Ma11y: A Mutation Framework for Web Accessibility Testing

Mahan Tafreshipour
University of California at Irvine
Irvine, USA
mtafresh@uci.edu

Anmol Deshpande
University of California at Irvine
Irvine, USA
adeshpa2@uci.edu

Forough Mehralian
University of California at Irvine
Irvine, USA
fmehrali@uci.edu

Iftekhar Ahmed
University of California at Irvine
Irvine, USA
iftekha@uci.edu

Sam Malek
University of California at Irvine
Irvine, USA
malek@uci.edu

ABSTRACT

Despite the availability of numerous automatic accessibility testing solutions, web accessibility issues persist on many websites. Moreover, there is a lack of systematic evaluations of the efficacy of current accessibility testing tools. To address this gap, we present the first mutation analysis framework, called Ma11y, designed to assess web accessibility testing tools. Ma11y includes 25 mutation operators that intentionally violate various accessibility principles and an automated oracle to determine whether a mutant is detected by a testing tool. Evaluation on real-world websites demonstrates the practical applicability of the mutation operators and the framework's capacity to assess tool performance. Our results demonstrate that the current tools cannot identify nearly 50% of the accessibility bugs injected by our framework, thus underscoring the need for the development of more effective accessibility testing tools. Finally, the framework's accuracy and performance attest to its potential for seamless and automated application in practical settings.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Human-centered computing → Accessibility design and evaluation methods**.

KEYWORDS

Software Accessibility, Mutation Testing, Web

ACM Reference Format:
Mahan Tafreshipour, Anmol Deshpande, Forough Mehralian, Iftekhar Ahmed, and Sam Malek. 2024. Ma11y: A Mutation Framework for Web Accessibility Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652113

1 INTRODUCTION

In today's world, websites play a crucial role in facilitating our daily routines, from commuting to staying connected to managing our finances [59]. With the ever-increasing usage of websites, it is imperative that developers prioritize accessibility to cater to the needs of everyone, including the 15% of the population with disabilities [60]. However, recent research reveals that a significant portion of websites are plagued with accessibility issues [57], including low-contrast text making it difficult for people with low vision to perceive the text. Additionally, nearly 60% of the analyzed websites lack alternative text for their images, which hinders individuals with visual impairments from understanding the content and context conveyed by the images.

Several studies have investigated the reasons behind such prevalence of accessibility issues and have identified developers' lack of awareness of website accessibility guidelines and lack of reliable, automated tools as some of the primary reasons [2, 3, 25]. In fact, manual testing is still the most reliable way to ensure website accessibility; however, it can be expensive to hire individuals and have them thoroughly examine each feature of the website while checking for adherence to all accessibility guidelines. Moreover, human evaluators are prone to making mistakes, and it is challenging to ensure comprehensive coverage of all features and various usage scenarios. While engaging individuals with disabilities in accessibility testing is valuable, it has similar limitations and may not always be possible due to a lack of access to individuals with disabilities. As a result, many developers opt for automatic testing to evaluate accessibility [15, 48].

However, automatic testing for accessibility has limitations. One limitation is incomplete coverage [48], as these tools may not address all accessibility guidelines. In addition, these tools have different degrees of accuracy and tend to produce inconsistent reports [26], requiring manual checks to verify the testing results. Furthermore, there is a lack of systematic evaluation methods for assessing the automatic testing tool's effectiveness. Existing efforts on evaluation of web accessibility testing tools, such as the Accessibility Tools Audit [5, 22], utilize manually constructed benchmarks that typically consist of overly simplistic test cases, often containing only a single or a few HTML tags. Such benchmarks do not capture the intricacy and diversity of real-world web applications, leading to a significant overestimation of the tools' abilities to identify accessibility issues across complex and dynamic web environments. This poses a challenge for software developers in selecting the most suitable tool for their specific needs. A systematic evaluation of the tools would not only benefit developers in making informed decisions but also highlight the shortcomings of the tools and provide

ISSTA '24, September 16–20, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652113

insights on how to improve them. In this paper, we aim to fill this gap.

We leverage mutation analysis to create an accessibility mutation framework, called Ma11y, aimed at evaluating web accessibility testing tools. Our mutation operators are crafted based on the Web Content Accessibility Guidelines (WCAG) version 2.1, with a specific focus on the success criteria and their corresponding failures [53]. These success criteria set the standards for accessible web content, and the associated failures are designed to reliably indicate non-compliance with these criteria. By mapping our 25 accessibility mutation operators to these failures, we guarantee that the injected accessibility issues are not only realistic but also accurately reflect the types of deficiencies that are likely to be encountered in actual web environments. Notably, Ma11y injects these operators into the final HTML DOM that loads in the browser. This design eliminates the need for the website's source code [62] while enabling the framework to seamlessly adapt to the dynamic nature of modern websites (dynamically added or removed elements and attributes within the program code), ensuring accurate generation of mutants. Additionally, our framework incorporates a fully automatic oracle for assessing a testing tool's ability to kill mutants (i.e., detect the presence of inaccessibility defects) by analyzing its output.

We investigate several key research questions to assess the effectiveness of our proposed accessibility mutation framework. Firstly, we aim to evaluate the quality of the 25 defined accessibility mutation operators and their contribution to each accessibility principle. Secondly, we assess the framework's ability to generate non-equivalent mutants and the accuracy of our oracle in successfully detecting mutants that are killed. Thirdly, we compare the performance of multiple web accessibility testing tools in their ability to detect various accessibility defects, and identify their strengths and weaknesses. Additionally, we explore potential approaches to improve the accuracy of these tools in identifying and addressing accessibility defects. These research questions aim to help us gain insights into the effectiveness of our approach and existing accessibility testing tools.

We evaluated Ma11y on 24 websites across different categories, along with 6 web accessibility testing tools. The results revealed the high applicability of the designed operators across diverse websites, while also demonstrating Ma11y's effectiveness in mitigating the generation of equivalent mutants. Additionally, the findings reveal that each tool exhibits distinct strengths and weaknesses. However, as a collective observation, these tools prove ineffective in detecting nearly 50% of the injected bugs, highlighting their limitations.

Overall, this article contributes the following:

- A set of 25 accessibility mutation operators derived from WCAG 2.1 failures.
- The first open-source, publicly available mutation analysis framework for web accessibility [45].
- Experimental results demonstrating the efficacy of the framework and the designed operators.
- An evaluation of existing tools, providing insights into their capability to identify real-world accessibility issues.

The remainder of this paper is organized as follows. Section 2 provides an overview of our mutation analysis framework, followed by the details of the fault model used in the design of our mutants in Section 3, and the implementation details in Section 4. Section 5 presents our evaluation of the framework. The paper concludes with an overview of the related work in Section 7, and a summary of our contributions in Section 8.

2 FRAMEWORK OVERVIEW

Ma11y's overview, depicted in Figure 1, consists of three main components. The first component is the Mutant Generator, which applies 25 mutation operators derived from the defect model based on WCAG 2.1 accessibility guidelines [54] to the website under test. We detail the mutation operators and their derivation process in Section 3. The Mutant Generator also includes checks to avoid generating equivalent mutants, discussed in detail in Section 4. The second component is the Tool Runner, which so far integrates the implementation of 6 popular accessibility testing tools through a unified interface. Once mutants are generated, the Tool Runner executes the accessibility testing tools on them. The final component is the Oracle, which compares the reports generated by each tool for both the original website and its corresponding mutated websites. By analyzing these reports, the oracle determines whether the accessibility issues were successfully identified by the tool. The framework generates a comprehensive report showcasing each tool's performance in detecting accessibility bugs, along with valuable insights and a mutation score assessment.

