implies that the definition should happen before the use. Therefore, we assert the happens-before relation between a definition and the corresponding use. In particular, assume n_u uses x; If n_1 is the only place where x is defined, we assert $n_1 \prec n_u$; If n_u may use multiple x from nodes $\{n_1, ..., n_i\}$, we assert $(n_1 \prec n_u) \vee ... \vee (n_i \prec n_u)$.

Repair Cost Encoding. We explicitly encode repair cost for two purposes: *race detection* and *cost-effective repair construction*. Different from using relational reasoning engines such as Datalog, it is tricky to determine if a given (def-use) pair is a member of a (happens-before) relation using an SMT engine (during race detection). Given a def-use pair n_d and n_u, if we assert $n_d \prec n_u$, the SMT engine will simply add the pair to the current happens-before relation if it does not cause any contradiction. In other words, the solver will always try to yield a satisfiable result by simply adding causal edges. Therefore, we leverage the cost encoding to address the issue.

We introduce a relation Ω *denoting the set of node pairs that do not have an edge in either direction*. If $(n_1, n_2) \in \Omega$, $(n_2, n_1) \in \Omega$.

For two nodes $(n_1, n_2) \in \Omega$, we define a function *cost* (n_1, n_2) to denote the cost when we introduce an additional edge from n_1 to n_2. The value of $cost(n_1, n_2)$ is determined using results from the previous analysis stage. In particular, if the order between n_1 and n_2 can be inferred from their source locations and they are in a same file, they are eligible for reordering. Note that if a node is asynchronous, the order cannot be inferred from the source locations. Since reordering has less runtime overhead, we assign a small value 1 to $cost(n_1, n_2)$. Otherwise, introducing synchronizations is the only way to enforce their order. Since it has more runtime overhead, we use a larger number 10 as the cost.

We use $C_{n_1 \rhd n_2}$ to denote the repair cost regarding an edge from n_1 to n_2. If an edge from n_1 to n_2 is added in a repair, $C_{n_1 \rhd n_2}$ equals to $cost(n_1, n_2)$. Otherwise, $C_{n_1 \rhd n_2}$ equals to 0. For any two nodes without edges in either direction, we encode their repair cost as follows.

$$\forall (n_1, n_2) \in \Omega,\ C_{n_1 \rhd n_2} = (n_1 \rhd n_2)\ ?\ cost(n_1, n_2)\ :\ 0$$

We use C_{repair} to represent the total cost, which is computed as $C_{repair} = \sum_{(n_1,n_2)\in\Omega} C_{n_1 \rhd n_2}$.

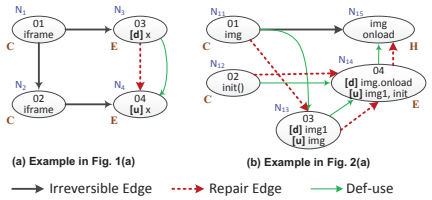

Figure 6: Repairs for the two illustrative examples.

Race Detection. To detect races, we encode the causal edges $\mathbb{E} \cup \mathbb{W}$ and def-use pairs as mentioned before. For node pairs that do not have any edges ($\mathbb{E} \cup \mathbb{W}$) in either direction, we encode their cost of adding new edges.

We set the total repair cost (C_{repair}) to 0 and query its satisfiability, which essentially checks whether the happens-before relations suggested by the def-use pairs can be satisfied without introducing new edges. If the constraint is SAT, no race is detected. Otherwise, there are race conditions.

Example. The encoding of Fig. 5(a) is

$$
\begin{aligned}
&(N_1 \rhd N_2) \wedge (N_1 \rhd N_3) \wedge (N_2 \rhd N_4) \wedge (N_3 \prec N_4) \\
&\wedge (\text{ if } N_3 \rhd N_4 \text{ then } C_{N_3 \rhd N_4} = 10 \text{ else } C_{N_3 \rhd N_4} = 0) \\
&\wedge (\text{ if } N_4 \rhd N_3 \text{ then } C_{N_4 \rhd N_3} = 10 \text{ else } C_{N_4 \rhd N_3} = 0) \\
&\wedge \ldots \\
&\wedge C_{repair} = (C_{N_3 \rhd N_4} + C_{N_4 \rhd N_3}) + (C_{N_1 \rhd N_4} + C_{N_4 \rhd N_1}) \\
&\qquad + (C_{N_2 \rhd N_3} + C_{N_3 \rhd N_2}) \\
&\wedge C_{repair} = 0 \\
&\wedge [1] \wedge [2] \wedge [3]
\end{aligned}
$$

Note $N_3 \prec N_4$ in the first row is introduced according to the def-use pair on x. The encoding for Fig. 5(b) is similar.

