UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

AccessBot – Assisted Assessment of Web Accessibility

Tânia Isabel Gomes Frazão Pina Santos

Mestrado em Informática

Dissertação orientada por:
Prof. Doutor Carlos Alberto Pacheco dos Anjos Duarte

2020

Resumo

A Web é um bem essencial para muitas pessoas com ou sem deficiência e deverá ser acessível para todos. Existem diferentes formas de deficiencia. Existem as deficiências visuais, como por exemplo, a cegueira ou daltonismo, as deficiências auditivas como a surdez ou dificuldade na audição, deficiências motoras que podem causar dificuldades em usar um rato ou até deficiências cognitivas que levam a uma pessoa ter maior dificuldade ao executar um ou mais tipos de tarefas mentais. Todas estas deficiências mencionadas deverão ser tidas em conta no desenvolvimento de websites ou aplicações da web, pois têm pré-requisitos de acessibilidade diferentes mas que se complementam em tornar um website acessível.

Para determinar se um website é acessível a todos, este precisa de ser avaliado e testado. O principal objectivo em testar a acessibilidade é verificar se todas as pessoas conseguem usar o website e dar feedback útil para promover modificações na implementação e no no design que melhore a usabilidade do website e a sua acessibilidade.

A avaliação da acessibilidade é baseada nos standards desenvolvidos pela W3C (World Wide Web Consortium). Estes standards consistem numa série de linhas de orientação como as WCAG (Web Content Accessibility Guidelines) que incluem vários critérios de sucesso e técnicas para tornar os diferentes componentes de desenvolvimento Web mais acessíveis; contudo, em certos países, existem também linhas de orientação governamentais específicas que complementam os standards da W3C.

As necessidades relativas à implementação de websites mais acessíveis fez com que, hoje em dia, a monitorização dos websites de entidades públicas em relação à acessibilidade seja obrigatória. No caso de Portugal o sector público deverá corresponder aos requisitos de acessibilidade com a entrada em vigor do Decreto-Lei, nº83/2018. A Agência para a Modernização Administrativa (AMA) é um instituto público responsável por garantir que os websites públicos governamentais e entidades semelhantes, melhorarão o seu grau de acessibilidade ao respeitar as regras de acessibilidade exigidas. Neste sentido, a AMA criou o Observatório Português da Acessibilidade dos Sítios Web e das Aplicações Móveis. A implementação deste Observatório de Acessibilidade leva a uma parceria com a Faculdade de Ciências da Universidade de Lisboa (FCUL). Neste sentido, o Access-Bot é um dos projectos sobre acessibilidade em desenvolvimento cujos pré-requisitos e as melhores práticas de usabilidade são definidas em cooperação com a AMA.

A implementação das linhas de orientação na identificação dos problemas de acessi-

bilidade poderão ser feitas de forma automática, usando testes automáticos; contudo, de acordo com estudos realizados estes apenas conseguem encontrar 30% dos problemas. Os problemas restantes deverão ser localizados usando testes manuais o que implica recurso a utilizadores. Estes testes manuais, também chamados de avaliação assistida, necessitarão de mais recursos que a avaliação automática mas têm a vantagem de detectar maior número de problemas de acessibilidade.

O AccessBot é uma aplicação desenvolvida como extensão do browser Chrome que tem por objectivo complementar a avaliação automática de acessibilidade feita pelo motor QualWeb. Este motor é mantido pelo Departamento de Informática da FCUL.

O AccessBot permite avaliar uma página web automaticamente recorrendo ao QualWeb mas complementa-a com avaliação assistida num ambiente de extensão do Chrome. As avaliações assistidas pelo AccessBot podem ser suportadas de diferentes maneiras; o AccessBot identifica todos os elementos da página web que foram alvo de avaliação automática e mostra o resultado final ao utilizador (avaliação automática) podendo este alterar o resultado se o desejar ou apresenta uma lista de passos que guiarão o utilizador a complementar a avaliação automática até chegar a um resultado final (avaliação semi-automática e manual). É de notar que a diferença entre avaliação semi-automática e manual, reside no facto de parte da avaliação semi-automática ter uma componente automática pelo QualWeb mas cuja avaliação precisa também do input do utilizador enquanto a manual é uma avaliação intrínseca e desenvolvida pelo AccessBot sem intervenção de avaliação do QualWeb. Na avaliação manual o utilizador faz a avaliação na íntegra seguindo uma série de passos até chegar ao resultado final.

Outro objectivo do AccessBot é permitir a interpretação inequívoca e implementação dos métodos de teste usando as regras definidas pela ACT Rules (Accessibility Conformance Testing) Community. Esta comunidade providencia orientação aos desenvolvedores em termos de informação e interpretação das linhas de orientação da WCAG. As ACT Rules permitem maior transparência e harmonização quanto aos métodos de teste.

Preliminarmente ao desenvolvimento do AccessBot, foi feito um estudo que consistiu na análise das ferramentas automáticas mais utilizadas actualmente e disponíveis como extensões do Chrome para os desenvolvedores. Este estudo permitiu atingir uma perspectiva geral sobre o uso destas ferramentos e analisá-las em termos de pontos fortes e pontos fracos e concluir possíveis melhoramentos que poderiam ser aplicados na implementação do AccessBot de forma a suprimir necessidades que os desenvolvedores tenham durante o uso destas ferramentas.

As ferramentas que foram analisadas foram aXe Chrome Extension, Tenon Check, Wave Chrome Extension, TotalValidator, ACCESS Assistant Community, Microsoft Accessibility Insights e ARC Toolkit. O estudo consistiu no uso de cada uma das ferramentas na avaliação de dez páginas. Estas foram selecionadas da lista de top websites da Alexa. As avaliações decorreram em ambientes de teste com browsers isolados de possíveis alterações externas.

No geral, as ferramentas foram de fácil instalação e uso mas não livres de falhas. Durante as avaliações, algumas funcionalidades de algumas ferramentas quebraram e foi necessário reiniciar o browser. Os resultados obtidos entre as ferramentas variaram em termos de critérios de sucesso que as ferramentas aplicam nas avaliações, em termos de como classificam os resultados da avaliação de acordo com o seu impacto para com o utilizador, por exemplo, para um mesmo resultado, umas podem atribuir como crítico enquanto outras não e na classificação dos resultados como erro ou alerta. Todas estas variâncias são uma consequência de como as especificações e as heurísticas são implementadas por cada ferramenta.

O estudo analisou o número de erros encontrados por critérios de sucesso pelas diferentes ferramentas. Apesar de haver algumas diferenças entre ferramentas, pois nem todas, usam os mesmos critérios de sucesso, obtiveram-se algumas semelhanças nos resultados concluindo que o critério de sucesso *4.1.1. Parsing*, foi o mais violado e refere-se à maneira como os web browsers e tecnologia assistiva (leitores de ecrã) interpretam um website, pois é importante que diferentes tecnologias possam interpretar o mesmo website sem perda de informação para o utilizador. Problemas neste critério poderão estar relacionados com o uso indevido dos elementos de HTML.

Foi também demonstrado por este estudo que as ferramentas, quando usadas individualmente, têm uma cobertura inadequada dos critérios de sucesso da WCAG. Mesmo que sejam usadas todas as ferramentas para avaliação da uma página web, a cobertura aumenta apenas com uma variação de 10% a 40% do que se só usar uma ferramenta. E tendo em conta esta premissa, a melhor opção é usar mais do que uma ferramenta automática para melhorar a acessibilidade e adesão ao desenvolvimento de websites acessíveis.

Apesar de não serem perfeitas e com limitações estas ferramentas são essenciais para ajudar os utilizadores a avaliar os websites de uma forma automática. Contudo, estas avaliações deverão sempre ser complementadas com procedimentos de testes manuais e os resultados analisados objectivamente. Os resultados do estudo foram incluídos no design do AccessBot permitindo agregar a informação prévia de forma a preencher as falhas das outras extensões do Chrome de avaliação de acessibilidade e corresponder às necessidades do utilizador.

As etapas de planeamento e execução do design, implementação e testes ao utilizador durante o desenvolvimento do AccessBot foram cumpridas dentro do tempo estimado. Apesar de ter havido alguns desafios durante o desenvolvimento, como por exemplo, encontrar as melhores soluções para integração do AccessBot com o QualWeb, a aprendizagem de componentes fundamentais de desenvolvimento de extensões Chrome, entre outras, estes foram ultrapassadas com sucesso.

A lógica de implementação do AccessBot tem dois aspectos fundamentais. Um aspecto é a integração com o QualWeb de forma a que o AccessBot consiga receber e processar os resultados da avaliação automática; o outro aspecto é a implementação dos algoritmos semi-automáticos e manuais, que permitem direcionar o utilizador durante a

avaliação assistida. Além desta implementação base do AccessBot e de forma a melhorar a interacção com o utilizador e usabilidade, foram também desenvolvidas funcionalidades, como por exemplo, a possibilidade de o utilizador escolher que tipo de avaliação pretende fazer, se automática, semi-automática e manual; opções de filtros que escondem resultados não pretendidos para visualização; contadores para guardar diferentes resultados; possibilidade de mudar o resultado das avaliações automáticas; fazer destaque dos elementos na página web que está a ser avaliada e por fim, guardar o resultado da avaliação ao fazer export de relatórios em formato CSV e EARL.

Após a implementação seguiram-se testes com utilizadores. Estes foram conduzidos remotamente devido à pandemia actual COVID-19. Os testes consistiram em duas rondas distintas com cinco participantes no total. Estes testes facultaram uma visão mais profunda e detalhada dos melhoramentos necessários e providenciaram um estudo sobre os comportamentos dos utilizadores do AccessBot e as suas preferências. Após a primeira ronda, melhoramentos em termos de cosmética e funcionalidade foram instituídos. Estes foram testados na segunda ronda verificando que as dificuldades encontradas inicialmente foram ultrapassadas tendo sido sugeridas novas alterações cosméticas e funcionalidades. Os testes de utilização do AccessBot permitiram oferecer uma miríade de benefícios para os futuros utilizadores e a identificação de problemas que antes não eram aparentes.

No geral o desenvolvimento da tese permitiu um contributo para campo da acessibilidade informática ao incorporar a importância da avaliação de acessibilidade por parte do utilizador juntamente com o reforço de como é essencial aplicar conceitos de acessibilidade na programação das páginas web desde o início do seu desenvolvimento. O projecto AccessBot é um projecto que é de desenvolvimento contínuo e actualizado ao longo do tempo incorporando regras ACT recentes, de forma a conseguir alcançar um maior número significativo de utilizadores.

Palavras-Chave: acessibilidade, WCAG, avaliação, assistida, QualWeb, ACT-rules, AccessBot.

Abstract

Nowadays, the World Wide Web is a necessity, and its content should be available to everyone. People with different types of disabilities have different needs in using the web and access the content. Developers should fulfill these needs by making websites accessible. Alongside this premise, worldwide government directives oblige public and private sector websites and apps to meet accessibility requirements. To achieve a determined level of accessibility conformance, developers should follow the WCAG 2.1 (Web Content Accessibility Guidelines) and use automatic testing tools to evaluate their websites. However, while creating an accessible website, they may find difficulties that make this a laborious process. After studying and comparing eight of the most well-known accessibility evaluation extensions for the Chrome web browser, I found that these difficulties arise from various factors. These are subjective guidelines interpretations and implementations, automatic testing tools that provide limited coverage of the success criteria, different results displayed for the same website, and some guidelines are not tested automatically, meaning developers should perform manual testing. After analyzing these results, this project, with the name AccessBot, tries to cover the automatic accessibility evaluation gaps. It is an assisted validation tool using the open-source QualWeb accessibility evaluation. AccessBot is a browser extension for Chrome. Being a chrome extension makes it easy to access, install, and use by developers and more accessible to the general public. Its implementation aims to help users by visually identifying the problem, and performing a step-by-step guided evaluation, complementing the automatic evaluation done by QualWeb. The accessibility testing considers the test rules developed by the ACT-Rules Community, which makes an effort to create detailed descriptions of WCAG.

Keywords: accessibility, WCAG, evaluation, assisted, QualWeb, ACT-rules, AccessBot.

Acknowledgements

The completion of the thesis could not have been possible without the participation and support of people. Their contributions are sincerely appreciated and gratefully acknowledge, particularly to the following:

I am very grateful for the support and readiness to help of my professor adviser, Prof. Doutor Carlos Duarte. He believed in me as a person and in my capacities to deliver such important work and contribute to the Accessibility area. He kept me going and helped me stay positive during the research and writing of the thesis without losing momentum in difficult times of COVID-19 pandemic.

I'm also very grateful for the companionship and love of my husband, Leandro Reis. He is also my teacher in the world of computer science. He kept me calm, positive, kept things in perspective, and had lots of patience in more difficult times. Always there from day one.

I want to thank AMA accessibility employees, especially Jorge Fernandes, for his input in the design of AccessBot and out of the box ideas to push AccessBot further.

QualWeb team, António Estriga, João Vicente, and Bruno Andrade for their availabilty in support if needed.

My relatives, especially mother Maria Frazão and cousin Rute Marques, are my number one fans and always give me faith to trust myself.

Special thanks to my colleague when I was working and studying, Dra Sónia Ribeiro, who gave me total support financially, physically, and when I needed more time to study or had an exam. She also gave me the courage to continue. Without her, it was impossible to be able to take my Master's in Informatics.

To my friends, Vicky and Lúcia, who in one way or another, shared their support.

To my pets, dog Kruger and cat Alice for always being present with their funny moments.

It has been an incredible three-year journey, and it is with a great sense of gratitude, joy, and peace that finally is reaching the end. I overcame my self-doubt in Informatics and gained the confidence to take action in this new field for me.

Finally, I'm thankful for FCUL for having the Master's degree in Informatics available for students from a different background than Informatics. These allow the integration of computer science with other areas of expertise, which pushes both fields forward.

I thank you all.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The Web is an essential-good for many people at work, at home, and on the road. In a few words, Web accessibility means that people with disabilities can use the Web equally and on the same level as people without disabilities (W3C Web Accessibility Initiative, 2005). Disabilities come in many forms, as shown in figure 1.1. For example, visually disabled people such as the blind, colorblind, and people who cannot see very well. When considering visual disabilities, developers need to think about how to use color; the website should not have a contrast between text and background, making it difficult to see the text; websites should be usable with screen readers operated by a keyboard and just read out the website's content. For the screen reader to read, for example, an image, it is necessary to specify an alternative text for the image to be read by the screen reader by using, for example, the alt attribute in the image tag element.

Accessibility seems to more often refer to vision impairment. In reality, accessibility also considers other disabilities, such as hearing, mobility, speech, and cognitive. Examples of hearing disabilities are deafness or hard-of-hearing. For these, appropriate solutions include subtitles for videos or transcripts for podcasts, among others. Motor disabilities that cause difficulties, such as using a mouse, are why a website should be completely accessible solely from the keyboard. People with cognitive disabilities are one of the largest groups of individuals with disabilities. This type of disability is harder to quantify precisely. However, by making information accessible for people with learning disabilities, a content author makes it accessible to young children or people who are not necessarily going to read and understand a long block of text.

To know if a website is accessible, it needs to be evaluated or tested. The principal objective in accessibility testing is to find errors and give useful feedback to developers to promote future design and implementation changes that improve the site's usability and accessibility. "Test early, test often" is an old software engineering saying. Accessibility evaluations should start right at the beginning of product design and continue following development iterations into the final delivery to ensure quality on time and within budget (W3C, 2019b).

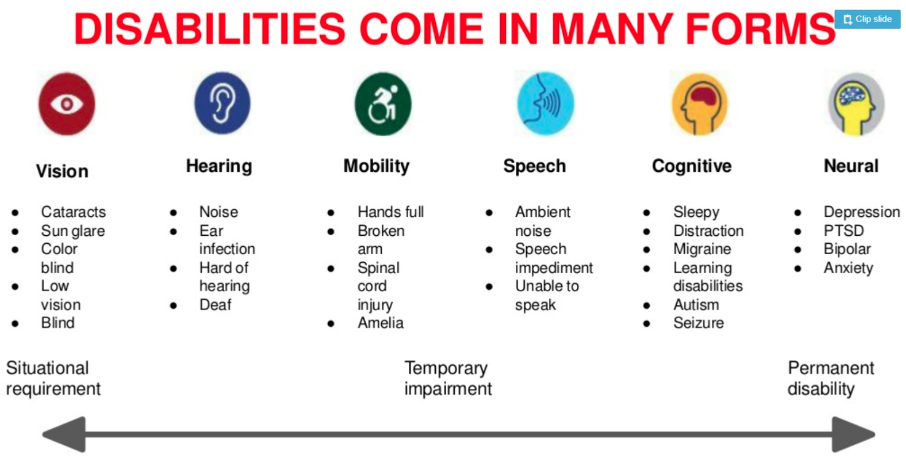


Figure 1.1: Examples of disabilities divided by different types (UXPA International et al., 2019).

1.1 Motivation

The disability market is composed of 1.3 billion people with disabilities globally (UXPA International et al., 2019), and this is how much will affect developers' roles and responsibilities. Moreover, looking at this number, friends, and families with a very close emotional connection to disability cannot be forgotten. They represent another 2.4 billion consumers (UXPA International et al., 2019). As a support mechanism, they want to support their friends and families with disabilities and not use an inaccessible website. Respectively, people with disabilities control \$2 trillion in income globally, and friends and family add \$8 trillion in annual disposable income (UXPA International et al., 2019).

When developers design websites and web applications well, they work for all people. Much accessibility can be built into the underlying code of websites and applications. The most considered standards to make websites and applications accessible are from the Web Accessibility Initiative (WAI)[1], a unit of the World Wide Web Consortium (W3C)[2]. The WAI has various resources to make the web-accessible. By describing real web users' stories, they acknowledge and formalize a set of disability types, mostly related to hearing, cognitive, physical, speech, and visual abilities, that usually restrict access to the Web. To achieve the purpose to make the Web accessible to everyone, the organization developed a set of guidelines with success criteria and techniques to support several components of web development (W3C Web Accessibility Initiative, 2019c):

- Web Content Accessibility Guidelines (WCAG) refers to the information on a web page or web application. The information can be natural information such as text, images, sounds, or code/markup that defines, for example, the structure and presentation;

- Authoring Tool Accessibility Guidelines (ATAG) refer to how to make the author-

[1]https://www.w3.org/WAI/
[2]https://www.w3.org/

ing tools themselves accessible so that people with or without disabilities can also create content for the Web;

- User Agent Accessibility Guidelines (UAAG) for browsers, browser extensions, and other applications that render web content.

Web standards from W3C, such as HTML, CSS, and many more, provide numerous accessibility features. For example, textual descriptions for images are read aloud by screen readers and used by search engines. Also, headings, labels, and other code supports accessibility and improves the overall quality.

The standards also led to the creation of a variety of Web accessibility evaluation tools with different characteristics: what guidelines version they cover, its language, type (if it is an API, Browser Chrome Extensions, Online Tool, along with others), if it generates reports or not, and so on. Important to note that across different countries, there are specific government guidelines that complement the W3C standards. The evaluation flow between tools is similar: they first retrieve the source code of the page, assess the techniques they implement, and output results that users analyze (Matos, 2017). In the end, the main objective is to help web developers and designers meet accessibility guidelines. At the moment, there are many applications, such as QualWeb, among others (W3C Web Accessibility Initiative, 2006) that will be discussed further.

Helping developers construct accessible websites is the motivation of this dissertation, alongside providing a tool for any user to perform accessibility evaluations on websites. Building accessible websites can be challenging for developers since they may need help understanding and implementing WCAG 2.1 guidelines' success criteria. Accessibility issues can quickly go unnoticed if the users do not have the correct tools to test for them. Automatic testing can help identify specific types of web accessibility issues and only find about 30% on average accessibility problems (Government Digital Service, Uk's Cabinet Office, 2018); the rest of the problems must be located using manual methods, and these developers need assistance to go through manual testing. This thesis's work proposes designing and implementing a tool to have a broader coverage of success criteria combining automatic testing with manual testing.

1.2 Objectives

There are typically two significant ways to evaluate accessibility: automatic and manual (also called assisted). The automatic uses tools from different types. They could be online tools, API (Application Programming Interface), browser extensions (for example, AccessBot), CLI (Command Line Tools), desktop applications, and others.

The work to be developed will complement automatic evaluations from the QualWeb engine with assisted evaluations, in a Chrome extension environment, by creating an autonomous program ("bot"), called AccessBot, that can interact with the webpage and the user. The open-source engine QualWeb, which performs the automatic evaluation, is maintained by the Department of Informatics at the Faculty of Science, University of Lisbon, Portugal.

The AccessBot will make the engine more useful with a different interface. The assisted evaluation needs more resources than automatic evaluation; however, it can detect many accessibility problems.

While identifying elements in the code or DOM can be automatically checked, the validation of how well the success criteria are fulfilled according to the recommendation is not, in most cases. For example, what appears to exist visually in a website may not correspond to the same in the code. Visually the website can appear to have headings, but these may not follow the correct syntax of HTML and be formatted using CSS. Another example, the 1.1.1 success criteria (WCAG 2.1), states, “All non-text content that is presented to the user has a text alternative that serves the equivalent purpose.” An automatic tool can detect if the alt attribute is present and not empty for this success criteria. However, it cannot evaluate if the textual description of the image is correct. At the present moment, only a human evaluator can verify the textual meaning of alt. The work will focus on testing web pages for various Success Criteria to verify if what appears on the website communicates, using all senses, correctly with all users. The Accessbot will show results similar to other automatic tools. The added feature will be in helping the developer with manual procedures. The manual evaluation can be supported in different ways; the bot can identify all the elements affected by one technique on the webpage or may present a procedure (list of steps) for a technique, guiding the evaluator during the process.

Another objective of the AccessBot is to enable unequivocal interpretation and implementation of testing methods by using the rules defined according to the Accessibility Conformance Testing (ACT) Rules Format 1.0 (W3C, 2019a), which defines a writing format for accessibility test rules. An ACT Rule is an explicit language description of testing a specific type of content for a specific aspect of accessibility requirement (W3C, 2019a).

These test rules can be used for developing automated testing tools and manual testing methodologies. It provides a standard format that allows any party involved in accessibility testing to document and share their testing procedures robustly and understandably. By using ACT rules description enables transparency and harmonization of testing methods, including methods implemented by accessibility test tools (Foley, 2019).

The tool is to be available for users as a Chrome extension. This way, it provides easy access and will not disturb the developer’s workflow. It is essential to mention that the extension is intended to be used by developers and, for example, by government bodies to check if overall websites are accessible.

1.3 Planning

The work was divided into different stages of development to achieve the objectives mentioned. The plan initially proposed was:

A Architecture definition and implementation support mechanisms for the evaluation presentation. These include the following steps:

  1 Familiarization with the QualWeb engine specifications.

  2 Learn how to make a Chrome extension;

  3 Develop a Chrome extension;

  4 Connect the extension to QualWeb.

  5 Implementing Evaluation and Report Language (EARL) for test reporting.

B Identify the ACT Rules that need a semi-automatic or manual evaluation.

