Automated Web Form Accessibility Labeling using Visual Cues

Anonymous Author(s)

ABSTRACT

Filling forms is a simple and common activity while browsing the web. However, for users with disabilities, it is not possible to access web forms if the forms do not have accessibility labeling markups. Large scale surveys show that such missing form accessibility labeling is one of the three most common web accessibility issues. However, there is currently little to no options in terms of automatically creating form labeling in order to make forms accessible. In this paper, we propose an approach that automatically infers labels by gathering and analyzing visual cues from a form, and then augmenting the DOM to incorporate the required accessibility labeling markup. We evaluate our approach on 15 real-world subjects and assess the accuracy of labeling inference, the safety of the DOM augmentation, as well as the labeling performance. The results show an average F1-measure of 88.1% for label inference, and an average run-time of around 1240 milliseconds.

KEYWORDS

web accessibility, web forms, accessibility repair, visual analysis

ACM Reference Format:

Anonymous Author(s). 2021. Automated Web Form Accessibility Labeling using Visual Cues. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Web accessibility impacts millions of people around the world. About 70 and 30 million people have software-related disabilities in the United States and the European Union alone [1, 2], respectively. Accessibility, in terms of software, refers to a set of software attributes that, if included in a piece of software, allows users with disabilities to access that software. Without such attributes, users with disabilities are not able to use or interact with the software, thereby presenting a major daily hurdle in our increasingly interconnected world.

Accessibility is gradually becoming a pressing legal issue. For instance, in the period from 2015 to 2018 the number of software accessibility lawsuits in US courts increased from 57 to 2258 [3, 4]). Similar accessibility laws exist or are being ratified in the European Union [5] requiring various entities (e.g., government agencies, businesses) to make their software accessible. When it comes to existing software development practice, however, the general adoption of accessibility in software projects has been rather low. Some developers and companies, perhaps under the pressure of other pressing business requirements, have not prioritized the issue of accessibility or included it in their requirements [6, 7].

In terms of research attention, the great majority of existing software accessibility related works come from a human factors research perspective [8–10], rather than a software engineering or tooling perspective. In such works, the goal is to document, categorize, and synthesize how exactly do users with disabilities use software. That is, issues like how do they understand the contents of an app, or what challenges do they come across while browsing. The objective is therefore to come up with high level accessibility guidelines that developers can keep in mind while designing or building their software.

However, the question of how those high level accessibility guidelines can be concretized and implemented in an automated fashion is a different topic that has received little attention. The standard existing practice for testing, analyzing, or repairing software accessibility has therefore remained a manual process [11–14], using manual cognitive walkthroughs or recruiting external auditors to manually evaluate the accessibility.

In this work, we propose a contribution that automates the repair of a certain aspect of web accessibility, which is *web form labeling*. In a large scale analysis of the top websites on the internet [15], it has been shown that this aspect of missing form labeling is one of the three most common web accessibility issues. Web form labeling accessibility requires programmatically declaring the labels of all form fields, in order to allow non-sighted users to access that information. Accordingly, if a non-sighted user reaches a form that has no accessibility labeling, they currently have little to no available options in terms of automatically repairing the form labeling in order to be able to access that form. This, of course, can be a significant obstacle because web forms are ubiquitous in all online services.

To that end, this paper proposes an approach that automates the inference and repair of web form accessibility labeling. The approach is based on visually analyzing the content and structure of a web form, and then augmenting the inferred labels into the Document Object Model (DOM). This analysis first converts the form into a set of uniform abstract elements in order to streamline subsequent analysis. Next, a set of visual cues are constructed from these elements, with the purpose of capturing essential visual information. Subsequently, the labeling process is formulated as an optimization problem, where various possible combinations of visual cues are optimized in order to reach the final labeling associations. These associations are finally converted into a standard markup in order to make the form accessible. In this work, we focus on visual disabilities as opposed to other forms of disability (e.g., hearing). The rationale for this is twofold. First, the web is predominantly a visual medium where most of the information is accessed visually as opposed to other senses. Second, surveys have shown that visual disabilities are the most relevant to web users with disabilities [16].

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper makes the following main contributions:

- A novel approach for automatically inferring web form accessibility labeling, which is the first to address this issue, to the best of our knowledge.
- A technique that is based on a combination of visual analysis and optimization formulation to repair the DOM of a web form in order to make it accessible.
- A qualitative and quantitative evaluation in terms of accuracy, safety, and performance. The results show an average F-measure of 88.1% for labeling accuracy, and an average run-time of around 1240 milliseconds.
- An implementation of our approach, available in a tool called AxeForm.

2 BACKGROUND AND MOTIVATING EXAMPLE

Figure 1 shows an example of an inaccessible web form. In a quick glance at the rendered form in Figure 1b, a sighted user can immediately understand the structure and content of the web form and navigate their way through it. For instance, the user would immediately recognize that they can write down their name, their address, and indicate whether or not they would like to receive daily coupons. Furthermore, the user readily and intuitively recognizes *which* fields are associated with the name or the address, for instance. The collection of information we just described is referred to as form labeling. That is, form labeling indicates what fields are present on the form and communicates a textual description for what each one of them represents.

While this understanding of form content and labeling happens naturally and instantaneously for sighted users, that is not the case for visually impaired (i.e., non-sighted) users. In *inaccessible* forms, such as the one in Figure 1, the labeling is communicated exclusively through *visual* design. This is because the HTML markup in Figure 1a is simply a collection of `<div>`s, `<p>`s, and `<input>`s that do not communicate any labeling associations to indicate what the various elements represent.

For instance, in Figure 1a, there is no piece of markup indicating what the `<textarea>` represents or means. It is not possible to find this information just by looking at the markup alone. That is, there is no markup telling a non-sighted user what information are they supposed to provide for that form field. The form is therefore completely unusable in this case. In contrast, sighted users can immediately understand that the `<textarea>` is where they should provide their address information, or that the first text input (with the placeholder label ‘Your name‘) is where they should provide their name. They understood this from the visual design and layout of the form. This implicit visual communication is intuitive and natural for sighted developers and users, but is unavailable for users who can not have *access* to visual information due to disabilities.

Accordingly, the web form is deemed *inaccessible*, because its markup is expressed in a fashion that does not provide any information about the labels of the form elements. The task of ensuring form accessibility would therefore rely on one key principle: any information that is visually perceivable by sighted users must also be available for non-sighted users. When a web form communicates its labeling exclusively through visual design, it does not follow this principle of accessibility.

The analysis and conclusion that we just made is currently being done manually, since it requires a high level analysis of the rendered form and its visual structure. Our goal in this work is to automate the reasoning we have just described in order to be able to automatically reach conclusions about the accessibility of form labeling, and then generate new markup to repair the Document Object Model (DOM) in order to make the form accessible.

