Automatically Detecting Reflow Accessibility Issues in Responsive Web Pages

Paul T. Chiou
University of Southern California
USA
paulchio@usc.edu

Robert Winn
University of Southern California
USA
rwinn@usc.edu

Ali S. Alotaibi
University of Southern California
USA
aalotaib@usc.edu

William G. J. Halfond
University of Southern California
USA
halfond@usc.edu

ABSTRACT

Many web applications today use responsive design to adjust the view of web pages to match the screen size of end users. People with disabilities often use an alternative view either due to zooming on a desktop device to enlarge text or viewing within a smaller viewport when using assistive technologies. When web pages are not implemented to correctly adjust the page's content across different screen sizes, it can lead to both a loss of content and functionalities between the different versions. Recent studies show that these reflow accessibility issues are among the most prevalent modern web accessibility issues. In this paper, we present a novel automated technique to automatically detect reflow accessibility issues in web pages for keyboard users. The evaluation of our approach on real-world web pages demonstrated its effectiveness in detecting reflow accessibility issues, outperforming state-of-the-art techniques.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software testing and debugging**.

KEYWORDS

Web Accessibility, WCAG, Software Testing, Response Web Design, Reflow, Keyboard Accessibility, Inclusive Design

ACM Reference Format:
Paul T. Chiou, Robert Winn, Ali S. Alotaibi, and William G. J. Halfond. 2024. Automatically Detecting Reflow Accessibility Issues in Responsive Web Pages. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639229

1 INTRODUCTION

The internet has become a key aspect in the daily lives of billions of people. Many people rely upon it to provide essential information and services, especially individuals who are disabled [25]. Though the Internet was designed to be inclusive for all [5], previous research has shown that a significant portion of web applications today are not accessible to users with various types of disabilities [9, 26]. A recent study published in 2023 found that over 96% of home pages on the Web violate the most widely used accessibility standards [35]. Moreover, another study published in 2019 found that nearly 70% of web pages contained "accessibility blockers" that severely impeded their use to disabled users [11]. Despite there being legislation in a few places around the world that mandates companies to provide equally accessible websites [61], the accessibility of web pages remains an ever-present issue.

One of the most prevalent modern web accessibility issues is when users cannot access one or more of a web page's intended functionalities via Assistive Technologies (ATs). Many individuals with disabilities rely on ATs that oftentimes display web pages in a different layout format than that of traditional devices. Unfortunately, many web applications are not designed to be user-friendly across modalities that display the User Interface (UI) differently. For example, a recent study showed that over 50% of well-known web pages are not implemented to correctly adjust the page's content across different screen sizes and viewports [88]. The loss of content between different layout versions of a web page is problematic for those who do not interact with the web page at the viewport at which the web page was originally designed. In this paper, we refer to the manifestation of these issues as *Responsive Accessibility Failures (RAFs)*, where users that rely on ATs with smaller screen sizes are unable to access certain functionalities of the web page that are otherwise available in the full-sized version. RAFs have been one of the key focuses among web accessibility practitioners in recent years [29, 37], as Responsive Web Design (RWD) is becoming more popular among modern web applications [10].

Detecting RAFs is a non-trivial task that is complicated by several challenges. First, developers must determine what functionalities are defined in a web page. This is a labor intensive process due to the UIs of modern web pages typically being complex and possibly consisting of many different states that all require exploration. Second, in addition to exploring a single UI, developers must render the UI in different modalities to observe the behaviors of of the identified functionalities and verify whether they are in sync from one screen size to another. Lastly, even knowing the available functionalities on a page, one must determine whether they are accessible via ATs by testing all possible ways they can be accessed by a user using ATs. These challenges make it difficult for developers to thoroughly and accurately identify RAFs in their web applications.

Recent research has focused on identifying reflow-related problems in web UIs. Techniques [50, 51, 87–89] have attempted to check for layout presentation failures that cause visual discrepancies in

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639229

the rendering of a web page from its intended appearance. While such techniques can detect deviations that affect the aesthetics and layout of UIs, they do not focus on the functional and interactive aspects of web pages that cause RAFs. Other research focused on functional (i.e., keyboard operable) aspects of web accessibility [56, 57]. However, these techniques target different sets of problems and, therefore, cannot be used to detect RAFs. Acceptance testing techniques [6, 91] use a quality assurance process to determine if the requirements of a web app's specification are met. While these techniques have the potential to detect RAFs, they require developers to manually write scenario-based test cases to check for the accessibility problems (i.e., RAFs) anticipated for each web page. Therefore, to date, detecting RAFs remains a substantial manual effort that requires the users to interact with a web page and explore all available functionalities across multiple versions of the UIs, which can be error-prone and time-consuming.

In this paper, we present an approach for automatically detecting RAFs in web applications. Our approach first analyzes the UIs of both the full-sized version and the reflow version of a web page to model the available functionalities at each viewport. The approach then defines and employs a detection algorithm to analyze these two models to find the functionalities that would be missing (i.e., RAFs) for a disabled user when interacting with the page in the reflow version. We implemented our approach as a prototype, SALAD, to evaluate how well our technique could detect RAFs in real-world web pages. The results show that our approach was effective in detecting RAFs. The key contributions of this paper are (1) the first-ever automated technique to detect RAFs and (2) an empirical study on real-world web pages that shows our approach is effective in detecting RAFs.

2 BACKGROUND AND MOTIVATION

Assistive Technologies (ATs) help to improve the functional capabilities of individuals with disabilities to access the Web [13]. One of the most common input methods shared across many ATs uses the keyboard interface for interaction [22]. For example, users with physical, visual, or cognitive disabilities who lack the ability to use a point-and-click-based interface use ATs, such as voice activation, switches, screen-readers, or special hardware keyboards, that map to the keyboard interface. W3C's Web Accessibility Initiative (WAI) requires all web applications to be keyboard compatible and "all functionality must be usable with the keyboard" (e.g., users can access and move between links, buttons, forms, and other controls using the Tab key and other keystrokes). The keyboard interface navigates web pages via keys (e.g., Tab, Shift+Tab, or Enter) to move the keyboard cursor around the UI and activate the selected element.

Responsive web design (RWD) is a modern design and implementation approach that enables developers to build web pages that provide an equivalent user experience regardless of the size of the end-user device [8]. Rather than having separate versions of web pages for mobile devices that have small screen sizes, RWD allows the web page to dynamically modify its layout to adapt to the size of a device's display. RWD poses several advantages, such as not requiring users to zoom in on page contents that are too small to read or needing to horizontally pan around pages that are too wide to fit on smaller screens. However, improperly designed or implemented RWD can cause accessibility issues for people with different disabilities, especially those that rely on the ATs and keyboard-based inputs. We describe how RAFs can manifest in web pages next.

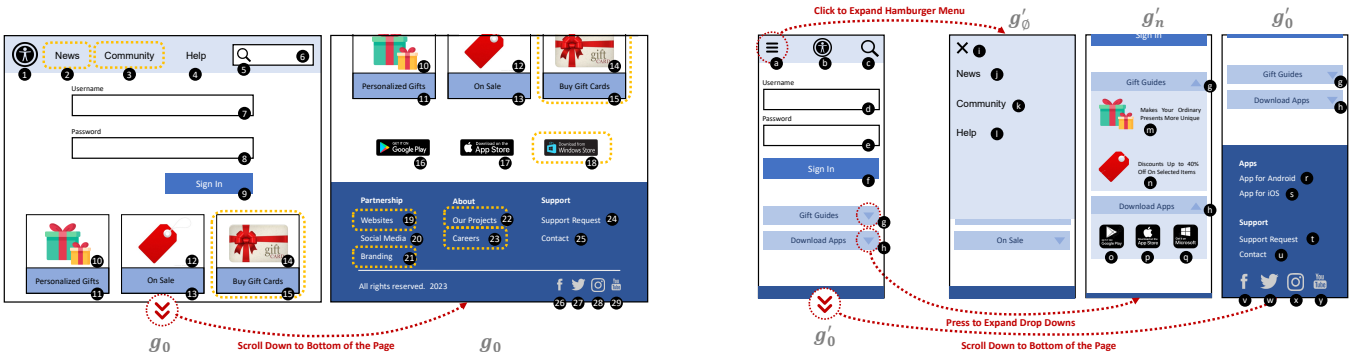
2.1 Responsive Accessibility Failures (RAFs)

2.1.1 Definition. In RWD, the concept of *reflow* [12] is the process of fitting web elements to the width of a web page so that the content remains within the boundaries of the browser's viewport when users zoom in to enlarge content or view the content via smaller devices. Web Content Accessibility Guidelines (WCAG) 2.1 Success Criterion (SC) 1.4.10 [23] requires that a web page has a reflow mechanism so that the page content can be presented without being cut off by the viewport and does not require scrolling in the direction of reading. The SC focuses on accessibility from both perceivable and operable aspects. For perceivable, it ensures that the content can be presented in one column and read without significantly increasing the effort required. From an operable standpoint, it requires that when displayed under reflow, the page content is without a "loss of functionality". The Success Criterion is considered to be met as long as all content and functionality are still fully available - either directly, revealed via accessible controls, or accessible via direct links.

RAFs generally affect users of various types of assistive technologies through the reflow version of web apps. Due to the versatility of assistive technologies on desktop apps, people who use these different kinds of setups are often using regular desktop browser agents with different-sized viewports (e.g., those that rely on screen magnification to enlarge the web content). SC 1.4.10 ensures that nothing is removed irrevocably for users on smaller viewports. WCAG defines the full-sized viewport as the default dimension rendered at 1280 × 1024 CSS pixels and the small (reflow) viewport as those rendered at this same default dimension with a 400% scale along the longer side of the web page (equivalent to 320 CSS pixels wide for vertically oriented pages or 256 CSS pixels tall for horizontally oriented pages) [23].