Finding a Cost-Effective Repair. Next we generate a repair if we found races in the previous step. A repair is a set of additional directed edges introduced to the causal graph such that the relations implied by def-use pairs are consistent with the new *happensBefore* relation.

We start by determining whether a repair exists. Unlike in race detection, this time *we only encode irreversible edges*. The reason is that ignoring the reversible edges allows the solver to explore reordering of the corresponding elements. We reuse the other constraint encodings from the previous step. This time, we do not restrict the repair cost, which can be achieved by removing the assertion $C_{repair} = 0$. We query the solver again. If the constraint is SAT, a repair exists. The solution provided by the solver will report the additional edges needed. It will also report a concrete value for repair cost C_{repair}. Assume its value is k. It means all races can be fixed with cost k.

However, the solution may not have the lowest cost. Therefore, ARROW repeatedly queries the solver with different cost assertions starting from 1. It stops when the constraints are SAT with the minimum cost.

We want to point out that the race repair scheme can be made independent from the race detection component and integrated with different race detectors.

Example. The example in Fig. 5(a) does not have any reversible edges. We get the repair constraint after removing the assertion $C_{repair} = 0$ from the race detection constraint. And the solver reports $N_3 \rightarrow N_4$ as the repair, which is shown in Fig. 6(a).

For the example in Fig. 5(b), after removing the reversible edges, we have more pairs of nodes that do not have edges between them. We encode the cost to introduce extra edges between these nodes. The solver reports four repair edges as shown in Fig. 6(b).

3.4 Page Transformation

In this step, we realize a repair by page transformation.

For each source file involved, we put back the reversible edges that do not involve any nodes in the generated repair edges and do not form any cycles with other edges in the graph. Recall reversible edges were removed during patch generation. We then perform a topological sort on the part of the resulting causal graph (with repair edges and restored reversible edges) corresponding to the file. The page elements denoted by the nodes are rearranged based on the order. Since the order between DOM elements remains the same (due to the irreversible edges), the new arrangement will not change relative positions among DOM elements so that the page's visual representation remains intact. Furthermore, the restored reversible edges also ensure the reordering is minimal for safety insurance (Section 5). After this, we satisfy the repair edges denoting element reordering.

However, there are still edges that cannot be satisfied by reordering, such as those across file boundaries and those involving asynchronous nodes. These edges have a high repair cost as discussed earlier. For these edges, ARROW introduces synchronizations as follows: it first introduces and sets a flag at the exit(s) of the head node. Then, before the tail node, ARROW inserts code to check whether the flag is set. If not, the inserted code reschedules the tail node to execute later using the timer function *setTimeout()*. An example of such transformation can be found in the second example in Section 2. Due to space limitations, we omit the page transformation algorithm.

Example. Let's revisit the two illustrative examples. Fig. 6 shows the repairs generated by the solver. For the first example in Fig. 6(a), three files are involved. After the per-file topological sort, we have $\{N_1, N_2\}$, $\{N_3\}$ and $\{N_4\}$, which do not trigger any reordering. To respect the repair edge $N_3 \rightarrow N_4$, ARROW introduces synchronization as shown in Fig. 1(c). Note that both N_3 and N_4 are asynchronous nodes because they are included in <*iframe*>'s so that we cannot determine their order from their script locations.

For the second example in Fig. 6(b), the topological sort produces $\{N_{12}, N_{11}, N_{13}, N_{14}, N_{15}\}$. The transformed page starts with N_{12}, followed by N_{11}, N_{13} and N_{14}, which denote event handler registration. Such a pattern is commonly seen and ARROW has a special rule to handle it. The rule groups the chain N_{11}, N_{13} and N_{14} into a handler registration inside the DOM tag as shown in Fig. 2. The updated topological order becomes $\{N_{12}, (N_{11}, N_{13}, N_{14}), N_{15}\}$. Instead of introducing expensive synchronization, the repair edges can be respected by reordering with the updated order. The repaired version is shown in Fig. 2 (c).

