QualState: Finding Website States for Accessibility Evaluation

Filipe Rosa Martins
frmartins@lasige.di.fc.ul.pt
LASIGE, Faculdade de Ciências,
Universidade de Lisboa
Lisboa, Portugal

Letícia Seixas Pereira
lspereira@ciencias.ulisboa.pt
LASIGE, Faculdade de Ciências,
Universidade de Lisboa
Lisboa, Portugal

Carlos Duarte
caduarte@edu.ulisboa.pt
LASIGE, Faculdade de Ciências,
Universidade de Lisboa
Lisboa, Portugal

ABSTRACT

Single Page Applications (SPAs) are characterised by changing a webpage's content without forcing a reload. While they increase the interactivity of web pages, they can also raise accessibility concerns, such as new content being displayed without screen reader users being aware. Another concern raised impacts the ability of current web accessibility evaluation tools to be able to assess SPAs. Current tools work by loading the DOM and assessing it. For SPAs, which change the DOM in response to user interaction, it is possible that evaluation tools do not have access to a significant part of a page's content. In this paper, we introduce QualState, a tool that can browse the different states of a SPA, and provide the DOM of the different states to QualWeb, an automated web accessibility evaluation tool, so their accessibility can be evaluated. We assessed QualState in a small set of SPAs and concluded that it increases the number of elements evaluated and improves QualWeb's ability to identify accessibility barriers.

CCS CONCEPTS

• **Human-centered computing → Accessibility**; **Accessibility systems and tools**.

KEYWORDS

Web Accessibility, SPA, Single Page Application, Crawler, Automated accessibility evaluation, QualWeb, QualState

ACM Reference Format:
Filipe Rosa Martins, Letícia Seixas Pereira, and Carlos Duarte. 2024. QualState: Finding Website States for Accessibility Evaluation. In *The 21st International Web for All Conference (W4A '24), May 13–14, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3677846.3677851

1 INTRODUCTION

Internet usage has grown at an unprecedented rate and shows no signs of slowing down. BroadbandSearch reports that between the years 2000 and 2022, Internet use increased by around 1355% [5]. This was accompanied by the need to change the paradigm of how web applications and websites are developed and used to accommodate more users, be faster and offer more functionalities [24]. During this advance emerged the concept of Single Page Application (SPA), e.g. Facebook, Google Maps, and Gmail. SPAs are dynamic pages that change their content in response to users' actions, without the need to reload a new page.

SPAs, like any other website, should be accessible, irrespective of their users' capabilities. SPAs, however, present some challenges that are not common in most other web pages. For example, new content being displayed on the page needs to be announced to screen reader users who are not capable of perceiving the visual change in the page. These challenges also extend to the domain of automated web accessibility evaluation. The vast majority of accessibility evaluation tools were developed considering static pages, obtaining the HTML (Hypertext Markup Language) and other content elements, such as images and JavaScript scripts [16], and then carrying out the accessibility evaluation. However, these tools are not prepared to handle the dynamic nature of SPAs, which change their content as a consequence of user actions. This dynamism is not triggered by the evaluation tool, meaning the tool will not access the content that can be presented to a user. Therefore, it becomes very difficult, if not impossible, to carry out a reliable accessibility assessment of SPAs with existing tools.

In response to this challenge, we introduce QualState, a solution designed to address the complexities of accessibility evaluation in SPAs. QualState uses a page state crawler designed to explore multiple states within a single web page. The crawler identifies elements that can be activated and triggers them. If no page reloads results from the activation, QualState compares the new state with the states that had already been visited to decide if it has not been seen before. QualState makes available another set of instructions to support a more efficient exploration of SPAs, such as the possibility of instructing the crawler to fill out form elements and give instructions about page elements that should not be activated. With this set of features, QualState enables a comprehensive exploration of dynamic web environments.

To support the accessibility evaluation of these environments, QualState was integrated with QualWeb [10], an automated web accessibility evaluation tool. An initial evaluation of the QualState-augmented QualWeb was conducted to provide an assessment of the benefits introduced by integrating QualState with QualWeb, highlighting its advantages over the core QualWeb system alone.

This work presents the following contributions:

- QualState, a crawler that can explore states in SPAs;
- QualWeb augmented by QualState demonstrating the feasibility of increased accessibility assessment of dynamic web environments;
- An initial evaluation of the benefits of increased page state assessments.

W4A '24, May 13–14, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1030-8/24/05
https://doi.org/10.1145/3677846.3677851

The remainder of the paper starts by reviewing related work. This is followed by a description of QualState, detailing its architecture, different components and integration with QualWeb. The following section presents the results of the initial performance evaluation, highlighting the possibility of reaching more content to be assessed. The final section concludes this work and discusses future challenges.

2 RELATED WORK

Our work is related to three domains: web accessibility, particularly web accessibility evaluation; Single Page Applications, emphasising the difficulties they create for accessibility evaluation; and web crawling and page state comparison.

2.1 Web Accessibility

With the growing use of the Internet and of its presence in our daily lives accessibility is beginning to be called into question because of the way it can exclude certain groups [6]. This became evident during the COVID-19 pandemic, when the quarantine meant that access to services became almost exclusively online, highlighting the many accessibility problems raised by the vast majority of websites. For example, Lazar [15] reports on how this problem has affected numerous colleges in the United States, identifying issues related to procurement and training, but also to the accessibility of digital resources.

Abuaddous et al. [2] had already categorized accessibility challenges into three main groups: (i) standards and guidelines, (ii) design and development, and (iii) accessibility evaluation. Regarding standards and guidelines, WCAG (Web Content Accessibility Guidelines) [29] is one of the most widely used sets of guidelines, serving as a reference when one wants to make a website more accessible. To illustrate their relevance, standards are being created around the world, such as the EN 301 549 [9] in Europe, and Section 508 [12] in the USA. However, WCAG is not without issues [6]. Ambiguity remains a concern, leading to subjective interpretations and poorly implemented solutions – it is noteworthy to mention that Accessibility Conformance Testing Rules (ACT-R) [30] have emerged to provide specific and objective criteria for accessibility testing. Additionally, relying solely on WCAG is deemed insufficient to ensure comprehensive accessibility.

In the context of design and development, a shortage of experts and trained personnel also hampers the implementation and assessment of accessibility, primarily due to a lack of awareness towards accessibility issues within the industry [2].