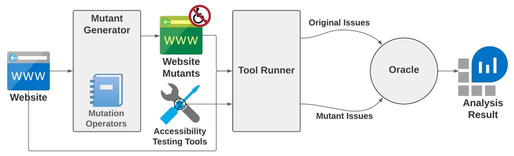

Figure 1: Mutation Testing Framework Architecture

3 MUTATION OPERATORS

In this section, we outline our approach for designing our defect model and subsequently deriving the mutation operators from them. Additionally, we provide detailed explanations of several mutation operators and delve into their implementation specifics in the subsequent sections.

3.1 Defect Model and Derivation of Operators

We started by analyzing the web accessibility guidelines that serve as a resource for making websites accessible to individuals with disabilities. Our goal was to identify violations of these guidelines, which represent accessibility bugs. We focused on the WCAG 2.1 guideline [53] among various available accessibility guidelines [40, 47, 54, 58, 61] due to its comprehensiveness, and its clear, testable success criteria. These criteria are crucial for evaluating web accessibility comprehensively, and each criterion is linked to detailed descriptions of failures. The failures illustrate how specific coding errors can lead to success criteria violations, often accompanied by practical examples. Such documentation of failures makes WCAG 2.1 an invaluable resource for identifying common web accessibility issues leading to its extensive adoption in prior work [32, 43, 55].

Table 1: List of proposed accessibility mutation operators

Principle	Guideline	Id	Name	Scope	Det.	Failure
Perceivable	Text Alternatives/Time-based Media	RID	**R**eplace **I**mage with **D**iv	T	Syn	F3
		CIA	**C**hange **I**mage **A**lt Text	A	Sem	F30
		RIA	**R**emove **I**mage **A**lt Text	A	Syn	F65
	Adaptable	RHP	**R**eplace **H**eader with **P**aragraph	T	Both	F2
		ASB	**A**dd **S**pace **B**etween Characters	C	Sem	F32
		RAS	**R**eplace **A**nchor Text with **S**pan	T	Syn	F42
		RHD	**R**eplace Table **H**eaders with Table **D**ata	T	Syn	F91
	Distinguishable	CFC	**C**hange **F**oreground **C**olor	S	Syn	F24
		RAD	**R**emove **A**nchor Text **D**ecorations	S	Both	F73
		CIF	**C**hange **I**nput **F**ont Size	S	Syn	F80
		CPF	**C**hange **P**aragraph **F**ont Size	S	Syn	F94
		ROA	**R**emove **O**utline from **A**nchor Text	S	Syn	F78
Operable	Enough Time	MTB	**M**ake **T**ext **B**link	S	Syn	F4
	Navigable	CPT	**C**hange **P**age **T**itle	C	Sem	F25
		CTO	**C**hange **T**ab Index **O**rder	A	Both	F44
		RAI	**R**emove **A**nchor Text **I**dentifier	S	Syn	F89
	Keyboard Accessible	CDP	**C**hange **D**evice-Specific **P**ointer	E	Syn	F54
	Input Modalities	CBL	**C**hange **B**utton **L**abel	A	Sem	F96
Understandable	Predictable	RFA	**R**emove **F**ocus from **A**nchor Text	S,E	Syn	F55
		CCC	**C**hange **C**ontext on **C**lick	E	Syn	F22
		CCB	**C**hange **C**ontext on **B**lur	E	Both	F9
		CCI	**C**hange **C**ontext on **I**nput	E	Both	F36
		CCS	**C**hange **C**ontext on **S**election	E	Both	F37
Robust	Compatible	RIN	**R**emove **I**nput **N**ames	T,A	Syn	F68
		MDI	**M**ake **D**uplicate **I**ds	A	Syn	F77

WCAG provides a set of 13 guidelines aimed at improving the accessibility of web content. The guidelines are organized into 4 principles: Perceivable, Operable, Understandable, and Robust [53]. Each one of these principles signifies an essential aspect of web accessibility and ensures that all users can discern and use the website without facing any accessibility issues. Each guideline includes testable success criteria that determine conformance to WCAG. Meeting the success criteria is necessary to achieve WCAG compliance. Significantly, the failures associated with each success criterion provide insightful descriptions that result in non-conformance. These failures are instrumental in our development of mutation operators, as they offer a direct link to real-world scenarios where web accessibility is compromised.

We utilized the failures to construct a defect model. We selected 25 failures from 13 WCAG 2.1 guidelines. We ensured that the selected failures covered different locations in an HTML DOM, referred to as “scope”. The scope of a failure can be categorized as Attribute, Event handler, Tree (Element Tag), Style, or Content, as defined in [62]. Some of these failures are syntactic, for example, issues with the HTML tags and their attributes. Some are semantic, requiring an understanding of the page’s content; for example: the Change Image Alt Text (CIA) operator changes the alternative text of an image to an unrelated text. And finally, some are a combination of both semantic and syntactic. One of our selection criteria entailed ensuring that selected failures do not involve the JavaScript logic of a website. This criterion has been set because the accessibility testing tools we used for evaluation analyze a web page after JavaScript is loaded. Therefore, we focused only on accessibility issues that arise after JavaScript loading. Failures that did not meet the above-mentioned criteria were excluded from the list. It is worth noting that three WCAG guidelines, named “Seizures and Physical Reactions”, “Readable”, and “Input Assistance”, do not have defined failures, and thus, we excluded them from the list.

We derived the mutation operators by analyzing the selected failures. To illustrate this procedure, let us consider failure F3, corresponding to success criterion *“SC 1.1.1: Non-text Content”* [51]. F3 occurs when developers use CSS to include images instead of using the HTML `<img>` tag. This creates an accessibility issue due to the absence of “alternative text” which is necessary for proper functioning of screen readers that are used by blind users. Our analysis of this failure led to the creation of a mutation operator called RID (Replace Image with Div). The RID operator replaces an `<img>` tag with a `<div>` tag containing the background-image CSS property. This mutation operator does not affect the image’s visibility or functionality, but introduces an accessibility issue by using CSS instead of the appropriate HTML tag.

In total, we designed 25 operators. The complete list of operators is provided in Table 1, where the first two columns indicate the principle and guideline violated by each operator. Columns 3 and 4 display the identifier and name of the operators, while the fifth column denotes the scope of the operator in the HTML DOM, i.e., the location where it applies. The sixth column specifies the detection analysis needed for this operator, which can be syntactic, semantic, or both. Syntactic operators solely alter the syntax without affecting the content, while semantic operators modify the content (e.g., replace an informative alt text for an image with random text). Some operators possess both semantic and syntactic properties. The last column provides the WCAG failure number from which the operator is derived. Due to space constraints, we will discuss a subset of these operators in detail in the following subsections. More information about all operators can be found on the companion website [34, 45].

3.2 Mutation Operators for Perceivability

Perceivability, as the first accessibility principle in WCAG, underscores the need for information and user interface components to be presented in a perceptible manner. This principle encompasses four guidelines: Text Alternatives, Time-based Media, Adaptable, and Distinguishable [53]. In this section, we focus on mutation operators derived from failures violating these guidelines, thereby compromising the perceptibility of web content.

For the Text Alternatives and Time-based Media guidelines, which are concerned with the presence of informative alternative text for non-text content, our mutation operators intentionally modify the alternative text or remove it entirely. For instance, the operator named **C**hange **I**mage **A**lt Text (CIA) alters the text alternative of an image to a randomly generated non-informative string.