C For each one of the rules, define the bot flow, and implement it. Different techniques have different flows, such as checking if the description of an image matches the actual image is one flow and checking the tab order is another flow.

D User testing, including preparation, execution, and result analysis of the tests performed to the users.

E The writing of the final report co-occurs with other phases.

1.4 Planning Execution

The planning execution details the steps taken to accomplish the phases listed in the plan above to accomplish the items referred.

The first phase A), includes the overall research about accessibility to keep up with state of the art, which is rapidly evolving, and to become acquainted with the accessibility guidelines of WCAG 2.1. The information sources on accessibility used during the thesis's development are the University of Lisbon repository[3], proceedings from W4All[4], information online from W3C WAI[5], and paper search engines like Google Scholar[6]. It also involved learning how to implement a Chrome extension by reading the developer Chrome extensions documentation[7].

[3]https://repositorio.ul.pt/handle/10451/12140
[4]http://www.w4a.info/
[5]https://www.w3.org/WAI/
[6]https://scholar.google.com/
[7]https://developer.chrome.com/extensions

Chapter three has a description of the “Study of Existing Accessibility Evaluation Chrome Extensions”. I did the study during the dissertation in order to achieve an overall perspective of what Chrome extensions exist at the present moment, which ones are most used by developers, their strengths and weaknesses. The study’s findings are also included in the design of the AccessBot, allowing to fulfill the gaps found in other Chrome extensions and match the user’s evaluation tools needs.

Since QualWeb is the engine of AccessBot, it is vital to familiarize with QualWeb, to know how the engine works, its architecture, and how it is implemented. The QualWeb code is available on Github[8]. Still, during the first phase, a challenge arose during integration. The problem resided in the fact that QualWeb is implemented in node.js, and AccessBot runs in the browser and is supposed to make HTTP requests to QualWeb. However, although node.js and the browser both run Javascript (ECMAScript), the environment is different. In the browser, we don’t have all the APIs that Node.js provides through its modules, like the filesystem access functionality alongside other differences. This phase involved a new replanning where QualWeb developers’ team needed to find the best solutions to ensure the Accessbot could integrate with QualWeb. Chapter five describes more information about this process, and the adaptations needed to complete the integration.

After the integration of browser extension and QualWeb, the next phases B) and C) were interchangeably done simultaneously. For each semi-automatic and manual ACT rule identified, the application flow was designed, and then the code was implemented. Phases B) and C) occupied a large amount of time of the plan.

The EARL report described in phase A) was postponed to phase C) in order to prioritize the implementation of the ACT rules.

The AMA (Agência para a Modernização Administrativa), responsible for developing administrative modernization in Portugal, uses the QualWeb engine for the accessibility evaluation of public government websites. Since the Agency is one of the primary users of QualWeb, they participated in the user testing referred to in phase D). User testing consisted of remote moderated testing of AccessBot because of the pandemic and included two testing rounds. The first round had three participants, two developers, and one AMA accessibility evaluator. After completing the modifications resulting from the first-round user’s insights, the second round had two participants, two AMA accessibility evaluators.

The final months of the work consisted of the majority of the writing. To sum up, the time for some of the tasks in each phase changed slightly concerning each other, but the work’s execution was within the estimated time frame.

[8]https://github.com/qualweb

1.5 Contributions

The dissertation starts by offering an analytical approach to the problems of accessibility evaluation by first studying the accessibility Chrome extensions tools for accessibility evaluation that exist at the moment for accessibility evaluators. The research on the most common automatic evaluation tools available as Chrome extensions for developers lead to publishing the article (Frazão et al., 2020).

Frazão, Tânia and Carlos Duarte (2020). "Comparing Accessibility Evaluation Plug-Ins". In: Proceedings of the 17th International Web for All Conference. W4A'20. Taipei, Taiwan: Association for Computing Machinery. DOI:10.1145/3371300.3383346.

This article reports the results of a study of eight of the most well-known automatic tools which are free or available under an open-source license. The tools were compared based on their feature set, their usability and their evaluation results of ten of the Alexa top websites. It was found that individual tools still provide limited coverage of the success criteria; the coverage of success criteria varies quite a lot from evaluation engine to evaluation engine and what are the most and least covered success criteria in automated evaluations. After analysing the results, the study recommends to use more than one tool (with a different engine) and to complement automated evaluation with manual checking since there is no ideal tool for identifying all the barriers a web page has, and they often lack human input evaluation.

The thesis then focuses on developing and implementing algorithms for human evaluators and uniting them with automatic evaluation based on the previous premise. It culminated in developing the Chrome extension tool, AccessBot. AccessBot extends the accessibility evaluation capabilities of an automatic evaluation engine, QualWeb, by adding the algorithms and making it easier to use as a Chrome Extension without complicated installations. As it is known, there are limitations of what coding can do to evaluate a web page for some guidelines that govern how accessible a web page has to be. AccessBot can surpass these limitations by including in the web page evaluation the user's input and appraisal. The tool shows that it is possible to go beyond automatic evaluation and improve an automated evaluation tool. In the end, this dissertation aims at contributing to the accessibility field by incorporating an importing dimension such as the manual user accessibility evaluation.

1.6 Document and Organization

The thesis consists of seven chapters. The first chapter, "Introduction" presents the context of work, the motivation, the main objectives, and describes the planning and its execution. The second chapter, "Related Work" provides a topical introduction about accessibility, its evaluation methods, discussing several references about problems with guidelines and accessibility tools that exist for developers to use and develops the theoretical framework of a Chrome extension. The third chapter, "Study of Existing Accessibility Evaluation Chrome Extensions" refers to an experimental study using eight browser accessibility tools, and presents the results obtained. The obtained results are interpreted and discussed, creating the foundation and need for the implementation of AccessBot. Chapter four refers to the design of AccessBot, describing what AccessBot should do. Chapter five is about AccessBot implementation. It explains how it was done, the difficulties encountered during the process, and at the end of the chapter, the user interaction section shows how it looks. In chapter six, AccessBot is put to testing by different evaluators. Some alterations needed to be done after valuable insights from the users. Chapter seven concludes the work of the thesis summarizing the results obtained during the development of AccessBot.

Chapter 2

Related Work

This section provides the context required to understand the fundamentals for developing the AccessBot and the parts involved. It begins by examining the concepts of usability and accessibility, the accessibility guidelines, its problems, the community's need to create ACT rules, and presenting an overview of existing evaluation tools. It also introduces the QualWeb engine and presents an overall view of a Chrome extension, its architecture, and how to create one.

2.1 Usability and Accessibility

Usability and accessibility are crucial concepts in the context of user experience. They are closely related to creating a web that works for people with or without disabilities. Because their goals and guidelines may overlap significantly, it is most effective to address them together when designing and developing websites. However, there are few situations when it is significant to focus on one aspect, for example, when developing standards and policies (W3C Web Accessibility Initiative, 2016).

Usability is about designing products to be effective and satisfying. It includes user experience design incorporating general aspects that impact everyone. Usability practice does not sufficiently address the needs of people with disabilities.

Accessibility addresses discriminatory aspects related to user experience for people with disabilities. A more accessible website means that people with disabilities can perceive, understand, navigate, and interact with websites and contribute without barriers. It covers requirements that are technical and relate to the source code rather than to visual appearance, for example, to ensure that websites work well with assistive technologies (screen readers, screen magnifiers, or voice recognition software). It also covers requirements related to user interaction and visual design, since inadequate design can cause barriers for people with disabilities (W3C Web Accessibility Initiative, 2016).

Usability and accessibility are closely connected and cannot be considered separately from each other; being accessibility more important for developers who, while taking into account accessibility, will at the same time make websites more usable for every people. Combining accessibility standards and usability methods with real people ensures that web design is technically and functionally usable by people with disabilities.

2.2 Web Accessibility Guidelines

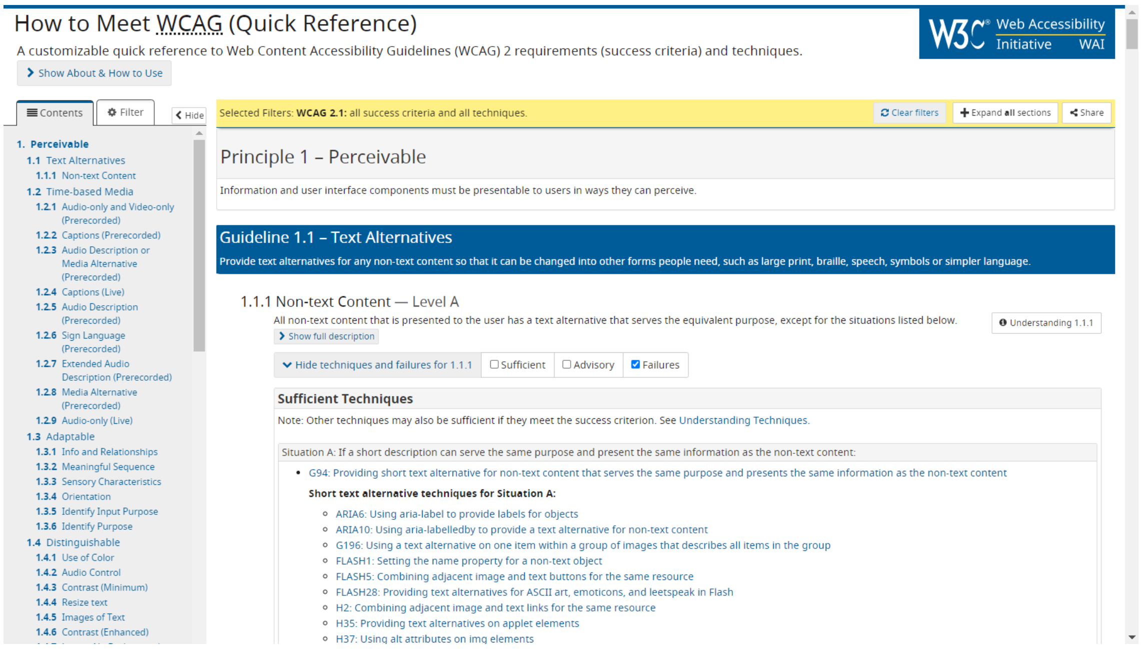
Over the years, the W3C Web Accessibility Initiative (WAI) has generated sets of guidelines to systematize what is required to produce and render accessible web content (W3C, 2019b; Hudson, 2011). These guidelines on accessibility are available in the form of checklists with hyperlinks to explanations and testing methods. The current set of guidelines for assessing the accessibility of websites are WCAG 2.1, which are organized around four accessibility principles (Stephanidis et al., 2009):

- Perceivable, which means that everyone should be able to perceive the information. Websites should not have any information hidden from a particular class of users; for example, if people are deaf, they should have transcripts they can read for audio podcasts. Otherwise, that information would be completely hidden.
- Operable, which means people need to access all information and all websites regardless of disabilities. For example, a button that can only be activated by clicking on it is undesirable because, otherwise, people that cannot use a mouse as a consequence of a motor disability will not be able to access that button;
- Understandable, which means all the information needs to be understood by the user and the website should be predictable;
- Robust, which means the website should behave similarly regardless of the user agent used to render it and still be accessible to everyone.

Each principle mentioned has a set of testable success criteria. The WCAG 2.1, published in June 2018, extended the number of success criteria in WCAG 2.0 by 17, taking the total to 78 testable success criteria. Figure 2.1 shows a snapshot of the W3C quick reference for the WCAG 2 success criteria.

Conformance to a standard means that a web page or website meets or satisfies the standard's "requirements". In WCAG, the 'requirements' are the Success Criteria. Conforming to WCAG means to satisfy the Success Criteria, and no content violates the Success Criteria.

Most standards only have one level of conformance. To accommodate different situations that may require or allow more significant levels of accessibility than others, WCAG has three conformance levels, and therefore, three levels of Success Criteria. The three conformance levels are: A (lowest), AA, and AAA (highest), making each level representing an increasing level of accessibility. Conformance at higher levels indicates conformance at lower levels also. For example, a web page that conforms to AA meets


How to Meet WCAG (Quick Reference)

A customizable quick reference to Web Content Accessibility Guidelines (WCAG) 2 requirements (success criteria) and techniques.

Show About & How to Use

Contents | Filter | Hide

1. Perceivable
   1.1 Text Alternatives
      1.1.1 Non-text Content
   1.2 Time-based Media
      1.2.1 Audio-only and Video-only (Prerecorded)
      1.2.2 Captions (Prerecorded)
      1.2.3 Audio Description or Media Alternative (Prerecorded)
      1.2.4 Captions (Live)
      1.2.5 Audio Description (Prerecorded)
      1.2.6 Sign Language (Prerecorded)
      1.2.7 Extended Audio Description (Prerecorded)
      1.2.8 Media Alternative (Prerecorded)
      1.2.9 Audio-only (Live)
   1.3 Adaptable
      1.3.1 Info and Relationships
      1.3.2 Meaningful Sequence
      1.3.3 Sensory Characteristics
      1.3.4 Orientation
      1.3.5 Identify Input Purpose
      1.3.6 Identify Purpose
   1.4 Distinguishable
      1.4.1 Use of Color
      1.4.2 Audio Control
      1.4.3 Contrast (Minimum)
      1.4.4 Resize text
      1.4.5 Images of Text
      1.4.6 Contrast (Enhanced)

Selected Filters: **WCAG 2.1:** all success criteria and all techniques. | Clear filters | Expand **all** sections | Share

Principle 1 – Perceivable

Information and user interface components must be presentable to users in ways they can perceive.

Guideline 1.1 – Text Alternatives

Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, braille, speech, symbols or simpler language.

1.1.1 Non-text Content — Level A

All non-text content that is presented to the user has a text alternative that serves the equivalent purpose, except for the situations listed below.

Understanding 1.1.1

Show full description

Hide techniques and failures for 1.1.1 | ☐ Sufficient | ☐ Advisory | ☑ Failures

Sufficient Techniques

Note: Other techniques may also be sufficient if they meet the success criterion. See Understanding Techniques.

Situation A: If a short description can serve the same purpose and present the same information as the non-text content:

- G94: Providing short text alternative for non-text content that serves the same purpose and presents the same information as the non-text content

  Short text alternative techniques for Situation A:

  - ARIA6: Using aria-label to provide labels for objects
  - ARIA10: Using aria-labelledby to provide a text alternative for non-text content
  - G196: Using a text alternative on one item within a group of images that describes all items in the group
  - FLASH1: Setting the name property for a non-text object
  - FLASH5: Combining adjacent image and text buttons for the same resource
  - FLASH28: Providing text alternatives for ASCII art, emoticons, and leetspeak in Flash
  - H2: Combining adjacent image and text links for the same resource
  - H35: Providing text alternatives on applet elements
  - H37: Using alt attributes on img elements

Figure 2.1: Snapshot of WCAG Quick Reference which presents a summary for WCAG 2 success criteria and techniques. (W3C Web Accessibility Initiative, 2019b)

both the A and AA conformance levels. Few studies have investigating the impact of the level of conformance of websites. However, it seems that implementing recommendations for a high level of web accessibility leads to improved accessibility. Furthermore, a high level of conformance proves to benefit both users with disabilities and non-disabled users (Kirkpatrick et al., 2018; Schmutz et al., 2016).

The guidelines provide techniques for developers to help them satisfying the success criteria's goals. The techniques are informative, meaning they are not required since success criteria can be met through other means. The basis for determining the conformance to WCAG 2.1 is the success criteria, not the techniques (Accessibility Guidelines Working Group, 2017). Content authors can implement different techniques; for example, the developer could develop a technique for HTML5, WAI-ARIA, or other new technology. W3C acknowledges that any techniques can be sufficient to satisfy the success criteria (W3C Web Accessibility Initiative, 2017c). If no content to which a success criterion applies fails, the success criterion is considered satisfied (W3C Web Accessibility Initiative, 2017b).

The three types of techniques described in WCAG 2.1 are sufficient, advisory, and failures. Sufficient techniques are techniques that, if implemented, means it meets the success criteria. The advisory techniques are suggested ways to improve web accessibility; they are helpful to some users accessing some types of content. They are distinct from the sufficient techniques because they may not be sufficient to meet the success criteria' full requirements. They may be based on technology not yet stable, they may not be tested, or some technologies may not work with them yet. For example, in some circumstances, assistive technologies may not be applicable or practical. They may increase accessibility for some users and decrease it to others. They may provide only related accessibility benefits and not address the success criteria itself. Failures are things that cause accessibility barriers and lead to success criteria failure. If the content has a failure, it does not meet WCAG success criteria unless an alternate version is provided without the failure. Each technique has tests to help verify if the technique is correctly implemented. The tests are not tests for conformance to WCAG. Evaluations must go beyond checking the sufficient technique tests to evaluate how content conforms to WCAG success criteria. Also, failing a technique does not necessarily mean failing WCAG because techniques, as said before, are not required, and content can meet WCAG success criteria in different ways (W3C Web Accessibility Initiative, 2017c; Accessibility Guidelines Working Group, 2017).

Web accessibility evaluation is an assessment procedure to analyze how well people with different disabilities can use the web page. Optimal results for accessibility evaluation are achieved using different approaches and taking advantage of specific benefits from each of them. The approaches range from automated testing provided by different tools (desktop applications, online tools, APIs, browser chrome extensions, and command-line tools) to manual inspection from accessibility experts or empirical evaluations. Automated tools can be used in isolation, but automated mechanisms often support even manual inspections.

2.3 Problems with accessibility guidelines

Different authors argue there are problems with the web accessibility guidelines (Stephanidis et al., 2009; Brajnik, 2008; Carvalho et al., 2018; Rømen et al., 2012; Vigo et al., 2013). One perspective that has been studied is the relation between guideline violations and real accessibility problems. Some studies found that evaluating the accessibility of web pages just by checking conformance with the WCAG results in overlooking several accessibility issues. This problem is aggravated if the accessibility check relies solely on automated tests that cannot validate the full number of success criteria in the guidelines. Another perspective that has been critiqued by practitioners is the applicability of the guidelines. Although guidelines would seem to present objective criteria against which to evaluate a system, they raise several difficulties. A large number of guidelines need much effort to learn and apply properly. For an accurate evaluation, every page should be evaluated against every applicable guideline, which would be very time-consuming. The guidelines help improve a website, but guidelines are generalizations, so there may be particular circumstances where guidelines conflict or do not apply and may be interpreted subjectively. To apply accessibility guidelines appropriately is needed expertise in the evaluation of detailed characteristics. However, this evaluation alone can never be sufficient, as this does not provide information to predict user behavior accurately. To be sure about the outcomes, evaluation of web accessibility also requires manual inspection and testing with users.

As stated above, guidelines can be hard to be interpreted or be interpreted in different ways by testers, organizations, and in different regions around the world.

2.4 ACT Rules

To make it easier for developers to reach consensus and uniformity on WCAG interpretation, the ACT-Rules Community (ACT-R) was created.

The ACT-R is a group of accessibility tool vendors, test procedure authors, and accessibility test experts that created an open community to set up a document and harmonize the interpretation of W3C accessibility standards, such as WCAG and WAI-ARIA, for testing purposes. Test rules are defined using the ACT Rules Format and reviewed by the community. The process of researching, documenting, and sharing knowledge from different perspectives within the group, builds towards a common understanding. By publishing such test rules, ACT-R hopes to motivate organizations to share their insights and adopt commonly agreed test rules. It aims to contribute to more consistent results, regardless of how the testing is done. To understand when something meets a requirement and when it does not should be clear and consistent. The ACT-R Community has no standing in the W3C and does not develop W3C recommendations or notes (ACT-Rules Community Group, 2019).

2.5 Accessibility Evaluation Methods

Success criteria from WCAG guidelines are written as testable criteria for objectively determining if content satisfies them. Testing the success criteria would involve a combination of automatic testing and human evaluation. Although content may satisfy them all, it may not be usable by people with a wide variety of disabilities. The W3C advises to include usability testing and the required functional testing that verifies if the content functions as expected. Performing usability testing helps determine how well people use the content for its intended purpose (W3C Web Accessibility Initiative, 2017b).

2.5.1 Expert evaluations

Expert testing is essential because experts understand how the underlying web technologies operate, and they can act as intermediaries for knowledge about different user groups (Brajnik, 2008). Expert evaluations should be done when initial prototypes are available. They serve to identify any accessibility issues in order to eliminate them before carrying user-based evaluations. These evaluations are also performed because it may not be possible to obtain actual users for evaluations or insufficient time for testing with users. Expert-based methods require one or more accessibility domain experts to work through a website looking for accessibility problems (Brajnik, 2008; Stephanidis et al., 2009; Kirkpatrick et al., 2018).

Expert testing includes four components:

- The tool-guided evaluation consists of a tool that looks for accessibility problems and presents them to the evaluator. While beginners may be mostly dependent on tool-guided evaluation, evaluators of all levels of experience know that even a single checkpoint may require several tests to check if it has been passed. Some of the tests can be automated, and some cannot. An automated tool can check whether there is an alternative description of every image, which can be a useful function in evaluating. However, no automatic tool can check whether the alternative descriptions are accurate and useful (Petrie et al., 2005).

- The screening and using of end-user assistive technology by experts can help them work through task scenarios representing what users would typically do (Stephanidis et al., 2009). Screening means experts try to reproduce the experiences of people with disabilities. They can use assistive technology to interact with a site or try to restrict one's abilities in some manner (W3C Web Accessibility Initiative, 2019a).

- Structural inspectors consist of experts' tools to probe how the various components of a web site work together. Inspection tools are designed to review the structures of web content. By definition, structures define the components of a web page and how they are related to one another. Experts can, by using the structural inspectors, check the HTML DOM (document object model), which is parsed by the browser. The browser associates different behaviors with particular components. Standard assistive technology does not process web document object models directly; they utilize the browser's and plugin's representation of web content in terms of structural systems. By using the DOM inspectors, these show the tree of elements, attributes, and text composed out of the HTML serialization, whereas the web accessibility inspectors abstract distinct components or relationships and list them (W3C, 2019b).

- Code review occurs when the evaluator looks directly at the code and assets of a web site to search for problems. After specific problems are resolved using a checker tool, the experts can pass on to manual testing and do a detailed inspection. From the above, it becomes apparent that even for expert evaluation, automated checking tools are paramount.