2.1 Accessible Rich Internet Applications

The aforementioned lack of form labeling markup in the code makes it difficult for non-sighted users to use web forms. This is because such users rely on *screen readers* to parse a web page for them since they can not directly access the page themselves. Screen readers are tools that speak out the various information, forms, and labels present on the page, and the non-sighted user would then select one of the options they heard from the screen reader. Screen readers can be thought of as web browsers for non-sighted users, with one major caveat. While a standard web browser simply renders the page as is and leaves it to the end user (i.e., a sighted user) to visually understand what the various elements mean or represent, screen readers require that the page markup contains semantic information or labeling of its content, which the reader would then announce audibly, giving non-sighted users an understanding of the overall semantic structure of the page.

The standard that is used by screen readers during their processing of web pages is the World Wide Web Consortium’s (W3C) *Accessible Rich Internet Applications* (ARIA) [17] standard. The ARIA standard specifies a set of markup attributes that should be included in the page’s HTML to make it accessible to screen readers or other assistive technologies.

Accordingly, when web developers incorporate these markups into their pages, the result is an accessible app that non-sighted users can use through their screen readers. Our objective in this work is to automate this process. That is, given a web form, we analyze it and infer the labels then incorporate them into the markup.

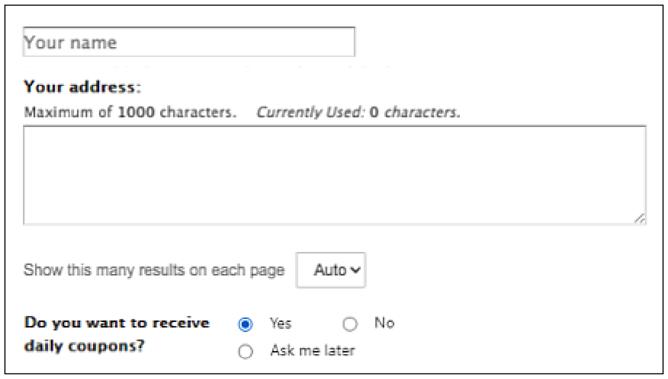
2.2 Syntax checks

Existing accessibility testing tools [18] are based on syntactic checking. They check the HTML against certain syntax rules. For instance, one common check is to assert that every `img` element has an `alt` text attribute (for text descriptions). Another check asserts that `input` elements must not be descendent of `a`. Another example is checking that every `ul` list element has a non-empty list item `li` child. In general, all syntactic checks follow the same template: if certain syntax A is present, assert that syntax B is true.

While such syntax checks can be useful simple assertions and are easy to automate, syntax checks are not capable of addressing the more important, and more challenging, high level perception and analysis that is required to infer and repair form labeling. To illustrate this, we revisit Figure 1(b), where we can visually see that there is a text box form field whose label is ‘Your address‘. We perceived this labeling association by visually looking at (b) but did not find it expressed in the markup in (a), and therefore concluded

```html
<form>
  ...
  <div>
    ...
    <p><textarea>...</textarea></p>
    ...
    <select>...</select>
  </div>
  ...
  <p>Your address:</p>
  ...
  <section>... <span>Yes</span> ...</section>
  ...
  <div><span>Show this many results on each page</span>
    <div><b>Do you want to receive daily coupons?</b></div>
  </div>
  ...
  <input type='radio' name='chk1' value="T">
  ...
</form>
```

(a) Sample of the HTML markup



Your name

Your address:

Maximum of 1000 characters. *Currently Used:* 0 *characters.*

Show this many results on each page Auto

Do you want to receive daily coupons? [✓ Yes] No Ask me later

(b) Rendered form

Figure 1: An example of an inaccessible web form.

that the form is inaccessible. We now pause to reflect and observe that there is no possible syntactic check that can automate the conclusion we just reached. That is, the steps that we just walked through requires our own visual perception as sighted users in order to reach the conclusion that there is a mismatch between the perceived visual content of the from, and the form's markup. This process can not be put in the form of a syntax assertion. In the next section, we describe how we construct an approach that automates the accessibility analysis procedure that we have just manually conducted.

3 APPROACH

In this paper, the objective is to automatically convert inaccessible web forms into accessible ones. We recall that the scope of this work focuses on form labeling because it has been shown [15] that missing form labeling is one of the top three most common web accessibility issues. Figure 2 shows an overview of our proposed approach. We adopt a strategy of visually analyzing the web form to infer labeling associations of form elements, and then repairing the DOM such that it's markup reflects the inferred labels in order to make the form accessible. The approach begins by loading the web page in an instrumented web browser (e.g. via Selenium). Next, any forms on the page are then abstracted in preparation for further analysis. Subsequently, a set of visual cues are constructed from each form. These visual cues are then used in the next stage, which is the optimization formulation. In this stage, we cast the labeling process as an optimization problem, where the minimization and maximization of certain combinations of visual cues yield the final form labels. Finally, the inferred labels are transformed into standard ARIA accessibility attributes and augmented into the DOM of the web page.

3.1 Form Abstraction

In this first stage, the goal is to abstract the form and extract essential elements, which would then be used in the remaining stages of the approach. More specifically, the abstraction process converts a form into a set of two classes of objects: *fields* and *texts*. Any other aspect of the form is discarded. The reason we do this abstraction is to have a uniform representation of the problem when we later conduct the inference. That is, the analysis will work with only two classes of objects instead of many possible permutations. We now describe the details of extracting fields and texts.

The process begins by traversing all form DOM elements in the rendered page. For each form element, all descendant nodes are traversed and classified into one of three categories: fields, texts, or discarded. The criteria for classification into one of these categories are described below.

Fields. Web forms contain multiple types of inputs that are expressed using many different markups and displayed in multiple ways. For instance, in Figure 1, we have `textarea`, `select`, and `input` elements with various combinations of attributes determining the type of input. In order to make the formulation of the labeling optimization tractable, we detect the many possible ways in which form inputs can be expressed and abstract them into a single category of fields. When it is time to perform the optimization and generate the labels (as described in Section 3.2), working with fields as an abstract category facilitates a more manageable optimization formulation. More concretely, we have:

$$f := \{E_i \mid \eta(E_i)\ \forall\ E_i \in \text{DOM}\} \tag{1}$$

where f is the set of abstracted fields, E_i is any descendant node of each form element, and $\eta(E)$ is a predicate which collects all possible kinds of form inputs (e.g., text areas, drop downs). This set of fields f will be used for all remaining steps.