Building on the discussion of accessibility challenges, the following section further explores the accessibility evaluation process, a critical aspect of this study.

2.2 Web Accessibility Evaluation

Accessibility evaluation is a demanding job that consumes time and resources and requires the participation of experts or users to carry out the evaluation [2]. As such, tools that carry out evaluations of websites automatically have naturally emerged.

These tools can be used online or locally, and typically check conformance with a set of accessibility standards [1], and significantly reduce the time and effort required for assessments. They remain highly beneficial for conducting initial accessibility evaluations, enabling rapid identification and filtering of a large number of accessibility issues.

Furthermore, these tools are also useful for conducting extensive evaluations of numerous websites collectively, a task particularly crucial, for instance, for monitoring accessibility status across various web platforms. They aid in understanding genuine accessibility levels and identifying areas requiring improvement interventions [14]. In this context, Santoro et al. [14] scrutinized 2.7 million web pages from Italian public administrations to assess adherence to WCAG 2.1 standards. Recent studies have also employed these tools for diverse purposes. Through an evaluation of around three million web pages, Martins and Duarte [20] investigated the correlation between web accessibility metrics, analyzing eleven metrics derived from automated evaluations across approximately three million web pages. In a further study with the same dataset, Martins and Duarte [19] characterized web accessibility and established connections between web technologies and accessibility levels, showing the feasibility of selecting technologies that enhance or, at least, do not compromise web content accessibility.

While these tools offer significant benefits, they also have limitations. A primary concern is that not all guidelines can be reliably assessed computationally [1, 2]. Additionally, there is inconsistency in how certain guidelines are interpreted by different programmers, leading to varying outcomes from different evaluation tools assessing the same page [2]. Furthermore, another limitation arises in dynamic contexts, as these tools may struggle to adequately assess pages with interactive elements or content that changes dynamically [3]. Given the focus of this work, which aims to address this gap, a more detailed discussion on this limitation will be provided later on. It is also important to highlight that, considering these limitations, a comprehensive accessibility evaluation requires manual assessment by experts, ideally involving users with disabilities [1].

Currently, various tools test the accessibility of a page, such as AChecker [7], WAVE [31], aXe [8], or QualWeb [10]. To illustrate the functioning of such tools, and considering we have integrated QualWeb and QualState, we will provide an overview of QualWeb's functioning.

To use QualWeb, the user has to provide a valid URL, choose which assessment modules to use and what output format is desired. Initially, the URL is processed in the DOM module, which is responsible for transforming the DOM obtained from the URL into objects that can be manipulated by QualWeb. This DOM is loaded in a Puppeteer [13] instance where the user-selected evaluation modules are injected. The evaluation modules are then executed, with the outcomes of each test being added to an object that contains the overall evaluation results. Finally, the final report is sent to a formatter to be converted into the desired format. Two formats are currently supported by QualWeb: its native JSON-based format and EARL, the Evaluation and Report Language [27] format.

2.3 Single Page Applications

In recent years, Single Page Applications (SPAs) have redefined web interaction by enabling dynamic content updates without full page reloads [22, 24]. SPAs operation starts with a server request,

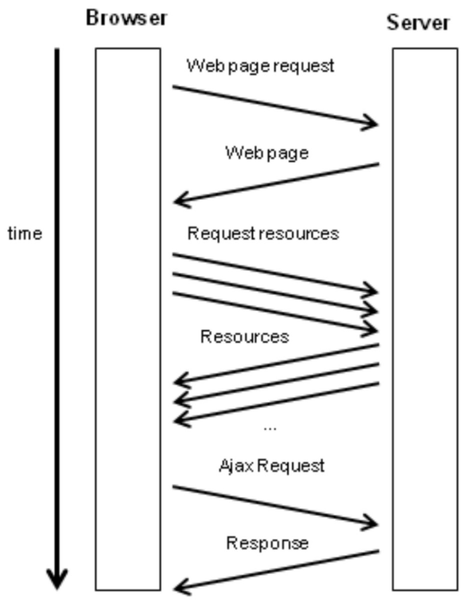

Figure 1: Resource processing in a web browser.

followed by DOM processing and rendering. New content is then obtained either through pre-loaded scripts or AJAX requests, enhancing interactivity [11, 21]. This communication process comprises three steps (see Figure 1):

1. **Request web page:** Initiates the primary resource request for content structure.
2. **Request resources:** Fetches additional resources to aid page construction, such as images.
3. **AJAX request:** Makes any number of HTTP requests for resources that are needed.

It is the final step that distinguishes a SPA from a traditional web page, allowing efficient resource retrieval for enhanced interactivity.

According to this processing model, the same page can take on various forms, with different elements and configurations. We will define a page *state* as a version of the page captured at one given instant. Page *states* can be represented in different ways. Some approaches use the DOM to define the state of a page [28], while other approaches use a combination of URL, links, forms, and a list of existing events [16].

Automatic accessibility evaluation tools were primarily designed to assess static pages. However, it is important to note another limitation of these tools: their functionality and effectiveness are limited in a dynamic context. Specifically, they can only analyze the original state of a page and are unable to recognize or understand when a state is updated due to user interaction [3]. In addition to this shortcoming, these tools are also unable to invoke different events in the SPA and capture new states to evaluate them. As such, it is not possible to obtain a reliable assessment of the accessibility of a SPA using traditional accessibility evaluation tools.

To be able to evaluate a SPA in its entirety, it is necessary to adapt the tools that currently exist. For that, mechanisms that can crawl the different states of a page are needed.

2.4 Crawlers

Web crawlers were initially built to explore all the links that a web application or website presents. The process starts with a URL which is then explored to find other links. All the links found are then placed in a list that will also be explored with the process continuing until all the entries in the list have been explored or a certain user-selected number of URLs have been found [17].

As a consequence of evolving web technology and the changes it promoted for websites and web applications, the concept of crawler has undergone some changes. It is now necessary to do a more comprehensive exploration to obtain more details about the page. Currently, a crawler works by obtaining HTML code, DOM, CSS, images, and Javascript scripts [16]. These elements are then processed to isolate elements, mostly HTML tags, that would allow the exploration to be extended. The elements found are then placed in a list for further exploration. Crawlers have appeared in the literature with two distinct approaches, automatic and semi-automatic, both of which have advantages and disadvantages.