The Adaptable guideline is concerned with content that can be presented in different ways without losing information. Our mutation operators under this guideline modify elements while maintaining their visual presentation, resulting in elements that visually resemble the originals but lack the intended information and structure for assistive technologies. For example, the Replace Anchor Text with Span (RAS) operator shown in Figure 2 replaces a hypertext link `<a>` tag with a `<span>` tag, replicating all the functionality and styles of the original `<a>` tag but omitting essential ARIA attributes. These attributes provide additional information to assistive technologies for better navigation and interaction [14]. As a result, the new `<span>` element remains unrecognized as a link by screen readers.

```
<a href="/what-we-offer/secure-2.aspx">
    Learn more
</a>
```

(a) Original Element

```
<span onclick="window.location.href='/what-we-offer/
     secure-2.aspx';" class="a-decoration"
>
    Learn more
</span>
...
<style>
.a-decoration {
    text-decoration: underline; !important;
    cursor: pointer; !important;
    color: blue; !important;
}
</style>
```

(b) Mutated Element

Figure 2: Example of Adaptability Mutation Operators

The Distinguishable guidelines underscore the importance of making elements on a page distinguishable through different foreground and background colors, font sizes, or text decorations. The mutation operators designed for this guideline aim to modify these style attributes and render elements inaccessible. For instance, Remove Anchor Text Decorations (RAD) removes any text decoration from the `<a>` tag, relying solely on color differences to distinguish the link. However, such reliance on color alone may lead to failures for individuals who cannot perceive color differences.

3.3 Mutation Operators for Operability

Operability, a foundational principle of web accessibility, aims to ensure that user interface components and navigation are easy to operate. This principle is guided by five crucial guidelines from WCAG: Keyboard Accessibility, Enough Time, Navigation, Seizures and Physical Reactions, and Input Modalities [53]. In this section, we present a set of mutation operators designed from failures that violate these guidelines, with the exception of Seizures and Physical Reactions, which lacks assigned failures.

The first guideline, Keyboard Accessibility, ensures that all website functionalities can be accessed and operated via a keyboard. An example of the operator designed to hinder keyboard accessibility is Change Device-Specific Pointer (CDP), which mutates an element with the `onclick` event handler and changes it to `onmousedown`. This mutation replaces the original keyboard-accessible control with a device-specific control, effectively making it unavailable to keyboard users. Consequently, individuals who rely solely on keyboard navigation and interaction may find this control inaccessible.

The Enough Time guideline emphasizes allowing users sufficient time to read and interact with content. The Make Text Blink (MTB) operator, illustrated in Figure 3, continuously blinks a `<span>` text without providing a mechanism to stop the blinking effect. This mutation hampers the user's reading experience, particularly for those with cognitive or visual impairments, by causing distractions and making it difficult to consume content within the provided time.

```
<span aria-hidden="false" class="input--wrap-label" data
     -mutation-id="F4">
```

(a) Original Element

```
<span aria-hidden="false" class="input--wrap-label
     blink_me" data-mutation-id="F4">
...
<style>
    .blink_me {
        animation: blinker 1s linear infinite;
    }
    @keyframes blinker {
        50% {
            opacity: 0;
        }
    }
</style>
```

(b) Mutated Element

Figure 3: Example of Enough Time Mutation Operators

The Navigation guideline stresses the importance of smooth navigation, content discovery, and clear indicators of the user's location within the interface. An operator violating navigation, Change Page Title (CPT), alters the page title to a randomly generated string unrelated to the content, misleading users and disrupting their ability to accurately determine their location within the interface. This modification may lead to confusion and hinder users from navigating effectively through the website especially when attempting to retrace their steps or understand the overall structure of the content.

The Input Modalities guideline encourages diverse input methods beyond the traditional keyboard, such as speech input, to enhance ways in which operations can be performed. The Change Button Label (CBL) operator, aiming to violate this guideline, alters the value of the `aria-label` attribute of a `<button>` element to a text unrelated to the button's visible label. This operator highlights situations where speech input users face difficulties in reliably activating controls due to mismatches between visible labels and accessible names. Addressing such issues would improve the accessibility of web interfaces for diverse user interaction modes.

3.4 Mutation Operators for Understandability

The principle of Understandability emphasizes the importance of users being able to comprehend both the information presented and the functionality of the user interface, without encountering content or operations that exceed their understanding. This principle encompasses three guidelines: Readable, Predictable, and Input Assistance, with only Predictable having assigned failures in WCAG [53].

The Predictable guideline emphasizes the need for consistent behavior on web pages, promoting a predictable and coherent user experience. The mutation operators within this section are designed to introduce unexpected changes to the page context, challenging its predictability. For example, the Change Context on Click (CCC) operator, as illustrated in Figure 4, unexpectedly opens a new window when a user clicks on a `<span>` element without any prior indication or warning. Such behavior can disrupt the user's focus and distract them from their current reading or task, potentially leading to confusion and difficulty in understanding the interface.

```
<span
    class="-img _glyph"
>
    Stack Overflow
</span>
```

(a) Original Element

```
<span
    class="-img _glyph"
    onclick="window.open('https://example.com')"
>
    Stack Overflow
</span>
```

(b) Mutated Element

Figure 4: Example of Predictability Mutation Operators

3.5 Mutation Operators for Robustness

The Robust principle emphasizes the significance of ensuring web content's reliable interpretation by various user agents, including assistive technologies, and its continued accessibility as technologies evolve over time. This principle is supported by the Compatible guideline [53], which plays a critical role in maximizing compatibility with present and future user agents and assistive technologies. By adhering to this guideline, web developers can ensure that their content remains accessible and functional across diverse platforms and evolving technologies.

One example of an operator that violates the Compatible guideline is Remove Input Names (RIN). This operator removes all labels and names associated with an `<input>` element within a `<form>`. Consequently, users may encounter difficulties in identifying the purpose of the form control, as the essential descriptive information is stripped away. Such mutations can lead to compatibility issues with assistive technologies, hindering users with disabilities from accurately interpreting and interacting with the form.

```
<label for="query">Search the NIH Website</label>
<input autocomplete="off" id="query" name="query" type="
    text">
```

(a) Original Element

```
<input autocomplete="off" id="query" name="query" type="
    text">
```

(b) Mutated Element

Figure 5: Example of Compatibility Mutation Operators

4 APPROACH

In this section, we discuss the implementation details of the three key components of the tool presented in this article.

4.1 Mutant Generator

The first component, called the Mutant Generator, is responsible for implementing and applying the mutation operators to websites. To achieve this, we utilize Puppeteer, a headless browser developed in JavaScript [20]. The decision to adopt a browser-based approach for implementing the operators was driven by several factors. Firstly, websites are dynamic in nature, which means their elements can change or be loaded at runtime, and attributes may be added or removed. To handle such variations effectively and avoid generating equivalent mutants (mutants that are equivalent to the original program), a solution capable of handling runtime changes is essential. Secondly, browsers provide the necessary tools to traverse and inspect a website's DOM, allowing for efficient search and evaluation of candidate elements. This capability enhances the effectiveness and accuracy of the mutation process.

While implementing mutation operators, we use Puppeteer to load the HTML DOM of each website. Once the loading process is complete, we assess each operator's applicability within the website's context. Applicability checking serves several purposes. Firstly, it verifies the presence of the desired element on the page. This ensures that the targeted element exists and can be manipulated. Secondly, it checks if the element is visible and can be accessed through the accessibility API. This is essential to ensure the operator's changes are observable and interactable. Note that assistive technologies (e.g., screen readers) rely on accessibility API for their implementation. Lastly, the applicability check confirms that the targeted element does not already possess the accessibility issue the operator aims to inject. This precautionary step is crucial to avoid generating equivalent mutants that do not introduce new accessibility issues.

To assess element visibility and its ability to be interacted with using accessibility API, Ma11y examines the properties of the elements shown in table 2, along with their respective values.

Table 2: Properties responsible for element's visibility

Property	Value
display	!= none
visibility	!= hidden
opacity	!= 0
role	!= presentation || != none
aria-hidden	!= true

The validation process for applying mutation operators in the framework involves checking the absence of the intended accessibility issue before execution. While straightforward for some operators like RIN, which simply removes labels from inputs, other operators are challenging. For example, validating ROA, which removes outlines from <a> tags during keyboard navigation, requires simulating keyboard navigation using the accessibility API and ensuring the target element receives a visible outline when focused, i.e., ensuring that in fact it can be accessed using accessibility API and that it does not already have an accessibility problem.