4. HANDLING PRACTICAL ISSUES

Searching for a cost-effective repair for real world pages is expensive. Assume there are n nodes in the graph. ARROW needs to select k edges from the $2n^2$ possible node pairs. The complexity is hence $O(2^{2n^2})$. To improve scalability, we apply the following optimizations.

4.1 Causal Graph Simplification

Real world web pages usually have a large number of DOM elements. For example, the home page of *shell.com* has 728

DOM elements. The causal graph is large if we use one node to denote each element. Fortunately, we observe that we can usually model multiple elements as one node.

As mentioned before, DOM elements in a web page (excluding those included by an *iframe*) are processed sequentially. A sequence of *consecutive* DOM elements with no script, event handler or iframe elements in between can be represented as one single node as there must not be any edges going in or out from inside the sequence. For example, the following is a piece of HTML script from *shell.com*.

```
<ul id="country_selector_list">
 <li><a href="http://www.shell.com/...">Algeria</a></li>
 ...
 <li><a href="http://www.shell.com.vn/">Vietnam</a></li>
</ul>
```

This is a drop-down list with 197 elements. None of them has an event handler. So, we use one node to model them. In our experience, most DOM elements do not have event handlers such that we can greatly reduce the number of nodes.

4.2 Constraint Simplification

Besides reducing the graph size, we also simplify the constraint encoding. It is very expensive to reason about quantifiers [3] in general. As an optimization, we eliminate quantifiers by transformation. For example, we eliminate the existential quantifier in theorem [2] in Section 3.3 by enumerating all the possible n_3 in [2], as shown in the following.

$$\forall n_x, n_y \in \mathbb{N}, n_x \prec n_y := n_x \rhd n_y \mid n_x \prec n_1 \wedge n_1 \rhd n_y \mid ... \mid n_x \prec n_i \wedge n_i \rhd n_y \quad [2']$$

Universal quantifiers are similarly removed by enumerating all nodes.

5. SAFETY AND LIMITATIONS

Safety In Repair. We assume the def-use analysis is complete in the absence of *eval()* and self-modifying JS code, which can be achieved in theory [1]. With this assumption, ARROW is safe during repair: *given a set of races in a page, it either determines that the page is not fixable or it guarantees that a repair must not introduce any new problems*, such as deadlocks or new exceptions.

First, our causal graph construction is conservative. For places that cannot be analyzed appropriately, such as *eval()*, ARROW conservatively introduces causal edges that prevent any transformation across these places. As such, the def-use information related to *eval()* is not necessary for ARROW. *Second*, theorem [3] in Section 3.3 about the irreflexive property of the *happensBefore* relation dictates that the causal graph is always cycle-free. Together with the conservative nature of the graph, a repaired page must be deadlock free. ARROW considers a page not fixable if cycles cannot be avoided. *Third*, according to our assumption, the def-use analysis is complete when `eval()` and self-modifying JS code are not considered. This ensures that the generated transformation must respect these def-use pairs such that new exceptions (e.g., undefined variables) cannot be introduced.

However, our current implementation is not complete due to the incomplete modeling of third party JS libraries. Real world pages make intensive use of third party JS libraries. Due to the sheer number of these libraries, we only model a subset that is commonly used (e.g. some *JQuery* functions). Note that it is usually not an option to analyze these libraries as part of the code base because many of them are fairly complex or even obfuscated. In practice, our limited modeling is sufficient. As shown in Section 6, most repair transformations are fairly local, not involving substantial global code re-ordering. As a result, they do not involve any complex library calls that may endanger safety.

Self-modifying pages are beyond the scope of ARROW. We assume the developer will collect the set of possible pages and analyze them individually.

Limitations In Race Detection. ARROW may have false negatives and false positives in race detection due to the dynamic features of pages and the approximations made during analysis. For example, ARROW will miss races involving `eval()`. Our def-use analysis involves over-approximations. Thus, ARROW may have false positives in race detection. To mitigate the problem, we only assert must-aliased def-use pairs in race detection. In our experiment, this strategy is effective. We did not observe any false positives. Another point we want to make is that false positives are not that problematic for ARROW as they only lead to redundant synchronizations or unnecessary reorderings.