2.5.2 Automatic Checking

Although automated accessibility checking has its role in evaluating websites, its strengths and weaknesses need to be understood because automatic tools cannot check many WCAG checkpoints automatically. According to Vigo et al. (2013), using evaluation tools reduces the burden of identifying accessibility barriers. However, an over-reliance leads to placing aside additional testing that necessitates expert evaluation and user tests. In a study about the sole reliance on automated tests, the researchers investigated the effectiveness of 6 state-of-the-art tools (AChecker, SortSite, Total Validator, TAW, Deque, and AMP). These tools had in common their capability to test web pages against the WCAG 2.0 guidelines. The coverage obtained was very narrow as, at most, 50% of the success criteria were covered. The most frequently violated success criteria were "1.3.1 Info and Relationships", "1.4.3 Contrast", "1.1.1 Non-text Content", "1.4.4 Resize Text" and "2.4.4 Link Purpose".

Therefore, relying on just automated tests entails that 1 out of 2 success criteria will not be analyzed (Vigo et al., 2013). Using only automated tools is not by itself a viable answer to the problem of evaluating accessibility. The W3C/WAI states: "Web accessibility evaluation tools can not determine the accessibility of Web sites, they can only assist in doing so" (W3C Web Accessibility Initiative, 2017a). Although automated checking tools are not sufficient to determine the accessibility of a resource, these tools are great to help developers meet accessibility standards. Developing better, automated, or semi-automated tools is essential. With the ever-growing dynamics of web pages (e.g., through AJAX and other JavaScript techniques), the state of a web page's content, structure, and interaction capabilities might become different when compared to the initial HTTP communication. Several dynamic content techniques allow for displaying or hiding information, injecting new content, and even removing Web pages' content. According to Fernandes et al. (2012) the automated evaluation must be applied to the content web browsers display, which can be an advantage for tools directly integrated on the browser. WAI website maintains an updated list of automated accessibility checking tools available to use (W3C Web Accessibility Initiative, 2006).

2.6 Introduction to QualWeb

QualWeb is an open-source[1] automatic accessibility evaluator developed over time by a group of researchers at LASIGE at the Faculty of Science of the University of Lisbon. Qualweb incorporates different contributions from different research projects. The first version is from 2008. It can perform an automatic analysis of the web page's content against a set of WCAG 2.1 level AAA standard Techniques and ACT Rules. This text describes how QualWeb works since it is the engine used and integrated into the javascript AccessBot project. Qualweb is an automatic evaluation tool. It accesses a representation of a web page's DOM after the browser has processed it, runs a series of scripts to obtain the outcomes of multiple rules and techniques, and outputs the results.

The web page accessibility evaluation by Qualweb presents the test target results with

[1]QualWeb is open-source and available at https://github.com/qualweb.

fail, pass, cannot tell, and inapplicable. Test targets are specific elements or nodes within an HTML page under testing. The HTML page is defined as the test subject, and this includes all embedded scripts, style, and images. Listing 2.1 provides examples for a rule that checks if image elements `img` have a text alternative; it presents a `passed` outcome on line one, an example of a `failed` outcome on line two, and an example of an `inapplicable` outcome on line three. The results are accompanied by outcome descriptions, informing the user why there is a specific problem with the web page. When one or more of the outcomes for a test target is failed, it means the accessibility requirements of the ACT Rule to test conformance to WCAG are not satisfied for the test subject. When all of the outcomes are passed or inapplicable (if there are no test targets), the accessibility requirements could be satisfied, or further testing is needed. In WCAG, success criteria do not evaluate to passed, failed, or inapplicable. They can be satisfied or not.

Listing 2.1: Example of HTML test cases for a rule that checks if img elements have a text alternative.

```
1 <img alt="W3C Logo" src="image/w3c.png">
2 <img src="image/w3c.png">
3 <input type="image" alt="W3C Logo" src="image/w3c.png">,
```

Qualweb syntactically checks the web page. For example, it verifies if images have an alt attribute. However, it does not verify the semantic relationship between elements. In the case of images, it is necessary to verify if the information conveyed in the alt text is related to the image and correctly describes it (Duarte, 2018).

Besides pass and fail outcomes, QualWeb shows possible problems with semantics, with `cannotTell` results. According to Santos Vicente (2018) this may be seen as an advantage when comparing with other accessibility evaluation tools. The result `cannotTell` points to a possible problem that needs manual assessment.

The integration with Accessbot extends the automatic evaluation by guiding the user to perform a manually semantic evaluation for some of the results.

QualWeb, in general, comprises the following elements:

- Core - the QualWeb source code receives the URL of the page and processes the web page for accessibility automatic evaluation. It contains four modules, besides the ACT-Rules module, which AccessBot uses, has the CSS and HTML techniques and best practices modules, which help meet WCAG success criteria and conformance requirements. The core can be integrated with any other source code on the server;

- CLI (command-line interface) - has the same core features but allows the user to perform an automatic evaluation of a web page from the terminal;

- Online - has the same core features and can be used online at website `http://qualweb.di.fc.ul.pt`. It has a graphical user interface to display the content of the web page evaluation.

2.7 Introduction to Chrome Extensions

2.7.1 What are Google Chrome Extensions

Chrome extensions are a viable way of enhancing web browsers' functionality by having access to almost all the features provided by the browser (Mehta, 2016). The objective is to provide targeted functionality to users. In the case of AccessBot to evaluate Web accessibility. They are built on web technologies such as HTML, Javascript, and CSS (Google, 2020g). They are also secure since they run in a sandboxed environment. Sandbox is a software container. It allows the execution of web technologies and also provides access to features of browsers. The Chrome Extensions framework provides APIs that help empower web applications by coupling with features provided by the Google Chrome web browser, for example, tabs, popup, actions, and search. Google Chrome supports browser extensions since 2010 (Mehta, 2016).

An important point to note is that browser extensions are not browser plug-ins since they are sandboxed within the host web browser while plug-ins are not. Plug-ins provide new support for particular media types to browsers. For example, an extension can allow users to save all the opened tabs, and a plug-in allows reading and rendering PDF files in the browser.

2.7.2 Advantages of Chrome extensions

Some Google Chrome users rely on extensions for increasing their productivity at work and solely for getting the most out of their web browser. The extensions provide a better workflow since they are of easy installation through the Chrome Web Store and simple access in the Chrome browser. They also provide a single purpose that should be narrowly defined and easy to understand, even if they include a range of functionalities (Mehta, 2016; Google, 2020g).

2.7.3 Technologies for Extensions Development

The technologies used to create Google Chrome Extensions are vanilla[2] HTML, CSS, Javascript, and JSON. Google Chrome Extensions can be built from any operating system.

HTML and CSS create the user interface (structure and styling, respectively). Javascript is used to provide the application logic and access the Google Chrome Extensions framework's APIs and components. JSON is applied to create the manifest file for the extensions where it provides information about the extension itself to the Google Chrome Browser.

[2]By definition vanilla is a term used in computer science when technologies are not customized from their original form, meaning that they are used without any customizations or updates applied to them.

2.7.4 Google Chrome Extensions API

As said earlier, Google Chrome Extensions are sandboxed, and this means that the code runs isolated, implying that different extensions cannot access code or memory belonging to another extension. Because of this sandboxed environment, there will not be name conflicts, even if there are extensions and files of different extensions. The Extensions framework provides many special-purpose APIs[3], for example, `chrome.runtime` API, but extensions can still use the standard APIs[4] that the browser provides to web pages. These are Javascript, and Document Object Model (DOM) APIs, HTML5 APIs, WebKit[5] APIs (for experimental CSS features such as animations, filters, and transformations), and V8 APIs[6] (such as JSON) are supported.

2.7.5 Development of a Chrome Extension

Chrome extensions can be simple or more complicated, depending on their purpose. Extensions are made of different but cohesive components such as background scripts, content scripts, options page, user interface elements, and logic files (Google, 2020c). Depending on the functionality, there will be different components with different options.

2.7.5.1 Components of a Chrome Extension

Various components are used to create Chrome Extensions. They are the building blocks. In a way, Chrome Extensions are no different from any other software application since the user interacts via inputs such as buttons, processes data, and displays the result. Like other development frameworks, the Google Extensions framework provides its developers with techniques to provide a user interface and other functionalities such as messaging or web requests between others. The components that are used to create Chrome Extensions are:

- Manifest components - every extension has a JSON formatted manifest, named `manifest.json`. The manifest provides information about the extension to the Chrome browser. The manifest contains the features that the extension will use, such as inputs (for example, Browser-Action), and defines their corresponding values. In the manifest, it is possible to declare the use of other APIs to access the Chrome Browser's functionalities, such as bookmarks, tabs, history, amongst others. The manifest file is the only reserved file name in the extensions. The other files can have different names (Mehta, 2016).

[3]For more information about these APIs on UR: https://developer.chrome.com/extensions/api_index.
[4]For more information about these APIs on URL: https://developer.chrome.com/extensions/api_other.
[5]WebKit is a web browser rendering engine to draw the HTML/CSS web page.
[6]V8 is Google's open source high-performance javascript engine, written in C++ and used in Chrome and node.js (can execute javascript within or outside of a browser.

- Input components - offer interactive functionality consisting of user interface and non-user interface Input elements (Google, 2020b). These are considered the entry points to the extensions' core logic, and they trigger certain responses from the scripting components or display a popup if the extension has one (figure 2.2). For example, the input component of choice during the development of AccessBot is browser/action, which allows putting icons in the Google Chrome Toolbar to the right. The browser action is used in an extension where common functionality is desired for every visited page. Note that Browser-Action APIs can be accessed from all scripting components, except content scripts.

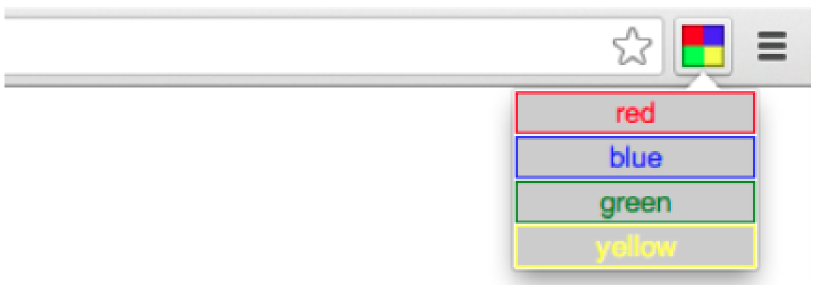


Figure 2.2: Representation of chrome browser/action input component. The multicolored square to the right of the address bar is the icon for a browser action. A popup is below the icon. (Google, 2020a)

- Scripting components - are the components that contain the application logic required when users interact with the extension. There are three types of scripting components: background, popup, and content scripts. Each scripting component has its separate scope. For example, popup scripts cannot use variables and functions defined in the background and vice-versa. The same happens for other pairs of scripts such as content scripts and popup scripts or content scripts and background scripts. The only way the scripts can access each other's data (variables and functions) is messaging (Mehta, 2016).

  According to Mehta (2016), the characteristics of each script are described in the following list.

  - Background scripts (event scripts):
    - Extensions are event-based programs used to modify or enhance the Chrome browsing experience, and these events are monitored in their background script, then reacting with specified instructions.
    - These events are browser triggers such as events fired from input components such as Browser-Action, for example, `chrome.browserAcion-onClicked` (Google, 2020d); in other

words, in this script there is a listener created to trigger the popup when users click on the browser icon.

- ∗ Another important use for the background script is to listen for events fired from the extension itself, such as `onMessage` that are only accessible from the `chrome.runtime` object. Most of these events are part of the messaging API provided by the Chrome Extensions framework.
- ∗ An event script's essential characteristic is the *persistence* of event-script. It can listen for events in a reliable manner because it is a long-running script (unlike the popup script, which is only executed when the popup is opened). Event scripts stay dormant and are automatically loaded when needed (for example, when the events they are listening to get fired), react with specified instructions, and are unloaded when they go idle.
- ∗ Background scripts are registered in the manifest (listing 2.2) under the `background` field and listed in an array, and `persistent` should be specified as `false` to make the event script active only on an event basis or `true` if it is always active (Google, 2020e).

– Popup Scripts:
  - ∗ The HTML file `popup.html` represents the view the extension's popup will have, and the javascript file `popup.js` will contain the application logic (listing 2.3).
  - ∗ They are an option view available to the Browser-Actions input components, consisting of an HTML page that only appears when the user clicks on the toolbar button. (Google, 2020a)
  - ∗ They have an essential feature: popups can access the Chrome Extensions API and all the Standard Javascript APIs, including listening for and responding to DOM events fired from the nodes within a popup.

– Content Scripts:
  - ∗ Type of scripting component that is injected into the visited web page(s).
  - ∗ Have minimal access to the Chrome Extensions API because they do not represent the extension runtime; they run in an isolated environment in the context of a web page and not the extension. They can read, modify the content, or add content (for example, HTML elements) of visited web pages using the DOM API.
  - ∗ They cannot use `chrome.*` APIs with the exception of `chrome.runtime` object, `chrome.extension`, `chrome.storage` provided by the Chrome Extensions framework.
  - ∗ They are declared in the manifest using the content script component, as displayed in listing 2.4, which takes the following properties in its definition - `matches`, `css`, and `js`.

Listing 2.2: The listing represents an excerpt of the manifest file where the background script is defined and is referred automatically in a generated HTML page with the name `background.html`.

```
...
{
"background": {
    "page": "background.html",
    "script": "background.js",
    "persistent": true
  },
}
...
```

Listing 2.3: src attribute specifies the path to the external script in the `popup.html`.

```
<script type="module" src="./dist/popup.js" defer></script>
```

Listing 2.4: The listing presents a code excerpt of the manifest file showing the content script component where `js` array contains the list of Javascript files to be injected into matching pages specified in the `matches` array; in this case all URLs.

```
{
...
"content_scripts": [{
     "matches": ["<all_urls>"],
     "js": ["./dist/contentScript.js", "act.js", "qwPage.js"]
    }
  ]
}
```

2.7.5.2 Chrome Extension Lifecycle

The extension's lifecycle is defined from the moment the extension is executed and ends when it is closed. The life cycle begins when the browser's extension loads, then the manifest is first to be read. The manifest provides permissions to access specific APIs, such as tabs API. Next, the views and scripts are loaded. Finally, the listener functions are assigned to input components to listen for events. The scripts that can listen for events fired from the input components, or other things that happen, represent the extension runtime. These include popup script and background script. Since the popup script is only executed when a popup is opened, the background script, a long-running script in the background, has a significant role since it listens for every event (Mehta, 2016).

2.7.5.3 Messaging API

Scripting components communicate with each other using the messaging APIs provided by the Google Chrome Extensions framework. To make it more straightforward, figure 2.3 represents the possible communications between the components but take into consideration that the extension architectures will vary based on functionality.

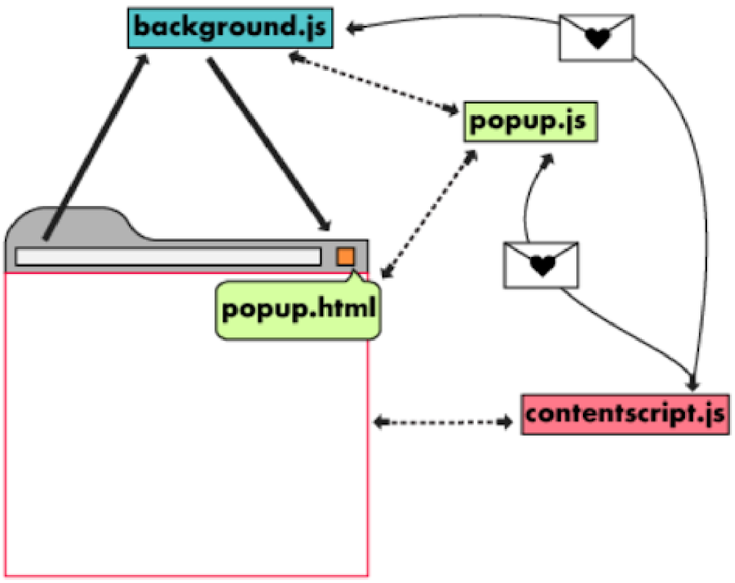


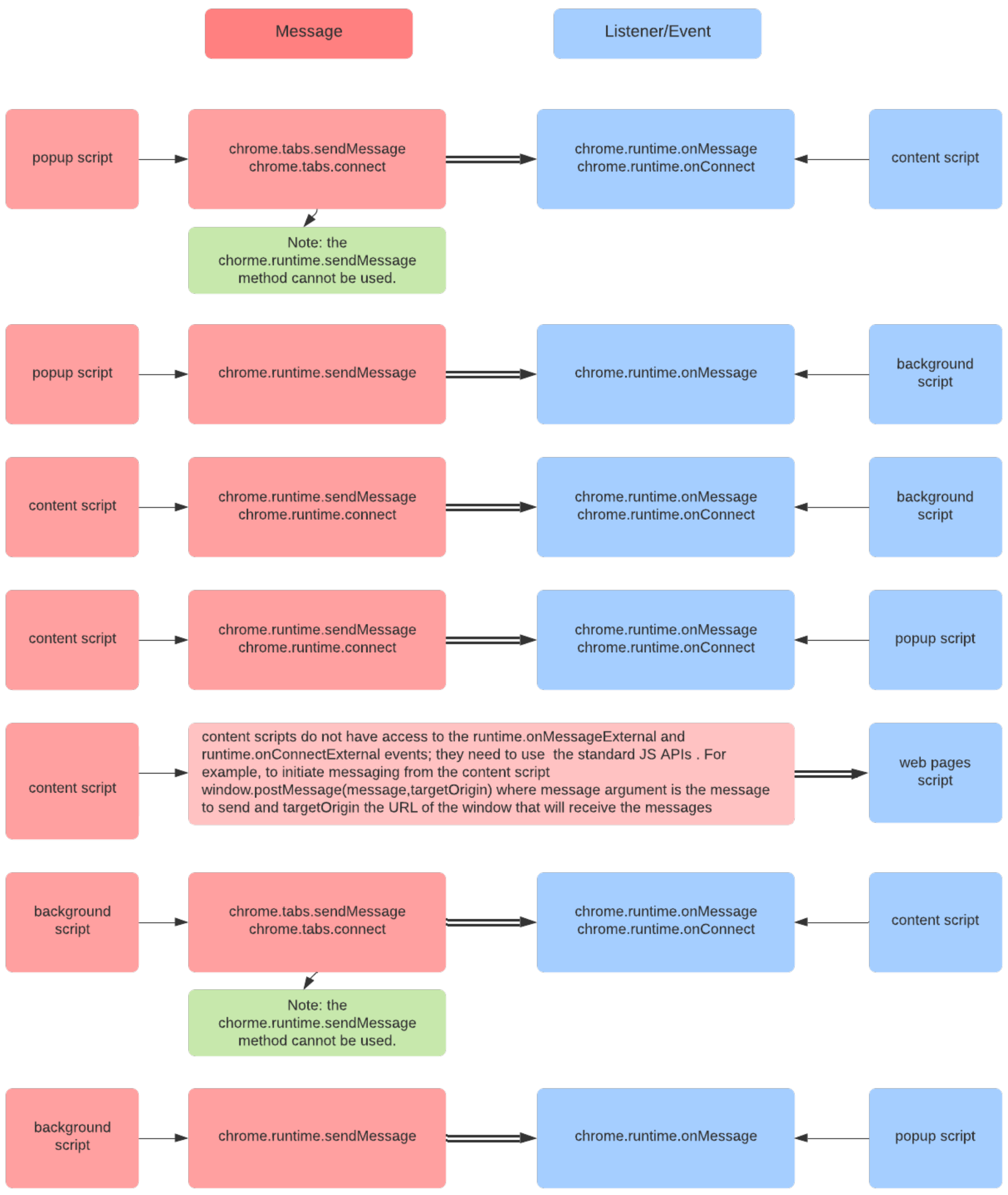
Figure 2.3: Communications representation between different scripts. The content scripts can communicate with the extension by exchanging messages represented by the envelopes symbols. (Google, 2020f)

Figure 2.4 demonstrates the different messages and listeners depending on which script is sending and which one is receiving.

2.8 Portuguese Laws and Obligations

As seen, making a website accessible means making sure as many people can use it as possible. The public sector websites are essential to everyone, and people who need them the most are often the people who find them challenging to use. Public sector websites need to meet accessibility requirements with the Decree-law nº83/2018 (Ministério da Ciência e da Tecnologia e Ensino Superior, 2018) which introduces in the Portuguese legislation the European directive of 2016/2102 about the accessibility of websites and mobile applications of the public sector. This decree states that government websites and mobile web applications need to be more accessible. The AMA (Agência para a Modernização Administrativa) should guarantee that the Government and the equal entities should fulfill the deadlines and the rules demanded in the decree to adapt to the European Union rules. Besides guaranteeing that the government websites comply, AMA should also create the Portuguese Accessibility Observatory. AMA is creating the Portuguese Accessibility Observatory[7] in cooperation with the department of informatics of

[7]https://observatorio.acessibilidade.gov.pt


Message
Listener/Event
popup script
chrome.tabs.sendMessage chrome.tabs.connect
chrome.runtime.onMessage chrome.runtime.onConnect
content script
Note: the chorme.runtime.sendMessage method cannot be used.
popup script
chrome.runtime.sendMessage
chrome.runtime.onMessage
background script
content script
chrome.runtime.sendMessage chrome.runtime.connect
chrome.runtime.onMessage chrome.runtime.onConnect
background script
content script
chrome.runtime.sendMessage chrome.runtime.connect
chrome.runtime.onMessage chrome.runtime.onConnect
popup script
content script
web pages script
background script
chrome.tabs.sendMessage chrome.tabs.connect
chrome.runtime.onMessage chrome.runtime.onConnect
content script
Note: the chorme.runtime.sendMessage method cannot be used.
background script
chrome.runtime.sendMessage
chrome.runtime.onMessage
popup script

content scripts do not have access to the runtime.onMessageExternal and runtime.onConnectExternal events; they need to use the standard JS APIs . For example, to initiate messaging from the content script window.postMessage(message,targetOrigin) where message argument is the message to send and targetOrigin the URL of the window that will receive the messages

Figure 2.4: Schema showing the different communications API between scripts. The red color represents the sending Chrome extension APIs for a message, and the blue color represents the respective message listeners' Chrome Extension APIs. Considering the script which sends and which script receives, some particularities are evident on the notes to take into account.

the Faculty of Science of the University of Lisbon (FCUL). AMA is working in partnership with FCUL in other different projects to develop different tools to evaluate website accessibility. AccessBot is one of those tools in development. The AccessBot prerequisites and usability best practices are implemented in agreement with AMA.

Chapter 3

Study of Existing Accessibility Evaluation Chrome Extensions

This chapter presents an experimental study of eight automated web accessibility evaluation extensions for the Google Chrome browser: aXe Chrome Extension, Tenon Check, Wave Chrome Extension, TotalValidator, ACCESS Assistant Community, Microsoft Accessibility Insights, and ARC Toolkit. These represent the most used tools among developers that can be freely accessed through the Google Web Store.

By using Chrome Extensions the developers are assisted by an automatic evaluation that allows them to visualize the webpage under evaluation. It is considered an advantage when comparing with other online automatic evaluation tools that are not Chrome Extensions. They simplify the work without the need to close the browser.