Similarly, operators like the CIA, altering the alternative text of images with unrelated text, necessitate determining if the original text was genuinely related to the image. To tackle this, we have implemented a series of heuristics designed to filter out text alternatives that may not be relevant to the image. These heuristics are derived from WCAG failure [49, 50], specifically drawing from failures 25 and 30, where certain alternative texts and page titles were found to be uninformative and unsuitable for use. For instance, we check that the original text does not contain specific patterns or phrases that indicate it is uninformative. Some of these checks are highlighted below:

- Trivial/template alt text: " ", "spacer", "image", "picture"
- Filename extensions in the alt text: "*.png", "*.jpg", "*.jpeg"
- Duplicate alt texts on a page
- Image URL as the only alt text

Before selecting an image for mutating its alt text, we verify that the alt text does not contain the aforementioned patterns. It is worth noting that these checks are primarily aimed at excluding alt texts that might be uninformative. However, they do not affirmatively establish an alt text is informative, as achieving that would require semantic analysis of both the alt text and the image, which falls outside the scope of this paper.

The framework tags the targeted element with a "data-mutation-id" attribute after verifying the operator's applicability on the website. The operator is then applied to the element, and the original DOM and mutated DOM are saved in separate files. Subsequently, the websites, with all operators applied, are hosted on a server to assess the effectiveness of web accessibility testing tools on these mutated versions. Finally, to ensure the quality and accuracy of the created mutants, two authors independently verified each one, complementing the automatic correctness checks of the mutants and their implementation.

4.2 Tool Runner

After generating and hosting the mutants on the server, the *Tool Runner* component takes over and executes the website accessibility testing tools on these web pages. Ma11y is designed such that any web accessibility testing tool can be integrated, so long that the output of the tool can be transformed to a particular JSON format.

The current version of *Tool Runner* integrates six web accessibility testing tools, namely, A11yLite [1], Access Continuum [31], axe [13], IBM Equal Access [24], QualWeb [39], and WAVE [56] as listed in Table 3. The implementation process differs for each tool, as some provide a RESTful API, while others are accessible through packages and programming libraries, necessitating integration through the respective libraries.

Given the diversity in the development and design of these tools, the output report format varies across them. To ensure consistency in the reports and integration with the oracle, we developed a transformation module that unifies the output format of these tool into the previously-mentioned JSON format. This format provides detailed information about the accessibility issues identified by each tool, including the unique problem code (generated by each tool), issue description, and pointers to each problematic HTML element. Listing 1 is showing an example of the JSON format.

```
{
  "code": "event_handler",
  "description": "Device dependent event handler",
  "pointer": "/HTML/BODY/DIV[3]/DIV/SPAN"
}
```

Listing 1: Unified JSON format

The unified JSON format reports accessibility issues using pointers like CSS selectors or XPaths, but since the Oracle component (discussed further below) needs to match problems detected by the tools to the mutated elements, access to the actual elements is required. The *Tool Runner* component uses Puppeteer once again to search the DOM based on the pointers provided by the tools. By mapping each pointer to its corresponding element in the DOM and verifying the presence of the "data-mutation-id" property, the *Tool Runner* collects all accessibility issues associated with the mutated elements for subsequent evaluation by the Oracle.

4.3 Oracle

The Oracle plays a crucial role in determining the effectiveness of the web accessibility testing tools by assessing their ability to identify (kill) the generated mutants. Upon receiving the unified tool reports for both the original and mutant websites from the Tool Runner, the Oracle compares the two lists of accessibility errors reported for these web pages. If the reported list for the mutant does not contain any new errors, it indicates that the tool failed to detect the mutant. Conversely, if a new error is present in the mutant list and corresponds to an accessibility bug that was injected into the website, the Oracle concludes that the tool successfully identified the problem, i.e., killed the mutant. Table 4 shows an example of the outputs examined by the Oracle for ROA operator and the IBM Equal Access tool.

Table 3: List of Tools implemented in Tool Runner

Tool Name	Publisher	License	API Type	Release Date
A11yLite [1]	A11yWatch LLC	Open Source	Library Package	2020-Jan-29
Access Continuum [31]	Level Access	Free Software	Library Package	2017-Oct-13
axe [13]	Deque Systems, Inc.	Open Source	Library Package	2015-Jan-10
IBM Equal Access [24]	IBM Accessibility	Open Source	Library Package	2020-May-18
QualWeb [39]	Faculdade de Ciências da Universidade de Lisboa	Open Source	Library Package	2008-Jan-01
WAVE [56]	WebAIM	Commercial	REST API	2014-Jan-01

Table 4: Example of the original and mutated issues examined by Oracle

Original Element's Issues	[style_focus_visible]
Mutated Element's Issues	[style_focus_visible, script_focus_blur_review]
New Issues	[script_focus_blur_review]
Killed	true

Table 5: Statistics of the selected websites.

Statistics	HTML Elements	# of Mutants
Total	33,866	366
Average	1,411	15
Standard Deviation	1,189	2.84
Maximum	4,662 (shein.com)	20 (nih.gov)
Minimum	73 (google.com)	9 (craigslist.org)

The Oracle component utilizes a mapping list that we developed between errors reported by tools and mutation operators. To that end, we extensively utilized each tool's documentation and conducted multiple executions of each tool on sample web pages with previously injected accessibility bugs. This process allowed us to observe the errors generated by each tool in response to the known accessibility issues present on web pages. Using this mapping list, the Oracle can determine whether a newly reported error on a mutated web page is due to detection of an accessibility bug injected by a specific mutation operator or not. When the reported error identifier matches a mutation operator that was applied to the mutated page, the Oracle counts that as the tool was successfully able to detected the mutant. Conversely, if no match is found, the Oracle concludes that the tool failed to detect the mutant. For example, in Table 4, the new issue detected by IBM Equal Access is `script_focus_blur_review`. Since the Oracle finds a match for this issue in the mapping list we created for the ROA operator, it concludes that this mutant was successfully detected (killed) by the IBM Equal Access tool.

To integrate a new tool, future researchers would need to provide our framework with their API specification and a mapping of error identifiers produced by the tool to mutation operators listed in Table 1.

5 EVALUATION

In this section, we present the experimental evaluation of our mutation operators and framework, as well as the assessment of the current state of web accessibility testing tools in detecting accessibility bugs. We aim to address the following research questions:

RQ1. *Operators*: What is the quality of accessibility mutation operators? What is the contribution of each operator to each accessibility principle?

RQ2. *Mutation Framework*: How effective is Ma11y in generating non-equivalent mutants? What is the accuracy of oracle?

RQ3. *Accessibility Tools*: How effective are web accessibility testing tools in detecting accessibility issues?

RQ4. *Performance*: How long does it take for Ma11y to generate and analyze the mutants?

5.1 Experimental Setup

5.1.1 Subject Websites: In order to select websites for testing Ma11y and web accessibility evaluation tools, we decided to compile a list of the most popular websites. We made use of top visited websites at the time of writing of this paper as tracked by Semrush [12]. Semrush hosts a list of the 20 most popular websites in 33 categories such as Banking, E-Commerce, Fashion, Finance, and others. To ensure comprehensive representation of different websites, we created a selection strategy based on the following criteria. We selected the most popular website worldwide from each of the 33 categories.

We then narrowed the list of websites to those that were in English. We manually examined the downloaded version of the websites to ensure that they closely resemble their online version in appearance and functionality. We did this check since some websites do not retain their styles when downloaded due to various factors, such as the website's server-side configurations, usage of dynamically loaded resources, or content security policies. Two researchers performed this task independently and arrived at a list of websites selected for testing. Out of 33 categories, we were able to select 24 unique, most visited websites. This is because in 4 cases there were the same website for different categories and we excluded 5 websites because they were equipped with detection scripts that identified when the site was accessed from any URL other than its original hosting location. This behavior rendered these websites unsuitable for our examination. The list of these websites are mentioned in the companion website [34, 45]. Table 5 provides a summary of the selected websites in terms of HTML elements.