6. EXPERIMENTAL RESULTS

ARROW is implemented in JavaScript, leveraging a set of Node.js utilities. For each subject web page, ARROW parses it and acquires its DOM tree using *htmlparser2*. For script elements, ARROW uses ECMAScript parser *Esprima* to parse them and generate AST trees. With the DOM tree and JS ASTs, the causal graph can be constructed. Our def-use analysis extends that in WALA [1]. The graph is then simplified and encoded to constraints, together with the def-use pairs. We use Z3 [7] to solve the constraints and find the optimal repair if any. Then we transform the input web page using AST transformation.

We randomly select 20 web sites from the *Alexa Top 500 Sites* and the *Fortune 500 2014 Sites* as the benchmark. As our current implementation only supports a set of third party libraries, we avoid those that make use a lot of other libraries. Table 1 shows the program size of the subject web sites. As we model the parsing of each HTML tag on the main page as a single node in our causal graph, we believe the line of code of the main page is an important metric that reflects the graph size. We also report the number and the line of code of all the JS files externally loaded by each site. Observe most of these pages are quite complex.

Columns `node` and `node(simp)` in Table 1 show the number of nodes in causal graph before and after simplification, respectively. The size ranges from small, with `United Continental Holdings, Inc.` at only 91 nodes, to quite large with `Delta Air Lines, inc.` at 1840 nodes. Observe that our simplification is effective. The reduction is usually two orders of magnitude, which is critical to efficiency.

Column `def-use` in Table 1 reports the def-use pairs between graph nodes caused by *global* variables/objects, when only must-aliasing is considered. These are the edges we assert to the causal graph to detect inconsistencies. Since most of the def-use pairs are respected by the causal graph, the number of races detected (column `races`) is much smaller.

We have validated that all races are real by inserting intentional sleep to the page to trigger the problematic orders. Note that we do not have false positives because our causal graph construction is conservative and we only consider must-aliased def-use pairs in race detection. False negatives cannot be studied due to the lack of ground truth. As in most existing works on bug repairs, we do not claim that ARROW can repair all races in a page.

Table 1: Program characteristics and Analysis Result

Site	Program Characteristics					Analysis Result												Causes of Races			
	HTML			JavaScript		Node	Node (simp)	Def-Use	Races	Solving time (s)	Sync edge	Reorder edge	Load Time			B/W Reorderings		Aysn script	Event handler	Timer	Ajax
	Index LOC*	#	LOC*	#	LOC*								before	after	%	LOC / %	lib calls				
360.cn	3851	1	3851	8	9574	1147	69	78	13(2)+	222.1	6	2	2.83	3.02	106.7	0 / 0%	0	7	6	0	0
alcoa.com	1167	12	17362	19	79425	768	88	150	5	117.9	3	4	2.16	2.28	105.6	3/ 0.2%	0	2	2	1	0
cardinal.com	1681	1	1681	19	34307	662	59	596	2	107.0	1	4	2.53	2.56	101.2	151/ 6.6%	8	0	1	1	0
chsinc.com	807	1	807	10	11915	416	28	13	5	2.4	1	2	1.65	1.76	106.7	4/ 0.5%	0	0	5	0	0
deere.com	924	1	924	16	21928	509	47	1000	1	51.5	1	1	0.91	0.96	105.5	0/ 0%	0	0	1	0	0
delta.com	5098	3	5104	29	72040	1840	136	766	2	39.6	1	12	6.49	6.34	97.7	76/ 1.3%	1	0	1	0	1
directv.com	3734	10	16969	40	105302	1216	149	1501	19	724.8	3	3	0.02	0.02	100.0	30/ 0.9%	0	17	2	0	0
dow.com	577	4	7812	15	34484	330	39	117	6	24.9	3	3	1.29	1.28	99.2	139/ 21.8%	0	3	2	1	0
dropbox.com	1199	3	1267	33	117455	594	65	586	12	155.9	5	0	6.51	6.44	98.9	0/ 0%	0	10	2	0	0
fc2.com	1588	12	2271	18	35426	415	67	71	11(3)+	11.2	3	1	1.99	1.98	99.5	0/ 0%	0	11	0	0	0
ge.com	1635	7	12743	17	53891	551	37	81	7	144.3	2	10	2.36	2.28	96.6	67/ 3.1%	1	5	2	0	0
jnj.com	1197	2	3707	23	29612	492	53	188	3	48.5	2	3	1.18	1.19	100.8	118/ 9.9%	0	1	1	1	0
manpowergroup.com	1143	3	3715	18	25491	627	46	94	4	55.1	2	5	1.04	1.07	102.9	112/ 9.7%	1	1	2	1	0
metlimos.com	504	2	1113	21	46227	369	52	69	9	605.8	7	6	1.31	1.31	100.0	17/ 3.4%	0	0	9	0	0
qq.com	14418	3	14699	18	21131	2440	101	92	20	356.3	10	0	14.2	14.8	104.2	0/ 0%	0	0	20	0	0
shell.com	1030	1	1030	25	32621	832	44	413	2	1.6	1	1	0.93	0.94	101.1	24/ 1.7%	0	1	0	0	1
statefarm.com	898	4	1052	15	13096	785	61	445	1	33.8	1	0	3.60	3.52	97.8	0/ 0%	0	0	1	0	0
tumblr.com	2237	2	2285	23	58331	701	100	967	22	24.9	3	1	19.4	21.6	111.3	0/ 0%	0	22	0	0	0
unitedcontinen-talholdings.com	148	2	2658	9	22670	91	21	82	3	76.2	2	3	2.36	2.21	93.6	46/ 24.6%	3	1	1	1	0
yahoo.com	9424	4	9998	7	36675	2011	55	124	4	110.9	3	3	6.20	6.22	100.3	0/ 0%	0	1	3	0	0