The initial tool selection criteria included, besides being freely accessible, that they were available as browser extensions. As depicted earlier, a web page is represented not just by its HTML source but also by a set of ancillary resources that are processed by the browser and transform the DOM contents. Using a browser extension to evaluate accessibility, the evaluator can expect that the content that is evaluated is not only what is presented to the user, but also that it is evaluated in the same conditions in which it is consumed by the user (for example, viewport size). Also, these tools might fit better into an expert evaluation workflow by providing support to the evaluator directly in the browser instead of requiring the use of additional applications.

The study centers on evaluating differences in the tools such as: how they are implemented, what success criteria they use, how many tests for each success criteria each one evaluates, the overall differences in the results of the reports generated, if the tools perform only automated tests or if they guide the developers through manual testing. It is essential to understand the impact of automatic evaluations of the accessibility of web pages in the browser environment. Consequently, the study examined the tool's differences, strengths, and limitations according to their evaluation results and considered their usability. The information gathered and its interpretation allows for future improvements

Table 3.1: Classification (0 min, 5 max) and the number of users of automated tools according to Chrome Web Store (data collected on Dec 4, 2019).

Tool	Classification	Users
Lighthouse -accessability	4.5	455,902
WAVE Chrome Extensions	4.2	186,810
aXe Chrome Plugin	4.6	102,196
Accessibility Insights	4.6	34,613
Arc toolkit	4.0	5,897
TotalValidator	2.3	5,239
ACCESS Assistant Community	4.6	3,059
Tenon Check	4.8	2,191
Average	**4.2**	**99,488.38**

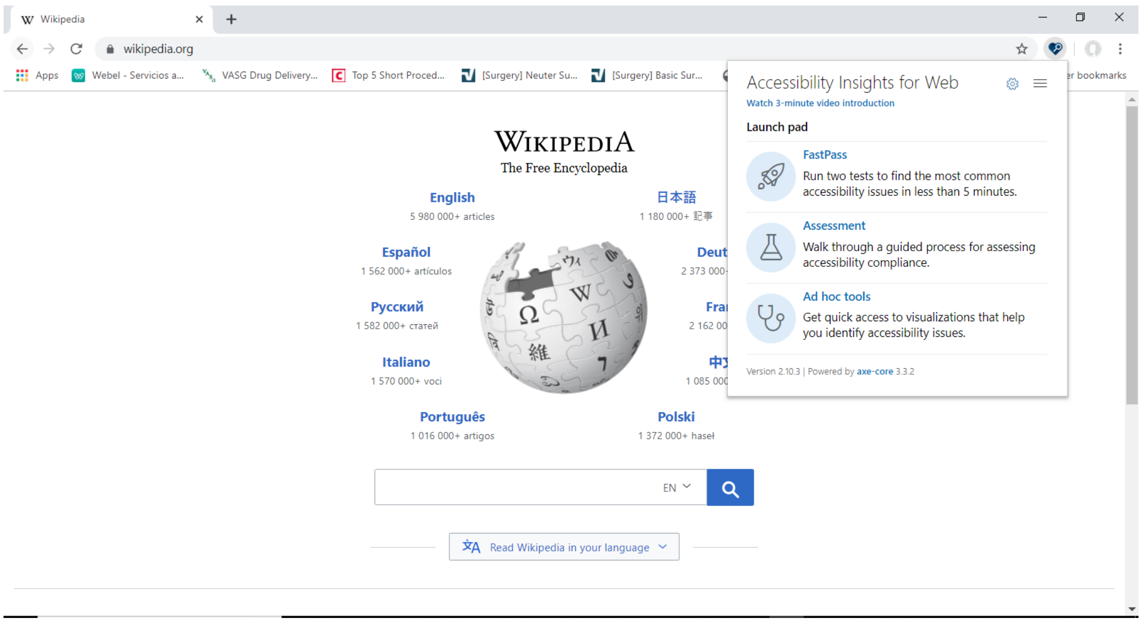
in the development of accessibility evaluation tools.

3.1 Tools

The criteria used to select the automated tools considered the type of tool. The tool needs to be available as an extension for the Google Chrome Browser, although some are also available for Firefox; be available for download in the Chrome Web Store and have a free or open-source license. Since this resulted in an extensive list of tools, the study's tools are the most prevalent among users. The final tool selection list is according to the tool's popularity, classification, and number of users. As can be seen in table 3.1, the most used tool is Lighthouse. However, aXe Chrome Extension has a better score with a larger number of users than the other tools also classified with the same score. The tools are listed in the W3C list of web accessibility evaluation tools (W3C, 2019b; W3C Web Accessibility Initiative, 2006) except for Lighthouse. It is essential to mention that most of these tools have upgrade products available except Microsoft Accessibility Insights, Lighthouse, and Wave. The following paragraphs introduce the selected tools.

3.1.1 Accessibility Insights

Microsoft Accessibility Insights supports two primary workflows, as shown in figure 3.1: FastPass and Assessment. The FastPass workflow helps developers identify common high impact accessibility issues. In FastPass mode, it begins with automated checks, and failures are highlighted directly on the target page. Clicking the failures shows details, including how to fix it and show the DOM errors. The FastPass also has an assisted manual test for tab stops explaining what problems to look for with a visual helper's aid. The second workflow is Assessment. It helps anyone with HTML skills conduct a thorough accessibility evaluation and has approximately 20 manual tests with test instructions. It also has detailed information about dos and don'ts, how to fix, and links to WCAG success criteria, techniques, and expected failures. The automated tests with the manual Assessment provide WCAG 2.0 AA coverage.



Accessibility Insights for Web

Watch 3-minute video introduction

Launch pad

FastPass

Run two tests to find the most common accessibility issues in less than 5 minutes.

Assessment

Walk through a guided process for assessing accessibility compliance.

Ad hoc tools

Get quick access to visualizations that help you identify accessibility issues.

Version 2.10.3 | Powered by axe-core 3.3.2

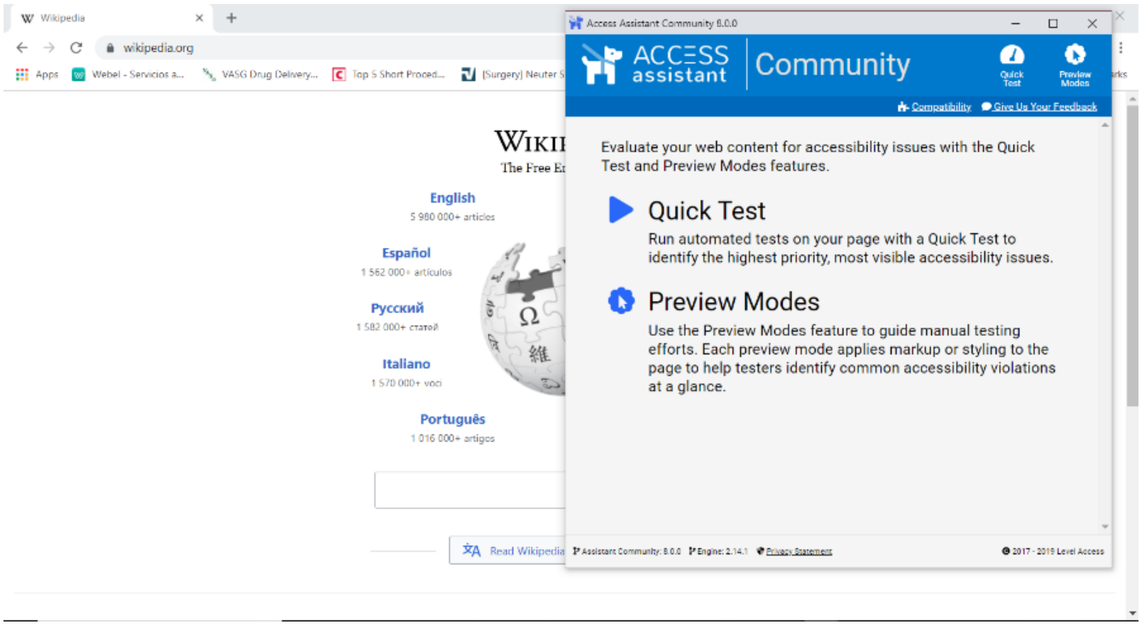
Figure 3.1: Screenshot of Microsoft's Accessibility Insight

3.1.2 ACCESS Assistant Community

With over 100 fully automated accessibility checks, Access Assistant Community (figure 3.2) runs a Quick Test on any web page to view details for all accessibility violations on the page, along with remediation guidance. This extension accesses URLs for all open tabs to enable the testing of pages on any open tab. Preview Modes features a guide for manual testing efforts. Each preview mode applies markup or styling to the page to help testers identify common accessibility violations at a glance.

3.1.3 ARC Toolkit

The ARC Toolkit (figure 3.3) uses the automated ARC Rules. It has a sidebar that shows ARC rules engine assertions organized in categories and sub-categories. Types of results are split into visible and hidden errors and warnings. Hidden are not visible in the browser but may impact assistive technology users, such as options in a menu that only appear when the menu is open. Warnings are potential issues that have been flagged but require manual verification. The test results show code followed by rules and assertions for the issue, along with a brief description and recommendation. Additional features will complement the manual accessibility process. For example, the "show and track focus" checkbox can enhance the visual focus with a bold red outline. It also has a feature for tab order visualization. When selected, the toolkit represents a keyboard user's experiences


Access Assistant Community 8.0.0

ACCESS assistant | Community

Compatibility | Give Us Your Feedback

Evaluate your web content for accessibility issues with the Quick Test and Preview Modes features.

Quick Test

Run automated tests on your page with a Quick Test to identify the highest priority, most visible accessibility issues.

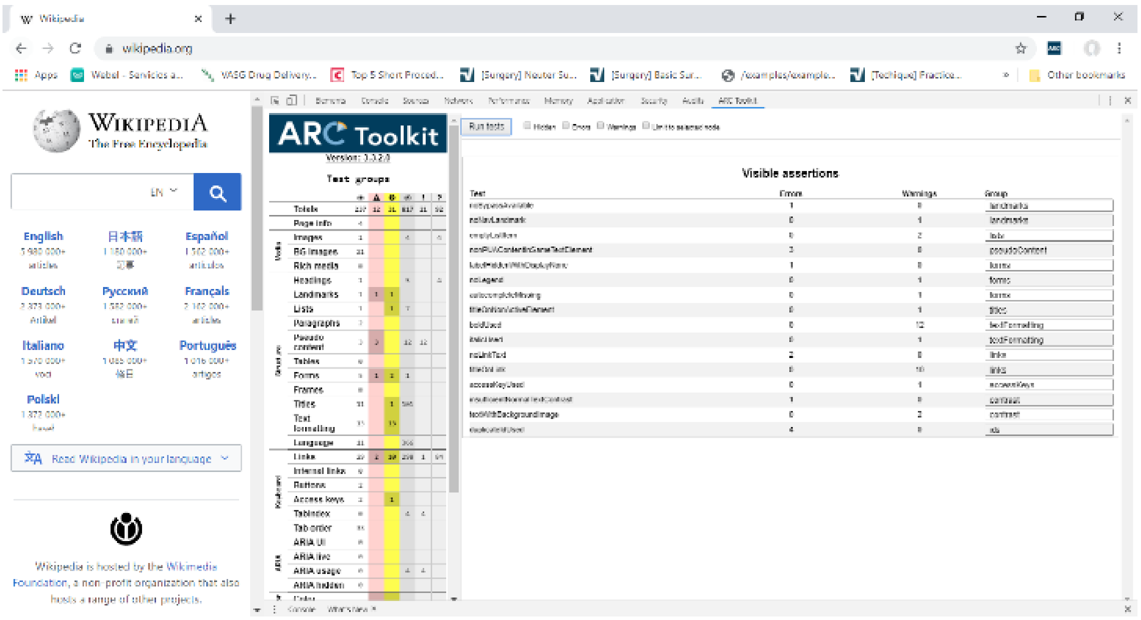
Preview Modes

Use the Preview Modes feature to guide manual testing efforts. Each preview mode applies markup or styling to the page to help testers identify common accessibility violations at a glance.

Assistant Community: 8.0.0 | Engine: 2.14.1 | Privacy Statement | © 2017 - 2019 Level Access

Figure 3.2: Screenshot of ACCESS Assistant Community.

when relying on the TAB key to navigate to each active element. It can submit the DOM or URL to the W3C Markup Validation Service.


ARC Toolkit
Version: 3.3.2.0

Test groups

	#					
Totals	237	12	31	817	31	92
Page info	4					
Images	1			4		4
BG images	31					
Rich media	0					
Headings	1			5		4
Landmarks	1	1	1			
Lists	1		1	7		
Paragraphs	7					
Pseudo content	3	3		12	12	
Tables	0					
Forms	3	1	2	1		
Frames	0					
Titles	31		1	580		
Text formatting	15		15			
Language	31			366		
Links	22	2	10	208	1	61
Internal links	0					
Buttons	1					
Access keys	1		1			
Tabindex	0			4	4	
Tab order	31					
ARIA UI	0					
ARIA live	0					
ARIA usage	0			4	4	
ARIA hidden	0					

Run tests | Hidden | Errors | Warnings | Limit to selected node

Visible assertions

Test	Errors	Warnings	Group
[illegible]	1	0	landmarks
noNavLandmark	0	1	landmarks
emptyListItem	0	2	lists
nonPUAContentInSameTextElement	3	0	pseudoContent
labelHiddenWithDisplayNone	1	0	forms
noLegend	0	1	forms
autocompleteMissing	0	1	forms
titleOnNonActiveElement	0	1	titles
boldUsed	0	12	textFormatting
italicUsed	0	1	textFormatting
noLinkText	2	0	links
titleOnLink	0	10	links
accessKeyUsed	0	1	accessKeys
insufficientNormalTextContrast	1	0	contrast
textWithBackgroundImage	0	2	contrast
duplicateIdUsed	4	0	ids

Figure 3.3: Screenshot of ARC Toolkit

3.1.4 aXe Chrome

aXe Chrome, as shown in figure 3.4, shows the user the accessibility violations and their number of occurrences on the page. Each violation includes the actual description of what is wrong and, if clicked, opens a page from Deque University that contains an even more detailed description. Under the actual issue description, there is also an excerpt of the code causing the issue, information on how it can be fixed, and an indication if it is a critical issue. The highlight option will highlight the offending component on the page.

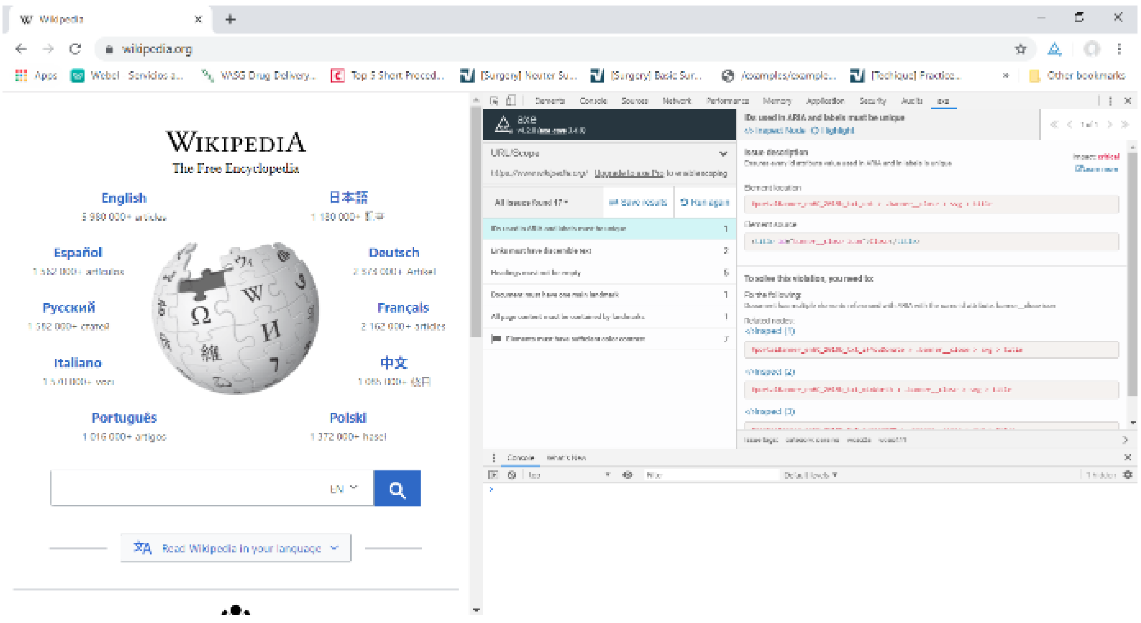


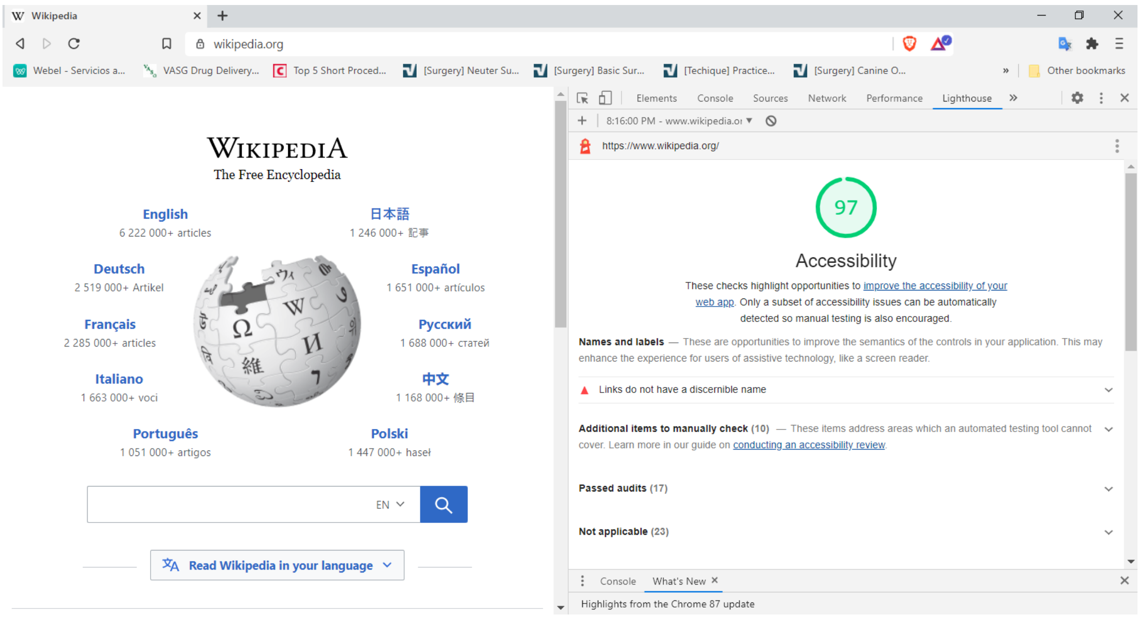
Figure 3.4: Screenshot of aXe Chrome.

3.1.5 Lighthouse

Integrated into Dev Tools of the Chrome browser (figure 3.5), Lighthouse is accessed through the "Audits" option. Lighthouse gives the error results, and after clicking it, it will explain more about the error and how to fix it. It also provides additional items to check manually. These items address areas that an automated testing tool cannot cover. It also informs the user about instances that have passed and also gives information about not applicable tests.

3.1.6 Tenon Check

Tenon Check operates based on an API and requires the user to sign up and obtain an API key and then configure the extension with the API key obtained. After clicking the extension button, Tenon asks the user to sign-in, and after that, the web page is evaluated. The Tenon website then shows a "prettified" version of the JSON received after the request is triggered by clicking the extension button, together with a dashboard with the results (figure 3.6). It presents how many issues were found, the URL, and other information. The left graph on the dashboard shows the number of tests that run and how many passed. The right graph shows the relative percentage of failures. The results show code snippets, which are actual HTML from the DOM. The Tenon pre-defined specifications consider an



These checks highlight opportunities to improve the accessibility of your web app. Only a subset of accessibility issues can be automatically detected so manual testing is also encouraged.

Names and labels — These are opportunities to improve the semantics of the controls in your application. This may enhance the experience for users of assistive technology, like a screen reader.

Links do not have a discernible name

Additional items to manually check (10) — These items address areas which an automated testing tool cannot cover. Learn more in our guide on conducting an accessibility review.

Passed audits (17)

Not applicable (23)

Figure 3.5: Screenshot of Lighthouse.

error as something that has 80% certainty. The priority score also presented is normalized considering other issues of the page. The users can export the results in .csv format.