5.1.2 Web Accessibility Testing Tools: We selected a set of 6 tools from the W3C's list of web accessibility evaluation tools [52]. There are 167 tools on this website. We filtered these tools on the basis of following factors. The tool must support the latest version of WCAG (i.e., WCAG 2.1), be available in English language, and provide an API to automate testing. This gave us a list of 14 tools. This list is inclusive of the open source and only plugin variations of the same tool. Some tools in this list have been deprecated and are no longer available. We decided to use the open source version of the commercial tools that provide an alternative, and commercial version of the tools that do not have any open source alternative. In total, we ended up with 6 unique tools, which are also some of the most widely used tools due to the nature of our selection criteria .

5.2 RQ1. Operators

The evaluation of the accessibility mutation operators focuses on understanding their prevalence, applicability, impact on different accessibility principles, their distribution based on the semantic or syntactic type, and their scope. To comprehensively assess these aspects, we conducted a mutation generation experiment on a diverse set of 24 subject websites described in Section 5.1.

The results of our experiments are summarized in Table 6, showcasing the applicability of the operators on each website. In total Ma11y could create 366 mutants for the 24 websites, with an average of 15 mutants per website. This indicates the high applicability of the defined operators for diverse web contexts.

Figure 6 illustrates the distribution of the created mutants based on the accessibility principles they violate. Notably, approximately half of the generated mutants pertain to the Perceivability principle, while the other half are associated with the principles of Operability, Understandability, and Robustness.

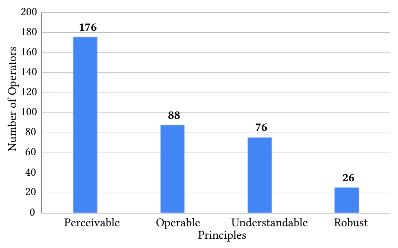

Figure 6: Distribution of the number of mutants generated per WCAG accessibility principles

Our findings reveal that 64% of the operators are associated with syntactic changes, 17% with semantic changes, and the remaining 19% exhibit both syntactic and semantic attributes. This diversity of operators enables a comprehensive evaluation of web accessibility testing tools, allowing for the identification of a wide range of potential accessibility issues across different dimensions.

Furthermore, we examined the distribution of mutants in terms of the corresponding operators' scope, which we defined in section 3. The analysis reveals that the mutants cover a diverse range of scopes, with 18% of the mutants targeting Event Handlers, 31% modifying the Style of elements, and 22% making alterations to HTML Attributes, 16% making changes to HTML Tree and the remaining 13% modify the Content. This comprehensive coverage ensures a thorough assessment of web accessibility testing tools across various scenarios and contexts.

5.3 RQ2. Mutation Framework

Despite our best effort to prevent the generation of equivalent mutants as detailed in Section 4.1, there are scenarios in which it is challenging to guarantee the absence of equivalent mutants. The three mutation operators, namely CIA, CPT, and CBL, fall under this category, where we cannot ensure complete avoidance of equivalent mutants. CIA alters the alternative text of images, CPT modifies the page title to a non-informative title, and CBL changes the button aria-label to a text unrelated to the button text. In Section 4.1, we briefly discussed employing heuristics, such as checking for template or trivial text or filenames, to mitigate the creation of equivalent mutants. To further assess the potential presence of equivalent mutants, a manual analysis was conducted on the three aforementioned operators and their corresponding mutants, as outlined in Table 6. This analysis involved comparing the original and mutated versions of the websites under these operators and identifying cases that led to equivalent mutants.

Remarkably, out of a total of 44 cases in which mutants were generated using these three operators, only one case produced an equivalent mutant. This outcome underscores the effectiveness of Ma11y by minimizing the occurrence of equivalent mutants and enhances its reliability for detecting genuine accessibility issues. Despite the challenges posed by semantic analysis, the framework's ability to avoid equivalent mutants in the majority of cases demonstrates its robustness and potential as a valuable tool in web accessibility assessment.

In Section 4.3, we presented the approach utilized by the oracle to determine whether a mutant is detected (killed). To ensure the accuracy and reliability of our oracle, a comprehensive manual analysis was performed on all 366 mutants, examining each tool's execution and cross-referencing with the mapping list previously created, which linked mutation operators to the error codes generated by each tool. To validate the oracle's classification of killed mutants, two authors independently scrutinized each decision made by the oracle. For mutants marked as "killed", the authors manually checked whether the tool indeed generated a new relevant error, affirming the oracle's correctness. Conversely, if no such error was produced, it indicated a false positive. A similar approach was adopted to identify false negatives.

During the manual analysis, we identified 15 cases of false negatives, resulting in an accuracy of approximately 96%. These false negatives occurred due to the oracle's reliance on error mapping between violated WCAG guidelines and the errors reported by accessibility testing tools. In some cases, the mapping between violated guidelines and the errors generated by the tools was not one-to-one. Notably, for a single injected accessibility issue, certain tools may produce different error codes on different pages. For example, the "Remove Input Names" (RIN) operator triggered varied error codes in the IBM Equal Access tool, such as `input_label_exists` or `input_label_visible`, depending on the specific case. Similarly, Access Continuum generated error codes 338 and 2440 for the same injected bug in different mutants. These inconsistencies led to decreased accuracy of our oracle. However, this is not a shortcoming of our oracle, but rather a shortcoming of existing tools, because in certain cases the existing tools do not produce a consistent output. In essence, this is similar to the problem of running flaky tests [33] for the conventional mutation testing. Due to the unpredictability of flaky tests, mutants may or may not be killed, which could affect the mutation kill score. Similarly, in certain situations existing accessibility testing tools are flaky, preventing our oracle from properly tracking their mutation kill score.

Ma11y exhibits strong capabilities in mitigating the generation of equivalent mutants and the accuracy of our oracle in identifying killed mutants provides a solid foundation for evaluating the performance of web accessibility testing tools.

5.4 RQ3. Accessibility Tools

In this evaluation, we investigate the performance of various web accessibility testing tools in detecting accessibility bugs within the 366 mutants generated Ma11y. The results are presented in Table 6, which displays the total number of mutants killed by each tool for each mutation operator.

The table presents an overview of the performance of various web accessibility testing tools in detecting accessibility bugs across the 366 mutants. On average, the tools detected 92 mutants, with IBM Equal Access achieving the highest number of 190 mutants killed and A11yWatch achieving the lowest with 52 mutants killed. Despite these figures, the mutation scores are disappointingly low (≈50%). This indicates that these accessibility testing tools cannot detect nearly 50% of the accessibility issues present in the websites. Surprisingly, even WAVE, a widely used commercial tool, exhibits a notably low mutation score of 25%.

Further analysis of the results revealed that two operators, MDI (Make Duplicate IDs) and RIA (Remove Image Alt Text), were the most easily detected by the tools. Almost all of the 41 mutants generated for these operators were successfully detected. These issues can often be identified through simple DOM inspections, making their detection relatively straightforward for the tools.

However, there are six operators for which none of the tools were able to detect a single mutant. These problematic operators include CPT (Change Page Title), CIA (Change Image Alt Text), CCS (Change Context on Selection), MTB (Make Text Blink), CTO (Change Tab Index Order), CCB (Change Context on Blur), and CPF (Change Paragraph Font Size). Among these, five operators are of the semantic or both syntactic and semantic types, highlighting the tools' limitations in semantic analysis of the website's content and their consequent failure to detect semantic accessibility bugs. Notably, the MTB operator, which uses animation to add dynamicity to text, cannot be detected by any of the tools, exposing their limitations in identifying dynamic accessibility bugs. Additionally, the CPF operator, a syntactic operator that changes the font-size of the paragraph, surprisingly eludes detection by all the tools.