*: After HTML/JavaScript Pretty-print. +: Number of Races That Form Cycles and ARROW Failed to Repair.

ARROW fixes races in the same page all together. Columns solving time, sync edge and reorder egde show the constraint solving time, the number of places where customized synchronizations are added, and the number of reordering edges in the repair. In some cases, the number of transformations is smaller than the number of races, suggesting that one transformation may fix multiple races. There are also cases where the number of reordering is larger than the number of races because reordering one node may cause other nodes to be reordered to respect existing def-use constraints. We have manually validated (using the aforementioned intentional sleep method) that the races are correctly fixed. We have also applied EventRacer [21], which is a dynamic race detector, to the pages. For the races that were detected by EventRacer before repair[2], they are no longer reported after repair. Note that for `360.cn` and fc2.com, ARROW failed to fix some of the races (as shown in column races) because the def-use edges and the irreversible edges form cycles.

Columns before and after report the average page load time before and after repair. We test the constraint solving time, and the page load time on a 1.3 GHz Intel Core i5 Mac machine with 4GB memory. The page load time is the average of 10 test runs recorded for each site on a clean Chrome (36.0.1985) browser. Observe that the runtime overhead of the repairs is small for most cases. In some cases, the repairs actually speed-up the load time. Further inspection seems to indicate that reordering may speed-up page loading.

The two columns of Between Reorderings show the lines of JS code that are reordered together with their percentage in the entire page, and the number of library function calls involved. This is to show that in practice, the reordering is mainly local and rarely involves library calls such that the repair is safe (i.e., no violations of def-use pairs from the original page due to incomplete modeling of library functions). We also want to point out that reordering may not be needed even though there are reorder edges from the solver (column reorder edge). These edges are likely to ensure def-use pairs caused by reversible edges. Since we put back the reversible edges before transformation, the reorder edges may not trigger any reordering.

Table 1 further classifies the races ARROW fixed into four categories. *Asynchronous script execution*: a race is caused by executing an asynchronous script where a read from/write to an object occurs. This execution is asynchronous to other parts of the program. Out of the 20 sites, 5 sites do not show races in this category. *Late event handler registration*: a DOM tag event handler is registered late in the program. In this case, it is possible the event fires before its handler is registered. *Timing events*: JS programs use *setTimeout()* or *setInterval()* to execute a function at specified time-intervals. During this time-interval, it is possible those variables referenced by the callback function are also read/written by other parts of the program. *AJAX*: before an AJAX request is sent out, a JS function is registered as its response handler and will be invoked once the server response arrives. Since the response may come at any time, this handler can interleave with other JavaScript executions.

[2]Since EventRacer is dynamic, it may not detect all the races reported by ARROW.

```
<!-- CHS Inc. Home.html -->
01 <form name="LoginPortletForm" action="...">
02   <input type="text" onkeypress="return executeViaEnter(event);">
03   <input type="password" onkeypress="return executeViaEnter(event);">
04   <div style="color: #ff0000; display: none;"> * Invalid Login </div>
05   <input type="submit" value="Log-in" onclick="doLogin();">
   </form>
   ...
06 <script src="main.js"></script>
```

Figure 7: Code snippet from site *chsinc.com*.