2.1.2 Motivating Example. A RAF occurs when functionalities in the full-sized version are not accessible in the reflow version. When a responsive web page is reflowed, fully visible contents in the full-sized layout are often collapsed into fewer items to fit the smaller horizontal screen space. Figure 1 shows an example web page where the navigation menu of the full-sized version of the web page is collapsed into a single hamburger menu button (ⓐ) [24] in the reflow version. Similarly, the "Personalized Gifts" and "On Sale" links are collapsed under the "Gift Guides" dropdown menu (ⓑ). This example also contains an RAF: the custom hamburger menu button is implemented using a element that is not accessible to the keyboard (i.e., it cannot receive keyboard focus). Thus, the collapsed menu cannot be expanded, and the functionalities associated with the contained menu items ("News" (ⓙ) and "Community" (ⓚ)) are unavailable to keyboard-based users.

The figure shows another example where a section of the web page's footer completely disappears from the page after reflow. In the reflow version, the "Websites", "Branding", "Our Projects", and "Careers" links are no longer present in the UI. As a result, the functionalities associated with those links in the footer are no



(a) Full-Sized rendering at 1280 CSS pixels wide

(b) Reflowed rendering at 320 CSS pixels wide (400% scale)

Figure 1: A responsive web page with functionalities not available for keyboard users under reflow.

longer available to the users. We use dotted rounded rectangles to highlight the RAFs in the web page's full-sized UI in Figure 2a.

2.2 Detecting RAFs

Detecting RAFs is inherently challenging because of the amount of manual labor and domain-specific knowledge involved in the process. A straightforward analysis that examines a page's structure or the syntax of interactive elements is insufficient for RAF detection. Instead, a developer must dynamically simulate all potential interactions available for a given web page in both the full-sized and reflow versions to ensure a comprehensive identification of the available functionalities. This process is further complicated because the UI functionalities may be designed and represented differently across the full-sized and reflow versions. For example, as shown in Figure 1, the full-sized version of the web page displays a search functionality as a search input, whereas in the reflowed version this functionality is represented as a link and is hidden under the search menu (not shown). Such examples showcase how difficult it can be for developers to determine whether a loss of functionality between modalities has occurred or if the functionality has simply been expressed differently. As discussed in Section 5, existing work in accessibility issue detection does not target RAFs. The closest work that focuses on RWD detects presentation problems that distort the layout of web pages [50, 51, 87–90]. These techniques model the layout of a web page over a range of different viewport widths to check collision (two elements overlapping), protrusion (an element overflowing its containing elements), or element(s) that disappear altogether off the edge of the viewable portion of the page as the viewport width changes. While these techniques are effective at identifying layout failures, they do not specifically target RAFs, and as we show in the evaluation, they may not lead to accurate RAF detection.

3 APPROACH

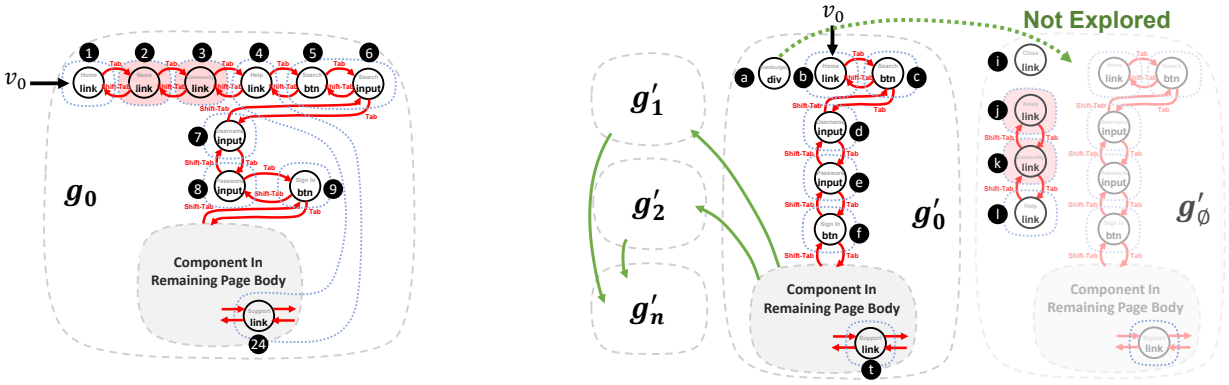
The goal of our approach is to automatically detect RAFs in a web page. Modern Rich Internet Applications (RIAs) have pages with structures and designs that complicate the automatic detection of RAFs. These web pages consist of various states, making it challenging to predict how keyboard users can navigate and determine the UI functionalities that should be accessible. By definition, identifying RAFs involves reasoning about functionalities across different modalities (i.e., full-sized versus reflow versions). This task requires understanding the keyboard navigation and UI functionalities, along with identifying discrepancies between the functionalities of the two versions. Our approach uses problem definition and domain-specific insights to overcome these challenges. Our primary insight is that the functionalities available in the full-sized version represent the functionalities intended by the developers and, therefore, represent the set of functionalities that should be checked for in the reflow version.

Figure 3 shows a workflow overview of our approach and a more formal version is shown as Algorithm 1. The input is a Page Under Test (PUT). The approach begins by creating two UI interactive models: one for the full-sized and one for the reflow UI (Section 3.1). Both of these models are then processed to extract their UI functionalities (Section 3.2). The approach then compares these UI functionalities while analyzing their UI interactive models to determine which functionalities are available via the keyboard in the full-sized but not the reflow version of the PUT (Section 3.3). The output of our approach is a list of detected RAFs for the PUT.

3.1 Modeling the UI Keyboard Interactivities

The first step in automatically detecting RAFs is to capture how a keyboard user can interact with a web page's UI via the keyboard. A challenge in capturing such keyboard interactions is that the way a web page behaves in response to keyboard inputs may not be apparent via the source code. Instead, it requires observing how the page reacts to different keystrokes' actions at different states of the UI. To overcome this challenge, the approach uses dynamic crawling techniques inspired by prior work on keyboard accessibility [56–58] to build a similar interactive model called the User Interface Interactive Model (UIIM). The UIIM represents the PUT's UI and possible behaviors based on a user's run-time interactions using the keyboard.

3.1.1 Definition. The UIIM is formally defined as G, a set of UI states the PUT can be in based on a user's keyboard input. Each UI state of the PUT $g \in G$ is itself a graph representing both the HTML elements available to a user and the way a user can navigate among them using the keyboard. Formally, we define a UI state g as



(a) UIIM for the Full-Sized rendering of the page

(b) UIIM for the Reflowed rendering of the page

Figure 2: A simplified version of the User Interface Interactive Models (UIIMs) for the web page in Figure 1, showing how the keyboard interaction navigates and changes the UI.

a directed graph of the form $\langle V, E, v_0, \Phi \rangle$, where V is the set of nodes that represent elements with interactive behaviors in the state; E is the set of directed edges that represent the corresponding navigation flow among the interactive elements in V; v_0 represents the entry node, which is the element that receives the initial keyboard focus in the state; and Φ is the set of keystroke actions that could be carried out by a keyboard-based user to navigate the UI, which includes {Tab, Shift + Tab, ↑, ↓, ←, →, Enter, Space, Esc} as specified by the W3C Authoring Practices [4]. Each UI state is uniquely identified by its set of nodes in the UI (i.e., $g.V$).

A node $v \in V$ is an element in the PUT that the user can interact with (i.e., navigate, activate, enter text, or provide input to). These interactive UI elements include (1) all the native control elements, such as HTML links, inputs, and form controls (i.e., <a>, <button>, <input>, <select>, and <textarea>) [3] as well as (2) non-native control elements that have been customized with interactivity (e.g., customized buttons that are implemented using or).

A directed-edge $e \in E$ is defined as a tuple $\langle v_s, \phi, v_t, \delta \rangle$ (where $v_s, v_t \in g.V$), indicating that when a source node v_s is in focus, the browser's keyboard focus shifts to a target node v_t by pressing key $\phi \in \Phi$. In cases where the key press causes new nodes to become present (or existing nodes to become absent) in the UI, we characterize the edge as an *inter-state edge*, since the key press also causes the user to transition to a new UI state. In other words, an *inter-state edge* has the property $v_s \in g_s.V$, $v_t \in g_t.V$ where g_s and g_t represent the old and new UI state respectively. In addition to focus transition, we use the δ Boolean property to represent whether the key ϕ causes any visible changes to the UI.

Examples of UIIMs are shown in Figure 2 as graphs where UI elements are nodes (solid circles), and the edges are arrows between each node, showing how the keyboard focus transitions from one element to the next during keyboard navigation. A UI state (i.e., g) is a big dotted bubble. Intra-state edges (red) are those whose endpoints belong in the same UI state, and inter-state edges (green) cross UI states that occur when its keyboard action changes the UI.

3.1.2 Construction. The construction of the UIIM dynamically explores the client-side UI using keyboard interaction. The approach first identifies the nodes (i.e., V) by rendering the PUT in a browser, then analyzing its Document Object Model (DOM) to identify each unique interactive element.

The approach then executes all the keyboard operations (i.e., Φ) on each interactive element $v \in V$ to identify the resulting keyboard navigation. After each action, the approach detects if a change in focus occurred by querying the browser to identify the element currently receiving focus in the PUT. In addition to identifying focus transition, the approach also queries the DOM to check if the action resulted in a new UI state. The approach creates *intra-state edges* for actions that transition focus without a change of state and *inter-state edges* for those that change states (i.e., $g_s \neq g_t$). For all the new UI states explored, the approach repeats the construction described above until no new navigation or UI state is found (i.e., the graph has reached a fixed point). In the case where the web page dynamically loads in content forever (e.g., infinite scrolling), our approach can be customized with an upper bound on the maximum depth of exploration to limit the number of new states to explore. The approach defines an edge's δ property to be `true` if ϕ's action triggered any visual changes to the DOM that is unrelated to changing the UI state.