Texts. In a similar fashion to the aforementioned abstraction applied to fields, we also perform text abstraction. Labels in web forms can be expressed using many possible textual elements. For example, in Figure 1, `p`, `div`, and `b` have been used, among many other possibilities. We again need a uniform representation for all these possibilities in order to make formulating the optimization problem tractable and manageable. The abstraction is achieved as follows:

$$t := \{E_i \mid \tau(E_i)\ \forall\ E_i \in \text{DOM}\} \tag{2}$$

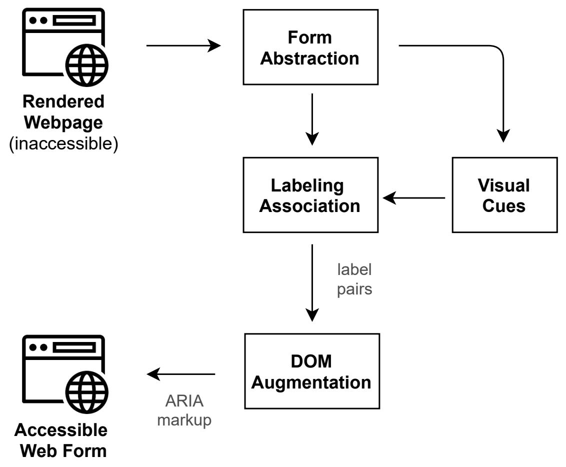


Figure 2: Overview of the major stages of the proposed approach.

where t is the set of abstracted texts, E_i is any descendant node of form elements, and $\tau(E)$ is a predicate that examines whether there is a text associated with E. More specifically, it returns non-empty nodes of DOM type `#TEXT`, which represent string literals. We note that the predicate is based on a node type, rather than an element (i.e., tag) type. This allows more robust abstraction because the predicate captures any text and does not make assumptions about how developers choose to place their text. In other words, regardless of the tag used for text data (e.g., `<span>, <div>`), text would still be stored in nodes of type `#TEXT`, even for custom HTML elements. This helps in making the approach more robust by reducing assumptions about tags and how they are used in the page.

3.2 Labeling Association

After the form abstraction process is completed, we proceed by formulating the labeling association as an optimization problem. The first step in formulating the problem is gathering visual cues from the abstracted form. Next, these visual cues are used to construct the decision variables that will make up the objective functions as well as the constraints of optimization. We describe the details of these steps in the following subsections.

3.2.1 Visual Cues. When sighted users browse a web form, they are able to instantaneously perceive all form labels and therefore understand what each form field signifies. They do this intuitively simply by visually glancing at the form, something that non-sighted users are unable to do. The purpose of this stage is therefore to automatically capture as much visual information as necessary that could help with inferring labels. In other words, we aim to emulate the sort of intuitive visual parsing that sighted users do.

To this end, we construct two types of visual cues to capture this sort of intuitive visual parsing. First, there are individual cues that are pertinent to each individual element. The second type of cues are pair-wise, which capture information between all possible element pairs. Individual cues capture *visual prominence*, while pair-wise cues capture *visual layout* aspects. We now describe what these cues represent and how they are constructed. We first emphasize, however, that no single cue is sufficient or conclusive on its own. Rather, it is the collective optimization of all possible cue combinations that yields the final labeling decision.

Visual Prominence. This cue aims to capture how visually prominent an object is on a rendered web page. Visual prominence indicates how quickly or likely would an element be noticed by a sighted user. For instance, objects that are larger or bolder are more likely to be perceived easier and quicker by sighted users, compared to objects that are very small and barely perceptible. The intuition behind constructing this visual cue is that form labels are more likely to have higher rather than lower visual prominence.

Visual prominence is constructed by the multiplication of three measurements: the font size, weight, and color contrast, which is a ratio of foreground to background color. First, each of these values are captured for every abstracted form element (as described in Section 3.1). We then take the maximums of each of these values and use that to compute a normalization factor. Finally, all visual prominence values are divided by this normalization factor to yield the final normalized visual prominence values. Only these normalized values are used in the remainder of the optimization formulation. More concretely, the visual prominence is computed as follows:

$$\phi_i = \frac{s_i w_i c_i}{n_p} \tag{3}$$

where ϕ_i is the visual prominence of the i-th element, s_i and w_i are the font size and weight, c_i is the color contrast, and n_p is the normalization factor computed from all elements as described in the previous paragraph.

Visual Layout. This cue aims to capture the visual arrangement of elements on the form. That is, we would like to get some information on how elements are placed with respect to one another to get a sense for the layout of the form. The intuition for adopting this cue is that, when a sighted user glances at a form, they would more likely make labeling associations for elements that are arranged next to each other, as opposed to elements that are on the opposite ends of the form, for instance. Accordingly, by capturing cues related to visual layout, we would have some information that can facilitate emulating the kind of visual parsing done by sighted users. This cue is of course not necessarily conclusive on its own, and that is the reason we combine multiple cues in order to reach final labeling decisions.

We capture this cue by measuring the pair-wise visual geometric Euclidean distance. This is done across all field-text pairs, but not performed on field-field and text-text pairs. The reason for not including these two later cases is twofold. First, we would like to keep the optimization problem as manageable and tractable as possible, and therefore need to minimize the number of pair-wise operations in order to reduce combinatorial explosion as much as possible. As such, by focusing on field-text pairs, we capture the minimum essential information on layout. The second reason is that the optimization formulation used for the labeling process (as discussed in Section 3.2) has constraints that eliminate the need for text-text or field-field information. Accordingly, these measurements provide the necessary raw information regarding the relative layout arrangements.

After each pair-wise measurement is collected, we proceed to determine the geometric bounds of all measurements. The reason for doing this is to normalize the geometric measurements, since this cue will be combined with other cues in the objective as well as the constraints of the optimization. Accordingly, for the optimization to function properly, the geometric measurements must first be normalized. To do this normalization we compute the geometric bound, which is the maximum possible pair-wise Euclidean distance for a collection of elements, and is therefore calculated once all distances are available. Finally, all distances are divided by the geometric bound and the resulting values are the final normalized values used in the remainder of the analysis. More concretely, the visual layout set can be expressed as follows:

$$L = \{\gamma_{i,j} \in \mathcal{R} \mid \gamma_{i,j} = \frac{E(f_i, t_j)}{n_g} \; \forall \; f_i, t_j\} \tag{4}$$

where $\gamma_{i,j}$ is the pair-wise visual layout cue, $E = \sqrt{\Delta x^2 + \Delta y^2}$ is the visual geometric Euclidean distance, where x and y are the visual locations on screen. n_g is the normalization factor computed from the geometric bound as explained in the previous paragraph.

3.2.2 Decision Variables. After all visual cues have been gathered, we proceed by constructing the decision variables that will be used to perform labeling optimization. The nature of the decision variables will of course depend on how the problem is formulated. The strategy we adopt is to formulate the optimization problem as a constrained binary program. The rationale for formulating the problem this way is twofold. First, the assignment of labels is essentially constrained by other label assignments. That is, if a solver has made labeling assignments involving some subset of texts, then this constrains the possible texts it can use for further labeling. Furthermore, the labeling process can be naturally thought of as a pair-wise binary function between all fields and all texts. For these reasons, a constrained binary program appears to be a suitable approach to formulate the labeling problem.

Accordingly, each decision variable is a binary value that represents the decision of assigning a particular label to a particular field. Each variable would therefore encode one possibility of labeling. The optimization solution represents which decision variables are applicable. That is, the decision variables is where the tradeoff occurs in terms of selecting various possible combinations of visual cues to reach final labeling. More concretely, we have:

$$d_{i,j} = \mathrm{T} \quad \Rightarrow \quad t_j = \lambda(f_i) \tag{5}$$

where $d_{i,j}$ is the decision variable and $\lambda(f)$ is an association function that assigns the label to each field f. Accordingly, once all decision variables have been determined, we are able to find the label associated with each field.