2.4.1 Automatic crawlers. We consider automatic crawlers as completely autonomous tools, that do not require user interaction, thus presenting themselves as an option in which no technical knowledge or knowledge of the application to be explored is required.

Crawljax [28] presents itself as "state-of-the-art" [26] and as one of the first SPA crawler tools. Crawljax is an open-source automatic tool which, given a URL, explores a web application by entering random values in input fields and invoking all the elements it identifies as clickable [16]. At the same time, it builds a graph with all the states it has found [16], with the nodes being the states and the edges being how the state was reached. Crawljax's biggest limitation is the small number of events it can invoke, over an equally small number of HTML tags. Crawljax considers only the `<a>`, `<div>`, `<input>` and `<img>` tags and the *click* and *mouseOver* events [23], thus restricting the number of states that can be explored, given other HTML tags and events can generate states. Crawljax does, however, provide a set of plugins that can extend its usefulness to various applications, including the possibility of expanding the list of tags and events that can be used [18].

Demodocus [4] presents a similar approach to Crawljax. Given a URL, all possible interactions are invoked to try and reach the maximum number of states. The graph also has a similar structure to what Crawljax builds, with the nodes being the different states reached and the edges being how they are reached, i.e., the interaction a user performs in the web application. Demodocus has the particularity of having a longer list of tags and events compared to Crawljax. The biggest limitation of this tool is the impossibility of entering values in input fields and thus explore the new states these might trigger. This problem limits this approach because, in many instances, a large portion of the web application only appears after interactions that require inputs. For example, an authentication page where username and password are required.

Finally, Autocrawler [17] is a crawler that builds on the concepts presented by Crawljax by adding mechanisms that can detect if there are pop-up windows and invoking CSS pseudo-classes. To deal with pop-up windows, Autocrawler creates a state for the pop-up window element. This state is then treated like any other in which all possible events are identified and invoked, thus exploring as many states as possible, until the event that closes the pop-up window is invoked and the remaining states are explored. The problem of invoking pseudo-class actions is solved by placing a proxy server between the browser and the web application. This proxy analyzes the code received from the web application to detect if there is any invocation of the `:hover` function. If there is, a hook function is placed that replaces the invocation of the `:hover` function with an `onClick` event, which can be invoked using JavaScript.

2.4.2 Semi-automatic crawlers. Semi-automatic crawlers require user intervention. This intervention may come in different ways, from interacting with the crawler to providing initial input.

AWET (Automated Web application testing based on Exploratory Testing) [26] is a tool that helps develop tests for web applications automatically. AWET uses a crawler with directions to explore as many states of a web application as possible and create more tests automatically. This tool initially needs a predefined set of tests as input. It is with these tests that the direction of the crawler is ultimately determined. The tests provided are then analyzed to remove inputs, interactions and their order of execution to create rules and strategies to apply to the crawler, thus exploring more states, and avoiding a disorganized and random process.

GUIDE [18] is another semi-automatic approach that relies heavily on user input. GUIDE is an interactive crawler that actively asks the user which input values to use in different scenarios, thus allowing users to interact without needing knowledge of the web application's underlying code. During the crawling process, if the crawler reaches a state where there is a way to proceed but user intervention is required, it will ask the user how to proceed by displaying screenshots in which the input fields are highlighted. The user can then define one or more values that the crawler should use. Using different values allows a user to create different directives, i.e., different options that the crawler will have to go through. This process is then repeated until there are no more states or until the user completes the process.

2.5 State comparison and evaluation

A crucial aspect of a crawler is the ability to compare states to ensure that the current state differs from all visited states.

Autocrawler [17] uses a simple solution for comparing states. Given a string with a state's HTML, it is compared with the HTML of the other states found, using the concept of "Edit Distance" [25].

Demodocus [4] compares states using the "Three-Stage Comparator Pipeline". The first step is to transform the HTML into a string and compare it with all the states already identified. If the string differs, then a new state was found. The second step is to compare the DOM of each state. The comparison focuses on the structure of the DOM to check whether nodes have been added or removed. If the comparison shows differences, the state is new. The last step uses the "Levenshtein technique", which compares the distance between nodes with text between the different states.

As mentioned before, the combination of a crawler to obtain states and the evaluation of their accessibility is a relatively new concept. As such, there are not many tools that do both without requiring changes or the use of auxiliary tools.

Demodocus [4], in addition to crawling a web application to obtain the various states, also performs an accessibility assessment. The accessibility assessment uses user models, i.e., user representations that simulate one or more disabilities, or standard accessibility requirements. Initially, when building the state graph, an *omniuser* is used, which can invoke all kinds of interactions and thus discover as many states as possible; once this process is complete, an *omnigraph* is created. Next, various subgraphs are constructed with the help of user models. A subgraph makes it possible to check if there are states that a user with disabilities cannot reach and thus confirm whether or not there are accessibility barriers.

3 QUALSTATE

This section presents QualState, our proposed solution to address the challenge of the evaluation of the accessibility of SPAs. QualState is an independent module that uses the Puppeteer [13] framework to control the browser and execute a set of actions to explore a SPA and find as many different states as possible.

Puppeteer is a Node.js framework that provides several APIs to control the browser through the Chrome DevTools protocol. The framework provides, in addition to these, other APIs that allow debugging websites and automating tests. Puppeteer is mostly used to automate tests in the browser. However, in the context of this work, it is used to carry out actions to reach different states.

The following sections introduce QualState, starting by introducing the main concepts and components it uses. This is followed by a description of QualState's architecture. Next, the workflow of the crawling process implemented in QualState is explained. Finally, we briefly describe how it was integrated with QualWeb to conduct accessibility evaluations of the states found by QualState.

3.1 Main concepts

This section introduces the main concepts required to understand QualState. To facilitate comprehension, we start with a generic description of how QualState operates. QualState starts by loading a page whose URL has been provided by the user. After the page is loaded, *actions* are executed on the page to try to generate new states. After each action, QualState checks whether the current *state* has already been found and saved. This process is repeated until there are no more actions or until the maximum number of states specified by the user has been reached.