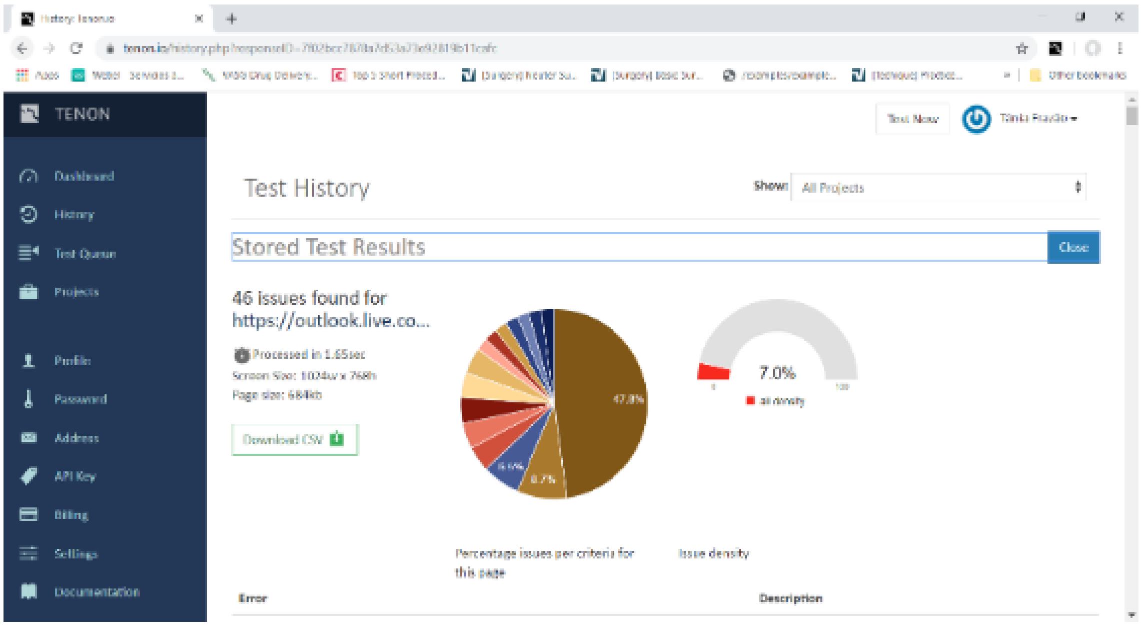
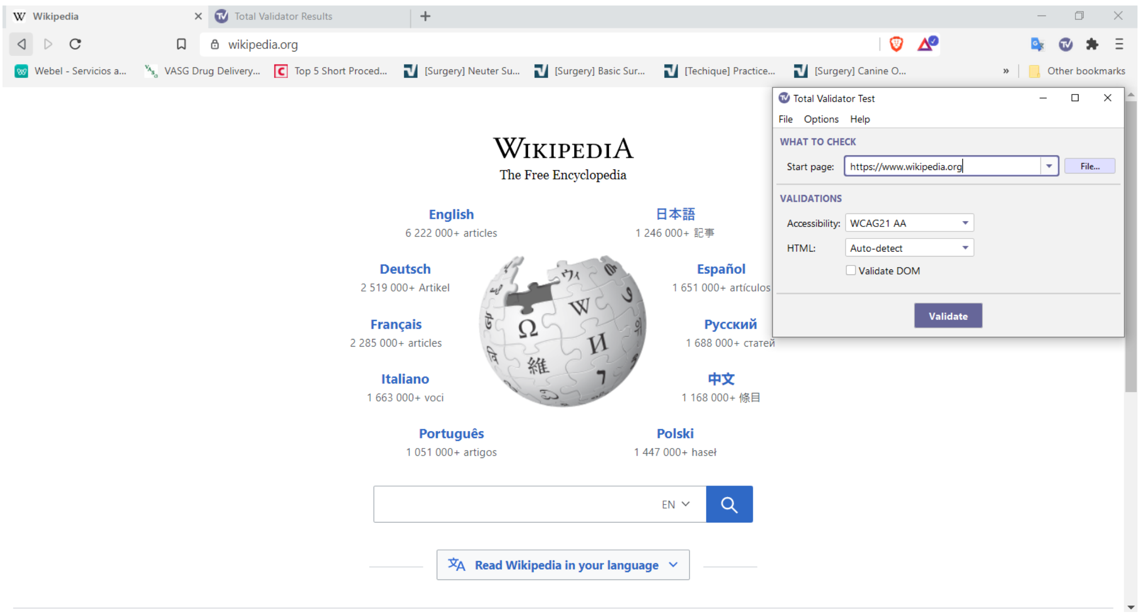


Figure 3.6: Screenshot of Tenon.

3.1.7 TotalValidator

After loading the page, the user clicks on the extension, and it opens the pre-installed Total Validator on the system (figure 3.7). The user has the option to first choose against which guidelines the page should be evaluated. The browser version is just a summary of the issues it finds (total errors and total warnings). In the free version, only the issue's id is shown (a number that identifies the issue in the TotalValidator documentation). Clicking on the id number opens the documentation explaining what the violation is and the success criteria it violates.



Start page: https://www.wikipedia.org

Accessibility: WCAG21 AA

HTML: Auto-detect

Validate DOM

Figure 3.7: Screenshot of TotalValidator.

3.1.8 WAVE

This tool presents the page with embedded icons and indicators using a color system (figure 3.8). Red icons indicate accessibility errors that need to be fixed. Green icons indicate accessibility features – things that probably improve accessibility (though these should be verified). The other icons and indicators, particularly the yellow alert icons, highlight other elements that the user should evaluate. The sidebar has a summary of the errors that were detected, alerts, and structural elements. It also provides manual contrast testing: by changing the color, the tool immediately tells us if it passes contrast checks. There is a "No Styles" option in the sidebar that will disable all original page styles. This option can help pinpoint where errors occur and ensure that the page's reading order is logical. Users can view a brief overview of what each icon means by clicking it and viewing its documentation.

3.2 Selection of Web Sites

A sample of web sites from the Alexa Top Sites list[1] was picked to compare the different tools' coverage. For each website, the page analyzed is the home page. Home pages

[1]https://www.alexa.com/topsites

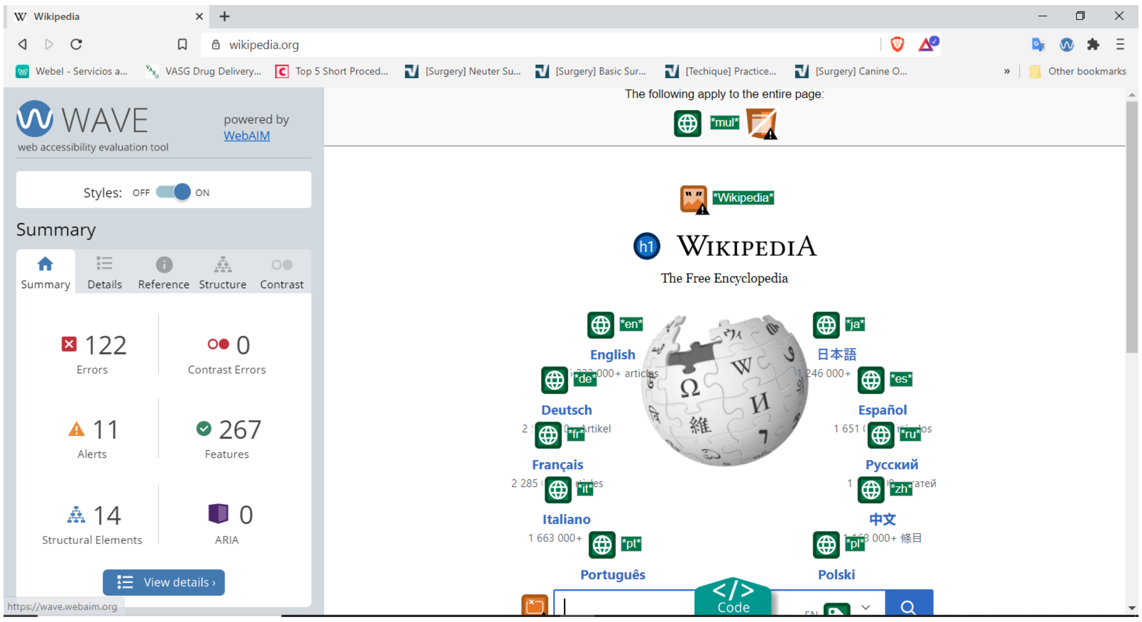


Figure 3.8: Screenshot of WAVE.

Table 3.2: Web sites selected for the evaluation (data collected on Dec 4, 2019).

#	Site	Language
1	`google.com`	English
2	`youtube.com`	English
3	`tmall.com`	Chinese
4	`wikipedia.org`	English
5	`yahoo.com`	English
6	`amazon.com`	English
7	`reddit.com`	English
8	`live.com`	English
9	`netflix.com`	English
10	`blogspot.com`	English

with merely a login page are discarded for the study (for example, Facebook). From the global list mentioned, the study consisted of selecting the first nine web sites using the English language and the first non-English language web site that met the criteria mentioned above. Table 3.2 presents the web sites studied.

3.3 Testing Environment

The testing environment uses a Chrome extension to trigger the execution of the evaluation within the browser. The version of the browser used is Version 78.0.3904.108 (Official Build) (64-bit). In order to have a fair comparison, two requirements were raised:

A All tools should evaluate the same web page. Given the fact that the web sites are dynamic, the evaluation started for all the tools within an interval of a few seconds.

I inspected the loaded pages to ensure there were no differences between them.

B There should be no external interference changing the page's DOM, apart from the testing tool's eventual scripts. There was a necessity to create 8 sandboxes to meet this requirement. In each sandbox, the only installed software was the browser (in the default configuration) with the corresponding extension (plus any additional software required by the extension). For the sandboxes, it was used Sandboxie (version 5.31.6).

3.4 Procedure

Before starting the evaluations, each tool is analyzed for its success criteria, and information is gathered. Then, one of the selected web pages is loaded in all browsers. Each browser contains one of the tools under testing. After the execution of the evaluations from all the tools, their outputs are collected. The remaining web pages repeat the same procedure. Once the collection of all webpages results, analysis is followed according to the accessibility violations found and success criteria coverage.

3.5 Results

3.5.1 Feature Analysis

Tables 3.3 and 3.4 present a comparison of the evaluated tool's features. The collection of the tool's information comes from the W3C list of accessibility evaluation tools, the tool's online website, and the usage to evaluate the sample of web pages. Notice that some tools are directly integrated into development tools (dev tools) of the browser and to access them, the user needs to click Inspect on the dev tools console. In contrast, others activate when the user click the extension button. Only TotalValidator required the pre-installation of software outside the browser environment. When the user starts the TotalValidator extension, it opens the pre-installed application. Some tools use the same engine, like Accessibility Insights and Axe-chrome, which use the aXe engine.

Tools differ in specifications concerning which success criteria they evaluate, how they implement the tests to analyze each success criteria, and how they classify the violation into error. On this topic, it is crucial to notice that one tool, Tenon, distinguishes between errors and warnings based on a predefined threshold percentage of uncertainty about the evaluation result. When the evaluation falls below the threshold, the tool is unsure if it identified a real issue and advises the user to do an additional manual evaluation. Some tools also classify the severity of the violation (for example, from minor to critical). This classification is based on the weight given to each test. The heavier-weighted tests have a more considerable impact on the overall accessibility of the web page. The weights

Table 3.3: Features of evaluated tools.

Automated tools	**Vendor**	**Version (Last Update)**	**Compliance**	**Browsers**	**Supported Formats**	**Types of Support**	**Can Check**	**Engine**
aXeChrome Extensions	Deque Systems	4.1.0 (24th October 2019)	WCAG 2.0, Section 508	Chrome	HTML, XHTML	Generating reports.	Single web pages; Restricted or password protected pages	axe-core
Tenon Check	Tenon LLC	2.0.4(12th October 2019)	WCAG 2.0, Section 508	Chrome	HTML, CSS	Generates report. Ability to export report in csv format.	Single web page and code snippet	tenon
WAVE Chrome Extension	WebAIM.org.	3.0.3 (3rd of November 2019)	WCAG 2.1, WCAG 2.0, Section 508	Chrome, Firefox	CSS, HTML, XHTML, Images	Generating reports; Displaying information within web pages Modifying the presentation of web pages.	Single web pages; Restricted password protected, locally stored, or highly dynamic pages.	wave
TotalValidator	Total Validator	5.0 (5th of April 2019)	WCAG 2.0, WCAG 1.0, Section 508,	Chrome, Firefox	HTML, XHTML	Generating reports of evaluation results	Single web pages, Groups of web pages;, Restricted or password protected pages	TotalValidator
ACCESS Assistant Community	Level Access	7.8.0.344 (31th October 2019)	WCAG 2.1, WCAG 2.0, Section 508	Chrome, Firefox	WAI-ARIA, HTML	Generating reports.	Single web pages, Restricted or password protected pages	Level Access
Accessibility Insights	Microsoft	2.10.3 (28th October 2019)	WCAG 2.1, WCAG 2.0, WCAG 1.0 Level AA.	Chrome, Edge	HTML	Generating reports; Providing step-by-step evaluation guidance; Displaying information within web pages	Single web pages	axe-core
ARC toolkit	The Paciello Group	3.2.0.0 (4th July 2019)	WCAG 2.0, WCAG 2.1, Section 508.	Chrome	WAI-ARIA, CSS, HTML, XHTML, SVG, Images, SMIL	Displaying information within web pages	Single web pages	ARC
Lighthouse - accessibility	Google	5.6.0 (18 October 2019)	WCAG 2.0, WCAG 2.1, Section 508.	Chrome	HTML	Generating reports.	Single web pages, Restricted or password protected pages	axe-core

Table 3.4: Behavior information, and tests of evaluated tools.

Tool	**Behavior**	**How to fix issues**	**Highlights issues in the code**	**Severity Classification**	**Information about the violated check**	**Upgrade version**	**Performs Semi-Automatic Evaluation**	Provide best practices evaluation	Provide manual evaluation
aXe Chrome Extension	Integrated in developer tools	Yes	Yes	Yes	Yes (Plus Other Resources)	Yes	Yes	Yes	No
Tenon Check	Displaying information within Tenon web app	Yes	Yes	Yes	Yes	Yes	No (If Using Only Extensions); Yes (If Using The Tenon)	No	No
WAVE Chrome Extension	Click on the extension	Yes	No	No	Yes	No	Yes	Yes	Yes
TotalValidator	Click on the extension to open Total-Validator desktop app.	No. (Redirects To W3c)	No	No	Yes	Yes	Yes	Yes	No
ACCESS Assistant Community	Clicking the extension to test all open tabs.	Yes	Yes	No	Yes	Yes	No	No	Yes
Accessibility Insights	Integrated in developer tools	Yes	Yes	No	Yes	No	Yes	No	Yes
ARC toolkit	Integrated in developer tools	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Lighthouse - accessibility	Integrated in developer tools	Yes	Yes	No	Yes	No	No	Yes	Yes

are based on heuristics defined by the tool vendors. When tests identify a problem, some tools help developers correct it by giving extra information on how to solve it.

A few tools also give information about instances that are considered as “Pass”, meaning that the website passed the test because the instances are present and correct. For some tools, like Accessibility Insights, they state that “Pass” also means there are “No matching/failing” instances, so the absence of a feature is considered a “Pass” in the results.

Besides automatic tests, tools may provide a semi-automatic evaluation where the tests identify elements or conditions to which success criteria apply but cannot evaluate if it is a “pass” or a “fail”. In these instances, tools indicate the instance is present and ask the user to check whether there is a violation of the success criteria.

Another category of tests is the manual checks. These are not tests made by the tools, but instead, are instructions to assist their users in conducting the evaluation, for example, to verify the site with keyboard-only navigation. The tools that provide manual procedures are Accessibility Insights, ARC Toolkit, Access Community, Lighthouse, and Wave. The procedures range from manual contrast evaluation in Wave to more elaborated and guided procedures in Accessibility Insights.

Some tools also provide “Best Practice” tests. They are not checks of success criteria but are tests to identify valid and “well-formed” HTML and CSS code. When the code is not up to the standard, the tools give suggestions on modifying it. Standards abiding code is halfway to implement the WCAG standards and design accessible websites (Hudson,

Table 3.5: Number of checks of each tool by the level of conformance.

Tool	A	AA	AAA	Total
aXe Chrome Plugin	35	7	2	44
Tenon Check	103	5	23	131
Wave Chrome Extension	152	28	0	180
TotalValidator	108	11	20	139
ACCESS Assistant Community	215	37	0	252
Accessibility Insights	55	7	2	64
ARC toolkit	136	9	0	145
Lighthouse -accessibility	39	4	4	47

2011).

All tools have unlimited use, except Tenon, limiting the number of requests by month to the API, besides being the only tool requiring the user to sign up and obtain an API key. All the tools, except TotalValidator, do not provide information on how well they support accessibility. TotalValidator informs that the creators regularly test if the tool can be used entirely by keyboard and the results generated are accessible. It also states to support screen readers.

Table 3.5 presents the number of tests each tool implements grouped according to the WCAG level of conformance. This information was gathered from the documentation available for each tool online. It is possible to analyze that ACCESS Assistant Community, ARC toolkit, and WAVE do not evaluate AAA success criteria. However, for A and AA levels, ACCESS Assistant Community provides a higher number of checks.

It is also essential to examine what the checks validate. Figure 3.9 presents a chart displaying the number of success criteria for each tool test. It is interesting to note if a user employs all tools, the total of individual success criteria tested is 62, still well below the total of 78 success criteria in WCAG. The previous observation is an example of the problems of over-reliance on automated tools.

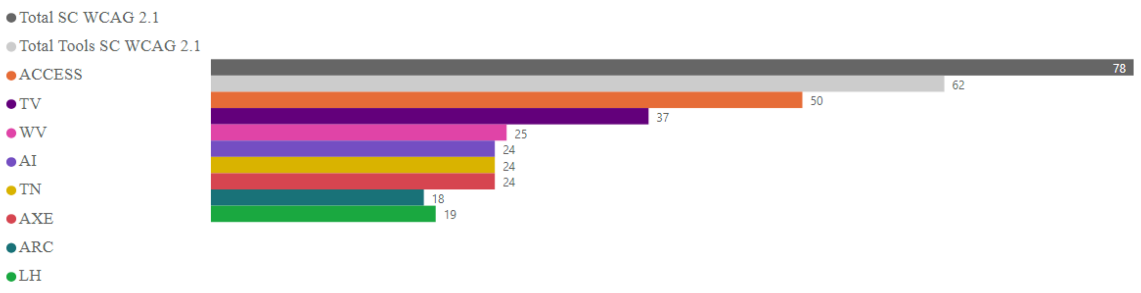


Figure 3.9: Number of success criteria tested by the tools.

The study analyzed which success criteria were overall most checked. Figure 3.10 presents a treemap where the areas are related to the number of checks across all tools that test that success criterion. Only success criteria with over 15 checks were included in the treemap in order to improve legibility.

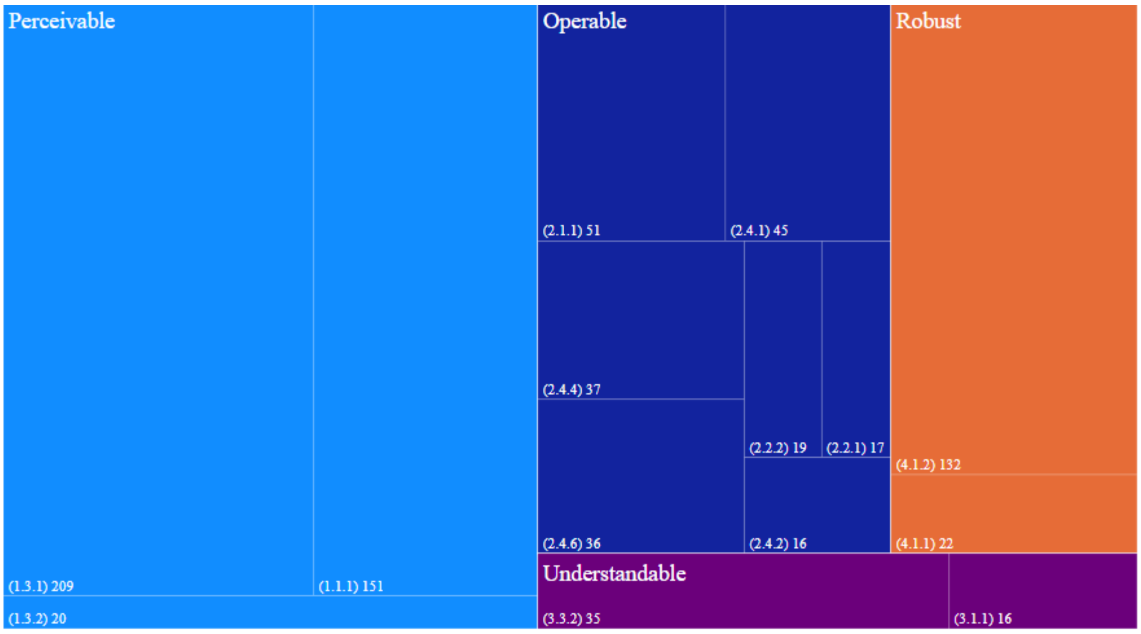


Figure 3.10: Number of checks by success criteria.

Most tested success criteria belong to the Perceivable principle. The other most tested principles are Operable, Robust, and Understandable by this order. Three success criteria stand out by the number of checks related to them: 1.3.1 Info and Relationships, 1.1.1 Non-text Content, and 4.1.2 Name, Role, Value. Perceivable is the principle with most checks only because of 1.3.1 and 1.1.1. In addition to those two, it has only one more success criterion with more than 15 checks. The Operable principle coverage is more uniform, with seven success criteria, each one between 16 and 51 checks. Both the Understandable and Robust principles have only two success criteria with more than 15 checks. For the Robust principle, this could be expected since it has only three success criteria (one of those was only introduced in WCAG 2.1, which is not covered by all tools). However, the Understandable principle has 17 success criteria, hinting at the difficulty of automating these success criteria. A similar argument can be made for the Perceivable principle that has 29 success criteria with only 3 having more than 15 checks and for the Operable principle with 29 success criteria also, although for this one the situation is not as pronounced.

3.5.2 Usability Analysis

In the following paragraphs, the strengths and limitations of each tool from my usability perspective are discussed.

3.5.2.1 Accessibility Insights

Strengths: Easy to use; Excellent user interface; Explanations about errors excellent and easy to follow; Manual testing procedures with visual help present; Good coverage of automated tests and approximately 20 manual guided tests.

Limitations: Sometimes, the tool crashed, and it was necessary to do a browser refresh.

3.5.2.2 aXe Chrome Extension

Strengths: Easy access to success criteria documentation; has good material about what is an issue and how to fix it; gives the possibility to save results; good, intuitive user interface; if tests are semi-automatic, asks and guides the user to evaluate the result.

Limitations: The extension stopped working a few times, especially when testing pages dense in information.

3.5.2.3 ARC Toolkit

Strengths: Easy access to success criteria documentation; Includes easy tests for focus order and new WCAG 2.1 success criteria Reflow and Text spacing and buttons to validate the code through the W3C Nu HTML validator.

Limitations: The user interface can be difficult to understand initially.

3.5.2.4 Tenon

Strengths: An id identified the errors; Possible to export results in CSV format; Good documentation available about how to use and success criteria evaluation.

Limitations: Necessary configuration of API password previously to use; when clicking on the browser extension button, it opens multiple browser windows with the reports instead of only one; results not aggregated by error.

3.5.2.5 WAVE

Strengths: Easy access to success criteria documentation; Simple results report; Possibility to turn-off CSS of the page; Direct contrast evaluation where user can manipulate colors and check for contrast directly with tool.

Limitations: Uses many icons on the page to help users locate instances, however with pages with many errors, this can be very confusing; Most of the information presented are alerts that there is content that can be a possible accessibility issue which it cannot evaluate automatically and needs user evaluation which is time-consuming.

3.5.2.6 TotalValidator

Strengths: Easy access to success criteria documentation; Possibility for a user to validate HTML; very concise information about errors and success criteria; Provides also information about best practices.

Limitations: Difficult to work with the extension because it needs to pre-install an application on the computer; Often, the extension, when clicked, cannot connect to the app to start evaluating the web page; Does not show the failing instances in code or how to solve it since it is a free version.

3.5.2.7 ACCESS Assistant Community

Strengths: Easy user interface; Provides manual testing guidance.

Limitations: Very difficult to access the documentation for the extension and success criteria; Does not show the count of instances failing per error.

3.5.2.8 Lighthouse

Strengths: Easy access to success criteria documentation; Easy to use; Generates a report that gives information on all of the tests that passed in addition to the ones that failed.

Limitations: Does not show the count of failing instances per error.

3.5.3 Web Pages Evaluation Analysis

The analysis begins by verifying the differences between the errors identified by each tool in the web pages evaluated. The failed checks are grouped by their success criteria to facilitate the analysis. The following charts present the results. The charts use logarithmic scales to accommodate the broad range of errors found.