Figure 7 shows the detection distribution based on the syntactic/semantic categorization of bugs. Out of 130 mutants categorized as semantic or both, the tools successfully detected 35 of them, amounting to only 26% detection rate. In contrast, the tools demonstrated a relatively better performance in detecting syntactic bugs, successfully identifying around 54% of them.

We also analyzed the results based on accessibility principles (Perceivability, Understandability, Robustness, and Operability). The tools exhibit satisfactory performance in detecting the operators violating the Robustness principles. However, their performance significantly drops when it comes to detecting operators violating the Operability principle, with WAVE exhibiting a detection rate of 0%. IBM Equal Access performed well in detecting mutants violating Perceivability and Understandability principles.

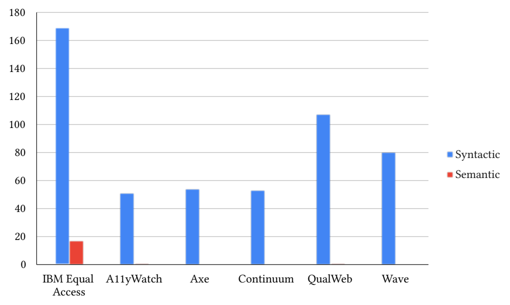

Figure 7: Number of Mutants Killed by tools

A11yWatch showed effectiveness in detecting Operability violations, while Continuum excelled in identifying Robustness issues.

Based on these observations, it is evident that each tool excels in detecting specific types of accessibility issues. As a result, we hypothesize that combining these tools may yield a more effective approach to detection. To assess this, we analyzed the results obtained from the combination of the tools, revealing a mutation score of 60%, surpassing the performance of the best individual tool by 8%. However, despite this improvement, the overall mutation score remains relatively low, with 40% of the accessibility issues remaining undetected. This indicates that while the combination shows promise, further enhancements are necessary to achieve more comprehensive and accurate accessibility testing.

Our study reveals a notable inconsistency in the effectiveness of tools in detecting accessibility bugs across various websites. In certain cases, a tool may identify a specific accessibility issue on one website but fail to do so on another. These inconsistencies, not previously highlighted in benchmark-based evaluations, challenge prior findings (e.g., [22]). For instance, while examining the CBL operator, we noticed when altering the button aria-label to unrelated text, IBM Equal Access and QualWeb could identify this issue on `capitalone.com` but not on `yahoo.com`. This contrasts with [21], where both tools were credited with detecting the bug. These findings highlight the inadequacy of relying solely on basic benchmark tests for a comprehensive evaluation of tools' ability to detect accessibility issues, since the design of a web page, how content is loaded, and how adaptive UI is rendered may impact the effectiveness of tools in detecting accessibility issues. Real-world web environments are more complex than benchmarks suggest. Our study, featuring intricate and realistic examples, offers a more accurate portrayal of tools' effectiveness in practical scenarios and opens up the opportunity for further investigation.

5.5 RQ4. Performance

To assess the performance of Ma11y, we measured the time required for generating mutants and analyzing the results produced by each tool to determine the detection status of the mutants. The experiments were conducted on a computer with a 1.4 GHz Intel Core i7 processor and 8 GB DDR3 RAM.

For evaluating the performance of the mutant generator component, we measured the time needed for analyzing the code, checking the applicability of the operator, and applying the operator. In Table

Table 6: Summary of web accessibility testing tools

Operator ID	RHP	CCC	CFC	CPT	RID	CIA	ASB	CCI	CCS	MTB	RAS	CTO	CDP	RFA	RIA	RIN	RAD	MDI	ROA	CIF	RAI	CCB	RHD	CPF	CBL	Total	Mutation Score
Targeted Accessibility Principle	P	U	P	O	P	P	P	U	U	O	P	O	O	U	P	R	U	R	P	P	O	U	P	P	O	-	-
Scope	T	E	S	C	T	A	C	E	E	S	T	A	E	E	A	T	S	A	S	S	S	E	T	S	A	-	-
Syntactic/Semantic	Bo	Sy	Sy	Se	Sy	Se	Se	Bo	Bo	Sy	Sy	B	Sy	Sy	Sy	Sy	Bo	Sy	Sy	Sy	Sy	Bo	Sy	Sy	Se	-	-
Total	17	23	19	24	17	17	20	7	1	24	24	20	5	24	17	2	15	24	20	5	12	6	1	19	3	366	-
Equivalent	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	-
IBM Equal Access Killed	0	19	11	0	17	0	15	4	0	0	23	0	0	24	17	2	0	24	20	0	11	0	1	0	2	190	**52**
QualWeb Killed	0	20	11	0	0	0	0	0	0	0	0	0	0	22	13	2	0	24	0	5	10	0	0	0	1	108	**30**
Wave Killed	10	23	16	0	0	0	0	0	0	0	0	0	0	0	16	1	0	24	0	0	0	0	0	0	0	90	**25**
Continuum Killed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	1	0	24	0	0	11	0	1	0	0	53	**14**
A11yWatch Killed	0	0	0	0	0	0	0	0	0	0	0	0	5	0	11	2	0	24	0	0	9	0	0	0	1	52	**14**
Axe Killed	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	2	2	24	0	0	12	0	0	0	0	56	**15**
Unique Killed	10	23	16	0	17	0	15	4	0	0	23	0	5	24	17	2	2	24	20	5	12	0	1	0	2	220	**60**

Table 7: Performance Metrics of Mutation Framework. (Total columns represent the sum of the average time per mutant across all projects)

Statistics	**Mutant Generation Time (s)**		**Mutant Analysis Time (s)**		**Overall Time (s)**	
	Total	**Per Mutant**	**Total**	**Per Mutant**	**Total**	**Per Mutant**
Total	669.56	3.25	2,060.99	136.82	2,730.56	140.08
Average	27.90	0.14	85.87	5.70	113.77	5.84
Standard Deviation	24.19	0.25	200.80	11.98	219.14	12.09
Maximum	106.51 craiglist.com	1.17 google.com	977.79 craiglist.com	57.52 craiglist.com	1,084.30 craiglist.com	58.06 craiglist.com
Minimum	5.86 live.com	0.02 nih.gov*	0.73 stackoverflow.com	0.10 discord.com*	7.56 live.com	0.12 discord.com

* More than one website had the same maximum/minimum time.

7, we report the summary statistics due to space constraints. Complete statistics for all 24 websites are available at [45]. As shown in Table 7, the average time taken to create each mutant by the generator was 0.14 seconds, with an average time of 27.90 seconds to create all mutants for a subject website. This indicates that the mutant generation process is swift and efficient.

Regarding the tool runner component, the analysis time heavily depends on the total number of errors and warnings generated by the web accessibility tool, which, in turn, is influenced by the size of the source code and the number of components rendered in the HTML DOM. On average, the analysis time for each mutant remains below 6 seconds, totaling an average of 86 seconds for all mutants on a website. Furthermore, the average running time of the tool for each mutant is 6 seconds, while the average analysis time for each website is 144 seconds. These results demonstrate the practicality and feasibility of using our tool in real-world scenarios.

6 THREATS TO VALIDITY

We have strived to eliminate bias and the effects of random noise in our study. However, it is possible that our mitigation strategies may not have been effective.

Our fault model is based on WCAG 2.1 guidelines, which, while comprehensive, are subject to updates and changes over time. It is important to note that the WCAG 2.1 guidelines may not cover all possible accessibility issues as it is not exhaustive. Possible errors in the tools used may affect our findings. To minimize this, we leverage tools that have been extensively used and validated in the literature. Moreover, we have extensively tested our implementation to ensure that there are no defects in the implementation of our tool.