3.1.1 Action. The *action* is a crucial concept in QualState. An *action* represents, as the name indicates, an action that can be performed on elements of the web page. An action is represented by a JSON object that contains information about how it should be carried out. There are two types of actions: automatic actions and manual actions. These are distinguished by the way they are obtained: automatic actions are those found by QualState through the processing of the web page. Manual actions are those added by the user, using the `interactions` configuration option.

Automatic actions always have the same structure. They are described by five properties:

- `user`: indicates whether the action is manual or automatic.
- `id`: identification of the element on which the action is to be performed.
- `className`: the element's class.
- `eventType`: type of event to be performed.
- `selector`: the element's selector.

The structure of manual actions depends on the information the user provides. The complete structure has 4 properties:

- `user`: indicates whether the action is manual or automatic.
- `beforeAction`: specifies what must be performed before the `endAction` is executed, usually associated with entering content on an input field.
- `endAction`: specifies the last instruction to be performed, associated with an event to be triggered on an element of the page.
- `wait`: the time to wait after any instruction carried out within the action.

3.1.2 *State.*

A *state* is a representation of the DOM of a webpage. During the crawling process, states go through two phases: comparison and persistence. QualState maintains one set at any time for the states that are found. When the crawling process reaches a new state, this is compared to the states already found. If it is a new state then it is persisted to the set, otherwise it is discarded.

3.1.3 *Interaction.*

As mentioned earlier, crawlers that are limited to "pressing" buttons are limited in the amount of exploration they can carry out, because, often, some kind of interaction is needed to expand the search for states. To facilitate this process, QualState offers the user the possibility of providing a set of data relevant to the exploration to be carried out. These instructions allow the user to provide as many inputs as appropriate, to allow the tool to explore the largest number of states. These states can also arise through inputs that generate errors, such as entering the wrong password or numerical values that exceed an established limit.

These instructions correspond to the *interaction* concept. QualState supports different types of instructions: `forms` instructions are used to add one or more *actions* associated with form elements found; `inputs` instructions are used to manage input elements identified in the crawling process; and `directions` instructions are used to direct the exploration to specific paths inside the SPA.

The `forms` instructions are characterised by three properties:

- `input`: Identifies the form elements that should be filled by QualState. It is a set of property-value pairs, where the property specifies the id of the form element to fill and the value is the content to be entered in the element.
- `action`: Identifies the form elements that will be the target of an event. This object contains the `id` of the form element and the type of event that is to be triggered on the element. For example, it can provide the id of the form's Submit button, and the *click* event to submit the form.
- `info`: Identifies the form where the `input` and `action` will be performed. The `info` object may also include the `wait` property to specify the time, in milliseconds, the crawling process should wait after any action within the form is executed.

The `inputs` object is used for QualState to manage input elements that may be encountered during crawling. This allows the user to test correct and incorrect values for every input element desired, thus finding states that arise when an incorrect input is presented and states that arise when the correct input is placed. The `inputs` instructions are characterised by two properties:

- `value`: Identifies the input element to be filled by QualState. It is a set of property-value pairs, where the property specifies the id of the input element and the value is the content to be entered in the element.
- `info`: Contains the `wait` property, used to specify the time, in milliseconds, that the crawling process should wait after filling an input element. This is useful, for example, in those instances where filling a field triggers a response that needs to fetch data from a server request. By specifying a delay before proceeding, the user can ensure there is enough time for the request to be answered.

During the crawl process, the user may feel that there is no need to explore certain paths, preferring to focus the resources on trying to reach other states they are more interested in. As such, QualState offers the user the option of providing instructions that direct the crawl to the desired path. The `directions` instructions are characterised by two properties:

- `actions`: Contain properties to direct how the crawl process should proceed. Each action is defined by three properties:
  - `values`: Contains one or more property-value pairs, defining the id of an element and the value that the user wants QualState to insert in the element during the crawling.
  - `action`: Specifies an element on the page and the type of event to perform on the element to trigger a desired result. The `action` object requires the `id` of the element, and the `event` that should be triggered on the element.
  - `info`: Contains the `wait` property representing the time, in milliseconds, the process must wait after any action performed.
- `info`: Specifies how the crawling should proceed after a direction instruction is executed. This can be specified by combining two properties. The `crawl` attribute indicates whether or not the crawl process should continue after completing a direction. This attribute can take one of two possible values: "stop" and "continue". The `save` attribute is used to indicate whether the states that are crossed when following the `directions` instructions should be saved or discarded. This allows the user to direct the crawling to a specific state of the SPA and start saving states only after reaching it.

3.1.4 *Configuration.*

Users can provide a set of configurations to customize state exploration and can add instructions to be executed during the crawling process. Even though this is not a concept specific to QualState, these settings are a crucial part of its operation.

- `url`: Specifies the URL of the SPA to explore.
- `headless`: Indicates whether Puppeteer should be run in headless mode (i.e., without the graphical UI) or not.
- `waitTime`: Specifies the time, in milliseconds, that QualState should wait before and after any action. Whenever QualState initially loads a page, it waits for it to complete loading before continuing with the process. However, it is common for a SPA to not load any more pages after the initial one, instead only updating the content of the already loaded page. Through this setting, the user can specify a waiting time between actions to ensure the SPA can complete the changes to a page before continuing the crawling.

- `maxStates`: Defines the maximum number of states the user wishes to find. When the maximum number is reached, the crawling is terminated.
- `numberOfProcess`: Number of crawling processes running concurrently to allow the process to be completed faster.
- `viewport`: Specifies the characteristics of the browser's viewport so the SPA can be crawled in different contexts. The following properties can be defined: `mobile`, `landscape`, `userAgent` and `resolution`.
- `log`: Selects where to display the crawl results. Two Boolean properties are used: `file` and `console`.
- `ignore`: Identifies which of the page's elements will be ignored during the crawl process. The user can specify those elements in two ways. Using the `ids_events` property, the user can specify elements where events should not be triggered during the crawling process. Using the `ids_compare` property, the user can specify which elements should be ignored when deciding if a state is a new state or not. The latter possibility is useful, for example, when there are elements with content that auto-updates (e.g., a clock or a stock ticker) and therefore would represent a new state with each update. Even though certain elements can be ignored during the crawling, they will always be present in the states saved as output of QualState.
- `cookies`: Allows users to dismiss the cookies notice. For that, three properties can be specified. `waitBefore` defines the time QualState should wait after the page is loaded. `btn` identifies the button QualState should trigger to dismiss the cookies notice. `waitAfter` defines the time QualState should wait after triggering a `click` event on the specified button.
- `login`: Allows users to specify login credentials required to authenticate in the SPA. The login object is an array of objects defining the steps required for logging in. Each of the objects can be of one type: `wait` specifying the time the process should pause before moving to the next step; `action` specifying an action to be performed, such as clicking a login button; and `credentials` used to specify the value that should be entered by QualState in input fields, such as a username or a password.