6.1 Case Study I

In this case we study three races ARROW fixed in page `CHS Inc. Home.html`. Part of the page is shown in Fig. 7. DOM tags are simplified to contain only relevant attributes. This piece of script denotes a login form containing two textboxes and one submit button. The textboxes register *executeViaEnter()* as the onkeypress event handler. The button registers *doLogin()* as the onclick handler.

Functions *executeViaEnter()* and *doLogin()* are defined in the external script *main.js* (Fig. 8), which is loaded after the login form is created (line 6 in Fig. 7). Function *executeViaEnter()* calls *doLogin()* if the input is '`enter`'. *DoLogin()* first performs some preprocessing, e.g., reads username, before it calls the *submit()* method on the login form. The server script specified in the `action` attribute of the login form (line 1 in Fig. 7) is used to process client side input. If a user enters the correct username and password, the user profile page will be displayed.

Script *main.js* is parsed after the creation of the login form, so it is possible these events fire before events' regis-

```
<!-- main.js -->
window.doLogin = function doLogin() {
    // preprocessing
    document.LoginPortletForm.submit();
}
window.executeViaEnter = function executeViaEnter(evt) {
    var charCode = ...;
    if (charCode == 13 || charCode == 3)  return doLogin();
    return true;
}
```

Figure 8: Event handlers defined in *main.js*.

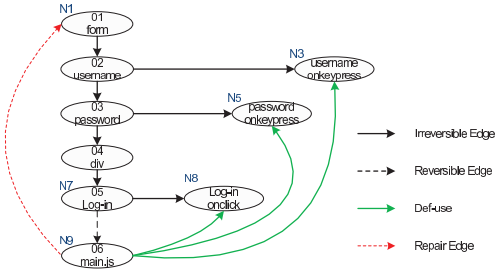

Figure 9: Causal Graph.

tration. We tested the effect of one of the races by intentionally delaying the execution of *main.js* to enforce that the onclick event happens before the definition of *doLogin()*. We found that even if the user enters the correct username and password, she will still be redirected to an error page (by the server side script).

The causal graph for this example is shown in Fig. 9. The def-use edges, $N9 \rightarrow N8$, $N9 \rightarrow N5$, and $N9 \rightarrow N3$, are not respected by the graph. For this case, the solver finds a single edge $N9 \rightarrow N1$ to fix all the three races. Note that the edge $N7 \rightarrow N9$ is reversible so that there is no cycle in the repair graph. ARROW then reorders the script *main.js* and puts it before the creation of the login form.

6.2 Case Study II

At the bottom of the page `metlimos.com`, there is a search bar composed of a text box and a search button, which allows the user to perform google search within the site. The background of this text box is a google custom search watermark image in gif format by default (see Fig. 10(a)). When user clicks the text box, the background should be set to be empty. When the text box loses focus, the background sets back to the default watermark image. Fig. 11 gives the code snippet of the onfocus and onblur events registration for the text box. These events are registered through an external script *brand(.js)* which is parsed after the creation of the search box. Before the external script is fetched, the onfocus and onblur events are not registered. At this point, if the user sets focus in the search box and types some text, the input text will be overlapped with the default watermark image as shown in Fig. 10(b).

The fixing of this case is similar to that of the motivating example in Fig. 6(b). The solver reports three repair edges. One edge from the textbox to the *brand(.js)* respects the def-use of the textbox. The other two edges start from the onblur and onfocus handler definitions and end at the calling of each event handler respectively. In order to implement the repair edges, ARROW reordered the handlers' registration at the point of textbox creation and also placed the handler definitions before the textbox.

7. RELATED WORK

Our work is closely related to client-side race detection of web applications [28, 19, 21, 10]. As mentioned in Section 1, these techniques only focus on detection but not repair. The

Figure 10: UI problem in *metlimos.com*

```
<!-- brand -->
01 var b = function() {if (q.value == '') q.style.background = '#FFFFFF
          url(http:\...\x2Fgoogle_custom_search_watermark.gif)';};
02 var f = function() {q.style.background = '#ffffff';};
03 q.onfocus = f;
04 q.onblur = b;
```

Figure 11: Event Handlers Registration in *brand(.js)*.