Figure 3.11 presents the number of errors found by Accessibility Insights in all web pages evaluated, grouped by success criterion. Most errors were found on success criteria 4.1.1, 1.4.3, and 4.1.2. The errors mostly address the Robust and Perceivable principles and level A of conformance in what concerns principles and conformance levels. The web pages with more errors found by Accessibility Insights were youtube.com, reddit.com, and tmall.com.

Figure 3.12 presents the number of errors found by the ACCESS Assistant Community in all web pages evaluated, grouped by success criterion. The success criterion with more errors was 4.1.1, mainly due to the evaluation of youtube.com. The success criteria 1.1.1 and 3.2.2 are the following criteria with more errors found on all pages. The principle violated more frequently is Robust, with level A conformance being dominant in the criteria violated. The web page with more errors found was youtube.com.

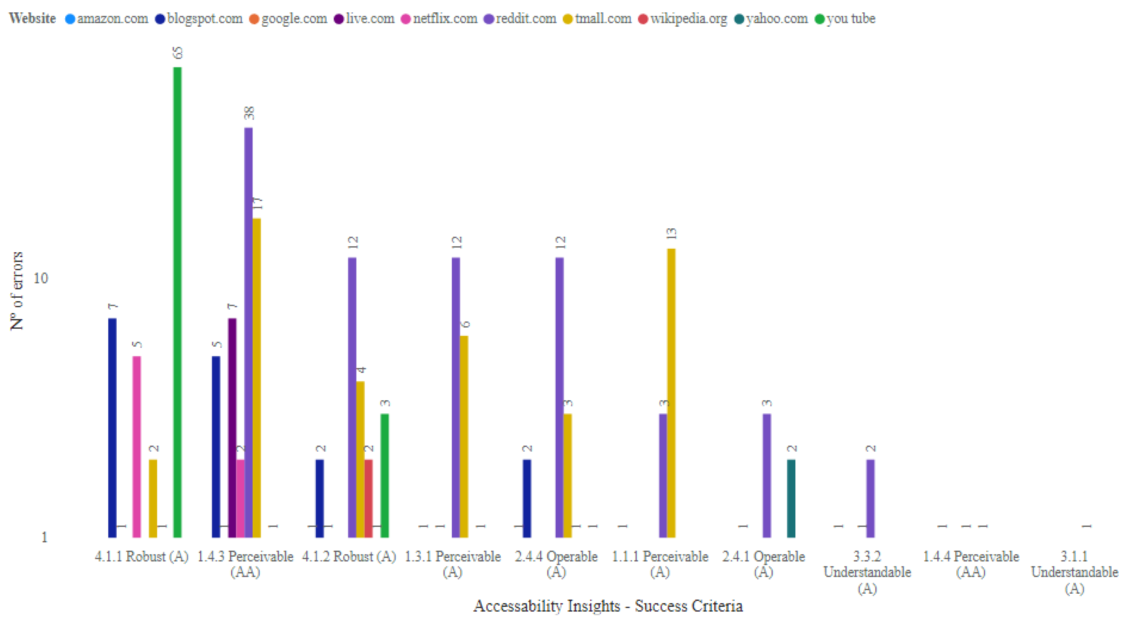


Figure 3.11: Number of errors found by Accessibility Insights.

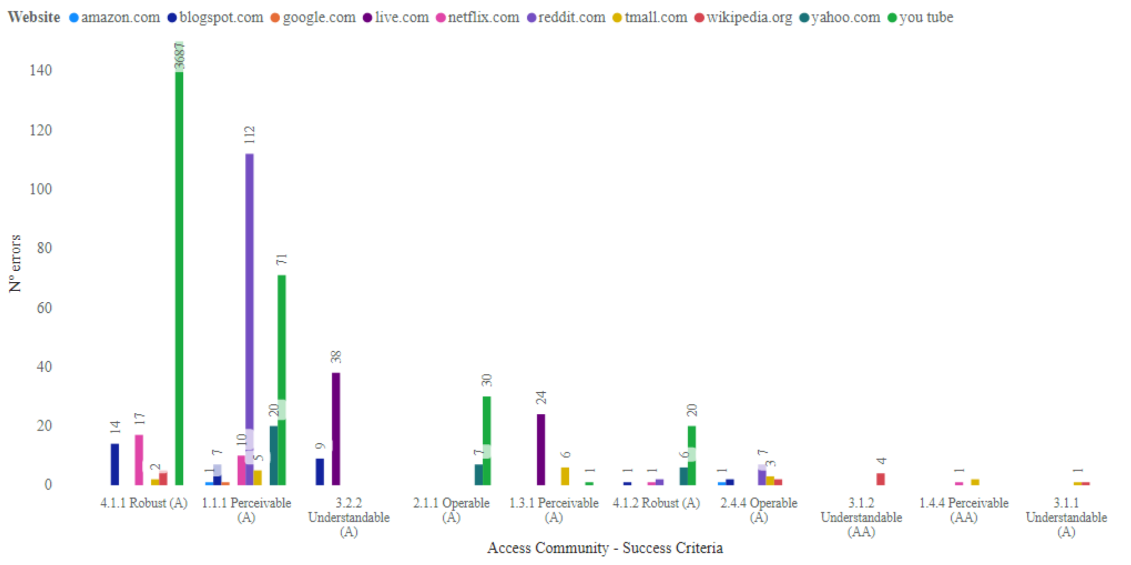


Figure 3.12: Number of errors found by ACCESS Assistant Community.

Figure 3.13 presents the number of errors found by ARC Toolkit in all web pages evaluated, grouped by success criterion. The success criteria with more errors are 1.4.3, 1.1.1, and 4.1.2. Although youtube.com is present with many errors, in the ten websites, ARC found 1.4.3 as the most common violation with more incidence for blogspot.com and amazon.com. For ARC Toolkit, Perceivable is the principle more frequently violated with several level A and AA criteria violated.

Figure 3.14 presents the number of errors found by aXe Chrome in all web pages evaluated, grouped by success criterion. For aXe Chrome, the most frequent violation happens for success criteria 4.1.1, 4.1.2, 1.1.1. Robust is the principle, and conformance level A

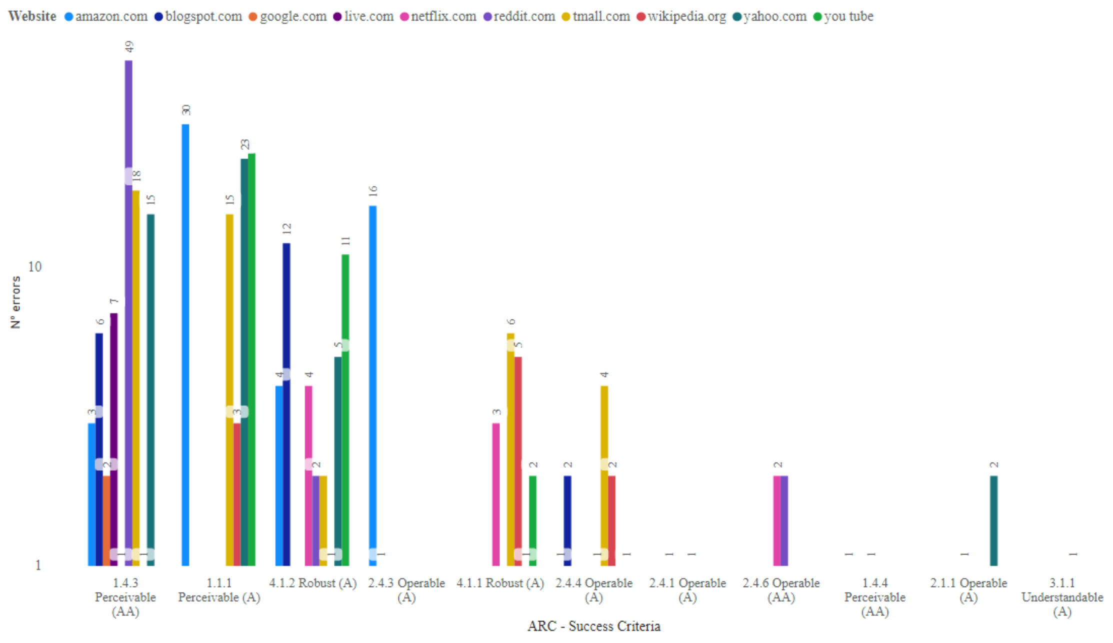


Figure 3.13: Number of errors found by ARC Toolkit.

success criteria are the most violated. The websites with more errors are youtube.com, tmall.com, and reddit.com. aXe Chrome and Accessibility Insights use the same engine. Even though there are small differences, the results are similar.

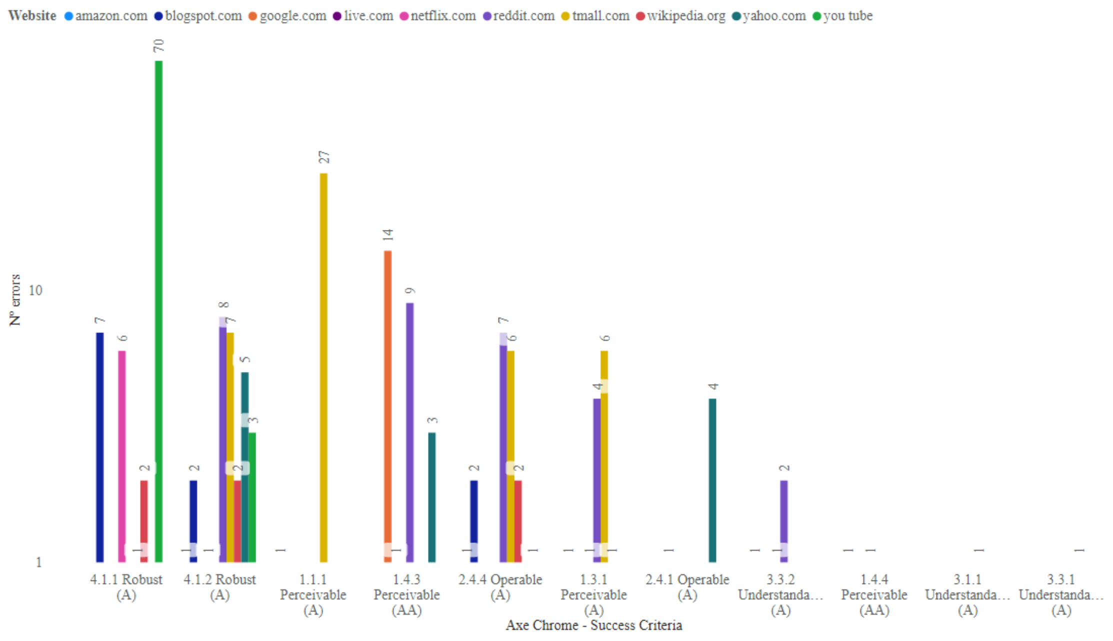


Figure 3.14: Number of errors found by aXe Chrome.

Figure 3.15 presents the number of errors found by Lighthouse in all web pages evaluated, grouped by success criterion. Lighthouse also uses the same aXe-core engine. As aforementioned, when tools use the same engine, the implementation and heuristics can be slightly different, but the results tend to be similar. In this case, the criteria found to have more errors are very similar to Accessibility Insights: 4.1.1, 1.4.3, and 4.1.2. The web pages evaluated with more errors are youtube.com, reddit.com, and tmall.com. Robust is the principle more violated and A the conformance level.

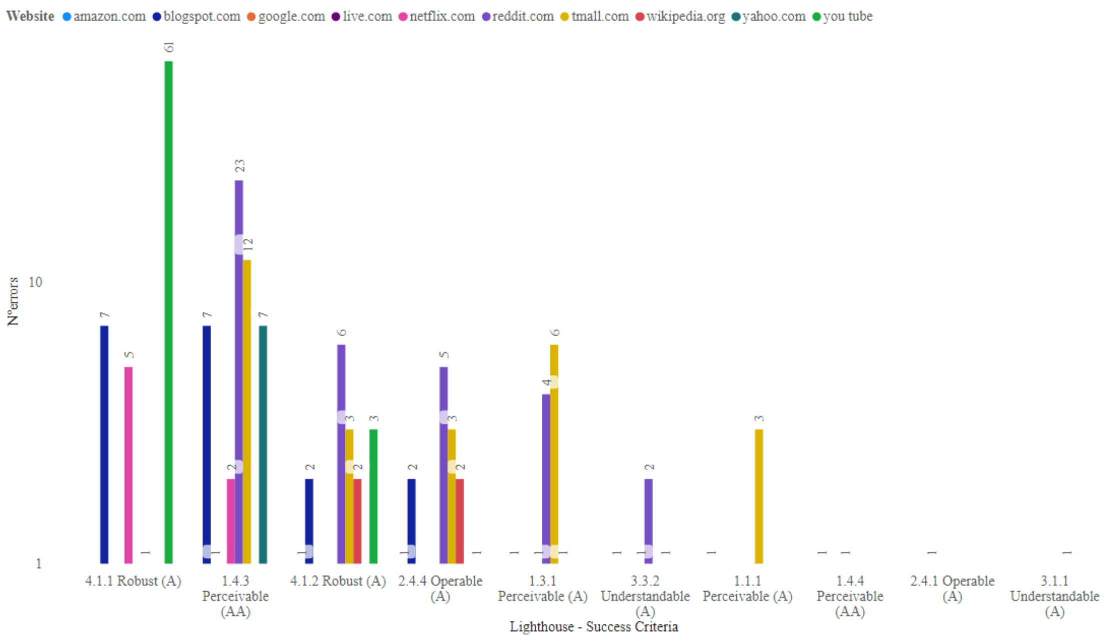


Figure 3.15: Number of errors found by Lighthouse.

Figure 3.16 presents the number of errors found by Tenon in all web pages evaluated, grouped by success criterion. The success criterion with most errors found by this tool was 1.1.1 followed by 2.4.9 and 2.4.4. The principles most violated were Perceivable and Operable, and the conformance level with more errors is A. Tenon evaluates more frequently conformance level AAA than the other tools. For Tenon, youtube.com is the website with more errors, followed by yahoo.com and live.com.

Figure 3.17 presents the number of errors found by TotalValidator in all web pages evaluated, grouped by success criterion. The success criterion with more errors detected by the tool was 1.4.3 followed by 2.4.6 and 1.1.1. The website with more errors is reddit.com, followed by youtube.com and amazon.com. More errors were found for conformance level AA than A. The principles violated more frequently were Perceivable and Operable.

Figure 3.18 presents the number of errors found by WAVE in all web pages evaluated, grouped by success criterion. The most prevalent errors are in success criteria 1.4.3, 1.1.1, and 2.4.4. The web pages evaluated by this tool with more errors are reddit.com, youtube.com, and tmall.com. The principle most often violated is Perceivable, and the level conformance most violated is A. It is important to recall that WAVE does not perform level AAA evaluations.

By inspecting the numbers of violations on all pages evaluated (table 3.6), a tendency of the current state of automated accessibility evaluation tools can be inferred. The con-

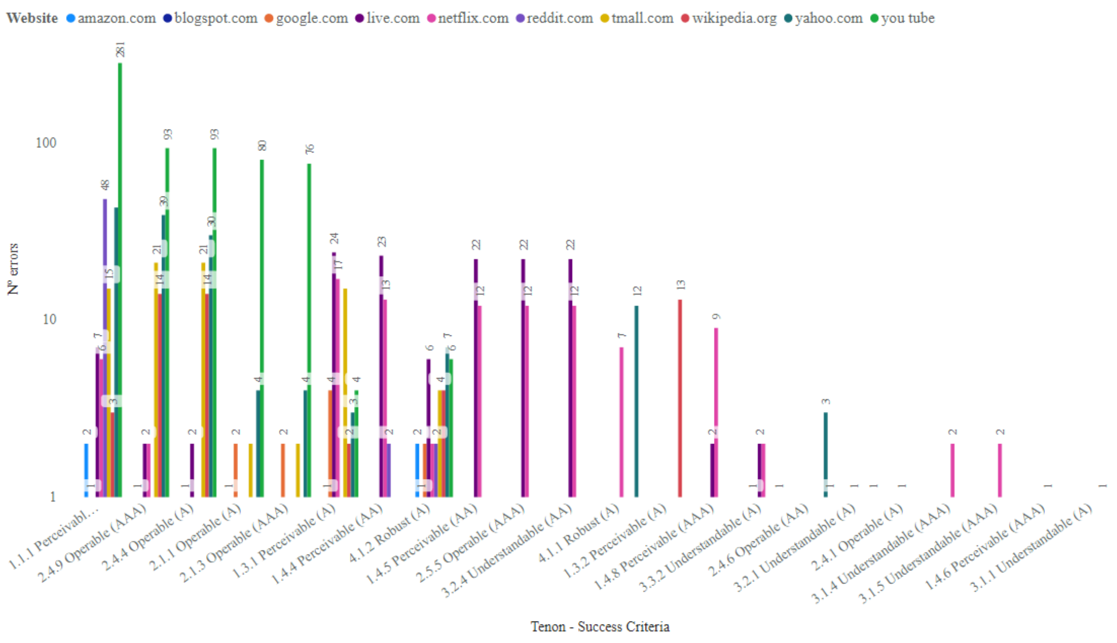


Figure 3.16: Number of errors found by Tenon

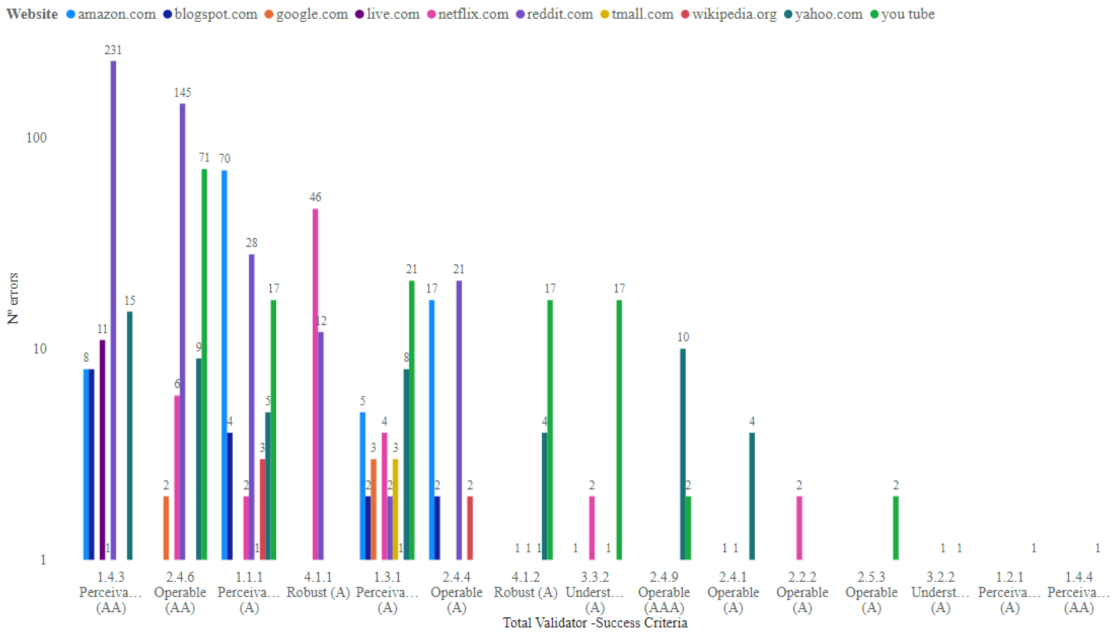


Figure 3.17: Number of errors found by TotalValidator.

formance level A is the one with most violations found, corresponding to a higher number of checks targeting this level's success criteria. On what concerns principles, Robust emerges as the one with more violations found. However, this is a direct result of a single page/tool combination. If the violations from ACCESS Assistant Community found on youtube.com are removed, then Perceivable becomes the more violated principle followed by Operable. The low number of violations in the Understandable principle should be highlighted due to the low number of checks in this principle.

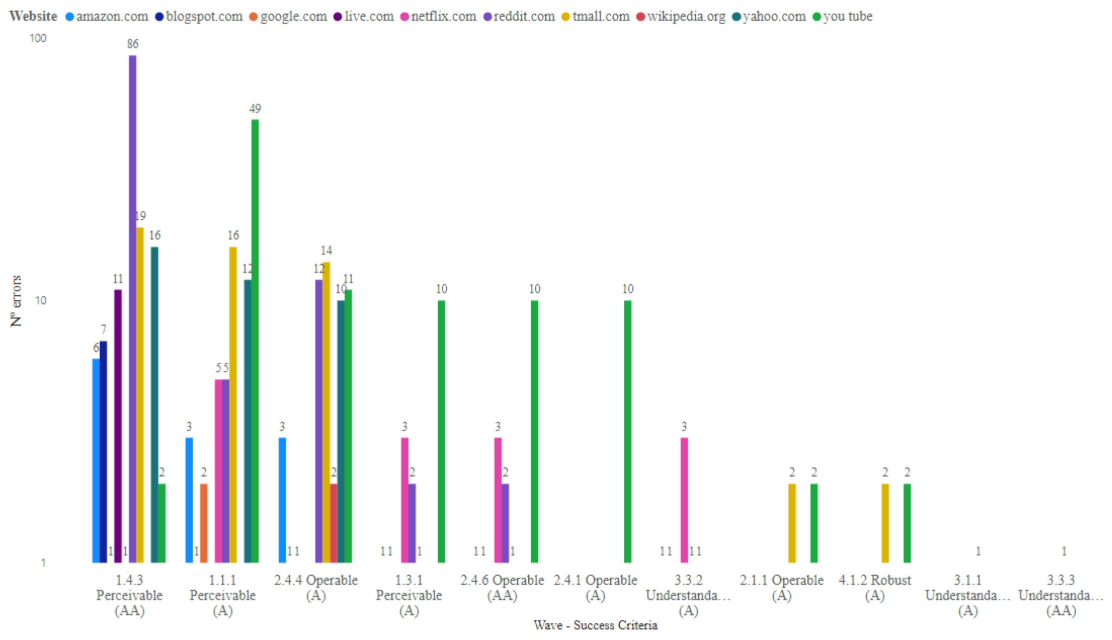


Figure 3.18: Number of errors found by WAVE.

Figure 3.19 provides an overview of what success criteria are most violated across all the pages tested, taking into account all tools' results. This, together with each tool's outcomes, paints a picture of the current coverage of automated accessibility evaluation tools.