3.2 Architecture

QualState architecture aims to be as modular as possible, making it a more scalable solution that is easy to reuse and maintain (Figure 2). In the following, we detail the main modules and their processing.

3.2.1 Events. The **events** module is responsible for finding all the events on a given page, to identify the set of actions to be performed.

Obtaining events is carried out in two phases. The first phase uses the `DomDebugger.getEventListeners` function provided by the Chrome DevTools protocol to check whether an element has an event or not, and, if so, what event it is. The second phase is a filtering process for all the elements on the page that can be activated by users without requiring an explicit event. The process filters all the elements `<a>` and `<input type = 'submit'>`, because, even without associated events, they are processed by user agents (i.e., browsers) as if they had a *click* event, which means they can generate a new state. Finally, after finding all the actionable elements, each one is transformed into an `action`.

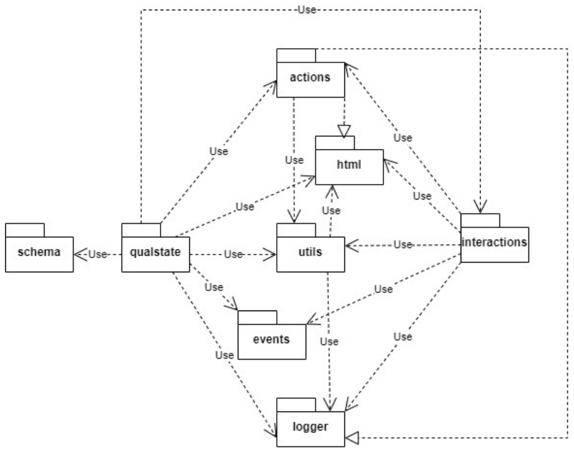

Figure 2: QualState's architecture

3.2.2 Actions. The **actions** module carryies out the actions necessary to reach a state. To do this, the `performAction` method is provided. The method receives an array of actions (see section 3.1.1) to perform. For automatic actions an event is triggered, while for manual actions there is the possibility of performing input actions.

3.2.3 HTML. QualState relies on being able to identify elements on a page to crawl the SPA. While many elements will already have an `id` provided by the developer, it is not possible to rely on that when trying to explore all states. For that reason, QualState automatically generates a selector for each element on the page and uses it to uniquely identify the element. The **html** module is responsible for processing the page to add or remove the `_selector` attribute to every element.

3.2.4 Interactions. The **interactions** module processes instructions for interactions that the user wants to happen. These are processed in three sub-modules: inputs, forms and directions.

The **inputs** module has two responsibilities: to check whether the input element exists on the page and to process the input instruction into a manual action. The module receives the set of inputs provided via the `inputs` option and the *state* the crawl is in. It then checks if all the elements of the input set are present in the state. If they are, the input is transformed into a manual action.

The **forms** module is responsible for processing the data from the `forms` option. This module processes the information in a very similar way to the **inputs** module. Initially, the module receives the set of forms the user entered, and the *state* the crawl is in. It then checks that the form element exists on the state. If it does, the form is transformed into a manual action.

The **directions** module has the responsibility of receiving and executing the `directions` instructions provided by the user. The process involves interpreting the execution of each instruction, transforming them into one or more actions.

3.2.5 Logger. The **logger** module is responsible for recording all the outputs resulting from the crawl. The logs can be made to the console or files and are divided into two categories: `info` and `error`. The console's logs display the information on the console, distinguished by category. The file's logs are written in two files: "logs.log" and "states.log". The former stores all the information about the actions performed, with each log entry associated with a hash representing the state where it originated. The latter stores all states found. Each state is described by a hash representing the state and the complete state string.

3.2.6 QualState. The **qualstate** module coordinates the entire crawling process. This module is responsible for starting and ending the crawling. It is also in this module that all the tasks are delegated, including: 1) validating the options given by the user, 2) searching for events, 3) managing how the `forms`, `inputs` and `directions` are controlled, and finally, 4) comparing states.

3.3 Workflow

This section provides further details about the processing of information by the QualState crawler.

Once the process has started, the first task is to check that the settings are correct. In the event of a problem with the structure of options or an incorrect option, an error message is displayed and the process is immediately terminated. After validating the settings, the initial setup is carried out. This includes initializing the variables and starting the browser, loading a page with the URL that the user has provided in the settings.

Before the crawl process starts, QualState processes any cookies or login configuration that the user provided in the configuration file. It then checks whether any directions need to be executed. If so, these are processed. The process starts by creating a queue just for the directions before starting the crawling. During the processing, a set of manual actions is created for each direction that must be executed. After each manual action, QualState checks whether or not the status should be saved according to what was specified in the `directions` configuration. The process of saving the state involves initially obtaining a hash of the state. It then checks for elements that are to be ignored, and if so, these elements are removed from the state. Then, regardless of whether or not elements have been removed, another hash of the state is obtained. This hash is then compared with the others in the comparing set. If the hash is repeated, then the state is repeated and the process ends. If the hash does not exist in the comparing set, QualState has found a new state. In this instance, the state is persisted by placing the changed hash in the comparing set and the original hash in the evaluating set along with the string version of the state. At the end of executing the set of directions, the `crawl` configuration is checked. If it is set to `continue`, QualState will collect the events, forms and inputs in this state. If it is set to `stop`, the process ends. This process is repeated until the directions queue is complete.

After executing the directions, QualState has created a set of automatic actions for the events and manual actions for the forms and inputs. The state is then saved and a new queue with all the actions obtained previously is created. After filling in the queue, the SPA is explored - for each element in the queue, the page is set up. This setup includes creating a tab within the browser using the `viewport` configuration. Once the page has been created, the actions are executed. Once the actions have been completed, the actions in the new state are obtained. These actions include automatic actions, i.e., all the events present in the current state, and manual actions, i.e., forms and inputs that may exist on the page.