The approach first builds the UIIM for the PUT rendered at both the full-sized version (i.e., G) and the reflow version (i.e., G'), which is shown in lines 2 and 3 of Algorithm 1. We omit the details of building the UIIM since the construction process is similar to related work [56–58].

3.2 Modeling the UI Functionalities

The second step in automatically detecting RAFs is to understand what UI functionalities are present in the PUT. Knowing the functionalities of the PUT is a key capability required in detecting RAFs because the approach must have a concrete idea of "what" component needs to be (or is not) accessible in the full-sized UI in order to compare with those components in the reflow UI. To date, there is no standardized way to define what a functionality is in the web UI. Existing work mainly focuses on identifying the semantics of "GUI widgets", which are single interactive elements [62, 79]. At a high level, we introduce the concept of "UI functionalities" to represent user operation(s) that accomplish a unique task when

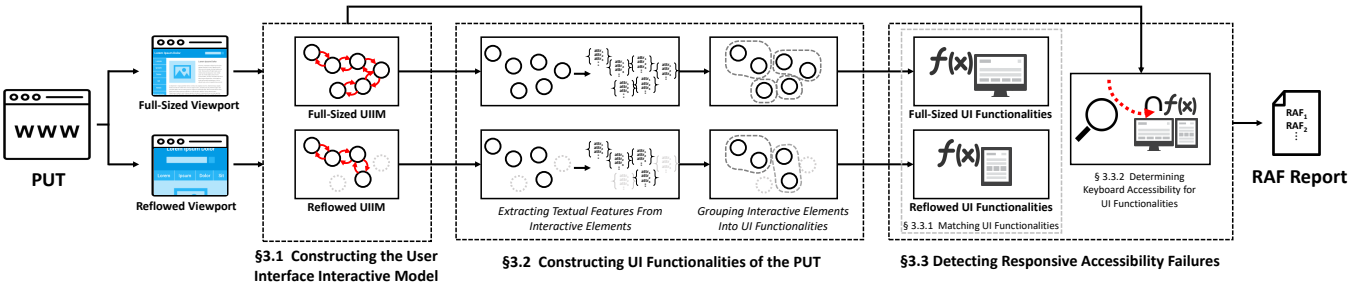


Figure 3: An overview of our approach.

interacting with the PUT's UI. In this section, we formally define the "UI functionalities" of the PUT and how to model them.

3.2.1 Definition. The approach defines a UI functionality (i.e., f) as a set of interactive elements that perform a similar task (e.g., buttons with similar Javascript function calls, drop-downs that contain similar content, or links that navigate to similar pages with different API parameters). The primary challenge in identifying UI functionalities is determining similar interactive elements. This is because multiple interactive elements can achieve the same functionality, but each may be implemented differently. For example, in our running example, both the Google Play store icon and the "App for Android" link serve the same functionality of taking the user to the page to download the Android app. However, one is a link, and the other is a customized button that calls for a JavaScript function to navigate to the download site. Simply relying on interactive elements' `id`s or labels alone is not enough to identify or differentiate functionalities.

To overcome these challenges, we examine the additional textual attributes associated with the elements to express the task being performed. Our insight is that the performed task is often depicted as the "action" defined by the associated attributes and text content description. Therefore, we formally define an interactive element as the tuple $\langle x, S_{fun}, S_{tag}, S_{inp}, S_{lab}, S_{txt} \rangle$ where x is the XPath that uniquely identifies the element in the DOM and $S_{fun}, S_{tag}, S_{inp}, S_{lab}, S_{txt}$ each is a set of strings that represents one of the five types of textual attributes associated to the element. We chose these five types of textual attributes (which we refer to as "features") that comprise all of the properties that are defined in HTML form control documentation [16–21] as the following:

(1) **function** – the destination directed by a link element (i.e., `href` attribute) or the event handler JavaScript function call signature of links and controls (e.g., buttons).
(2) **tag type** – the type of element such as <a>, <button>, <select>, <input>, etc.
(3) **input attributes** – the attributes of input controls including general attributes i.e., `id`, `type`, `name`, `value`; boolean attributes i.e., `readonly`, `disabled`, `multiple`, `required`, `autofocus`, `autocomplete`; value attributes i.e., `size`, `maxlength`, `min`, `max`, `pattern`, `placeholder`, `step`, `list`; and form attributes i.e., `form`, `formaction`, `formenctype`, `formmethod`, `formtarget`, `formnovalidate`, `novalidate`.
(4) **labels** – the descriptive labels associated with the element.
(5) **textual properties** – text-based DOM (type 3) nodes wrapped inside of the element. (e.g., text of buttons or custom controls).

3.2.2 Constructing UI Functionalities of the PUT. The approach analyzes the full-sized and the reflow UIIMs and identifies the available functionalities in their UIs. To do this, the approach defines a function $\mathcal{F} : G \rightarrow F$ that takes a UIIM (i.e., G) as the input and identifies all of the functionalities in it. The output is a set of functionalities across all the possible UI states of the PUT for G. The function $\mathcal{F}$ is shown in lines 16 to 36 in Algorithm 1.

The function $\mathcal{F}$ starts by analyzing the dynamically rendered DOM at every explored UI state of the UIIM, which allows us to get accurate run-time properties that represent exactly what the users would see as they interact with the PUT. The function first analyzes the DOM representation of each UI state (i.e., $g \in G$) and iterates over all their interactive elements to create a set of unique interactive elements (i.e., V_G) based on their XPaths. Next, the function groups interactive elements that perform similar tasks together as the same UI functionality (i.e., V_{sim}). This is done by using a function $\mathcal{S}$ (shown in Equation (2), used on line 26 of Algorithm 1) that takes a pair of interactive elements (i.e., (v_a, v_b)) to determine if they have equivalent features. The result of this step (i.e., $\mathcal{F}$) is the set of functionalities available across all of the states in the full-sized UI (i.e., F_G) and those of the reflow (i.e., $F_{G'}$) UI of the PUT.

In Figure 2, a functionality is shown as blue dotted ovals within a UI that groups node(s) (i.e., interactive element(s)) together. For example, in the initial UI state of the full-size UIIM (i.e., g_0), the "Help" (❹) and "Support Request" (㉔) links are grouped together because they are considered a single functionality that takes the user to the help/support page.

Identifying Interactive Elements That Perform Similar Tasks. To determine which interactive elements perform similar tasks, we use NLP techniques (e.g., the Word2Vec model [74]) to capture the context from their associated features. A characteristic of Word2Vec is that semantically related words are close together in terms of their cosine similarity. The approach defines a similarity function $\text{sim}(v_a, v_b)$ that returns an overall similarity score between a given pair of interactive elements. The function first computes the cosine similarity between each pair of corresponding features (e.g., $i_a.S_{lab}$ and $i_b.S_{lab}$). The function then assigns a weight to each of the five features based on the relative importance of the feature. Our insight is that in the context of web applications, interactive elements that achieve the same task may share commonalities in certain features over others. For example, the "function" and "tag type" features would be considered to have more weight than the "input attributes" feature. The reason is that in design, a link (or input field) is more

likely to be implemented as the same type of element or with the same actions/function-calls.

The overall similarity score is then calculated by computing the weighted average of the cosine similarities across the five feature vectors as shown in Equation (1). In the equation, the respective weights of the five features (i.e., w_k) are multiplied by their corresponding cosine similarities and then divided by the sum of all the weights. The approach defines the helper function $\mathcal{S}$ and uses a threshold θ to determine if the two interactive elements (i.e., v_a and v_b) are similar enough to be equivalent. v_a and v_b do not have equivalent semantics if their overall similarity is less than θ.

$$\text{sim}(v_a, v_b) = \frac{\sum_{k \in \{fun,tag,inp,lab,txt\}} \left[w_k \cdot \text{sim}_{cos}(S_k \in v_a, S_k \in v_b) \right]}{\sum_{k \in \{fun,tag,inp,lab,txt\}} w_k} \tag{1}$$

$$\mathcal{S}(v_a, v_b) = \begin{cases} \text{true}, & \text{if sim}(v_a, v_b) \geq \theta \\ \text{false}, & \text{otherwise} \end{cases} \tag{2}$$

Fitting the Feature Weights and Similarity Threshold. To determine the weight of each feature and the value θ, we experimented on a small sample of web pages on the Internet. We began by examining websites returned from *Discuvver.com* (a service that returns a random website) and selecting 150 that implement RWD. For each page, we manually identified and grouped all interactive UI elements (e.g., buttons, inputs, links, drop-down menus) that achieve the same functions on the page, such as links with the same destination. We considered these groups our ground truth, representing elements serving the same functionality. We then applied our similarity function $\mathcal{S}$ to these elements to categorize them into functional groups within the UI. We measured the accuracy of how well the groups identified by $\mathcal{S}$ matched the manually determined ground truth. With over a thousand groupings, we tuned the weights associated with computing the similarity score (Equation (1)) and the similarity threshold (Equation (2)) for an optimal accuracy of approximately 97%. To help account for the functionalities whose textual features may slightly differ from the change in design in RWD, the groupings also included those with the same functionalities across their respective full-sized and reflow versions. We present the specific numerical values for the weight of each feature set and the similarity threshold θ in Section 4.1.

3.3 Detecting Responsive Accessibility Failures

The approach detects RAFs by analyzing and comparing the constructed UIIMs (i.e., G and G') and the identified set of functionalities (i.e., F_G and $F_{G'}$) for the PUT. An RAF is defined as those functionalities that are keyboard accessible in the full-sized version but not in the reflow version. Algorithm 1 shows the detection process from line 6 to line 14.