3.2.3 Objective and Constraints. Now that we have specified the decision variables, the next step is to create the objective function and the constraints. In order to do this, we first need to consider the kind of labeling markup provided by the ARIA standard. This is because the final DOM repair process will include generating the necessary markup to convey the inferred labels, and this markup has to follow the ARIA standard.

The ARIA standard provides two types of form labeling: individual and group labels. Individual labeling represents the label associated with every single form input on its own. For instance, in Figure 1b, consider the text box under 'Your address'. Individual labeling creates an association between the text box field and the element representing the field's textual description (i.e., 'Your address'). ARIA defines markup for group labeling as well. For instance, consider the 'Yes', 'No', and 'Ask me later' options at the bottom of Figure 1b. There are three different individual labels in this case, associating each of the three textual descriptions with the corresponding radio button for each of them. While these are indeed correct labels providing the correct textual description for each radio button, ARIA provides a group labeling in this case. This is because while the individual labels on their own are necessary, they are not sufficient in this case. The group labeling for the three radio buttons is marked by the element 'Do you want to receive daily coupons.'

The strategy we adopt to identify both label types is a bottom-up multiple-passes approach. We first infer the individual labels, and then use that information to perform another pass to identify the group labels. The reason for adopting this strategy is that it allows us to optimize the problem for each label type separately, and also simplifies group labeling by using information from the individual labeling. Therefore, each of these label types will have a different objective function and set of constraints.

We start by creating the coefficients to be used in the objectives. Each decision variable is associated with a coefficient, and the values of these coefficients are what imparts context and meaning to the binary decision variables. The magnitude of each coefficient describes the pair-wise and per-element visual cues. For individual labels, we have:

$$\alpha_{i,j} = \frac{\gamma_{i,j}}{\phi_i} \tag{6}$$

where $\alpha_{i,j}$ is the coefficient of the decision variable, ϕ_i is the visual prominence cue, and $\gamma_{i,j}$ is the visual layout cue, as described in Section 3.2.1. We then proceed to formulate the objective function and the constraints:

$$\begin{aligned} A \;=\; \operatorname{argmin} & \sum_i \sum_j \alpha_{i,j} d_{i,j} \\ \text{s.t.} \quad & \sum_j d_{f_1,j} = 1, \cdots \sum_j d_{f_N,j} = 1 \\ & \sum_i d_{i,t_1} \le 1, \cdots \sum_i d_{i,t_M} \le 1 \end{aligned} \tag{7}$$

with A being the labeling associations result. α and d are the coefficients and decision variables, respectively, as described in the preceding paragraphs. We make a number of remarks regarding the above equations. First, we can see that the visual cues (as captured in α) cover all possible labeling combinations. The decision variables then determine which combination of visual cues yields the optimal result. We also note that we have two sets of constraints. The first set of constraints subjects the optimization to the requirement of finding exactly one label. Each form field f gets its own constraint equation, containing entries for all possible labels. The rationale for this constraint is twofold. First, this prevents the optimization from yielding the trivial solution (i.e. zero decision variable selections). Second, this avoids solutions where only a few fields are associated with all labels by virtue of yielding a smaller objective value, thereby leaving many other fields without a labeling association. The second set of constraints subjects the optimization to restrictions on the possible associations that can be done for text elements. Each text t gets its own constraint equation, paired with all possible fields. Paired with the first set of constraints, these second constraints enforce a couple requirements. First, that the labeling association for each text entry is optional. That is, it encodes the expectation that not every text field needs to be a label. This is true because forms have many texts that are not labels (e.g. descriptions, hints). Furthermore, the constraints ensure that each potential label is associated with at most one form field. This enforces the requirement that a solution is not accepted if only a few text elements end up being associated with many fields due to a smaller objective value.

Once the individual labeling associations are obtained, we proceed to infer the group labeling. First, we discard all texts that have received a labeling association from the individual labeling stage (i.e. all $t \in A$). Next, our target is to identify groupings of the fields. To achieve this, we take an agglomerative clustering approach [19]. The reason for adopting this approach is that we use agglomerative clustering in a way that enables us to identify groupings without requiring thresholds or parameters. This helps in increasing the robustness of analysis, and reduces the often cumbersome manual effort required to tune any parameters.

We now describe the grouping process. First, we construct a fields distance matrix. The matrix contains the visual geometric Euclidean distance between each pair of fields, which is calculated as follows: $M_{i,j} = \sqrt{\Delta x^2 + \Delta y^2}$, where M is the distance matrix, i and j represent the pair of fields, and the deltas refer to the minimum distance between the pair. Subsequently, we determine the linkage function to be used for the agglomerative clustering. This function determines which pair of clusters to merge in the next level of agglomeration. For the current task of identifying visual groupings, we adopt the single-linkage criterion [19]. Single-linkage agglomeration occurs, in our case, when the minimum visual geometric distance between clusters is smallest.

Accordingly, we run the agglomeration process as per the aforementioned steps. Subsequently, we obtain all heights of the obtained hierarchy of clusters. In our case, these heights would signify the progressively increasing visual geometric distances between clusters. We then compute the median value of all heights (excluding heights of zero). This median value is then used as a cutoff to the hierarchy of clusters. That is, only clusters whose within-cluster visual geometric distance are below the cutoff are retained. The final result is a set of field groupings. Once the groupings have been identified, we proceed to infer the labeling associations for those groups. We achieve this by performing a second optimization pass and formulate the problem as follows:

$$\begin{aligned} B \;=\; &\operatorname{argmax} \sum_i \sum_j \frac{d_{i,j}}{\alpha_{i,j}} \\ \text{s.t.} \quad &\sum_j d_{f_1,j} \le 1, \cdots \sum_j d_{f_N,j} \le 1 \\ &\sum_i d_{i,t_1} \le 1, \cdots \sum_i d_{i,t_M} \le 1 \end{aligned} \tag{8}$$

where B is the group labeling associations result, and α and d are the same variables as those that have been used in the first optimization pass, with the exclusion of of all individual labeling associations that have been obtained from that first pass. We make a number of remarks regarding the above equations. First, we note that the structure of the problem is different from that in the first optimization pass. We now have a maximization problem instead of a minimization. The rationale for this is as follows. First, we note that group labels may or may not exist in a given form, which is different from individual labels that are always present. The problem formulation therefore needs to take this into account. This brings us to the second observation, which is regarding the constraints. In the previous pass, we had constraints to have the decision variables finding exactly one label (i.e. the equality constraints). In this pass, however, no such requirement is made. Instead, we have an upper bound inequality constraint on the decision variables pertaining to all form fields (the first set of constraints in Equation (8)). This encodes the fact that group labeling associations may or may not be present on a given form, and if it does have one, that its doesn't result in a degenerate case where many texts are associated with a given field. However, since we now have an upper bound inequality instead of an equality constraint, the problem can no longer remain a minimization. This is required in order to avoid trivial solutions (i.e. zero decision variable selections). Accordingly, the problem is formulated as a maximization. The placement of the coefficients of decision variables have been adjusted accordingly to reflect the change in the objective. However, their computation remains the same as before, using the same visual cues we previously described. We also note that the second set of constraints (pertaining to the text elements) have remained the same as before, since we still have the same expectation as the first case that not all texts need to partake in a labeling association.