To minimize the potential threats to validity related to selection of websites, we aimed to ensure maximum coverage of different types of websites. We selected the most popular websites from a wide range of domains allowing us to increase the generalizability of our findings.

7 RELATED WORK

Web accessibility testing/evaluation: Web accessibility guidelines have been instrumental in aiding numerous studies and tools to evaluate both web pages [1, 13, 24, 31, 39, 56] and mobile apps [7–9, 16, 23, 28, 35, 36, 42]. Tools such as Siteimprove [44], Google Lighthouse [19], and Tenon [27] were designed to assess web accessibility based on these guidelines. AI-based approaches like AccessiBi [4], Equally AI [17], and Applitools [10] have emerged for web page accessibility evaluation. Studies have focused on detecting specific accessibility issues, such as improving HTML tables' accessibility for individuals with visual impairments [55] or enhancing data visualizations with interactive JavaScript plugins [43]. Others have utilized assistive technologies to identify dynamic web accessibility problems, such as detecting and localizing keyboard accessibility failures in web applications using modeled keyboard navigation flow [11]. Despite having a plethora of accessibility testing tools, a critical gap exists in the form of a unified framework to assess their effectiveness. Absence of such framework hinders effective comparison and identification of the best tools available. Our work addresses this gap by providing a framework that can be integrated with new accessibility testing tools, enabling researchers to evaluate their effectiveness accurately.

Web accessibility testing tools: Prior studies have compared web accessibility testing tools. In these studies, manually constructed HTML pages with intentional accessibility errors [46] or simplified web pages, often containing only a single or a few HTML tags [5, 22], are employed to assess the detection abilities of the compared tools. Other research has focused on comparing tool features, usability, and web page evaluation [18, 30], or on evaluating tool performance on real-world websites with known accessibility issues [6, 26, 29, 41, 48]. Additionally, there is research that manually created HTML pages with accessibility bugs based on web accessibility guidelines to assess web development frameworks like Angular for accessibility warnings [32]. However, these methods are not automated and require significant manual effort to assess new accessibility testing tools. In contrast, our work presents a novel and systematic approach by automatically injecting various accessibility bugs into existing web pages. By utilizing mutation analysis, we rigorously evaluated the effectiveness of the tested tools in detecting bugs, addressing the limitations of benchmark-based and manual approaches. Our framework's ability to simulate real-world scenarios, including the complexity and dynamicity of modern web applications, provides a more accurate and comprehensive assessment of tool performance.

Mutation testing for web: Various studies have delved into web application mutation testing from different perspectives. One notable study by Yandrapally and Mesbah [62] introduced MAEWU, a web GUI mutation framework that evaluates UI test quality by applying mutation operators to the final DOM of the website. Nishiura et al. [38] focused on client-side JavaScript mutation operators specifically designed for DOM manipulation in web applications. Another research [37] developed mutation operators tailored to JavaScript in web applications, along with generic JavaScript operators. While these studies have made significant contributions to web application mutation testing, our work differs in its primary focus on defining accessibility-aware mutation operators. We aim to assess the capabilities of web accessibility testing tools in detecting accessibility-related bugs introduced by these operators, which no previous research has done.

8 CONCLUSION

This research introduced a novel accessibility mutation framework, called Ma11y, with 25 operators derived from WCAG 2.1, covering a wide range of accessibility guidelines and principles across different scopes and types of issues. The integration and evaluation of 6 popular web accessibility testing tools using Ma11y highlighted their strengths and weaknesses. The study revealed that on average, the tested tools detected less than 50% of the injected accessibility issues, underscoring the need for further improvement in web accessibility testing tools. We also found that the tools mostly fail to eliminate mutants that modify the semantic content of page elements or operators related to the dynamics of web pages. Therefore, one future direction for accessibility tool designers could be to focus on considering the dynamic nature of web pages and incorporate semantic and contextual analysis into their tools.

Additionally, a key finding from our study was the inconsistency in the tools' ability to detect accessibility issues across different websites. This highlights that a tool's effectiveness can vary significantly depending on the specific web context, challenging the reliability of benchmark-based evaluations.

Our framework is designed in ways that can support integration of various accessibility testing tools with minimal effort. We demonstrated this through the integration of 6 popular web accessibility testing tools. We believe its adoption by future researchers as a common platform for comparing and contrasting web accessibility testing tools can significantly foster progress and research in this area. For our future work, we aim to expand the diversity of mutation operators supported by Ma11y, which would increase its coverage of accessibility issues that may occur in practice. Specifically, Ma11y currently does not include operators that can mutate the JavaScript logic of a website. We believe expanding the framework with such operators can further enhance its utility.

The research artifacts for this study are available publicly at the companion website [34, 45].

ACKNOWLEDGMENTS

This work was supported in part by award numbers 2211790 from the National Science Foundation. We would like to thank the anonymous reviewers of this paper for their detailed feedback, which helped us improve the work.

REFERENCES

[1] a11yWatch. 2022. a11yWatch. https://a11ywatch.com/.
[2] Iyad Abu Doush and Ikdam Alhami. 2018. Evaluating the Accessibility of Computer Laboratories, Libraries, and Websites in Jordanian Universities and Colleges. *International Journal of Information Systems and Social Change* 9 (04 2018), 44–60. https://doi.org/10.4018/IJISSC.2018040104
[3] Iyad Abu-Doush, Ashraf Bany-Mohammed, Emad Ali, and Mohammed Azmi Al-Betar. 2013. Towards a more accessible e-government in Jordan: an evaluation study of visually impaired users and Web developers. *Behaviour & Information Technology* 32, 3 (2013), 273–293. https://doi.org/10.1080/0144929X.2011.630416 arXiv:https://doi.org/10.1080/0144929X.2011.630416
[4] AccessiBe. 2023. AccessiBe. https://accessibe.com/.
[5] Alphagov. 2023. Acessibility Tools Audit. https://alphagov.github.io/accessibility-tool-audit/.
[6] Abdullah Alsaeedi. 2020. Comparing Web Accessibility Evaluation Tools and Evaluating the Accessibility of Webpages: Proposed Frameworks. *Information* 11, 1 (2020). https://doi.org/10.3390/info11010040
[7] Android. 2023. Accessibility Scanner - Apps on Google Play. https://play.google.com/store/apps/details?id=com.google.android.apps.accessibility.auditor&hl=en_US. Last Accessed: December 7, 2023.
[8] Android. 2023. Espresso : Android Developers. https://developer.android.com/training/testing/espresso. Last Accessed: December 7, 2023.
[9] Android. 2023. Improve your code with lint checks. https://developer.android.com/studio/write/lint?hl=en. Last Accessed: December 7, 2023.
[10] Applitools. 2023. Applitools. https://applitools.com/.
[11] Paul T. Chiou, Ali S. Alotaibi, and William G. J. Halfond. 2021. Detecting and Localizing Keyboard Accessibility Failures in Web Applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 855–867. https://doi.org/10.1145/3468264.3468581
[12] Semrush Company. 2023. Semrush, Online Marketing can be easy. https://www.semrush.com/.
[13] Deque Systems. 2023. axe-core. https://github.com/dequelabs/axe-core.
[14] MDN Web Docs. 2023. ARIA. https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA.
[15] Iyad Abu Doush, Khaled Sultan, Mohammed Azmi Al-Betar, and et al. 2023. Web accessibility automatic evaluation tools: to what extent can they be automated?

CCF Transactions on Pervasive Computing and Interaction (2023). https://doi.org/10.1007/s42486-023-00127-8

[16] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*. ICST, Västerås, Sweden, 116–126.

[17] Euqally AI. 2023. Equally AI. https://equally.ai/.