The approach first checks if each functionality $f_G \in F_G$ can be accessed via the keyboard (we describe how to determine whether a functionality is keyboard accessible in Section 3.3.2). Those that are keyboard accessible represent the functionalities "available" to keyboard users in the full-sized UI. For each of these "available" functionalities, the approach further analyzes to check if it is also keyboard accessible in the reflow UI. To do this, the approach first identifies the functionality in the reflow UI (i.e., $f_{G'} \in F_{G'}$) that corresponds to the "available" functionality f_G by matching any functionality that has equivalent features (we discuss the process for matching functionalities across UIs in Section 3.3.1). We call the equivalent functionality (i.e., $f_{G'}$), the counterpart of f_G. The approach then checks if this counterpart $f_{G'}$ is keyboard accessible. For those "available" functionalities $f_G \in F_G$ that either (1) have no equivalent counterpart, or (2) whose equivalent counterpart $f_{G'}$ is not keyboard accessible, the approach considers them as RAFs.

Algorithm 1 Our Approach

Input: Page Under Test (*PUT*),
Output: $\bar{F}$: Set of unavailable functionalities under reflow

```
1:  Initialize F̄ = ∅
2:  G ← construct UIIM model at full-size rendering of the PUT        ▷ /* Section 3.1 */
3:  G′ ← construct UIIM model at reflow rendering of the PUT          ▷ /* Section 3.1 */
4:  F_G = F(G)                                                        ▷ /* Section 3.2 */
5:  F_G′ = F(G′)                                                      ▷ /* Section 3.2 */
6:  for all f_G ∈ F_G do
7:      if ¬ isKeyboardAccessible(f_G) then continue;                 ▷ /* Section 3.3.2 */
8:      isFuncInReflowAndAccessible ← False
9:      for all f_G′ ∈ F_G′ do
10:         if S̄(f_G, f_G′) ∧ isKeyboardAccessible(f_G′) then         ▷ /* Equation (3), Section 3.3.2 */
11:             isFuncInReflowAndAccessible ← True
12:     if ¬ isFuncInReflowAndAccessible then
13:         add f_G to F̄
14: return F̄
15:
16: procedure F(G)
17:     V_G ← ∅                                   ▷ /* Set of unique interactive elements of the PUT */
18:     for all g ∈ G do
19:         for all v ∈ g.V do
20:             V_G ← V_G ∪ v
21:     V_simSet ← ∅                              ▷ /* Set of set of similar interactive elements of the PUT */
22:     for all v_a ∈ V_G do
23:         isExist ← False
24:         for all V_sim ∈ V_simSet do
25:             for each v_b ∈ V_sim do
26:                 if S(v_a, v_b) then isExist ← True                ▷ /* Equation (2) */
27:         if ¬ isExist then
28:             V_new-sim ← ∅ ∪ v_a
29:             V_simSet ← V_simSet ∪ V_new-sim
30:         else
31:             V_exist-sim ← the set of interactive elements where similarity was found
32:             V_exist-sim ← V_exist-sim ∪ v_a
33:     F ← ∅                                     ▷ /* Set of UI functionalities of the PUT */
34:     for all f ∈ V_simSet do
35:         F ← F ∪ f
36:     return F
37:
38: procedure isKeyboardAccessible(f)
39:     V_sim ← the set of similar interactive elements of f
40:     Initialize isAccessible ← False
41:     for all v ∈ V_sim do
42:         v_0 ← the initial node in g.V where v resides
43:         if isReachable(v_0 → v) then                              ▷ /* Check for keyboard focusability */
44:             E ← the set of outgoing edges from v
45:             for all e ∈ E do
46:                 if e.ϕ = Enter ∨ e.ϕ = Space then                 ▷ /* Check for keyboard actionability */
47:                     if e.δ then isAccessible ← True
48:     return isAccessible
```

3.3.1 *Matching UI Functionalities Between Full-Sized and Reflow UIs.*

To determine if a functionality present in the full-sized UI is also present in the reflow UI of the PUT, it is necessary to identify the same functionality across these different versions of UIs. Automatically identifying the same functionalities across the full-sized and the reflow version of a web page is non-trivial because functionalities in F_G and $F_{G'}$ are, by nature, displayed in different screen modalities. Traditional web analysis techniques [43, 44, 67–70] have used elements' XPaths as identifiers for elements in the UIs. However, because the reflow version of the PUT may differ

in layout and have completely different DOM tree structures, the same functionality can result in different XPaths. Therefore once the UI undergoes reflow and the design changes to another version, we can no longer rely on certain properties (e.g., XPath, label text, visibility, CSS appearances, etc.) to identify the functionalities. To address this challenge, we leverage the textual attributes associated with a functionality's interactive elements, as discussed in Section 3.2.2. Our insight is that the syntax of interactive elements usually remains similar despite changes to its visual-related layout properties after reflow.

Our approach matches equivalent functionalities between the full-sized and reflow UIs using a mechanism similar to that which matches similar interactive elements, as described in Equation (1). The approach defines a function $\widehat{\mathcal{S}}$, shown in Equation (3), to check if two functionalities (e.g., f_a and f_b) are equivalent regardless of which UI they reside in. The function examines the set of interactive elements in the two functionalities (i.e., respectively V^a_{sim} and V^b_{sim}) and does a pairwise matching using the function $\text{sim}(v_a, v_b)$ in Equation (1) to check if they are equivalent. The equivalence is determined by checking if the highest similarity score between all the interactive element pairs (i.e., all the ordered pairs (v_a, v_b) in the Cartesian product of V^a_{sim} and V^b_{sim}) is greater than the value θ.

$$\widehat{\mathcal{S}}(f_a, f_b) = \begin{cases} \text{true}, & \text{if } \left(\max_{(v_a, v_b) \in V^a_{sim} \times V^b_{sim}} \text{sim}(v_a, v_b) \right) > \theta \\ \text{false}, & \text{otherwise} \end{cases} \tag{3}$$

The approach uses this function $\widehat{\mathcal{S}}$ in line 10 of Algorithm 1 to find f_G's corresponding counterpart in the reflow UI (i.e., $f_{G'} \in F_{G'}$) to further check if $f_{G'}$ is keyboard accessible, which we describe in the next subsection.

3.3.2 Determining Keyboard Accessibility for UI Functionalities. We use the procedure isKeyboardAccessible (line 38 to line 48) in Algorithm 1 to determine if a functionality is keyboard accessible. The procedure takes a functionality (i.e., f) as the input and examines all of its associated interactive elements (i.e., $v \in V_{sim}$) to see if any satisfies the two conditions that make them keyboard accessible [31, 32]. We classify the functionality as accessible if there exists at least one interactive element that passes both conditions below, representing that a keyboard-based user would be able to focus on and activate the given functionality via that element.

Condition One – Focusability. The procedure first checks if an interactive element is included in the PUT's keyboard navigation (lines 41 to 43). Our insight is that the UIIM is constructed by crawling the UI with all the possible navigation keystrokes to explore new UI states. Each time a new state is explored, the approach repeats the analysis to build the sets of nodes and edges (i.e., V and E) for this new state. Therefore, reachability from a UI state's v_0 would indicate that a keyboard user can use the standard keyboard navigation keys to put the keyboard focus on the interactive element to access its associated functionality. The procedure performs a reachability analysis on the UI state (i.e., g) where the given functionality's associated interactive element (i.e., v) resides. The analysis examines if v is reachable from the starting point of navigation for keyboard users of the UI state (i.e., $v \in g.V$ can be reachable through the set of directed edges $g.E$ starting from $g.v_0$).

Condition Two – Actionability. The second condition for an interactive element v to be accessible is for it to be actionable through the keyboard. A naive way to check this would be to examine if v has a keyboard event handler associated with it. However, it is difficult to correctly identify whether the event-handling function contributes to activating the element to make something happen. Our approach overcomes this by analyzing the UI state and its DOM. Our insight is that during the crawling, any activation can be represented by a change to the PUT's visible attributes and reflected through an edge's δ Boolean property. In lines 46 and 47, the procedure checks whether executing keystrokes Enter or Space, would also cause such a change to δ. An edge whose ϕ is either Enter or Space, with a δ = `true` indicates that after the focus is set on the element, a keyboard user can activate their underlying functionalities to trigger a reaction using these activation keystrokes.

In Figure 2, the RAFs in the header of the page are represented by the red/shaded dotted ovals (i.e., the functionalities associated with "News" (❷) and "Community" (❸)). while these functionalities are reachable from the starting navigation point (i.e., v_0) in the full-size UIIM, they are not reachable in the reflow UIIM. This is because the hamburger menu element (i.e., ⓐ) is not accessible; therefore, the derived UI state (i.e., $g'_{\varnothing}$ where these functionalities reside) is never explored via the keyboard. Although the help/support functionality is also under this unexplored UI state, it can be reached via the "Support Request" (❶ equivalent in $g'_{\varnothing}$) link and is not a RAF.

4 EVALUATION

To assess the effectiveness and usefulness of our approach, we conducted an evaluation focusing on the following three research questions:

RQ1. What is the accuracy of our approach in detecting RAFs in web pages?

RQ2. How fast is our approach at detecting RAFs?

RQ3. How impactful are the RAFs that our approach detected?

4.1 Implementation

We implemented our approach as a Java-based prototype tool called re**S**ponsive **A**ccessibi**L**ity f**A**ilure **D**etector (SALAD). The approach utilized the Selenium (ver. 3.141.5) WebDriver API to drive the Firefox browser (ver. 92.0) to render the PUT in different display modalities. The full-sized version of the PUT is rendered by setting Firefox's viewport dimension to 1280 × 1024, while the reflow version of the PUT is rendered at the viewport dimension of 320 × 1024. Note that evaluating at viewports of 1280px and 320px follows the WCAG evaluation protocol [39] and is considered the industrial standard in testing for SC 1.4.10 (i.e., reflow) violations [38].