The next step is to check whether the state has already been found and, consequently, decide whether or not it should be saved. The process of confirming whether the state exists starts by removing any elements that have been marked for not comparing and then confirming whether or not the state is repeated by comparing it with the states saved so far. If the state is not repeated it is saved. The process then continues to the next phase.

The last step of each execution is to populate the queue with the new actions found on the state, thus inserting more elements into the queue which will continue its process until it runs out of elements. While the queue is being processed there is only one stopping condition: if at any time the maximum number of states is reached, then the queue is terminated and the process moves on to the final phase. In the final phase, the browser is closed and the necessary information is presented to the user via the console or stored in a file, depending on the configuration selected.

The whole processing workflow is visually depicted in Figure 3.

3.4 QualWeb integration

One of the goals of this work is, in addition to developing a solution that can obtain as many states as possible, to evaluate the accessibility of the page taking into account its different states. Thus, it was necessary to integrate QualState with QualWeb. As such, when using QualWeb, users can configure it to use QualState and thus have an accessibility report for all the states found.

A change to QualWeb's reporting format was required when using QualState. The change reflects that there may be one or more state evaluations for the same URL. As such, the new reporting structure for each page includes an array, where each element of the array contains the accessibility assessment for each identified state in the URL. Given this change in the reporting structure, at the moment, the option to generate reports in EARL format in QualWeb is not available when using QualState.

4 EVALUATION

To perform an initial evaluation of QualState's performance and contribution to accessibility evaluation, we conducted two sets of tests on different SPAs. All the tests were carried out on a computer with an Intel(R) Core(TM) i7-7700HQ processor running at 2.80GHz - 2.81 GHz, with 12 GB of RAM, on Windows 10.

The tests were carried out on the following SPAs:

- Plant22[1] - SPA with a limited set of identifiable states comprising different media elements.
- PLACM[2] - SPA with visualizations of accessibility data, with a very large number of states.
- Portuguese Accessibility Observatory[3] - SPA with tabular presentation of accessibility data, with a very large number of states.

[1]https://plant22.co/ (accessed February 2024)
[2]http://qualweb.di.fc.ul.pt/placm/assertions/continent/ (accessed February 2024)
[3]https://observatorio.acessibilidade.gov.pt/directories (accessed February 2024)

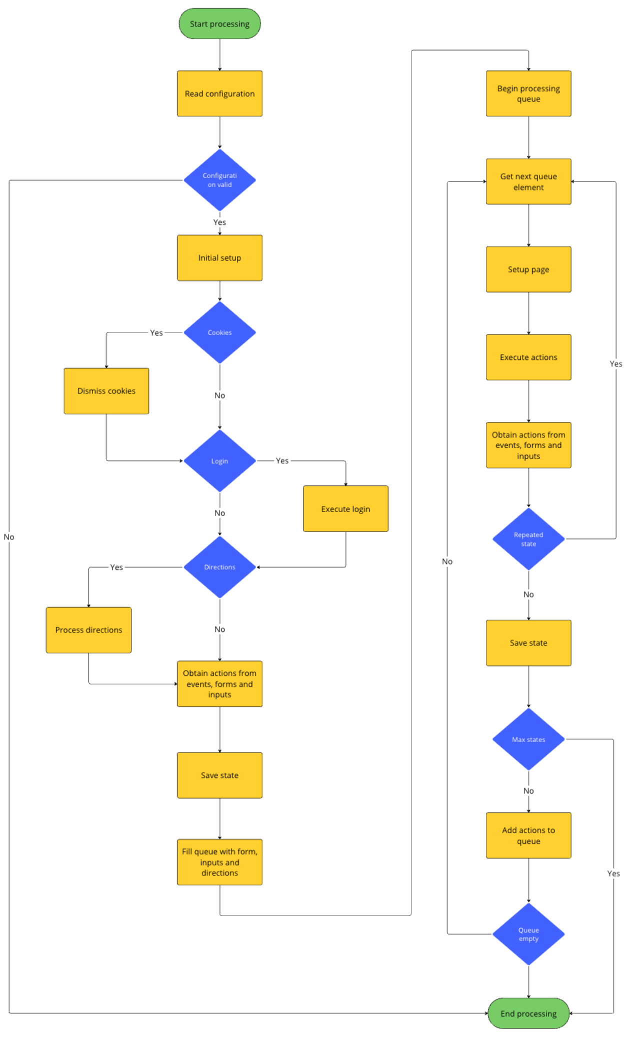

Figure 3: QualState process flow

To assess the contribution of QualState to the performance of QualWeb, we compared the outcomes of QualWeb with and without QualState. The first set of tests consisted of exploring how many additional accessibility tests can be done by using QualWeb with QualState when compared with QualWeb without QualState. The test consisted of running QualWeb without QualState and running QualWeb with QualState limited to 2, 3, 5 and 10 maximum number of states, to understand the impact of exploring more states.

Tables 4, 5 and 6 present the number of tests by outcome that were executed by QualWeb for the Plant22, PLACM and Portuguese Accessibility Observatory SPAs, respectively, averaged by the number of states that were assessed.

The QualWeb version that was used contained a total of 82 tests. In every state, 82 tests were executed, which explains that the total number of tests averaged by the number of states is always equal to 82. The impact of QualState can be seen in the decrease in the average number of tests that are inapplicable when the number of states increases. Correspondingly, the number of passed, warning and failed tests increases. What this translates to, is that, by exploring more states, QualWeb can find more elements that are applicable to tests that were not applicable in the initial state. For instance, in the Plant22 evaluation of the initial state, the WCAG technique test "QW-WCAG-T8", which tests whether the content of a text alternative for an image is appropriate or not, was found to be inapplicable, meaning no images with alternative text had been found. When evaluating the second state, the same test returned a warning, meaning that an image with a text alternative had been found (and needed to be checked by a human for its appropriateness).

As can be observed by analysing the tables, there is a clear tendency for the decrease of the number of inapplicable tests as the number of states increases. Nevertheless, it is also clear that the

Max States	1	2	3	5	10
Passed tests	11	27	38	66	126
Warning tests	7	16	25	43	88
Failed tests	3	6	10	16	35
Inapplicable tests	61	115	173	285	571
Total tests	82	164	246	410	820

Table 1: Number of tests executed by QualWeb on Plant22 per maximum number of states evaluated.