3.3 DOM Augmentation Markup

So far, the form has been abstracted, visual cues generated, and the optimizations yielding the labeling associations have been solved. At this stage, we perform the final step needed to ensure form accessibility, which is repairing the DOM. This stage involves taking all the labeling associations obtained so far, and transforming them into ARIA accessibility markups. These markups are then added to the DOM of the page in order to encode all the labeling associations that have been inferred. We now describe the details of this markup augmentation process.

The ARIA standard provides markup for both individual and group labels. We will start by describing the encoding for individual labels, and then proceed to describe to the case of groups. First, we take all the individual labeling associations obtained in A (Equation (7)). This provides a mapping of f_i to t_i, representing pairs of fields and labels, respectively. Next, we iterate over each field f_i, and examine the associated label t_i. If the t_i element in the page has an id (i.e. a non-zero `id` attribute), then we obtain that id. Otherwise, a unique random id is generated for that element and added as an `id` attribute for that element in the DOM. Subsequently, for each field f_i, we add the `aria-labelledby` standard ARIA attribute. The value of this attribute is then set to the value of the `id` attribute of the associated t_i, whether that id already exists or was generated. If the associated t_i is not an explicit element (e.g., the textual label is contained in a placeholder attribute, such as the “Your name” placeholder label in Figure 1), then we add an `aria-label` instead of `aria-labelledby`, and we set its value to be the content of the text, since assigning an id in this case is not possible. The ARIA standard allows labeling elements using both ids and the actual textual value of the label.

We then move on to process the group labeling associations obtained from B (Equation (8)). In this case, the mapping result is different from the case of individual labeling associations. Here the mapping is from a set of fields $\{f_1, \ldots, f_n\}$ to the associated inferred label t_i. We begin by iterating over each set of fields, and examine the associated label. We obtain the id of the associated label, or generate a unique one if it does not exist, in the same fashion as done in the previous case for individual labels. Next, we find the *nearest common ancestor* in the DOM for the $\{f_1, \ldots, f_n\}$ set. The reason for doing this is the nature of the standard ARIA attribute used to group related elements, which is `role="group"`. It should contain as descendants the group of elements its representing [20]. Accordingly, for the nearest common ancestor node of a set $\{f_1, \ldots, f_n\}$, we add the attribute `role="group"` to that node to communicate the presence of a labeling group. Next, we add the `aria-labelledby` attribute to the same nearest common ancestor node, and set the value of the attribute to the id attribute (whether existing or generated) of the mapped label t_i associated with the set $\{f_1, \ldots, f_n\}$.

3.4 Implementation

We implemented the proposed approach using server-side JavaScript (Node). We used the Selenium WebDriver to instrument browsers and extract DOM information and computed attributes. We used Lpsolve [21] to formulate and solve the optimization problems and OpenCV [22] for image operations. To make the study replicable, we made available online a link to our data and tool [23], which we called AxeForm (short for accessible forms).

4 EVALUATION

We conducted qualitative and quantitative studies to answer the following research questions:

RQ1 How accurate is the labeling associations inference?
RQ2 Are the labeling accessibility repairs conducted in a safe manner?
RQ3 How does the performance scale with the size of real-world web forms? Is it suitable for real-time usage?

In the following subsections, we discuss the details of the experiments that we designed to answer each research question, together with the results and discussions.

4.1 RQ1: Inference Accuracy

In this question, the objective is to assess how accurate is the inference of the labeling associations in a given web form. The rationale for evaluating this aspect is that the approach first performs the labeling association inference, and then uses this inference to generate the form repair markup. Accordingly, we first need to assess the inference process itself.

Table 1: List of the 15 subjects used for evaluation.

Subject	Form description	# elements in form	total # elements
wikipedia.org	Page links search	45	518
opinionlab.com	Experience feedback	116	136
zoom.us	Live demo request	437	924
netflix.com	New titles suggestion	17	432
microsoft.com	Careers support ticket	86	444
dropbox.com	Business account inquiry	398	721
stackoverflow.com	Help center	106	743
etsy.com	Product design careers	68	340
zendesk.com	Customer service inquiry	428	1323
bing.com	Search customization	1552	1642
github.com	Account support request	229	283
intuit.com	Career events sign-up	326	1293
salesforce.com	Website quality feedback	102	764
indeed.com	Account registration	54	222
paypal.com	Search for jobs	144	410

We evaluated this research question as follows. First, we collected 15 random subjects that were sourced from the Alexa top websites list [24]. The way we selected the evaluation subjects are as follows. To start, we get a random subject url from the pool. We then load that url in a browser in order to inspect the website. This inspection process is conducted manually, and its goal is to find a page on the website that contains a web form. For each website, we manually inspect its various sections and pages until a web form is found. If no web forms are found within five minutes of manual examination, the subject is skipped and a new random url is obtained from the pool. The only additional criterion we have on web forms is that they are reachable without requiring registration or payment. This criterion was made in order to simplify the process of collecting subjects. The final list of subjects is shown in Table 1, and the full urls are available online [23]. The final list of subjects covers a variety of topics from a variety of websites, with a varying number of elements in forms, ranging from 17 up to 1552. Subsequently, we feed the web form to our approach and obtain the output labeling associations. We then examine each generated association and classify the results into true positives, false positives, and false negatives. We now describe each one of these outcomes.

False positives. These are cases in which the inferred labeling association for a given field *does not* match the visually perceived labeling. For instance, in Figure 1b, if the inferred labeling associates the highlighted blue radio button to the label ‘Ask me later‘, then this is a false positive labeling, because it does not match the correct visually perceived labeling, which is the ‘Yes‘ option.

True positives. These are cases in which the inferred labeling association for a given field correctly matches the visually perceived labeling. For instance, in Figure 1b, if the inferred labeling associates the large text box input to the label ‘Your address‘, then this is a true positive labeling.

False negatives. These are cases in which no labeling associations were inferred for a field that *should* have been associated with a label. For instance, in Figure 1b, if no labels were associated to the

Table 2: Evaluation of the inference accuracy of labeling associations.

	Proposed approach			**Baseline (randomized)**		
Subject	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**
wikipedia.org	3	0	0	0	5	1
opinionlab.com	6	2	0	0	11	1
zoom.us	12	0	2	0	13	4
netflix.com	3	0	2	1	4	3
microsoft.com	5	0	0	0	15	0
dropbox.com	9	0	1	0	7	4
stackoverflow.com	4	1	0	0	6	1
etsy.com	3	0	2	0	2	3
zendesk.com	6	0	0	0	9	1
bing.com	11	7	1	0	5	4
github.com	5	0	0	0	9	2
intuit.com	6	0	0	1	7	2
salesforce.com	3	1	1	0	8	1
indeed.com	5	2	1	0	6	2
paypal.com	4	0	0	1	3	1
	Prec.	Rec.	F1	Prec.	Rec.	F1
	86.7%	89.4%	88.1%	2.7%	9.1%	4.1%

highlighted blue radio button, then this is a false negative case, because there should have been a label (i.e., the ‘Yes‘ option).