The WebDriver is used to (1) execute keyboard-based [14, 15] actions to interact with the rendered PUT and (2) execute JavaScript to analyze the functionalities' properties from the PUT's UI, to construct the UIIMs. Note that for the UI exploration, we set a maximum depth of five to terminate the crawling process for the

practicality of evaluation. The weights for the similarity function presented in Equation (1) and the similarity threshold (i.e., θ) in Equation (2) used in the paper's evaluation were the following: $w_{fun} = 0.45$, $w_{tag} = 0.14$, $w_{inp} = 0.18$, $w_{lab} = 0.14$, and $w_{txt} = 0.09$, $\theta = 0.98$. These values were found most effective and were determined via the experiment discussed in Section 3.2.2. The setting mentioned above is configurable, allowing the user of SALAD to select a threshold that best fits the desired trade-off between the number of false positives and true negatives reported by the tool.

The approach made use of the deeplearning4j (dl4j) library [7] to carry out NLP-based operations (e.g., pre-processing the extracted textual data using tokenization and stop-word elimination) and build a Word2Vec model, which is then used to measure the cosine similarities between functionalities. Lastly, the evaluation was run on an AMD Ryzen Threadripper 2990WX with 64GB memory running 64-bit Ubuntu Linux 18.04.4 LTS. The complete set of experiment data has been made public to the community [40]. Other details about the tool and the Appendix of the paper are available via the project website [41].

4.2 Experiment Setup

4.2.1 Selecting Subject Web Pages. We conducted our evaluation on a dataset of 62 subject web pages. We selected the subject web pages from the Moz Top 500 most visited website list [33] and the artifacts of two related works: ReDeCheck [88] and KAFE [57]. The subject web pages from the Moz Top 500 were randomly selected based on the WCAG-EM sampling strategy [39], with an aim to collect a representative sample of real-world, live web pages. Our identified set of sample web pages includes common web pages of the websites with varying styles, layouts, structures, and essential functionalities. Their page size averages 273 UI elements with a minimum, maximum, and median of 19, 4013, and 165 elements, respectively. Due to space constraints, the complete list and size properties of all subjects are included in the Appendix.

We compiled an initial set of 126 web pages from the previously mentioned three sources, with 40 subjects being acquired from the Moz Top 500, 26 from ReDeCheck's evaluation dataset [87], and 60 from KAFE's evaluation dataset [57]. From this initial set, we filtered out 7 subjects that were either (1) not designed according to responsive design principles [27], as such websites fall outside the scope of this study, or (2) could not be executed due to issues with the cached versions of the subjects from related works. From the filtered set of 119 subjects, we selected all 47 subjects that were found to have contained RAFs. From the same set of 119, we also randomly selected an additional 19 subjects without any RAFs to account for potential false positives. Lastly, from the chosen set of 66 subjects, we excluded 4 subjects on which SALAD could not be ran due to issues with the Selenium WebDriver API being unable to properly interact with the web page. Thus, our finalized resultant set of subjects consisted of 62 web pages: 19 from the Moz Top 500, 9 from the evaluation of ReDeCheck, and 34 from the evaluation of KAFE.

4.2.2 Building the RAF Ground Truth. We built the ground truth for a given subject web page by following the testing procedure described in WCAG Technique F102 [38]. In particular, we manually interacted with the full-sized and the reflow versions of the web page and identified functionalities that existed in the full-sized version but were either absent or inaccessible in the reflow version. Any such functionalities were recorded as RAFs. The process described above was independently carried out by two authors, who both had prior experience in web development and accessibility testing. After the two independent sets of ground truths were completed, the two authors identified the discrepancies between their respective sets. The two authors had a high agreement (99%), with only five discrepancies being identified where then a third author acted as a validator to resolve these discrepancies. We discuss the potential threat this protocol may impose on our results in Section 4.6. In total, our subjects contained 559 RAFs, with an average of 13 RAFs per subject that was found to have contained RAFs.

4.2.3 Selecting Tools for Comparison. We selected four state-of-the-art tools to include in our evaluation to aid in evaluating SALAD's performance: ReDeCheck [88], KAFE [57], Qualweb [59], and WAVE [36]. The first three tools were selected from (1) popular accessibility testing tools described in previous literature, and the fourth tool was selected from (2) the Web Accessibility Evaluation Tools List [1]. Note that the commercial tool Axe [28] was not included because its Guided Test mode is not fully automatic and requires manual human assistance to detect keyboard accessibility issues.

A challenge that we faced in our comparison was that none of these four tools directly detected RAFs. Therefore, we used the tools and/or interpreted each of their detection results in a way that would be most successful in detecting RAFs while not including in the accuracy calculation, detections of other types of accessibility issues that could, by definition, not be RAFs. For ReDeCheck, since it reports relative layout failures at different viewports, we only considered the results at the viewport equal to the reflow version (i.e., 320 pixels), and then evaluated its reported failures against the RAF ground truth. For KAFE, which detects specific types of keyboard accessibility issues, we evaluated all issues reported by the tool against the RAF ground truth. Finally, for Qualweb and WAVE, we evaluated only their results for the subset of WCAG guideline checks that dealt with keyboard or reflow-related accessibility failures.

4.3 RQ1. Effectiveness of SALAD

4.3.1 Protocol. We measured the effectiveness of SALAD by evaluating how accurate, in terms of precision and recall, it was in identifying RAFs and compared its results against those of ReDeCheck, KAFE, Qualweb, and WAVE. Precision was calculated by dividing a given tool's total number of correctly identified RAFs by the total number of RAFs that the tool identified. Recall was calculated by dividing a given tool's total number of correctly identified RAFs by the total number of RAFs in the ground truth.

Table 1: Approaches' RAF Detection Accuracy

	SALAD	ReDeCheck	KAFE	WAVE	Qualweb
Precision	85%	18%	1%	0.2%	0%
Recall	94%	4%	0.6%	0.2%	0%

4.3.2 Results. The results of RQ1 are displayed in Table 1. SALAD achieved a precision of 85% and a recall of 94%, which outperforms

the other evaluated state-of-the-art tools, on our dataset. Out of the total 559 RAFs in our subjects, SALAD was able to correctly detect 526 (94% of all RAFs). Note that ReDeCheck and KAFE fail to produce an output on a number of different subject web pages, 5 and 6, respectively. Thus, their precision and recall are derived solely from the results of the subjects on which they are able to run.

Upon further investigation, we find that our approach does not correctly detect RAFs in two general scenarios: (1) our approach fails in the functionality grouping procedure; and (2) an incomplete UIIM. The first scenario occurs due to inaccuracies when identifying functionalities in the UI. In the first scenario, our approach either incorrectly groups different functionalities together as a single functionality resulting in two false negatives, or incorrectly detects non-RAFs as RAFs when multiple interactive elements that have the same functionality are not grouped together, resulting in ten false positives. An example of this occurs in the subject *bowiestate*, where the approach groups the "Save" and "Save and Continue" buttons as the same functionality. The approach does not detect the "Save" functionality that disappears in the reflow version because the "Save and Continue" functionality is considered equivalent and still present. This occurs because our approach is unable to differentiate "Save" functionality from the "Save and Continue" functionality due to their similar JavaScript function signatures. This is a limitation of our approach; it cannot analyze the syntax of JavaScript code, which is a well-known challenge in this problem space [42, 45, 75]. We also investigated inaccuracies in the functionality grouping and discovered that they were primarily due to the lack of information that could be extracted from the web page. We believe that adding more features, such as those visually related to images/icons [85], could improve the expressiveness of the functionalities' interactive semantics.

The second scenario that caused detection inaccuracies is due to an incomplete UIIM. This occurred because of implementation limitations. For example, the Mutation Observer API [2] occasionally did not recognize some changes to the DOM during keyboard interactive crawling. As a result, the underlying UI functionalities nested within some menus were not dynamically explored and were incorrectly considered to be absent. In the subject *airbnb*, 23 false positives were detected because Selenium was unable to navigate through scrolling carousel components due to a common "*Stale*" exception that is known to affect React client-side frameworks [34].

We also reviewed the results of the included state-of-the-art tools. ReDeCheck's precision was negatively impacted by the fact that the tool only identifies relative layout failures concerning visible elements. The tool is unable to detect failures with elements that completely disappear from the UI, which is a frequent root cause of RAFs. ReDeCheck's recall result is due to the tool detecting layout issues that did not impact keyboard-based accessibility. KAFE's detection accuracy is the result of the tool being unable to detect reflow keyboard accessibility issues, as it focuses on the relationship between the mouse and keyboard. The RAF detection results of both WAVE and Qualweb reflect that the tools have limited testing coverage on WCAG rules relating to keyboard or reflow accessibility. Note that although the four included state-of-the-art tools are effective at detecting the issues for which they were designed, they are not effective at detecting RAFs. This, along with other factors, exemplifies the need for a new approach (e.g., SALAD) that is able to accurately detect RAFs.

4.4 RQ2. Run-time Performance

4.4.1 Protocol. To answer RQ2, we measured the running time of all evaluated tools on each subject web page. SALAD's, ReDeCheck's, and KAFE's running time included the time required to input a subject and run the respective tool. WAVE's and Qualweb's running time consisted of the time elapsed from the initial interaction with the respective tools' browser extension to when the results were displayed.