[18] Tânia Frazão and Carlos Duarte. 2020. Comparing Accessibility Evaluation Plug-Ins. In *Proceedings of the 17th International Web for All Conference* (Taipei, Taiwan) *(W4A '20)*. Association for Computing Machinery, New York, NY, USA, Article 20, 11 pages. https://doi.org/10.1145/3371300.3383346

[19] Google. 2023. Google Lighthouse Accessibility. https://developer.chrome.com/docs/lighthouse/accessibility/.

[20] Google Chrome Developers. 2023. pptr.dev - Puppeteer Documentation. https://pptr.dev/.

[21] ACT Rules Community Group. 2023. ACT-Rules - Visible label is part of accessible name. https://act-rules.github.io/rules/2ee8b8.

[22] ACT Rules Community Group. 2023. ACT-Rules Community. https://act-rules.github.io/.

[23] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM New York, NY, USA, Bretton Woods, New Hampshire, USA, 204–217.

[24] IBM Accessibility. 2023. Equal Access. https://github.com/IBMa/equal-access.

[25] Yavuz Inal, Kerem Rızvanoğlu, and Yeliz Yeşilada. 2019. Web accessibility in Turkey: awareness, understanding and practices of user experience professionals. *Universal Access in the Information Society* 18, 2 (2019), 387–398. https://doi.org/10.1007/s10209-018-0639-3

[26] Rita Ismailova and Yavuz Inal. 2022. Comparison of Online Accessibility Evaluation Tools: An Analysis of Tool Effectiveness. *IEEE Access* 10 (2022), 58233–58239. https://doi.org/10.1109/ACCESS.2022.3179375

[27] Karl Groves. 2023. Tenon. https://tenon.io/.

[28] KIF. 2023. Keep It Functional - An iOS Functional Testing Framework. https://github.com/kif-framework/KIF. Last Accessed: December 7, 2023.

[29] Shashank Kumar, Jeevithashree Divya Venkatesh, and Pradipta Biswas. 2021. Comparing ten WCAG tools for accessibility evaluation of websites. *Technology and Disability* 33 (04 2021), 1–23. https://doi.org/10.3233/TAD-210329

[30] Shashank Kumar, JeevithaShree DV, and Pradipta Biswas. 2020. Accessibility evaluation of websites using WCAG tools and Cambridge Simulator. *arXiv preprint* (2020). arXiv:2009.06526 [cs.HC] https://doi.org/10.48550/arXiv.2009.06526

[31] Level Access. 2023. Access Continuum. https://www.levelaccess.com/access-continuum/.

[32] Michael Longley and Yasmine N. Elglaly. 2021. Accessibility Support in Web Frameworks *(ASSETS '21)*. Association for Computing Machinery, New York, NY, USA, Article 47, 4 pages. https://doi.org/10.1145/3441852.3476531

[33] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. https://doi.org/10.1145/2635868.2635920

[34] Ma11y. 2023. Ma11y-tool-analyzer-Artifacts. https://shorturl.at/cgtC2.

[35] Forough Mehralian, Navid Salehnamadi, Syed Fatiul Huq, and Sam Malek. 2022. Too Much Accessibility is Harmful! Automated Detection and Analysis of Overly Accessible Elements in Mobile Apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, ACM New York, NY, USA, Rochester, Michigan, USA, 13 pages.

[36] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 107–118.

[37] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2015. Guided Mutation Testing for JavaScript Web Applications. *IEEE Transactions on Software Engineering* 41, 5 (2015), 429–444. https://doi.org/10.1109/TSE.2014.2371458

[38] Kazuki Nishiura, Yuta Maezawa, Hironori Washizaki, and Shinichi Honiden. 2013. Mutation Analysis for JavaScript Web Applications Testing. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE* 2013.

[39] University of Lisbon. 2023. QualWeb Evaluator. http://qualweb.di.fc.ul.pt/evaluator/.

[40] Penn State University. 2023. Penn State Accessibility. https://accessibility.psu.edu/.

[41] Marian Pădure and Costin Pribeanu. 2019. Exploring the differences between five accessibility evaluation tools. In *Proceedings of RoCHI 2019*.

[42] Navid Salehnamadi, Forough Mehralian, and Sam Malek. 2022. GroundHog: An Automated Accessibility Crawler for Mobile Apps. In *2022 37th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, ACM New York, NY, USA, Rochester, Michigan, USA, 13 pages.

[43] Ather Sharif, Olivia H. Wang, Alida T. Muongchan, Katharina Reinecke, and Jacob O. Wobbrock. 2022. VoxLens: Making Online Data Visualizations Accessible with an Interactive JavaScript Plug-In. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 478, 19 pages. https://doi.org/10.1145/3491102.3517431

[44] Siteimprovie. 2023. Siteimprove. https://www.siteimprove.com/.

[45] Mahan Tafreshipour. 2023. Ma11y-tool-analyzer-GitHub-Repository. https://github.com/mahantaf/web-a11y-tool-analyzer.

[46] Claudia Timbi-Sisalima, Cecilia I. Martin Amor, Silvia Otón Tortosa, José R. Hilera, and Juan Aguado-Delgado. 2016. Comparative Analysis of Online Web Accessibility Evaluation Tools. In *Information Systems Development: Complexity in Information Systems Development (ISD2016 Proceedings)*, Jerzy Gołuchowski, Malgorzata Pańkowska, Chris Barry, Michael Lang, Henry Linger, and Christoph Schneider (Eds.). University of Economics in Katowice, Katowice, Poland. http://aisel.aisnet.org/isd2014/proceedings2016/CreativitySupport/3

[47] United States Access Board. 2023. Information and Communication Technology (ICT) Standards and Guidelines. https://www.access-board.gov/ict/.

[48] Markel Vigo, Justin Brown, and Vivienne Conway. 2013. Benchmarking Web Accessibility Evaluation Tools: Measuring the Harm of Sole Reliance on Automated Tests. 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 1, 10 pages. https://doi.org/10.1145/2461121.2461124

[49] W3C. 2023. Failuer of success criterion 1.1.1. url-https://www.w3.org/WAI/WCAG21/Techniques/failures/F30.

[50] w3c. 2023. Failure of success criterion 2.4.2. https://www.w3.org/WAI/WCAG21/Techniques/failures/F25.

[51] W3C. 2023. Understanding success criterion 1.1.1. https://www.w3.org/WAI/WCAG21/Understanding/non-text-content.html.

[52] w3c. 2023. Web Accessibility Evaluation Tools List. https://www.w3.org/WAI/ER/tools/.

[53] W3C. 2023. Web Content Accessibility Guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/.

[54] W3C Web Accessibility Initiative. 2023. Web Content Accessibility Guidelines (WCAG) 2.1 Techniques. https://www.w3.org/WAI/WCAG21/Techniques/.

[55] Yanan Wang, Ruobin Wang, Crescentia Jung, and Yea-Seul Kim. 2022. What Makes Web Data Tables Accessible? Insights and a Tool for Rendering Accessible Tables for People with Visual Impairments. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 595, 20 pages. https://doi.org/10.1145/3491102.3517469

[56] WebAIM. 2023. WAVE. https://wave.webaim.org/.

[57] WebAIM. 2023. WebAIM: Million. https://webaim.org/projects/million/.

[58] WebAIM. 2023. WebAIM Web Content Accessibility Guidelines (WCAG). https://webaim.org/standards/wcag/.

[59] WebsiteBuilder.org. 2023. Internet Statistics. https://websitebuilder.org/blog/internet-statistics/.

[60] World Health Organization. 2023. World Health Organization. https://www.who.int/publications/i/item/9789241564182.

[61] Yale University Usability and Accessibility Group. 2023. Web Accessibility. https://usability.yale.edu/web-accessibility.

[62] Rahulkrishna Yandrapally and Ali Mesbah. 2021. Mutation Analysis for Assessing End-to-End Web Tests. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 183–194. https://doi.org/10.1109/ICSME52107.2021.00023

Received 16-DEC-2023; accepted 2024-03-02