Finally, in order to have a more thorough and informative evaluation, we included a baseline in our experiments. However, since there are no existing tools that perform the labeling associations and repairs, our baseline consists of a random selection process. In this process, random elements from a given web form are selected. Next, a labeling association is generated to another randomly selected element. This set of randomly selected elements and their associations is then taken to be the baseline.

4.1.1 Results and Discussion. Table 2 shows the results of evaluating the accuracy of inferring the labeling associations. The columns show the true positive (TP), false positive (FP), and false negative (FN) results for each subject. The last row of the table shows the net precision, recall, and F-1 measure across all subjects. The two groups of columns, labeled “Proposed approach” and “Baseline”, show the accuracy of inferring labeling associations using either the proposed approach or the baseline approach, respectively. The key outcome of this evaluation is the F-1 measure, which is at 88% for the proposed approach. This indicates a rather effective inference process. Precision and recall were comparable, at 86% and 89%, respectively. Figure 3 shows a sample of the inferred labeling associations corresponding to the motivating example. Each entry represents the outcome of a decision variable, as given by Equation (5).

In order to further understand the limitations of the approach, we investigated the false positive and false negative cases. We identify a few trends. First, false positives occurred in forms that had visual prominence cues that did not follow the proposed optimization model (Section 3.2.3). In these cases, the forms had a high ratio of texts relative to fields and all the texts had similar visual prominence cues. This resulted in the optimization process yielding suboptimal labeling associations. False positives also occurred in cases where, in addition to the aforementioned visual cues, the visual layout was dense, which made it difficult to compensate for the layout density using visual prominence information.

As for the false negatives, the most predominant case was due to non-standard form elements. That is, these cases did not use a `type=radio`, for instance, to indicate radio buttons. Instead, a non-standard element (e.g., `div`) was used, and then only styled to appear like a radio input. The proposed approach is incapable of handling these cases, where the input elements themselves are not marked up as input elements. Another related cause of false negatives is where the `form` elements were replaced by non-standard markup (e.g., `div`), and therefore the approach was incapable of delineating the form region from the rest of the page. For future work, we plan to explore how these cases can be handled. We predict that this might require formulating another set of visual cues to detect the missed cases.

4.2 RQ2: Markup Safety

In the previous research question, we examined how accurate the labeling association inference is. Once that aspect has been evaluated, we need to examine whether the insertion or augmentation of the inferred associations into the DOM is safe. The rationale for evaluating this aspect is that we want to make sure that any markup repairs applied to a page does not cause any unintended or unaccounted for breakages or failures to the page.

We evaluated this research question as follows. First, we continued using the same test subjects used in the first research question. Each subject is then loaded in a browser and had their labeling associations inferred and the DOM repaired. To assess the safety, we check two different aspects. First, a visual check is made comparing images of the page before and after the repair. The rationale is to assess whether the generated markup would cause unintended breakage to the rendering of the page. Second, a functionality check is made. Here, we randomly manipulate the form (e.g., selecting various radio options, expanding drop down menus) to assess whether the generated markup would cause unintended breakage to the functionality of the form. Finally, we note down the outcome (i.e., pass or fail), and note down the number of failures.

4.2.1 Results and Discussion. Table 3 shows the results of evaluating the safety of the markup repair. For each subject, the columns show the outcome of the safety check and the number of failures. Subjects for which no failures were observed have ‘-‘ under the number of failures column. For most subjects, we did not observe safety failures (as described in Section 4.2). For two of the subjects, as shown in Table 3, we did observe a couple of failures. In both these cases, the DOM repair process have failed. The reason for these failures is locator breakages. This occurs because after the labeling association for a field is determined, locators are used locate the field in the DOM, then insert the markup repairs at that location. But in the case of the observed failures, the locators became stale, where locators that were previously valid become unusable. This occurred because the subject’s DOM was highly dynamic and therefore the DOM has changed from the time the locator was acquired.

In this work we used DOM locators in the form xpath strings. The issue of stale locators and possible approaches to address them have been also reported in other web testing and analysis works [25, 26] and is still an open research problem. For future work, we plan to explore different possibilities to capture the locators and address these issues. One option would be devising a more robust timing strategy at which to capture the locators in order to minimize the probability of going stale. Another option would be using visual locators instead of DOM locators.

4.3 RQ3: Runtime Scalability

After evaluating the accuracy and safety aspects in the previous research question, this question examines the runtime performance (i.e., total time of execution). The rationale for evaluating this question is as follows. First, since a couple of aspects in the optimization formulation are combinatorial in nature, we wanted to assess whether or not this is going to cause any performance penalty, and more importantly how does the performance scale with the size of forms. Second, since the goal of this work is to make web forms accessible, it would be useful to know if the runtime performance is good enough if this approach were to be used for real-time repair by end users during their browsing activities.

We evaluated this research question as follows. For each subject, we record the number of DOM elements in the forms as a measure of form size. This will be used to measure how does the runtime scale with form size. Then, the total runtime is measured from the moment the subject is loaded until the final DOM markup is augmented.

4.3.1 Results and Discussion. Figure 4 shows the results of the runtime performance evaluation. The x-axis is semi-log and shows the number of DOM elements within the form. The y-axis is linear and shows the runtime in milliseconds.

The average runtime is 1238 ± 534 milliseconds. Minimum and maximum runtimes are 702 and 2691 milliseconds, respectively. The data range of DOM elements covers three orders of magnitude

```
{
  // associations.json
  // each entry is f_i -> t_i (Equation 5)

  ".../form/div[1]/p/textarea":
      ".../form/p",

  ".../form/input[@name='chk1']":
      ".../form/section/span",

  ".../form/div[1]/select":
      ".../form/div[2]/span",

  // remaining elements
  // ...
}
```

Figure 3: Sample of the generated association decisions output (corresponds to Figure 1).

Table 3: Evaluation of accessibility markup augmentation safety.

Subject	Outcome	# failures
wikipedia.org	pass	-
opinionlab.com	pass	-
zoom.us	pass	-
netflix.com	pass	-
microsoft.com	pass	-
dropbox.com	pass	-
stackoverflow.com	pass	-
etsy.com	pass	-
zendesk.com	pass	-
bing.com	pass	-
github.com	pass	-
intuit.com	fail	2
salesforce.com	pass	-
indeed.com	pass	-
paypal.com	fail	3

of form sizes, making it suitable for assessing scalability. The runtime changes linearly with the size of the DOM, indicating good scalability with respect to form complexity. We recall that we are implementing the optimization in the Lpsolve [21] solver. In addition to being simple to setup for the type of optimization we are conducting, part of the reason for this choice is that this solver has low-to-medium performance [27], and therefore any runtime evaluation would give a more conservative estimate. Accordingly, while the observed average performance of around 1.2 seconds is arguably suitable for run-time repair, the performance will likely improve further when more capable solvers are used. The key outcome, however, remains the same, which is that the performance scales linearly with respect to form complexity.