Max States	1	2	3	5	10
Passed tests	19	42	63	105	210
Warning tests	10	20	30	50	100
Failed tests	2	4	6	10	20
Inapplicable tests	51	98	147	245	490
Total tests	82	164	246	410	820

Table 2: Number of tests executed by QualWeb on PLACM per maximum number of states evaluated.

Max States	1	2	3	5	10
Passed tests	24	51	77	131	268
Warning tests	9	21	33	57	120
Failed tests	1	7	13	25	60
Inapplicable tests	48	85	123	197	372
Total tests	82	164	246	410	820

Table 3: Number of tests executed by QualWeb on the Portuguese Accessibility Observatory per maximum number of states evaluated.

States	1	2	3	5	10
Passed	11	13.5	12.7	13.2	12.6
Warning	7	8	8.3	8.6	8.8
Failed	3	3	3.3	3.2	3.5
Inapplicable	61	57.5	57.7	57	57.1
Total	82	82	82	82	82

Table 4: Number of tests per outcome executed by QualWeb on Plant22 averaged by number of states evaluated.

States	1	2	3	5	10
Passed	19	21	21	21	21
Warning	10	10	10	10	10
Failed	2	2	2	2	2
Inapplicable	51	49	49	49	49
Total	82	82	82	82	82

Table 5: Number of tests per outcome executed by QualWeb on PLACM averaged by number of states evaluated.

States	1	2	3	5	10
Passed	24	25.5	25.7	26.2	26.8
Warning	9	10.5	11	11.4	12
Failed	1	3.5	4.3	5	6
Inapplicable	48	42.5	41	39.4	37.2
Total	82	82	82	82	82

Table 6: Number of tests per outcome executed by QualWeb on the Portuguese Accessibility Observatory averaged by number of states evaluated.

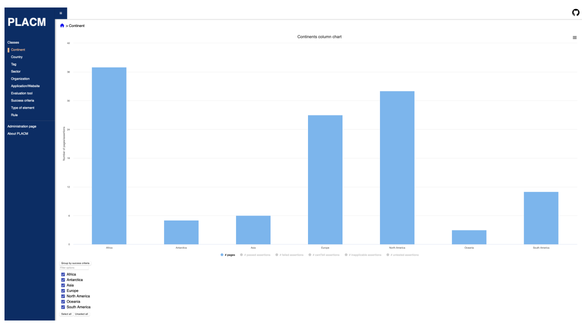

Figure 4: The PLACM SPA showing a bar chart of accessibility data grouped by continent.

impact is dependent on the characteristics of the web application. This is clear in the numbers for PLACM. PLACM's presentation is very similar in all renderings. As can be seen in Figure 4, PLACM presents a bar chart of accessibility data and a set of filters to control the presentation. The chart can be selected to drill down on the data, which can also be achieved by selecting different data levels on the left side menu. Except for the "Administration" and "About" pages, all the pages follow this same structure. This is the reason why there is a decrease in the number of inapplicable outcomes from the first to the second states, but no further evolution for the next 8 states. Because we limited the number of states to be analysed to 10, in the evaluation we did not reach other types of "pages" of the PLACM SPA. However, we see that, for both Plant22 and the Portuguese Accessibility Observatory, even a small increase in the number of states evaluated leads to a consistent increase in the tests that can be applied by QualWeb, which reveals the ability to find more elements to evaluate.

The second set of tests was focused on the performance of QualState. These tests tried to gauge how the number of processes running concurrently influences the number of states found at the same time. The tests were carried out on the same set of websites. Tables 7, 8 and 9 present the number of different states found running QualState during 1, 2 or 3 minutes with 1, 5 or 10 concurrent processes.

	1 process	5 processes	10 processes
1 minute	6	13	13
2 minutes	13	14	14
3 minutes	13	14	20

Table 7: Number of different states found per search time and number of processes running for the Plant22 SPA.

	1 process	5 processes	10 processes
1 minute	18	80	128
2 minutes	48	230	415
3 minutes	80	377	715

Table 8: Number of different states found per search time and number of processes running for the PLACM SPA.

	1 process	5 processes	10 processes
1 minute	6	21	31
2 minutes	15	68	100
3 minutes	25	120	150

Table 9: Number of different states found per search time and number of processes running for the Portuguese Accessibility Observatory SPA.

The tables show that, as expected, increases in both the time for crawling and the number of crawling processes lead to increases in the number of states found.

Even with only just this initial, small evaluation, it was possible to verify that QualWeb using QualState performs a higher number of accessibility checks during a single evaluation. The new exploration functionality results in a more complete accessibility assessment, which is not limited to the initial state of a web page.

5 CONCLUSIONS AND FUTURE WORK

QualState presents a solution to the problem of automatically assessing the accessibility of dynamic web pages, i.e., pages that update the content being presented to the user without loading a new page. QualState is capable of crawling the different states of a web page and can be configured to interact with certain elements and ignore others. This allows users to trigger, for example, error and success states in forms. It can also be configured to fill out login information and dismiss cookie banners, thus supporting the exploration of, for example, intranets and other websites that require authentication.

By integrating QualState with QualWeb, an automated web accessibility evaluation engine, we have benefited from the crawling ability to augment the usefulness of this accessibility evaluation tool. Early testing shows improvements in the number of applicable elements found on pages that have multiple states, compared to the elements that can be assessed without the crawling possibility.

Despite the positive initial results, further research into aspects of this tool would be extremely beneficial. Two perspectives in particular need to be considered. One is the ability to automatically explore different types of events. Currently, QualState only responds to *click* events. Even though these are the more common events associated with interactivity and navigation in web pages, in the future, to ensure a more complete crawling process, it would be useful to consider other types of events. The second perspective is related to the way states are compared. Different combinations of user agents and assistive technology can result in different renderings of the same DOM. It would be interesting to explore how these could impact state comparison and exploration but also to use that information to automate state exploration for users with different functional needs. For example, a change in the colour of an element could represent a new state relevant to a user who can see the element but would be irrelevant to a screen reader user. Finally, it is paramount to conduct a more in-depth evaluation of the benefits of state crawling for web accessibility evaluation.