Table 2: Approaches' Runtime Performance (mm:ss)

		SALAD	ReDeCheck	KAFE	WAVE	Qualweb
Modeling	Avg.	60:04	n/a	28.23	n/a	n/a
	Mdn.	21:43	n/a	17:05	n/a	n/a
Detection	Avg.	02:51	02:22	00:01	00:01	00:02
	Mdn.	02:39	00:50	00:01	00:01	00:02

4.4.2 Results. The results for all of the tools' running times are displayed in Table 2, with each cell representing the average and median runtime per subject. In SALAD's and KAFE's case, we split the runtime into two parts. The "Modeling" row represents the time spent constructing the UIIM and exacting the contained *interactive semantics* of all interactive elements on the web page. The "Detection" row refers to the time spent on analyzing keyboard accessibility, loading the language model, grouping the *interactive semantics* into functionalities, and matching the available functionalities across the two versions of the UI. The other tools do not build models but, instead, statically analyze the web page. Thus, we mark their time spent on modeling as "n/a." SALAD has the slowest runtime with an overall average of 62:55 minutes and a median of 24:32 minutes per subject. The detailed runtime of each subject for each tool is included in the Appendix.

The run-time of SALAD was slower than the other approaches due to the overhead of initializing the necessary environment for its analyses. Approximately 85% of the total time was spent on the modeling phase, and only 15% was on the detection phase. We analyzed the run-time breakdown of each step in SALAD and found that the model construction itself spent an average of 67% (26 mins) of the time initializing the subject proxy and WebDrivers; 3% (1 min) of the time extracting and building nodes from UI elements; 26% (10 mins) of the time crawling the UI to build navigation edges; and 5% (2 mins) of the time extracting the semantics of UI functionalities. For the detection phase, 92% (2.5 mins) of the time was spent loading and initializing the language model.

We think that SALAD's run-time is reasonable in the context of software engineering's continuous integration testing process. Since SALAD is fully automated, it can be optimized by running unattended on multiple machines or deploying Selenium's processing across multiple cloud computing instances. The longer runtime of SALAD is a tradeoff for a significant improvement in its detection accuracy and its ability to find a much larger number of issues.

4.5 RQ3. Impact of Detected RAFs

4.5.1 Protocol. To address RQ3, we examined the impact of the detected RAFs from two perspectives: (1) how essential an identified missing functionality is to the overall usage of a subject web page (referred to as *essentiality*) and (2) how the RAFs manifested in a subject web page (referred to as *manifestation*). To determine the *essentiality*, two of the authors examined the functionality associated with each RAF and determined how essential it was to the scope of the subject. We determined the scope of a subject by following Step 1.a of the WCAG-EM [39]. RAFs were classified into one of three categories: *supportive*, *important*, or *crucial*. The *supportive* category included functionalities that directly provide access to additional information to the user, such as a link to privacy policy or the ability to share content on a social media platform. The *important* category included functionalities that provide an essential function or information to the user but are not necessary to meet the designated purpose of the web page. An example of an *important* RAF would be a "Get Help" button or a link to the homepage of a website. Lastly, the *crucial* category included functionalities that may prevent the primary purpose of the web page from being fully achieved if absent. For example, an RAF consisting of the functionality that allowed a user to log in would be categorized as *crucial*. If any discrepancies in the *essentiality* categorization procedure arose between the two authors, then a third author was consulted until a consensus could be reached. We discuss the potential threat this protocol may impose on our results in Section 4.6.

To determine *manifestation*, every RAF detected by SALAD was manually examined and categorized into one of two categories: *completely-missing* or *inaccessible*. The *completely-missing* category included functionalities from the full-sized UI that were found to be absent in the reflow UI. The *inaccessible* category included functionalities from the full-sized UI that also exist in the reflow UI, but were not accessible via the keyboard.

4.5.2 Results. The results of the *essentiality* categorization are the following: 29 *crucial*, 189 *important*, and 308 *supportive* RAFs. The functionalities of the *crucial* RAFs varied based on the purpose of the containing web page, but included functionalities, such as search and login, that could have a significant impact on the user experience. For example, a *crucial* RAF in the subject *discordapp* removes the ability to log in. Specifically, the login feature is unavailable to a keyboard-based user in the reflow version due to the associating elements completely disappearing from the UI. This prevents users from accessing the web page's primary goal of sending messages, which can only be done once logged into their account. Another example of a *crucial* RAF is in the subject *wiktionary* where users navigating the page via the keyboard cannot search for a dictionary of a specific language of their choosing. The loss of the language search functionality makes finding a dictionary of a language other than English much more time-consuming and tedious.

We also found that RAFs in the *important* category many times prevented people who use keyboard-based assistive technologies from accessing useful information. For instance, a keyboard-based user navigating the reflow version of the subject web page *raise* is unable to access the FAQ page, which includes answers to commonly asked questions by new users and other important information such as the ability to contact support related to the website's usage. Another example involves the *shutterstock* subject web page, where direct access to information concerning different paid plans disappears in the reflow version.

The majority of the RAFs detected by SALAD fell into the *supportive* category. Despite being supportive in nature, the absence of *supportive* functionalities can hinder disabled users from accessing the same level of information and services as their able-bodied counterparts. For example, in the subject *venmo*, access to the data and privacy policy is entirely missing from the reflow version. Similarly, in the subject *usgsgov* web page's reflow version, access to valuable educational information about earthquakes is entirely missing.

For the manifestation categorization, we found that 403 (77%) of the detected RAFs by SALAD manifest as *completely-missing* and 123 (13%) manifest as *inaccessible*. Upon further investigation, we found that *completely-missing* RAFs typically occur when the browser's rendering engine switches to a CSS style sheet specified for the reflow viewport, where it then hides the associated interactive element(s) via the `display:none` property. This, in addition to the prevalence of *completely-missing* RAFs, might suggest that, whether intentional or not, additional screen space is favoured over the inclusion of more features, which can frequently lead to violations of WCAG success criterion 1.4.10 [38]. For example, in the subject *cloudflare*, the functionality to change languages completely disappears from the UI in favor of a clear header bar. Regarding the *inaccessible* category, a little over 95% of the *inaccessible* RAFs still exist on the web page but are simply hidden under an inaccessible drop-down menu. For example, all of the RAFs detected in the subject *gizmodo* are inaccessible due to a non-keyboard-navigatable hamburger button, which appears in the reflow version as a substitute for the full-sized header bar. Furthermore, in almost all cases of *inaccessible* RAFs, the inaccessible elements that cannot be navigated also lack keyboard event handlers to make them actionable, which may imply that developers often overlook keyboard accessibility completely during implementation.

4.6 Threats to Validity

A potential threat to the external validity of this study is that we only tested for reflow accessibility issues rendered at the viewport width of 320 CSS pixels. This can be a threat because responsive web design allows different reflow layout versions to be applied at different viewport widths, and RAFs may exist at different width intervals [88]. However, evaluating at 320px is in accordance with the WCAG evaluation protocol for SC 1.4.10 violations [38]. One should note that the 320px width is equivalent to using a screen magnifier on a desktop browser (i.e., a standard 1280-pixel wide viewport) with a maximum zoom of 400% [30].

Another threat to external validity is that our tool is implemented using Selenium's FirefoxDriver. Potential keyboard navigation discrepancies between web browsers could result in different models with varying sets of UI paths, potentially impacting the generalizability of RAF detection. Despite the popularity of Chrome/Chromium-based browsers, we opted to evaluate using the Firefox browser because we wanted to be consistent with the related tools (i.e., KAFE and ReDeCheck) we compared against, which were implemented using Firefox.

A threat to internal validity is the upper bound that our implementation uses to handle situations such as infinite scrolling. This

could be a threat because our approach might not model the unexplored content, which may lead to a false negative. In practice, we think this is unlikely to happen since infinite scrolling is generally repeating templated content, which is checked until it reaches the upper bound of reappearing during the exploration. Although we did not find RAFs that appeared in content beyond the upper bound, if this were to occur, it would result in a lower recall for our approach.

A potential threat to construct validity is whether our concept of RAFs matches the intended idea of these accessibility issues in the real-world setting. Our defining and identification of RAFs is reasonable because it is based on WCAG Success Criterion (SC) 1.4.10. The success criterion is designed to be objective and reliable, allowing practitioners to determine what constitutes as an RAF. It also uses examples to demonstrate the failures in different use-case scenarios. As further validation, we attempted to contact the developers of our subject web pages to report the discovered accessibility issues for feedback. We contacted 37 webmasters of the 43 subject web pages containing RAFs, with the remaining web pages having no way to contact their respective webmasters. A total of 6 (16%) webmasters responded to our message. All of the 6 responses received acknowledged our report and forwarded it to their relevant development team. A senior accessibility specialist from one of our subject websites mentioned in their response that they were able to replicate the RAF and that they had immediately worked on addressing the issue. Another commented, "Even with all the testing tools available and ongoing testing, we do miss things, so your findings are appreciated." This can serve as an indirect confirmation that the issues we are targeting are real-world problems.

An additional threat to construct validity is our *essentiality* categorization for detected RAFs. By nature, the *essentiality* definition is subjective. To mitigate this threat, we employed a consensus-oriented protocol for the categorization. While even this protocol could lead to an incorrect categorization, we note that even the least impactful RAFs cause a loss of functionality for people using ATs to interact with a web page's reflow version and are still a violation of the corresponding WCAG guidelines.

5 RELATED WORK

Current automated accessibility tools are limited and cannot be used to detect RAFs. Web accessibility evaluation tools [1] that scan web pages for conformance only analyze the rendering of the pages' UI. VizAssert [76, 77] uses formal verification methods and SMT queries to automatically assert violations of accessible layout properties. AxeRay [52] is an approach that infers semantic groupings of elements across various regions of a web page to test if elements violate their WAI-ARIA roles' semantic structure. However, they do not account for RAFs that manifest in RWD where the size of the browser's viewport changes. Researchers have also built tools to detect or repair different types of UI accessibility issues in mobile apps [46–49, 64–66, 72, 73, 82–84, 93]. These techniques are limited to the mobile app domain and do not focus on RAFs.