happens-before graphs in [19, 21] share some similarity with our causal graph. But their graphs are based on JS objects, constructed from trace, dynamic and precise. Our graphs are based on events, static and conservative. Our work is relevant to program synthesis and program repair. Le Goues et al. [14] apply heuristics based genetic programming to repair C programs. SearchRepair [12] generates patches based on semantic code search over large repositories of candidate code. DirectFix [16] considers semantic factors and generates the simplest program repairs for C programs. LeakFix [9] automatically fixes memory leaks in C program via static analysis. SemFix [17] generates patches using semantic program analysis via dynamic symbolic execution. Program transformations have also been successfully applied to fixing concurrency bugs [11, 8, 22]. Our work is also related to automatic web application repair. Selakovic and Pradel [24] proposed a pattern-based detection and repair method for performance problems in JavaScript programs. Nguyen et al. [18] and Samimi et al. [23] proposed an automatic technique to fix HTML generation errors in PHP scripts. Alkhalaf et al. [6] presented an automatic differential repair for vulnerable input sanitizers. Our work is related to preventions of concurrency related violations in general [27, 15, 13]. ConcBugAssist [13] aims to fix assertion violations by enforcing extra schedule orders. It encodes the detection of violating schedule as a Max-SAT problem and computes fixes by reducing it to a set covering problem. Besides introducing extra schedule orders as fixes, ARROW could additionally reorder the code to avoid synchronization overheads. However, ConcBugAssist relies on the observed dynamic executions. The fixes may introduce violations for unobserved executions. Comparatively, ARROW doesn't introduce new races, as it considers all def-uses via static analysis.

8. CONCLUSION

We develop a solver-based technique ARROW that can automatically fix race issues on client side web pages, while guaranteeing safety and achieving the cost-effectiveness. In our experiment, we use ARROW to fix 151 races from 20 real world commercial web sites.

9. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 14096 68, 1320444, and 1320306, ONR under contract N000141410 468, National Basic Research Program of China (973 Program) (Grant No. 2014CB340702), National Natural Science Foundation of China (Grant No. 61272080), and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

10. REFERENCES

[1] IBM, The T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/.

[2] Native JavaScript: sync and async. http://blog.getify.com/native-javascript-sync-async/.

[3] Quantifier Vs Non-Quantifier. http://stackoverflow.com/questions/10011478/quantifier-vs-non-quantifier.

[4] race condition for 'loaded' callback. https://github.com/mixpanel/mixpanel-js/issues/11.

[5] Race condition when loading images dynamically. http://web.onassar.com/blog/2013/10/09/.

[6] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 225–236, New York, NY, USA, 2014. ACM.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '08/ETAPS '08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] DongDong Deng, GuoLiang Jin, Marc de Kruijf, Ang Li, Ben Liblit, Shan Lu, ShanXiang Qi, JingLei Ren, Karthikeyan Sankaralingam, LinHai Song, YongWei Wu, MingXing Zhang, Wei Zhang, and WeiMin Zheng. Fixing, preventing, and recovering from concurrency bugs. *Science China Information Sciences*, 58(5):1–18, 2015.

[9] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe memory-leak fixing for c programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 459–470, Piscataway, NJ, USA, 2015. IEEE Press.

[10] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting concurrency errors in client-side javascript web applications. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 61–70, Washington, DC, USA, 2014. IEEE Computer Society.

[11] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association.

[12] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, Lincoln, NE, USA, November 2015. DOI: 10.1109/ASE.2015.60.

[13] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 165–176, New York, NY, USA, 2015. ACM.

[14] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, Jan 2012.

[15] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 318–329, New York, NY, USA, 2014. ACM.

[16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.

[17] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[18] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and T.N. Nguyen. Auto-locating and fix-propagating for html validation errors to php server-side code. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 13–22, 2011.

[19] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian 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.

[20] Sreeram Ramachandran. Web metrics: Size and number of resources. https://developers.google.com/speed/articles/web-metrics. Last updated: 26 May 2010.

[21] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[22] Veselin Raychev, Martin Vechev, and Eran Yahav. Automatic synthesis of deterministic concurrency. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 283–303. Springer Berlin Heidelberg, 2013.

[23] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE '12, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[24] Marija Selakovic and Michael Pradel. Automatically fixing real-world javascript performance bugs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 811–812, Piscataway, NJ, USA, 2015. IEEE Press.

[25] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon 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.

[26] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[27] Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 1–12, New York, NY, USA, 2014. ACM.

[28] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 805–814, New York, NY, USA, 2011. ACM.