4.3.2 Threats to validity. In order to avoid any selection bias, we chose test subjects (i.e., web sites) randomly with the mentioned criteria in Section 4.1. The subjects are diverse and complex enough to be representative of real-world scenarios as they have been collected from a pool of high traffic sites on the Internet [24], thereby mitigating the external validity of the study by making the results generalizable. To make the study replicable, we made available online a link to our tool implementation and evaluation data [23].

5 RELATED WORK

There has been little work in the software engineering community in terms of automating or creating tools for the various aspects of accessibility guidelines, especially the high level semantic guidelines that rely on human perception and assessment [12]. The great majority of existing software accessibility related works come from a human factors research perspective, rather than from a software engineering or tooling perspective. For instance, one line of work [8–10, 28–31] is based on analyzing existing websites (or categories of websites, such as educational or banking) by manually observing how non-sighted users would interact with them. The goal is to discover any issues that might surface during usage, and then come up with improved accessibility guidelines. The hope is that these

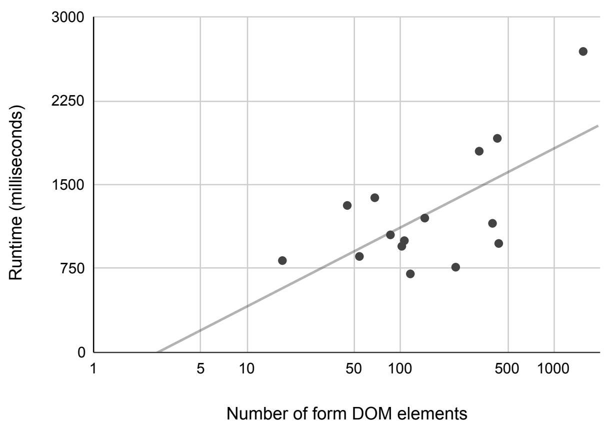


Figure 4: Evaluation of the runtime performance scalability for various sizes of input web forms. (Runtime in milliseconds. Form size in number of DOM elements in a form.)

guidelines will be later kept in mind while developing software. A related area of research aims to discover how the software development practices themselves can be improved such that it is more likely that the end product software is more accessible. Krainz et al. [32] explores how model-based development contributes to accessibility compliance. Sanchez et al. [33] and Bai et al. [34, 35] focus on agile software engineering practices and how do they impact the accessibility of the product.

A few tools do exist that provide basic *syntax* checks for accessibility. These checks consist of basic rules that are readily automatable, such as: any `ul` list element must contain non-empty `li` child elements, or no `input` elements must not be descendent of `a` elements. The goal of such checks is to provide quick and simple assertions that are easily automatable. Patil et al. [36] and Eler et al. [37] check for absence of predefined attributes in mobile applications, such as the absence of required alternative text for all user interface images. These are then flagged and reported to developers, allowing them to identify and fix these potential issues. Other tools [38, 39] perform similar syntax checks, differing mainly in what are the attributes that are being checked. Such syntax checkers, however, have been shown to be of limited efficacy. An empirical study of the most popular accessibility checkers found that these tools revealed only %26 of a known small set of accessibility violations present in tested web pages. One paper [40] does utilize higher level semantic analysis. However, the paper only performs testing and does not perform repairs, and is not related to web form labeling. In contrast, the proposed approach in this paper conducts repairs for web form labeling accessibility, which has been shown [15] to be one of the three most common web accessibility issues. To summarize, syntax checks, such as the checks conducted in the aforementioned works, can not be used to handle the high level perceptual analysis required to infer and repair accessible form labeling associations, as illustrated and discussed in Section 2.

Another line of research is related to generating test inputs for web testing tasks. These works have proposed techniques geared towards improving the generation, utility, or robustness of test inputs. One goal of this vast area of research is to create better sequences of test operations or data in order to achieve a certain testing objective (e.g., test coverage). Some papers [41–44] adopt a strategy of randomly generating the inputs. In these cases, the focus is not mainly on the generated input data itself but on other aspects of testing, such as traversing the event space or assertion generation, with the raw input data being randomly generated, or predefined by the user. Another line of work [45, 46] gives more attention to the raw input data. In these cases, specific types or formats of input values are used instead of randomly generated inputs. The intuition is that properly formatted and typed inputs allow better exploration and testing. The aforementioned works, however, are not related to accessibility testing or repair, nor generate the required labeling associations to be able to perform accessibility repair.

Finally, a few existing papers have explored the use of some visual analysis aspects for testing or analyzing web applications. Burg et al. [47] propose a program understanding tool that is geared towards front-end development projects and understanding how the user interface changes during interactions with the application. It allows a developer to select a certain element that they are interested in, and the tool tracks how the UI is changing with respect to that element, and tracks the corresponding code changes. Another work [48] analyzes the UI of a web page in order to generate reusable components. Its goal is to visually observe any patterns in design mockups or templates, and then create reusable web components that captures those patterns. Choudhary et al. [49] focuses on cross-browser compatibility, and proposes a technique that checks for difference between how a given web page is rendered across browsers. A given web app is loaded in two different browsers, and an image diff is performed to locate the areas where the two browsers differ, therefore helping the developer to identify issues that may not otherwise be easy to identify.

6 CONCLUSION

Filling web forms is a key activity while browsing the web. This task that can be easily completed by sighted users, is a significant hurdle for non-sighted users if the form does not contain the required accessibility labeling. This issue of missing form labeling is consistently one of the top three most common web accessibility issues. Unfortunately, however, when a non-sighted user is faced with a non-accessible form, there are currently little to no options available to access that form. To this end, this work proposed an approach that automatically analyzes web forms and makes them accessible. The approach first abstracts a given web form, then generates visual cues from the form, in an effort to emulate how sighted users would visually perceive the form. The visual cues are then used in a constrained binary optimization program to solve for the form labeling associations. These are finally translated into standard ARIA accessibility markups and augmented into the DOM to repair the form and make it accessible. We evaluated our approach on 15 real-world subjects and assessed the accuracy of labeling inference, the safety of the DOM augmentation repairs, as well as the labeling performance. The results show an average F1-measure of 88.1% for label inference, and an average run-time of around 1240 milliseconds.

REFERENCES

[1] Centers for Disease Control and Prevention, National Center on Birth Defects and Developmental Disabilities, Division of Human Development and Disability., "Disability and health data system (dhds) data." https://dhds.cdc.gov, 2016.

[2] European Union's Eurostat, "European union labour force survey (eu-lfs). prevalence of disability," http://appsso.eurostat.ec.europa.eu, 2014.

[3] Seyfarth Shaw Law Firm, LLP., "Americans with disabilities (ada) act, title iii report." https://www.seyfarth.com/news-publications, 2018.

[4] United States Courts, Court Records., "Public access to court electronic records (pacer)." https://www.uscourts.gov/court-records, 2020.

[5] Publications Office of the European Union, "Directive (eu) 2016/2102 of the european parliament on the accessibility of the websites and mobile applications of public sector bodies." https://eur-lex.europa.eu/eli/dir/2016/2102/oj, 2016.