Several techniques exist for detecting presentation failures in RWD. ReDeCheck [88] uses a layout graph to model the relative alignment of a web page's HTML elements with respect to one another as the viewport width of the page changes. It is able to detect layout inconsistencies of a web page across different versions of the UIs using implicit oracles generated by their model [87]. Viser [50] and Verve [51] are built on this and use image comparison techniques to help developers confirm and classify the failure reports created by ReDeCheck. In addition to analyzing the UI of RWD from a snapshot of a single UI state of the web page's DOM, VFDetector [81] processes the web page using mouse *click* interactions to explore responsive layout failures that are located in different UI states (e.g., those triggered by CSS and/or JavaScript). While these techniques may be able to detect RAF in situations where an element becomes visually cut off, as shown in the evaluation, they are not designed to handle RAFs that are dynamic in nature. Another technique X-PERT [80, 86] detects cross-browser issues (XBIs) by identifying presentation failures that occur when a web page is rendered with one particular browser, but is free of failures on another browser. It renders a web page over different browsers but does not model the same page over a range of viewport sizes to detect issues related to RWD.

There has been extensive work in trying to understand the semantics of UIs [55]. Test reuse or test transfer techniques is a line of research that specializes in identifying similar functionalities across different UIs [53, 54, 60, 63, 71, 78, 85]. Another work shares a similar goal to our functionality procedure described in Section 3.2, and aims to identify and map similar functionalities between web and Android UIs [62] and across web UIs [79]. Similarly, work proposed by Yandrapally et al. defines a framework that identifies functional "near-duplicates" as two web page states that contain functionalities that are identical or highly similar to each other [92]. Given the limitations of our approach highlighted in Section 4.3.2, we believe that incorporating one of the previously mentioned works into SALAD may improve our work and lead to overall better results.

Current state-of-the-art tools that are able to map similar functionalities between different UIs [62, 79, 92] cannot be utilized to detect RAFs. Specifically, they are unable to detect accessibility issues that require interacting with the app via different input methods. KAFE [57], on the other hand, analyzes web pages' keyboard interactivity to detect keyboard accessibility failures. However, KAFE does not identify nor track functional similarities in different versions of UIs nor RWD as a whole.

6 CONCLUSION

Responsive web design (RWD) is a web design approach that adapts web page layouts to fit different platforms and screen sizes. This can create a better user experience for many; however, poorly designed or implemented responsive web pages can lead to Responsive Accessibility Failures (RAFs) – a situation where keyboard users are unable to access core functionalities in a web page. This paper introduced an automated approach to detect Responsive Accessibility Failure (RAF). The evaluation showed that our approach can detect RAFs in real-world web pages with high accuracy. Furthermore, positive responses from web developers indicate that our approach targets real-world problems and can produce useful detections.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant 2009045.

REFERENCES

[1] 2019. Web Accessibility Evaluation Tools List. https://www.w3.org/WAI/ER/tools/. Updated: 2016-03.

[2] 2020. MDN Web Docs: MutationObserverInit.childList. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserverInit/childList. Accessed: 2020-08-16.

[3] 2020. Usability.gov: User Interface Elements. https://www.usability.gov/how-to-and-tools/methods/user-interface-elements.html. Updated: 2020-11-09.

[4] 2020. W3C: WAI-ARIA Authoring Practices 1.1 - Keyboard Interaction. https://www.w3.org/TR/wai-aria-practices-1.1/#keyboard-interaction. Accessed: 2020-08-28.

[5] 2022. W3C WAI: Introduction to Web Accessibility. https://www.w3.org/WAI/fundamentals/accessibility-intro/. Updated: 2022-03-31.

[6] 2022. Cucumber: Tools & techniques that elevate teams to greatness. https://cucumber.io/. Updated: 2022-09-13.

[7] 2022. Deeplearning4j: Deeplearning4j Suite Overview. https://deeplearning4j.konduit.ai/. Accessed: 2022-12-01.

[8] 2022. Google Search Central: Documentation – Responsive Web Design. https://developers.google.com/search/mobile-sites/mobile-seo/responsive-design. Accessed: 2022-10-05.

[9] 2022. LevelAccess: The 2022 State of Digital Accessibility Report. https://www.levelaccess.com/sodar2022/. Accessed: 2022-10-05.

[10] 2022. Mdn web docs: Responsive design. https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design. Accessed: 2022-10-05.

[11] 2022. Nucleus Research: The Internet is unavailable. https://nucleusresearch.com/research/single/the-internet-is-unavailable/. Accessed: 2022-10-05.

[12] 2022. Responsive Web Design (RWD) Best Practices Guide. http://web.simmons.edu/~menzin/CS321/Unit_7_Mobile/Chapter_13_ResponsiveWebDesign/RWDBestPracticesIBM.pdf. Accessed: 2022-10-11.

[13] 2022. State of Connecticut Department of Developmental Services: What is Assistive Technology? https://portal.ct.gov/DDS/General/AssistiveTechnology/What-is-Assistive-Technology. Accessed: 2022-10-07.

[14] 2022. The Selenium Browser Automation Project: WebDriver - Keyboard. https://www.selenium.dev/documentation/en/webdriver/keyboard/. Accessed: 2022-12-01.

[15] 2022. The Selenium Browser Automation Project: WebDriver - Mouse. https://www.selenium.dev/documentation/webdriver/actions_api/mouse/. Accessed: 2022-12-01.

[16] 2022. W3 Schools: HTML Form Attributes. https://www.w3schools.com/html/html_forms_attributes.asp. Accessed: 2022-10-05.

[17] 2022. W3 Schools: HTML Form Elements. https://www.w3schools.com/html/html_form_elements.asp. Accessed: 2022-10-05.

[18] 2022. W3 Schools: HTML Forms. https://www.w3schools.com/html/html_forms.asp. Accessed: 2022-10-05.

[19] 2022. W3 Schools: HTML Input Attributes. https://www.w3schools.com/html/html_form_attributes.asp. Accessed: 2022-10-05.

[20] 2022. W3 Schools: HTML Input form* Attributes. . Accessed: 2022-10-05.

[21] 2022. W3 Schools: HTML Input Types. https://www.w3schools.com/html/html_form_input_types.asp. Accessed: 2022-10-05.

[22] 2022. WAI Web Content Accessibility Curriculum: Guideline – Design for device-independence. https://www.w3.org/WAI/wcag-curric/gid10-0.htm. Accessed: 2022-10-05.

[23] 2022. WCAG 2.1: Understanding Success Criterion 1.4.10: Reflow. https://www.w3.org/WAI/WCAG21/Understanding/reflow.html. Accessed: 2022-10-05.

[24] 2022. Wikipedia: Hamburger button. https://en.wikipedia.org/wiki/Hamburger_button. Accessed: 2022-10-11.

[25] 2023. COVID-19 Pushes Commerce Online, Making ADA Website Compliance More Important Than Ever. https://www.law.com/legaltechnews/2020/05/19/covid-19-pushes-commerce-online-making-ada-website-compliance-more-important-than-ever/. Accessed: 2023-05-19.

[26] 2023. 3Play Media: Key Takeaways from UsableNet's 2023 Mid-Year Digital Accessibility Lawsuit Report. https://www.3playmedia.com/blog/key-takeaways-usablenets-ada-web-app-report/. Accessed: 2023-07-19.

[27] 2023. 8 RWD problems (and how to avoid them). https://www.creativebloq.com/rwd/responsive-design-problems-12142790. Updated: 2014-07.

[28] 2023. Axe® Accessibility Testing Tool. https://www.deque.com/axe/.

[29] 2023. Bureau of Internet Accessibility: Why Reflow Is Essential for Web Accessibility. https://www.boia.org/blog/why-reflow-is-essential-for-web-accessibility. Accessed: 2023-07-31.

[30] 2023. Knowbility: Exploring WCAG 2.1 — 1.4.10 Reflow. https://knowbility.org/blog/2018/WCAG21-1410Reflow. Accessed: 2023-07-31.

[31] 2023. Mdn web Docs: Keyboard - Clickable elements must be focusable and should have interactive semantics. https://developer.mozilla.org/en-US/docs/Web/Accessibility/Understanding_WCAG/Keyboard#clickable_elements_must_be_focusable_and_should_have_interactive_semantics. Accessed: 2023-08-01.

[32] 2023. Mdn web Docs: Keyboard - Interactive elements must be able to be activated using a keyboard. https://developer.mozilla.org/en-US/docs/Web/Accessibility/Understanding_WCAG/Keyboard#interactive_elements_must_be_able_to_be_activated_using_a_keyboard. Accessed: 2023-08-01.

[33] 2023. Moz's list of the most popular 500 websites on the internet. https://moz.com/top500.

[34] 2023. Reflect: How to deal with StaleElementReferenceException in Selenium. https://reflect.run/articles/how-to-deal-with-staleelementreferenceexception-in-selenium/.

[35] 2023. The WebAIM Million: An annual accessibility analysis of the top 1,000,000 home pages. https://webaim.org/projects/million/. Updated: 2023-04-01.

[36] 2023. WAVE Web Accessibility Evaluation Tool. https://wave.webaim.org/.

[37] 2023. WCAG SC 1.4.4 Resize Text 1.4.10 Reflow . https://yatil.net/blog/resize-text-reflow. Accessed: 2023-07-31.

[38] 2023. WCAG Technique F102: Failure of Success Criterion 1.4.10 due to content disappearing and not being available when content has reflowed. https://www.w3.org/WAI/WCAG21/Techniques/failures/F102.

[39] 2023. Website Accessibility Conformance Evaluation Methodology. https://www.w3.org/TR/WCAG-EM/.