ACKNOWLEDGMENTS

This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020)

REFERENCES

[1] Julio Abascal, Myriam Arrue, and Xabier Valencia. 2019. *Tools for Web Accessibility Evaluation.* Springer, London, 479–503. https://doi.org/10.1007/978-1-4471-7440-0_26

[2] Hayfa.Y. Abuaddous, Mohd Zalisham Jali, and Nurlida Basir. 2016. Web Accessibility Challenges. *International Journal of Advanced Computer Science and Applications* 7, 10 (2016), 172–181. https://doi.org/10.14569/IJACSA.2016.071023

[3] Humberto Lidio Antonelli, Leonardo Sensiate, Willian Massami Watanabe, and Renata Pontin de Mattos Fortes. 2019. Challenges of automatically evaluating rich internet applications accessibility. In *Proceedings of the 37th ACM International Conference on the Design of Communication* (Portland, Oregon) *(SIGDOC '19)*. Association for Computing Machinery, New York, NY, USA, Article 32, 6 pages. https://doi.org/10.1145/3328020.3353950

[4] Trevor Bostic, Jeffrey Stanley, John Higgins, Daniel Chudnov, Justin Brunelle, and Brittany Tracy. 2021. Automated Evaluation of Web Site Accessibility Using A Dynamic Accessibility Measurement Crawler. arXiv:2110.14097 [cs.IR] https://arxiv.org/abs/2110.14097

[5] BroadbandSearch.net. 2024. Key Internet Statistics in 2024. https://www.broadbandsearch.net/blog/internet-statistics#post-navigation-0.

[6] Milton Campoverde-Molina, Sergio Luján-Mora, and Llorenç Valverde García. 2020. Empirical Studies on Web Accessibility of Educational Websites: A Systematic Literature Review. *IEEE Access* 8 (2020), 91676–91700. https://doi.org/10.1109/ACCESS.2020.2994288

[7] Cantan Group. 2023. ACHECKS Accessibility Checker. https://www.achecks.org/.

[8] Deque. 2024. Axe Accessibility Testing Tool. https://www.deque.com/axe/.

[9] ETSI. 2021. Accessibility requirements for ICT products and services. https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf.

[10] Faculdade de Ciências da Universidade de Lisboa. 2024. Qualweb Accessibility Evaluator. https://qualweb.di.fc.ul.pt/evaluator/.

[11] Nádia Fernandes, Ana Sofia Batista, Daniel Costa, Carlos Duarte, and Luís Carriço. 2013. Three Web Accessibility Evaluation Perspectives for RIA. In *Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility* (Rio de Janeiro, Brazil) *(W4A '13)*. Association for Computing Machinery, New York, NY, USA, Article 12, 9 pages. https://doi.org/10.1145/2461121.2461122

[12] General Services Administration. 2023. Section 508. https://www.section508.gov/.

[13] Google. 2024. Puppeteer. https://pptr.dev/.

[14] Nicola Iannuzzi, Marco Manca, Fabio Paternò, and Carmen Santoro. 2023. Large Scale Automatic Web Accessibility Validation. In *Proceedings of the 2023 ACM Conference on Information Technology for Social Good* (Lisbon, Portugal) *(GoodIT '23)*. Association for Computing Machinery, New York, NY, USA, 307–314. https://doi.org/10.1145/3582515.3609549

[15] Jonathan Lazar. 2022. Managing Digital Accessibility at Universities during the COVID-19 Pandemic. *Univers. Access Inf. Soc.* 21, 3 (aug 2022), 749–765. https://doi.org/10.1007/s10209-021-00792-5

[16] Manuel Leithner and Dimitris E. Simos. 2020. XIEv: dynamic analysis for crawling and modeling of web applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing* (Brno, Czech Republic) *(SAC '20)*. Association for Computing Machinery, New York, NY, USA, 2201–2210. https://doi.org/10.1145/3341105.3373885

[17] Yan Li, Peiyi Han, Chuanyi Liu, and Binxing Fang. 2018. Automatically Crawling Dynamic Web Applications via Proxy-Based JavaScript Injection and Runtime Analysis. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. IEEE, 242–249. https://doi.org/10.1109/DSC.2018.00042

[18] Chien-Hung Liu, Woei-Kae Chen, and Chi-Chia Sun. 2020. GUIDE: an interactive and incremental approach for crawling Web applications. *The Journal of Supercomputing* 76 (03 2020). https://doi.org/10.1007/s11227-018-2335-4

[19] Beatriz Martins and Carlos Duarte. 2023. A large-scale web accessibility analysis considering technology adoption. *Universal Access in the Information Society* (July 2023). https://doi.org/10.1007/s10209-023-01010-0

[20] Beatriz Martins and Carlos Duarte. 2024. Large-scale study of web accessibility metrics. *Universal Access in the Information Society* 23, 1 (March 2024), 411–434. https://doi.org/10.1007/s10209-022-00956-x

[21] MDN contributors. 2022. Populating the page: how browsers work. https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work#parsing.

[22] Ali Mesbah and Arie van Deursen. 2007. Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 181–190. https://doi.org/10.1109/CSMR.2007.33

[23] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-Based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering* 38 (2012), 35–53.

[24] Seyedamirhossein Mousavi. 2017. Maintainability Evaluation of Single Page Application Frameworks : Angular2 vs. React. , 39 pages.

[25] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Comput. Surv.* 33, 1 (mar 2001), 31–88. https://doi.org/10.1145/375360.375365

[26] Nezih Sunman, Yiğit Soydan, and Hasan Sözer. 2022. Automated Web application testing driven by pre-recorded test cases. *Journal of Systems and Software* 193 (07 2022), 111441. https://doi.org/10.1016/j.jss.2022.111441

[27] W3C. 2018. Evaluation and Report Language (EARL) Overview. https://www.w3.org/WAI/standards-guidelines/earl/.

[28] W3C. 2022. Understanding the Four Principles of Accessibility. https://www.w3.org/WAI/WCAG21/Understanding/intro#understanding-the-four-principles-of-accessibility.

[29] W3C. 2023. WCAG 2 Overview. https://www.w3.org/WAI/standards-guidelines/wcag/.

[30] W3C. 2024. About ACT Rules. https://www.w3.org/WAI/standards-guidelines/act/rules/about/.

[31] WebAIM. 2024. WAVE Web Accessibility Evaluation Tool. https://wave.webaim.org/.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009