[6] C. Vendome, D. Solano, S. Liñán, and M. Linares-Vásquez, "Can everyone use my app? an empirical study on accessibility in android apps," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2019, pp. 41–52.

[7] S. Harper and A. Q. Chen, "Web accessibility guidelines," *World Wide Web*, vol. 15, no. 1, pp. 61–88, 2012.

[8] A. Alshayban, I. Ahmed, and S. Malek, "Accessibility issues in android apps: state of affairs, sentiments, and ways forward," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE, 2020, pp. 1323–1334.

[9] S. Snider, W. L. Scott II, and S. Trewin, "Accessibility information needs in the enterprise," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 12, no. 4, pp. 1–23, 2020.

[10] S. Bhagat and P. Joshi, "Evaluation of accessibility and accessibility audit methods for e-governance portals," in *Proceedings of the 12th International Conference on Theory and Practice of Electronic Governance*, 2019, pp. 220–226.

[11] A. Bai, H. C. Mork, T. Schulz, and K. S. Fuglerud, "Evaluation of accessibility testing methods. which methods uncover what type of problems?" *Studies in health technology and informatics*, vol. 229, pp. 506–516, 2016.

[12] P. Acosta-Vargas, S. Luján-Mora, T. Acosta, and L. Salvador-Ullauri, "Toward a combined method for evaluation of web accessibility," in *International Conference on Information Theoretic Security.* Springer, 2018, pp. 602–613.

[13] G. Brajnik, "A comparative test of web accessibility evaluation methods," in *Proceedings of the 10th international ACM SIGACCESS conference on Computers and accessibility.* ACM, 2008, pp. 113–120.

[14] S. Abou-Zahra, "Web accessibility evaluation," in *Web accessibility.* Springer, 2008, pp. 79–106.

[15] Institute for Disability Research, Policy, and Practice. Utah State University., "Annual accessibility analysis," https://webaim.org/projects/million/, 2021.

[16] WebAIM Non-profit Organization, "Screen reader user survey results." 2019. [Online]. Available: https://webaim.org/projects/screenreadersurvey8/

[17] "World wide web consortium. accessible rich internet applications 1.2. 2019."

[18] United Kingdom Government's Office of Digital Services, "Accessibility tools audit." https://alphagov.github.io/accessibility-tool-audit/.

[19] G. Gan, C. Ma, and J. Wu, *Data clustering: Theory, algorithms, and applications.* Society for Industrial and Applied Mathematics, 2021.

[20] T. W. W. W. Consortium, "Using grouping roles to identify related form controls," https://www.w3.org/TR/WCAG20-TECHS/ARIA17.html, 2021.

[21] "Lpsolve mixed-integer linear programming system," https://cran.r-project.org/web/packages/lpSolve/index.html.

[22] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[23] https://cutt.ly/CWodmOq.

[24] "Alexa top sites," https://aws.amazon.com/alexa-top-sites/.

[25] M. Leotta, A. Stocco, F. Ricca, and P. Tonella, "Robula+: An algorithm for generating robust xpath locators for web testing," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 177–204, 2016.

[26] H. Kirinuki, H. Tanno, and K. Natsukawa, "Color: correct locator recommender for broken test scripts using various clues in web application," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2019, pp. 310–320.

[27] A. Luppold, D. Oehlert, and H. Falk, "Evaluating the performance of solvers for integer-linear programming," Tech. Rep., 2018.

[28] S. Yan and P. Ramachandran, "The current status of accessibility in mobile apps," *ACM Transactions on Accessible Computing (TACCESS)*, vol. 12, no. 1, pp. 1–31, 2019.

[29] A. S. Ross, X. Zhang, J. Fogarty, and J. O. Wobbrock, "Examining image-based button labeling for accessibility in android apps through large-scale analysis," in *Proceedings of the 20th International ACM SIGACCESS Conference on Computers and Accessibility*, 2018, pp. 119–130.

[30] G. Agrawal, D. Kumar, M. Singh, and D. Dani, "Evaluating accessibility and usability of airline websites," in *International Conference on Advances in Computing and Data Sciences.* Springer, 2019, pp. 392–402.

[31] T. Domínguez Vila, E. Alén González, and S. Darcy, "Website accessibility in the tourism industry: an analysis of official national tourism organization websites around the world," *Disability and rehabilitation*, vol. 40, no. 24, pp. 2895–2906, 2018.

[32] E. Krainz, K. Miesenberger, and J. Feiner, "Can we improve app accessibility with advanced development methods?" in *International Conference on Computers Helping People with Special Needs.* Springer, 2018, pp. 64–70.

[33] S. Sanchez-Gordon and S. Luján-Mora, "A method for accessibility testing of web applications in agile environments," in *Proceedings of the 7th World Congress for Software Quality (WCSQ). En proceso de publicación.(citado en la página 13, 15, 85)*, 2017.

[34] A. Bai, K. S. Fuglerud, R. A. Skjerve, and T. Halbach, "Categorization and comparison of accessibility testing methods for software development," 2018.

[35] A. Bai, V. Stray, and H. Mork, "What methods software teams prefer when testing web accessibility," *Advances in Human-Computer Interaction*, vol. 2019, 2019.

[36] N. Patil, D. Bhole, and P. Shete, "Enhanced ui automator viewer with improved android accessibility evaluation features," in *2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT).* IEEE, 2016, pp. 977–983.

[37] M. M. Eler, J. M. Rojas, Y. Ge, and G. Fraser, "Automated accessibility testing of mobile apps," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2018, pp. 116–126.

[38] ASLint, "Accessibility linter." 2020. [Online]. Available: https://www.aslint.org/

[39] Google Inc., "Accessibility scanner." 2020. [Online]. Available: https://support.google.com/accessibility/android/answer/6376570

[40] M. Bajammal and A. Mesbah, "Semantic web accessibility testing via hierarchical visual analysis," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, 2021, pp. 1610–1621.

[41] D. Adamo, D. Nurmuradov, S. Piparia, and R. Bryce, "Combinatorial-based event sequence testing of android applications," *Information and Software Technology*, vol. 99, pp. 98–117, 2018.

[42] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[43] W. Song, X. Qian, and J. Huang, "Ehbdroid: beyond gui testing for android applications," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2017, pp. 27–37.

[44] A. Sadeghi, R. Jabbarvand, and S. Malek, "Patdroid: permission-aware gui testing of android," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 220–232.

[45] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, "Mobolic: An automated approach to exercising mobile application guis using symbiosis of online testing technique and customated input generation," *Software: Practice and Experience*, vol. 48, no. 5, pp. 1107–1142, 2018.

[46] M. Dhok, M. K. Ramanathan, and N. Sinha, "Type-aware concolic testing of javascript programs," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 168–179.

[47] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining visual changes in web interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology.* ACM, 2015, pp. 259–268.

[48] M. Bajammal, D. Mazinanian, and A. Mesbah, "Generating reusable web components from mockups," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2018, pp. 601–611.

[49] S. R. Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on.* IEEE, 2012, pp. 171–180.