[40] 2023. Zenodo: Dataset for paper "Automatically Detecting Reflow Accessibility Issues in Responsive Web Pages". https://zenodo.org/records/10140605. Accessed: 2023-11-17.

[41] 2024. SALAD's Project Web Site. https://sites.google.com/usc.edu/salad/home.

[42] Christoffer Quist Adamsen, Anders Møller, Saba Alimadadi, and Frank Tip. 2018. Practical AJAX Race Detection for JavaScript Web Applications. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 38–48. https://doi.org/10.1145/3236024.3236038

[43] Abdulmajeed Alameer, Paul T. Chiou, and William G. J. Halfond. 2019. Efficiently Repairing Internationalization Presentation Failures by Solving Layout Constraints. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 172–182. https://doi.org/10.1109/ICST.2019.00026

[44] Abdulmajeed Alameer, Sonal Mahajan, and William G. J. Halfond. 2016. Detecting and Localizing Internationalization Presentation Failures in Web Applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 202–212. https://doi.org/10.1109/ICST.2016.36

[45] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2014. Understanding JavaScript Event-Based Interactions. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 367–377. https://doi.org/10.1145/2568225.2568268

[46] Ali S. Alotaibi, Paul T. Chiou, and William G.J. Halfond. 2021. Automated Repair of Size-Based Inaccessibility Issues in Mobile Applications. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 730–742. https://doi.org/10.1109/ASE51524.2021.9678625

[47] Ali S. Alotaibi, Paul T. Chiou, and William G.J. Halfond. 2022. Automated Detection of TalkBack Interactive Accessibility Failures in Android Applications. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 232–243. https://doi.org/10.1109/ICST53961.2022.00033

[48] Ali S. Alotaibi, Paul T. Chiou, Fazle M. Tawsif, and William G. J. Halfond. 2023. ScaleFix: An Automated Repair of UI Scaling Accessibility Issues in Android Applications. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 147–159. https://doi.org/10.1109/ICSME58846.2023.00025

[49] Abdulaziz Alshayban and Sam Malek. 2022. AccessiText: Automated Detection of Text Accessibility Issues in Android Apps. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3540250.3549118

[50] Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. 2019. Automatic Visual Verification of Layout Failures in Responsively Designed Web Pages. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 183–193. https://doi.org/10.1109/ICST.2019.00027

[51] Ibrahim Althomali, Gregory M. Kapfhammer, and Phil McMinn. 2021. Automated visual classification of DOM-based presentation failure reports for responsive web pages. *Software Testing, Verification and Reliability* 31, 4 (2021), e1756. https://doi.org/10.1002/stvr.1756 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1756 e1756 stvr.1756.

[52] Mohammad Bajammal and Ali Mesbah. 2021. Semantic Web Accessibility Testing via Hierarchical Visual Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1610–1621. https://doi.org/10.1109/ICSE43902.2021.00143

[53] Farnaz Behrang and Alessandro Orso. 2018. Test Migration for Efficient Large-Scale Assessment of Mobile App Coding Assignments.

[54] Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 54–65. https://doi.org/10.1109/ASE.2019.00016

[55] Chunyang Chen, Sidong Feng, Zhengyang Liu, Zhenchang Xing, and Shengdong Zhao. 2020. From Lost to Found: Discover Missing UI Design Semantics through Recovering Missing Tags. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW2, Article 123 (oct 2020), 22 pages. https://doi.org/10.1145/3415194

[56] Paul T. Chiou, Ali S. Alotaibi, and William G.J. Halfond. 2023. BAGEL: An Approach to Automatically Detect Navigation-Based Web Accessibility Barriers

for Keyboard Users. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. https://doi.org/10.1145/3544548.3580749

[57] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. 2021. Detecting and Localizing Keyboard Accessibility Failures in Web Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. https://doi.org/10.1145/3468264.3468581

[58] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. 2023. Detecting Dialog-Related Keyboard Navigation Failures in Web Applications. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1368–1380. https://doi.org/10.1109/ICSE48619.2023.00120

[59] Nádia Fernandes, Daniel Costa, Sergio Neves, Carlos Duarte, and Luís Carriço. 2012. Evaluating the Accessibility of Rich Internet Applications. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*. Association for Computing Machinery, New York, NY, USA, Article 13, 4 pages. https://doi.org/10.1145/2207016.2207019

[60] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/3236024.3236055

[61] Jonathan Lazar. 2019. Verifying that web pages have accessible layout. In *The potential role of U.S. consumer protection laws in improving digital accessibility for people with disabilities*. U. Pa. JL Soc. Change 22 (2019), 185.

[62] J. Lin and S. Malek. 2022. GUI Test Transfer from Web to Android. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society. https://doi.org/10.1109/ICST53961.2022.00011

[63] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 42–53. https://doi.org/10.1109/ASE.2019.00015

[64] Zhe Liu. 2022. Woodpecker: Identifying and Fixing Android UI Display Issues. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 334–336. https://doi.org/10.1145/3510454.3522681

[65] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2021. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3324884.3416547

[66] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang. 2023. Nighthawk: Fully Automated Localizing UI Display Issues via Visual Understanding. *IEEE Transactions on Software Engineering* 49, 01 (jan 2023), 403–418. https://doi.org/10.1109/TSE.2022.3150876

[67] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. Automated Repair of Layout Cross Browser Issues Using Search-Based Techniques. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. https://doi.org/10.1145/3092703.3092726

[68] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2017. XFix: An Automated Tool for the Repair of Layout Cross Browser Issues. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3092703.3098223

[69] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. 2018. Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 215–226. https://doi.org/10.1109/ICST.2018.00030

[70] Sonal Mahajan, Krupa Benhur Gadde, Anjaneyulu Pasala, and William G. J. Halfond. 2016. Detecting and Localizing Visual Inconsistencies in Web Applications. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 361–364. https://doi.org/10.1109/APSEC.2016.060

[71] Leonardo Mariani, Mauro Pezzè, Valerio Terragni, and Daniele Zuddas. 2021. An Evolutionary Approach to Adapt Tests Across Mobile Apps. 70–79. https://doi.org/10.1109/AST52587.2021.00016

[72] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2023. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/3551349.3560424

[73] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-Driven Accessibility Repair Revisited: On the Effectiveness of Generating Labels for Icons in Android Apps. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3468264.3468604

[74] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 3111–3119.

[75] Shabnam Mirshokraie. 2014. Effective Test Generation and Adequacy Assessment for JavaScript-Based Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/2610384.2631832

[76] Pavel Panchekha, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2019. Modular verification of web page layout. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 151:1–151:26. https://doi.org/10.1145/3360577

[77] Pavel Panchekha, Adam T. Geller, Michael D. Ernst, Zachary Tatlock, and Shoaib Kamil. 2018. Verifying that web pages have accessible layout. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. Association for Computing Machinery, Philadelphia, PA, USA, 1–14. https://doi.org/10.1145/3192366.3192407

[78] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI Test Cases from IOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 284–295. https://doi.org/10.1145/3293882.3330575

[79] Andreas Rau, Jenny Hotzkow, and Andreas Zeller. 2018. Transferring Tests Across Web Applications. In *Web Engineering*, Tommi Mikkonen, Ralf Klamma, and Juan Hernández (Eds.). Springer International Publishing, Cham, 50–64.

[80] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-Browser Issues in Web Applications. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 702–711.

[81] Yeonhee Ryou and Sukyoung Ryu. 2018. Automatic Detection of Visibility Faults by Layout Changes in HTML5 Web Pages. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 182–192. https://doi.org/10.1109/ICST.2018.00027

[82] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-Case and Assistive-Service Driven Automated Accessibility Testing Framework for Android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3411764.3445455

[83] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2023. Groundhog: An Automated Accessibility Crawler for Mobile Apps. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3556905

[84] Yuhui Su, Chunyang Chen, Junjie Wang, Zhe Liu, Dandan Wang, Shoubin Li, and Qing Wang. 2023. The Metamorphosis: Automatic Detection of Scaling Issues for Mobile Apps *(ASE '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3556935

[85] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. 2021. UI Test Migration Across Mobile Platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 756–767. https://doi.org/10.1109/ASE51524.2021.9678643

[86] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2014. X-PERT: A Web Application Testing Tool for Cross-Browser Inconsistency Detection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 417–420. https://doi.org/10.1145/2610384.2628057

[87] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. Automated Layout Failure Detection for Responsive Web Pages without an Explicit Oracle. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3092703.3092712

[88] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2017. ReDeCheck: An Automatic Layout Failure Checking Tool for Responsively Designed Web Pages. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3092703.3098221

[89] Thomas A. Walsh, Gregory M. Kapfhammer, and Phil McMinn. 2020. Automatically identifying potential regressions in the layout of responsive web pages. *Software Testing, Verification and Reliability* 30, 6 (2020), e1748. https://doi.org/10.1002/stvr.1748

[90] Thomas A. Walsh, Phil McMinn, and Gregory M. Kapfhammer. 2015. Automatic Detection of Potential Layout Faults Following Changes to Responsive Web Pages (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 709–714. https://doi.org/10.1109/ASE.2015.31

[91] Willian Massami Watanabe, Renata P. M. Fortes, and Ana Luiza Dias. 2012. Using acceptance tests to validate accessibility requirements in RIA. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility (W4A '12)*. Association for Computing Machinery, Lyon, France, 1–10. https://doi.org/10.1145/2207016.2207022

[92] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-Duplicate Detection in Web App Model Inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 186–197. https://doi.org/10.1145/3377811.3380416

[93] Yuxin Zhang, Sen Chen, Lingling Fan, Chunyang Chen, and Xiaohong Li. 2023. Automated and Context-Aware Repair of Color-Related Accessibility Issues for Android Apps *(ESEC/FSE 2023)*. https://doi.org/10.1145/3611643.3616329