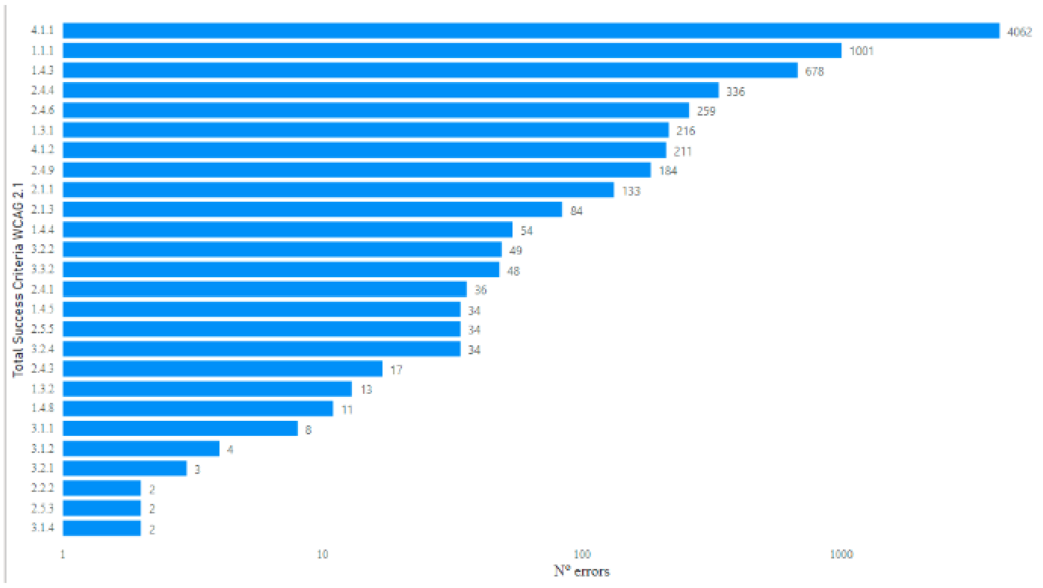


Figure 3.19: Number of violations found by all tools grouped by success criteria.

Analyzing the success criteria evaluated by tools, some criteria were unique to some tools. The unique success criteria are 3.1.2 and 3.2.2 for ACCESS Assistant Community;

Table 3.6: Number of violations reported by each tool across all pages tested, grouped by conformance level and principle (P - Perceivable; O - Operable; U - Understandable; R - Robust).

	Conformance			Principle			
	A	AA	AAA	P	O	U	R
Accessibility Insights	179	75		114	26	5	109
ACCESS	4114	7		261	52	53	3755
ARC	190	110		201	40	1	58
aXe	187	30		71	24	6	116
Lighthouse	131	56		73	15	6	93
Tenon	808	110	306	574	546	49	55
TotalValidator	339	508	12	455	298	23	83
Total	**5948**	**896**	**318**	**1749**	**1001**	**143**	**4269**

2.4.3 for ARC Toolkit; 3.3.1 for aXe Chrome; 2.1.3, 3.2.4 and 2.5.5 for Tenon; 2.5.3, 2.2.2 and 1.2.1 for TotalValidator; and 3.3.3 for WAVE. Lighthouse and Accessibility Insights do not have any unique success criteria as expected since they share the same engine (aXe Chrome probably as a different version of the engine, justifying its unique success criteria). On the other side of the spectrum, the success criteria that appear in every tool are 1.1.1, 4.1.2, and 2.4.4. Success criteria 1.3.1 and 2.4.4 appear in all but one of the tools. The success criteria with more errors are 4.1.1, 1.4.3, and 1.1.1. However, different tools find more violations in different sets of success criteria. One distinguishing factor between tools seems to be their level of support for the success criteria of the "Robust" principle. One group of tools composed by Accessibility Insights, aXe Chrome, Lighthouse (the three sharing the same engine), and ACCESS Assistant Community edition all find several errors for criteria 4.1.1. The remaining tools, ARC Toolkit, Tenon, TotalValidator, and WAVE, find a much smaller number of violations for the success criteria 4.1.1. Three of these tools (ARC, TotalValidator, and Wave) find more for 1.4.3, and Tenon finds more errors for 1.1.1.

Only five of the eight tools presented had results specifying warnings, and these were Accessibility Insights, Axe, ARC Toolkit, Wave, TotalValidator. Warnings are tests that only identify the instance but request manual testing by the user. The success criteria with more warnings are 4.1.1, 4.1.2, and 2.4.1. The conformance level with the higher number of warnings is A with a total warning of 6263; AA had only 1170 and 2 for AAA. With more warnings, the WCAG principle is Robust with 2672 in total, followed by operable with 2524, perceivable with 2177, and understandable with 62.

3.6 Discussion

There have been discussions and arguments about how to measure the accessibility of websites (Brajnik et al., 2007). Ideally, any thorough accessibility evaluation should involve automatic and manual approaches, but that is not always possible due to different constraints (for example, time, cost, expertise). Automatic approaches use automatic tools. Some of these are available for the browser. After triggering the evaluation, the tool evaluates the web page and generates a report. An ideal report flags accessibility issues, describe them, and gives guidance on how to fix them.

In this experimental study, accessibility tools freely available as browser extensions were analyzed. Browser extensions are easily accessible and do not require the installation of additional software. For that reason, it is paramount that these extensions are easy to use since they can be found and installed by users without previous accessibility expertise. Such tools facilitate developers and the general public's workflow, for example, government bodies that need to verify if public websites are corresponding to the mandatory accessibility directives.

Overall, the tools were easy to install and use but not flawless since during evaluations, they crashed, and it was necessary to refresh the browser. In these instances, care was taken to check if the reloaded page was still unchanged compared to the pages evaluated by the other tools. However, of the tested chrome extensions, the one that demanded the user to install an application in addition to the chrome extension was TotalValidator. The results obtained between tools varied in what success criteria they evaluate (for example, WAVE does not evaluate level AAA conformance criteria), varied in classifying the results according to its impact on the user, and classifying the result as an error or a warning. All these variances are a consequence of how specifications and heuristics are implemented for each tool. This study analyzed the number of errors found per success criteria by the different tools. Although there were some differences in success criteria evaluated between tools, some similarities in the results could be found, with 4.1.1 being the most violated success criteria. The analysis showed that individual tools have low coverage of the WCAG 2.1 success criteria. If all are used together, the coverage increases, affording approximately 10% to 40% more coverage than using only one tool. To improve the evaluation performance, developers should fill the gaps between tools using more than one. There are tools, for example, TotalValidator or ARC Toolkit who can perform, besides accessibility evaluation related to success criteria, a validation of the HTML code evaluation in order to leverage the evaluation. A well-written code (for example, abiding by standards) is halfway to meet accessibility requirements also. Considering the usability issues found during this study and the results of the tools' evaluations, users may struggle with accessibility evaluations because of the inherent variance between implementations, how to use the tools, and how to interpret the results. Given that the best option seems to be to use more than one tool to improve accessibility and adherence to developing accessible websites, there should be more consensus on how tools are developed, concepts defined, and results presented. In what regards concepts, semi-automatic tests need human intervention. However, this can be due to tools only being able to detect an instance of the tested elements without being able to evaluate if the element meets the criteria (for example, Wave); or a tool may need human intervention because it detects a failing instance but with an uncertainty factor too high to be sure about the correctness of its decision (for example, Tenon). Another example is pass or fail: some tools consider a pass even when the instance is absent (for example, Accessibility Insights). Others, like Lighthouse, consider a pass only if the instance is present and passed the test. In this situation, the other tests that were not performed because the instance is absent are considered "not applicable". All these differences in concepts impact how results are presented and interpreted. Overall, automatic tools available as browser extensions allow users to assess accessibility quickly. Although not perfect and with limitations, they remain essential to help users evaluate websites to find most of those violations of success criteria that can be automated. However, they should always be complemented with manual testing procedures, and the results need to be analyzed objectively with reasoned judgment.

Chapter 4

Design of AccessBot

AccessBot is an extension for Chrome that helps users find and fix accessibility issues in web pages. This chapter introduces AccessBot and presents its features. Since QualWeb is the engine behind AccessBot, an important step is to integrate AccessBot with QualWeb. After this section, and taking into account AccessBot's features, the architecture of AccessBot is described in detail. Since AccessBot performs semi-automatic and manual evaluations, it is necessary to design the algorithms for these tests to complement the automatic evaluation tests. These are also presented in this chapter, along with the automatic tests.

4.1 AccessBot Features

AccessBot allows users to verify that a web page is compliant with WCAG. The tool supports the ability for the user to choose between three types of accessibility evaluations. The user can pre-select before starting an evaluation, an automatic, a semi-automatic, and manual evaluation. AccessBot allows the user to select all types at the same time or a specific type. QualWeb is the engine that performs automatic evaluation where the tool automatically checks for compliance. The semi-automatic and manual evaluations need user input in order to achieve a final result. To accomplish this, AccessBot provides exact questions to orientate the user in identifying critical accessibility issues with additional rule descriptions and test instructions. After all questions for a test have been answered, AccessBot notifies the user with a final result.

Currently, the number of semi-automatic rules is twenty-four, and the number of manual rules is two. More information about the rules is located at the end of this chapter. It is essential to point out that there are rules with automatic tests and semi-automatic tests simultaneously.

After clicking on the button start evaluation, a result popup window provides the user with a result analysis dashboard.

The rules are aggregated in categories to be easier to identify. AccessBot also allows the user to filter the rules according to the result of all the rules; for example, list only rules that have passed or failed. The user also has the information about what type of rule is, if it is automatic, semi-automatic, or manual. Inside each rule, the user can filter tests. The filter shows the rules with tests that have passed, for example.

For all the evaluations' results, AccessBot provides a visual helper that identifies the element that is being evaluated quickly and more straightforward. The visual helper appears as a red square around the element on the web page that is being evaluated. If there is more than one element highlighted on the evaluated web page, there is a feature to remove all highlights using the "Remove highlights" button.

For every evaluation, even for the automatic tests, the user can manually change the outcome of the test and can add an observation for a test result. These changes are automatically saved. After an evaluation, the user can also insert the evaluator's name and save the results by exporting them in two formats, EARL (Evaluation and Report Language) and CSV (Comma-Separated Values) for later analysis.

4.2 AccessBot Integration with QualWeb

QualWeb is the accessibility evaluation engine of AccessBot. It is an evaluator implemented in `noje.js`. As is known, both the browser and Node use javascript as their programming language. However, building AccessBot that runs in the browser is different since there is no access to `node.js` APIs like the filesystem access functionality. Since AccessBot is designed to work as a Chrome extension, it can only use javascript to run in the browser. QualWeb uses several libraries to execute its logic, so instead of using the entire QualWeb application, AccessBot uses the critical libraries that evaluate the HTML and CSS. This decision not only allows AccessBot to be lighter but also allows AccessBot to work completely independent of any external call necessary, meaning that it can be used to evaluate web pages offline.

There are three libraries that Accessbot uses from QualWeb; these are:

- `qw-page`[1] - `qw-page` exposes a class with the same name `qwpage` that takes as arguments the document object and the window object. It generates a new object, which is sent to the execute method provided by the `act-rules` library.

- `act-rules`[2] - is the main library responsible for doing the evaluation of the `qw-page` object that it receives. It has as a dependency on the entire list of evaluations for each ACT rule. `act-rules` processes the HTML and CSS, and returns an object that comprises all the elements tested and their results. This object is responsible for all the automatic evaluations on the page and is also a starting point to further enhance it with semi-automatic and manual tests.

[1]Code hosted in https://github.com/qualweb/qw-page.

[2]Code hosted in https://github.com/qualweb/act-rules .

- `earl-reporter`[3]- EARL is a vocabulary to describe test results in a machine-readable format. Developed by the W3C, it allows applications to interchange an XML format that describes accessibility evaluation results. Both QualWeb and AccessBot allow users to export an EARL report file. AccessBot uses this library developed for QualWeb as the basis for generating an object with all the information required to be EARL compliant. AccessBot uses this object to generate HTML, JSON, and CSV files to save on their machine.

4.3 AccessBot Architecture

Here is presented the AccessBot web application architecture's blueprint, which describes its components, their relations, and how they interact with each other. It is a client-side process, meaning its processes take place on the user's computer without needing access to a web server. The absence of a web server can be seen as an advantage since the AccessBot application can run the same scripts on an HTML page even if the user is disconnected from the internet.

The AccessBot analyzes the webpage in the opened tab of the Chrome Browser. Figure 4.1 represents the flow between the architecture's components.

When the user clicks to start the evaluation, a message is sent to `background.js` from `popup.js` with the selected options. The `background.js` script is responsible for keeping communication with the entire system. When receiving the message from the `popup.js`, `background.js` verifies the user clicked options, in this case, if the user selected automatic, semi-automatic, and manual tests to perform the accessibility evaluation. If the user requires automatic and semi-automatic tests `background.js` then communicates with `content.js` to get the content of the tested page. The `content.js` script then fetches and manipulates the active page. When `content.js` gets this event, it creates a `qw-page` object with the document and window object. The `qw-page` object is generated from a class in the imported library `qw-page.js`, as shown in figure 4.1. The `qw-page` object is used as an argument for the `execute` method from another imported library called `act-rules` that resides in `act.js` file. The method then generates the accessibility evaluation result object. The resulting object from `act-rules` method is used as an argument of the callback function defined in `background.js` but used in the `content.js`. This callback function allows the `background.js` to handle the `content.js` response.

The object received has much information that we do not require for AccessBot. When `background.js` gets this information, it filters the original object received to build a new object with selected properties. With this process completed, `background.js` creates a call to the browser to open `result.html` as a popup. If the user only requires

[3]Code hosted in https://github.com/qualweb/earl-reporter.

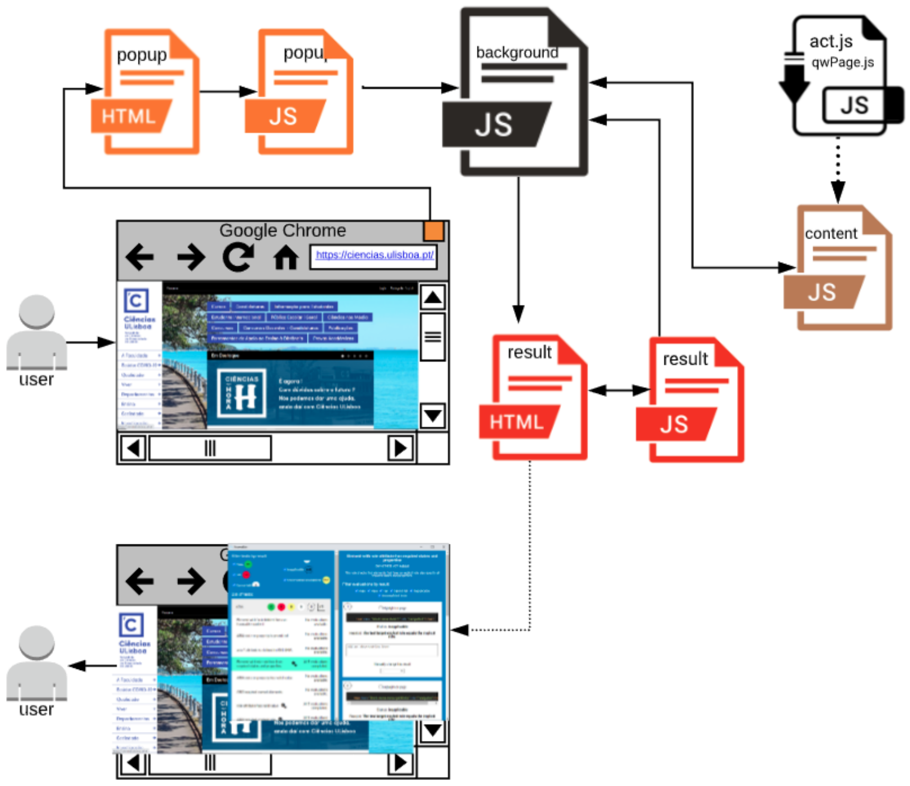


Figure 4.1: Representation of AccessBot architecture. The user clicks on the AccessBot (orange square), and the subsequent flow follows with the succeeding output.

manual tests, `background.js` does not need to communicate with `content.js`, and it jumps straight to calling the browser to open `result.html`.

As soon as `result.html` is fully loaded, it notifies `background.js` and `background.js` sends the filtered object `result.js`. `result.js` is responsible for the entire upkeep of AccessBot, meaning it is the main file that presents the result and is also responsible for AccessBot's user interaction. `result.js` will modify the active page when the user decides to highlight an element that is being evaluated. For that to happen, it will send an event to `background.js`, and `background.js` will notify `content.js` of the necessary change. The same is true when the user removes the highlight. The highlights are visible as red borders around the selected element on the evaluated web page.

4.4 Rules implemented in AccessBot

The rules for accessibility testing implemented on AccessBot are detailed in table 4.1. In this table, QualWeb rules identified with code QW-ACT-Rxy can be used for developing automated and semi-automatic testing methodologies. One rule can have multiple automatic or semi-automatic tests applied to different elements, but each element being evaluated by the rule can only have one test.

The QualWeb rules are based on ACT Rules identified by an ACT Rule ID in the third column of table 4.1. Each rule also has a description that provides a brief explanation of what the rule does.

In table 4.1 and AccessBot, the rules are divided into categories to organize rules and ease access to them by identifying the rules with shared characteristics, namely the success criterion and guideline group they belong. Each guideline corresponds to a different type of content. In the end, several categories emerged. Each category contains multiple rules, and each rule can only be in one category.

In the last column of table 4.1, each rule has the information if it has automatic and semi-automatic tests or if it is manual. Each rule's source code of automating testing has exit points in QualWeb called "result code" (RC), which means the exit code status to obtain a final result of a pass, fail, inapplicable, or cannot tell. The semi-automatic test algorithm picks on the result code that needs user evaluation and prompts a series of questions before obtaining a final result. After the user answers the questions, the algorithm reaches a final evaluation result for the tested element. The manual tests do not require result codes since they are only questions made to guide the user and independent from the QualWeb result evaluation.

Semi-automatic and manual algorithms examples are presented in the next section.

4.5 Semi-automatic Test Algorithms

This section illustrates how semi-automatic algorithms were designed by introducing a couple of examples. Figure 4.2 is the algorithm for R8. R8 belongs to the category Image, and this rule checks if image elements that use their source filename as their accessible name do so without loss of information to the user. As presented in the figure, this is a simple diagram representing a semi-automatic rule. It indicates that, although QualWeb evaluates the element and finds instances of images with an accessible name equal to the filename, the machine code cannot determine if the image filename has a loss of information or not and consequently gives the result "Cannot Tell". The result corresponds to the result code (RC) RC1. The resulting code is the starting point for user evaluation and decision making. The decision represented by a rhomboid shape with white background has a query to the user, and depending on the answer, the final result will be a pass (green square) or a fail (red square). The other semi-automatic rules' algorithms are in the appendix A of the thesis.

Figure 4.3 represents a more complex algorithm. This algorithm is for Rule 17, which verifies if an image has an accessible name. In this algorithm, the automatic evaluations performed by QualWeb are represented by rhomboid shapes colored with a gray background representing machine code decision making. In this particular case, there are two different result codes, RC1, and RC3, that result from the decision-making code of R17.

Table 4.1: AccessBot rules description for webpages evaluation. Rules are described by category, id, and what type of tests they address.

Category	QualWeb Rule ID	ACT Rule ID	ACT Rule Name	Type of tests
Title	QW-ACT-R1	2779a5	HTML Page has a title	Automatic Semi-automatic
Language	QW-ACT-R2	b5c3f8	HTML has lang attribute	Automatic Semi-automatic
Language	QW-ACT-R3	5b7ae0	HTML lang and xml:lang match	Automatic
Time	QW-ACT-R4	bc659a	Meta-refresh no delay	Automatic
Language	QW-ACT-R5	bf051a	Validity of HTML Lang attribute	Automatic
Image	QW-ACT-R6	59796f	Image button has accessible name	Semi-automatic Automatic
Orientation	QW-ACT-R7	b33eff	Orientation of the page is not restricted using CSS transform property	Automatic
Image	QW-ACT-R8	9eb3f6	Image filename is accessible name for image	Semi-automatic
Link	QW-ACT-R9	b20e66	Links with identical accessible names have equivalent purpose	Automatic Semi-automatic
iFrame	QW-ACT-R10	4b1c6c	iframe elements with identical accessible names have equivalent purpose	Automatic Semi-automatic
Button	QW-ACT-R11	97a4e1	Button has accessible name	Automatic Semi-automatic
Link	QW-ACT-R12	c487ae	Link has accessible name	Automatic Semi-automatic
ARIA	QW-ACT-R13	6cfa84	Element with ARIA-hidden has no focusable content	Automatic
Sensory and Visual Clue	QW-ACT-R14	b4f0c3	meta viewport does not prevent zoom	Automatic
Audio and Video	QW-ACT-R15	80f0bf	audio or video has no audio that plays automatically	Automatic Semi-automatic
Form	QW-ACT-R16	e086e5	Form control has accessible name	Automatic Semi-automatic
Image	QW-ACT-R17	23a2a8	Image has accessible name	Semi-automatic
Parsing	QW-ACT-R18	3ea0c8	id attribute value is unique	Automatic
iFrame	QW-ACT-R19	cae760	iframe element has accessible name	Automatic Semi-automatic
ARIA	QW-ACT-R20	674b10	role attribute has valid value	Automatic
Image	QW-ACT-R21	7d6734	svg element with explicit role has accessible name	Automatic Semi-automatic
Language	QW-ACT-R22	de46e4	Element within body has valid lang attribute	Automatic Semi-automatic
Audio and Video	QW-ACT-R23	c5a4ea	video element visual content has accessible alternative	Automatic Semi-automatic
Form	QW-ACT-R24	73f2c2	autocomplete attribute has valid value	Automatic
ARIA	QW-ACT-R25	5c01ea	ARIA state or property is permitted	Automatic
Audio and Video	QW-ACT-R26	eac66b	video element auditory content has accessible alternative	Automatic
ARIA	QW-ACT-R27	5f99a7	This rule checks that each ARIA- attribute specified is defined in ARIA 1.1.	Automatic
ARIA	QW-ACT-R28	4e8ab6	Element with role attribute has required states and properties	Automatic
Audio and Video	QW-ACT-R29	e7aa44	Audio element content has text alternative	Semi-automatic
Form	QW-ACT-R30	2ee8b8	Visible label is part of accessible name	Automatic Semi-automatic
Audio and Video	QW-ACT-R31	c3232f	Video element visual-only content has accessible alternative	Automatic
Audio and Video	QW-ACT-R32	1ec09b	video element visual content has strict accessible alternative	Automatic
ARIA	QW-ACT-R33	ff89c9	ARIA required context role	Automatic
ARIA	QW-ACT-R34	6a7281	ARIA state or property has valid value	Automatic
Heading	QW-ACT-R35	ffd0e9	Heading has accessible name	Automatic Semi-automatic
Table	QW-ACT-R36	a25f45	Headers attribute specified on a cell refers to cells in the same table element	Automatic Semi-automatic
Contrast	QW-ACT-R37	afw4f7	Text has minimum contrast	Automatic Semi-automatic
ARIA	QW-ACT-R38	bc4a75	ARIA required owned elements	Automatic
Table	QW-ACT-R39	d0f69e	All table header cells have assigned data cells	Automatic
Sensory and Visual Clue	QW-ACT-R40	59br37	Zoomed text node is not clipped with CSS overflow	Semi-automatic
Form	QW-ACT-R41	36b590	Error message describes invalid form field value	Semi-automatic
Object	QW-ACT-R42	8fc3b6	Object element has non-empty accessible name	Automatic Semi-automatic
Keyboard	QW-ACT-R43	0ssw9k	Scrollable element is keyboard accessible	Automatic
Link	QW-ACT-R44	fd3a94	Links with identical accessible names and context serve equivalent purpose	Automatic Semi-automatic
Parsing	QW-ACT-R48	46ca7f	Element marked as decorative is not exposed	Automatic
Keyboard	-	80af7b	Elements focusable with keyboard	Manual
Time	-	efbfc7	Text content that changes automatically can be paused, stopped or hidden.	Manual

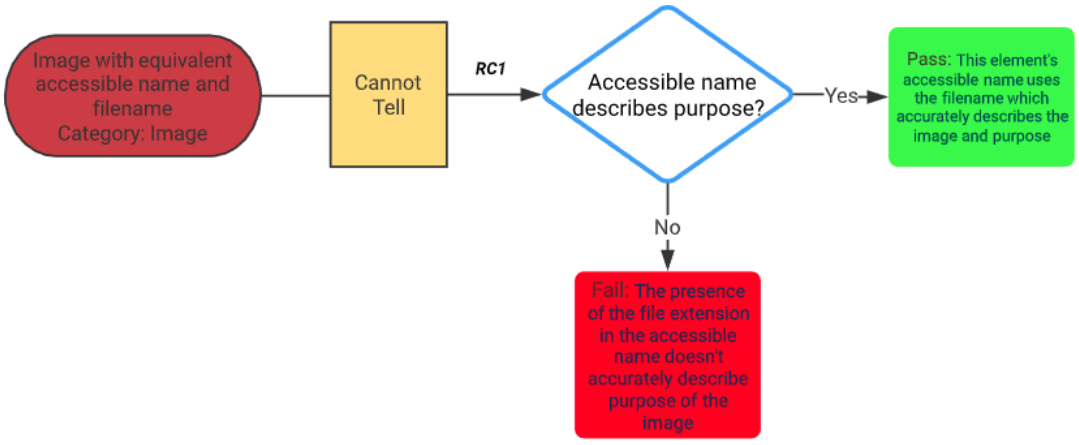


Image with equivalent accessible name and filename
Category: Image
Cannot Tell
RC1
Accessible name describes purpose?
Yes
Pass: This element's accessible name uses the filename which accurately describes the image and purpose
No
Fail: The presence of the file extension in the accessible name doesn't accurately describe purpose of the image

Figure 4.2: Diagram flow of Rule 8. The rule checks that image elements that use their source filename as their accessible name do so without loss of information to the user.

Consequently, each RC leads to a different pathway of questions to the user. The questions are represented by the rhomboid shape but with the white background color. Depending on the answer of the user, this will lead to a pass or fail result.

4.6 Manual Test Algorithms

The algorithm represented in figure 4.4 is for a manual test. As stated, for these tests, QualWeb does not intervene, and they are composed of a series of instructions to guide the user. Figure 4.4 represents the algorithm for keyboard manual test to verify if the web page is operable using a keyboard. A series of questions and instructions are presented to the user, reaching a final result of pass or fail. There is another manual algorithm test, but this is presented in the appendix B.

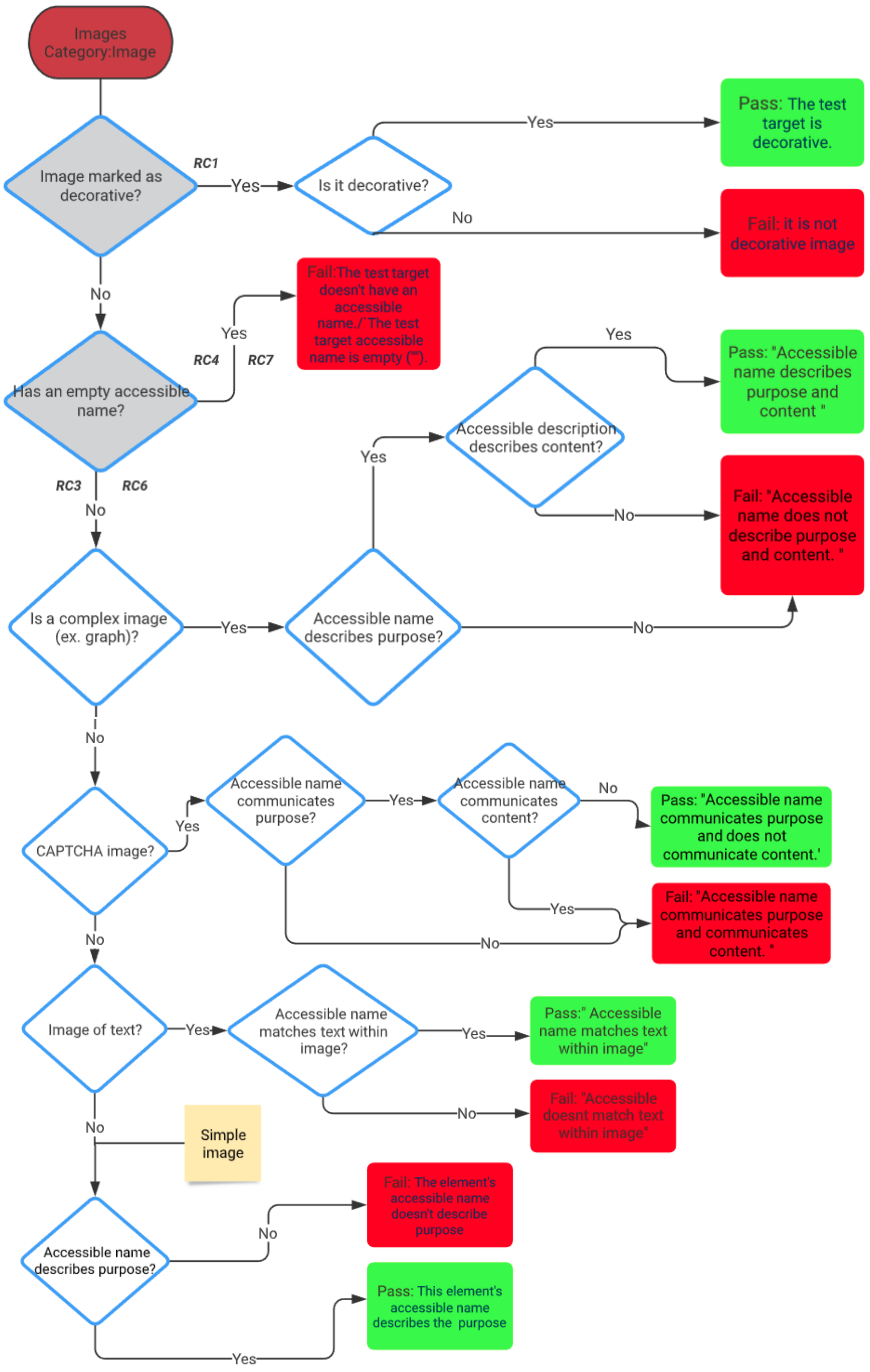

Images Category:Image
Image marked as decorative?
RC1
Yes
Is it decorative?
Yes
Pass: The test target is decorative.
No
Fail: it is not decorative image
No
Has an empty accessible name?
RC4
RC7
Yes
Fail:The test target doesn't have an accessible name./'The test target accessible name is empty ('').
RC3
RC6
No
Is a complex image (ex. graph)?
Yes
Accessible name describes purpose?
Yes
Accessible description describes content?
Yes
Pass: "Accessible name describes purpose and content "
No
Fail: "Accessible name does not describe purpose and content. "
No
No
CAPTCHA image?
Yes
Accessible name communicates purpose?
Yes
Accessible name communicates content?
No
Pass: "Accessible name communicates purpose and does not communicate content.'
Yes
No
Fail: "Accessible name communicates purpose and communicates content. "
No
Image of text?
Yes
Accessible name matches text within image?
Yes
Pass:" Accessible name matches text within image"
No
Fail: "Accessible doesnt match text within image"
No
Simple image
Accessible name describes purpose?
No
Fail: The element's accessible name doesn't describe purpose
Yes
Pass: This element's accessible name describes the purpose

Figure 4.3: Diagram flow of Rule 17. This rule checks that each image either has a non-empty accessible name or is marked up as decorative.

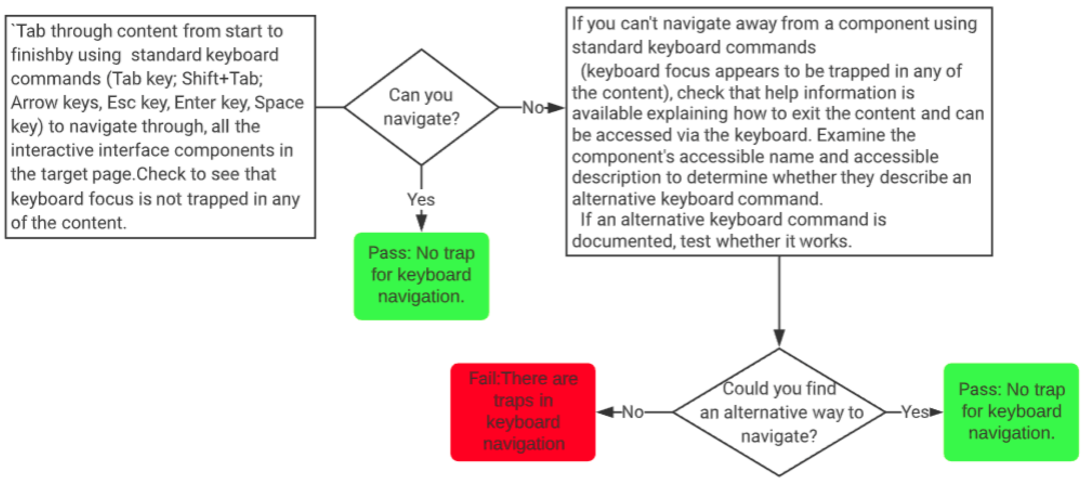

`Tab through content from start to finishby using standard keyboard commands (Tab key; Shift+Tab; Arrow keys, Esc key, Enter key, Space key) to navigate through, all the interactive interface components in the target page.Check to see that keyboard focus is not trapped in any of the content.
Can you navigate?
No
Yes
Pass: No trap for keyboard navigation.
If you can't navigate away from a component using standard keyboard commands (keyboard focus appears to be trapped in any of the content), check that help information is available explaining how to exit the content and can be accessed via the keyboard. Examine the component's accessible name and accessible description to determine whether they describe an alternative keyboard command. If an alternative keyboard command is documented, test whether it works.
Could you find an alternative way to navigate?
No
Fail:There are traps in keyboard navigation
Yes
Pass: No trap for keyboard navigation.

Figure 4.4: Rule ACT ID 80af7b. This rule checks for keyboard traps. This includes use of both standard and non-standard keyboard navigation to navigate through all content without becoming trapped.

Chapter 5

AccessBot Implementation

This chapter provides an understanding of all significant aspects of the implementation of AccessBot. First, the technologies used are described, then the development process, including a description of the main and features logic, and lastly, it presents the difficulties encountered during the development process.

5.1 Technologies used on AccessBot

For implementing the AccessBot Chrome extension, the Javascript programming language is used as the scripting language without frameworks, alongside HTML, CSS, and JSON. For compiling Javascript modules, the tool used is Webpack[1]. It generates a few files that run AccessBot besides bundling the resources and compiling TypeScript to Javascript. The libraries of QualWeb mentioned in the previous chapter are written in TypeScript resulting in more robust software. Webpack needed configuration, which was done during the development of the project. Listing 5.1 represents the configuration of Webpack with its entry points and the output results. For version control, Git is used.

Listing 5.1: Webpack configuration file.

```

const path = require('path')

module.exports = [{
  entry: {
    result: './result.js',
    popup: './popup.js',
    background: './background.js',
    contentScript: './contentScript.js',
  },
  output: {
    filename: '[name].js',
```

[1]For more information https://webpack.js.org/ .

```
    path: path.resolve(__dirname, 'dist'),
    libraryTarget: 'var',
    library: ["[name]"],
  },
  optimization: {
    minimize: false
},
}]
```

5.2 AccessBot Processing

This section describes the code implementation process of AccessBot. For the implementation to be straightforward, the logic is going to be divided into smaller sections. Note that these sections may not be sequential. During development, some coding overlap exists; for example, a feature is developed, but new functionalities are implemented and added, or the feature needs correction.

The implementation is divided into automatic logic, which deals with manipulation of QualWeb automatic results; semi-automatic and manual logic, which refer to the implementation of the algorithms and how these are presented to the user; and finally, the features logic that provides a more easy interpretation and manipulation of the results by the user. It is also mentioned the setup of the manifest file, which is the JSON file that specifies the assets of AccessBot. The popup section refers to the entry point of the AccessBot.

5.2.1 Setup of manifest file

The creation of any Chrome extension always starts with the manifest file. This file contains all the information needed to create AccessBot, as shown in the listing 5.2. The manifest file is updated as needed during the AccessBot coding. It is a JSON file that contains all the information that defines the extension: the description of AccessBot; a content security policy that introduces strict policies that make the extension more secure; permissions which is the information AccessBot can access; and allows to specify also aspects of AccessBot functionality such as background scripts, content scripts, and browser actions.

Listing 5.2: AccessBot manifest file configuration.

```

{
  "name": "AccessBot",
  "version": "1.1.6",
  "description": "Assisted Evaluation powered by QualWeb.",
  "manifest_version": 2,
  "content_security_policy": "script-src 'self'; object-src 'self'"
     ,
  "permissions": ["storage", "tabs", "activeTab", "http:
     //127.0.0.1:9222/*"],
  "background": {
    "page": "background.html",
    "script": "background.js",
    "persistent": true
  },
  "browser_action": {
    "default_popup": "popup.html"
  },
  "icons": {
      "48": "icons/robot_48.png"
  },
 "content_scripts": [{
     "matches": ["<all_urls>"],
     "js": ["./dist/contentScript.js", "act.js", "qwPage.js"]
  }
 ]
}
```

5.2.2 Popup

After reflecting on how the user would interact with the extension, the best solution was to implement a browser action. An icon on the Google Chrome toolbar is created for the user to click. When the user clicks the icon, a popup appears, which corresponds to the file popup.html, allowing the user to define the options that will control the evaluation and start the evaluation itself. When the user clicks the button to start the evaluation on the popup, an event is triggered. An object that stores the user options for the evaluation (manual, semi-automatic, and automatic) is sent to `background.js`. The user is required to select an option; otherwise, the start evaluation button is disabled. Chrome uses an observer pattern with events and listeners to communicate between different parts of the application. In this case, `chrome.runtime.sendMessage()` with options object as an argument has a listener on `background.js` that is waiting to receive the object once it is created.

Upon receiving the event from the popup, in the `background.js` listener defined using `chrome.runtime.onMessage.addListener()`, the options object received is verified to find if the user selected manual, semi-automatic, or automatic.

This step is essential because it will determine if we need to use QualWeb to generate an evaluation for us to display to the user in semi-automatic and automatic evaluations, contrary to manual evaluations.

5.2.3 Automatic and Semi-automatic Shared Logic

There are a few steps that are common to both logics. In `background.js`, steps consist of determining the active chrome tab's id, which corresponds to the web page that will be evaluated. With that information, an event message to the corresponding `content.js` of the tab is sent from `background.js`.

With `chrome.tabs.sendMessage()` the code can communicate with the open tab on Chrome, bearing in mind that in order to do that, we are required to create a `content.js` file. The `chrome.tabs.sendMessage()` has three arguments. The first argument is an integer value corresponding to the tab id we wish to communicate. The second, the object we wish to send, and the third is an optional callback function. We use this callback when sending the message to the `content.js` to ease the communication process between `background.js` and `content.js` by reducing listeners' number.

The code can never directly manipulate a page with Chrome Extension. Chrome requires to define on the manifest JSON a `content_script` with a rule for URLs that will load the `content.js` file. In the case of AccessBot, we applied the rule `all_urls` on the manifest. Every website that users open when we have the AccessBot extension enabled will have the defined `content.js` loaded to manipulate the page. The `content.js` from the tab that gets this message will create a `qwPage` object, by using the imported library with the same name `qwPage`, from the page Document and Window object (listing 5.3). The Window object represents an open window in a browser, while the Document object represents the HTML displayed in that window. The Document object has various properties that refer to other objects which allow access and modification of the document content.

Listing 5.3: Code excerpt from file `content.js` that shows the use of the two imported libraries from QualWeb. The code creates a `qwpage` object using `QWPage` library. Then this object is processed by `ACTRules` library.

```

if(request.message === "getDocument") {
    const result = new QWPage.QWPage(document, window);
    let act = new ACTRules.ACTRules()
    const actResult = await act.execute({},result,[]);
    sendResponse(actResult);
  }
```

With `qwPage` object, the code executes the QualWeb assertions for the evaluation web page using another library named `ACTRules`. This logic needs to be inside `content.js` because we can only access the document and the window object inside the tab execution context. After we get the final result, in this case, variable `actResult` from the rules assertions, we use the callback defined in the `background.js` with the `actResult` as an argument.

An example of `console-log` of the original QualWeb object obtained after the evaluation of the webpage `https://ciencias.ulisboa.pt/` is presented:

```
Object
    assertions:
        QW-ACT-R1: {
            name: "HTML Page has a title",
            code: "QW-ACT-R1",
            mapping: "2779a5",
            description: "This rule checks that the HTML page has a title.",
            metadata: {...},
            ...}
        QW-ACT-R2: {
            name: "HTML has lang attribute",
            code: "QW-ACT-R2",
            mapping: "b5c3f8",
            description: "This rule checks that the html element has (...) attribute.",
            metadata: {...},
            ...}
        QW-ACT-R3: {
            name: "HTML lang and xml:lang match",
            code: "QW-ACT-R3",
            mapping: "5b7ae0",
            description: "The rule checks that for the html element, (...) are used.",
            metadata: {...},
            ...}
        (...)
```

The callback function saves on a variable, accessible throughout `background.js`, a simplified object used for the application (listing 5.4). As the name says, this simplified object is a simplified modification of the original QualWeb evaluation result that contains all the selected properties used in AccessBot. Afterward, the code asks Chrome to create a new popup window where it will display the interface for the user to interact with the evaluation results.

Listing 5.4: Part of the function in `background.js` that modifies the original Qualweb results in an object containing the properties AccessBot will use.

```
 ...
               return {
                code: rule.code,
                      description: rule.description,
                      results: results,
                      name: rule.name,
                      id: rule.mapping,
                      url: rule.metadata.url,
                      accessiblename: rule.accessiblename
                    }
                  });
...
```

As soon as the popup opens (which corresponds to the file `result.html`), it notifies `background.js` using the same event system previously mentioned. `Background.js` will then send back the filtered results, the user-defined options, the website URL, and the original QualWeb result. The original QualWeb result is going to be used to export the EARL report. Inside `result.js` resides the logic to present the evaluation of the results to the user, update the DOM that is representative of `result.html` when the user interacts with it, and export reports of evaluations. A new object is created to make this happen. For clarity reasons, it is called the main object (Appendix C). The first thing `result.js` does when it gets the information from `background.js` is to generate the main object responsible for the `result.html` state using the function `generateCategoriesData(result, options)`. This object contains the entire state of the `result.html` meaning it contains the information `result.js` needs to manipulate the DOM.

The updating process of the DOM is continuous because when it needs to be updated, most of the HTML is removed, and multiple functions containing various parts of the HTML are called to add HTML back to the DOM. The process of continuously "destroying" and "recreating" all HTML nodes simplifies the updating process since AccessBot is not using any particular javascript framework for DOM manipulation. This logic is used to avoid potential issues with updating multiple parts of the DOM at the same time due to a change of state. In sum, interactions with `result.html` can lead to update of a property of the main object; everytime a part of the main object is altered a function is called to destroy and reconstruct DOM again.

The main object created contains the counters for the multiple results of each evaluation, the remaining tests to be completed (this is especially important for the semi-automatic evaluation), and the total number of tests. It also stores an attribute called categories. The categories were created to organize better the many rules required for semi-automatic and automatic evaluations. They are not part of the QualWeb object, and they exist only on AccessBot. A separate javascript file was created to map the categories to the rules as shown on listing 5.5.

Listing 5.5: `categories.js` file containing the object that attributes the rules to the categories for better organization of the automatic and semi-automatic rules.

```

import category from "./const.js";

export default {
    [category.IMAGE]: ["R6", "R8", "R17", "R21"],
    [category.TITLE]: ["R1"],
    [category.KEYBOARD]: ["R43"],
    [category.LANGUAGE]: ["R2", "R3", "R5", "R22"],
    [category.TIME]: ["R4"],
    [category.ORIENTATION]: ["R7"],
    [category.SENSORYVISUALCLUES]: ["R14", "R40"],
    [category.AUDIOVIDEO] : ["R15", "R23", "R26", "R29", "R31", "
       R32"],
    [category.PARSING] : ["R18", "R48"],
    [category.ARIA] : ["R13","R20", "R25", "R27", "R28", "R33", "
       R34", "R38"],
    [category.FORMS]: ["R16", "R30", "R24", "R41"],
    [category.HEADINGS]: ["R35"],
    [category.TABLES]: ["R36","R39"],
    [category.CONTRAST]: ["R37"],
    [category.LINKS]: ["R9", "R12", "R44"],
    [category.IFRAMES]: ["R10", "R19"],
    [category.BUTTONS]: ["R11"],
    [category.OBJECT]: ["R42"],

}
```

The original object received from `background.js` and obtained after the evaluation done by QualWeb, contains a list of rules and their results, as exemplified before. Each rule contains a list of test results.

The function `generateCategoriesData(result, options)` is used to generate the categories array in the main object (listing 5.6). The function takes each object member from the original object and evaluates it for its rule code. The rule code is obtained from the QualWeb Rule ID, for example, QW-ACT-R42, then the string is split to obtain R42 to map the categories to the rules.

Listing 5.6: Creating categories object using the function `generateCategoriesData(result, options)`

```
if (getCategoryIndex === -1) {
    semiManualTests.categories.push({
        name: currentCategory,
        fixedName: currentCategory.replace(/ /g, '').replace(/[^
           A-Za-z0-9]/g, ''),
        total: total,
        count: 0,
        pass: 0,
        fail: 0,
        inapplicable: 0,
        warning: 0,
        missing: 0,
        selected: false,
        index: categoryNextIndex,
        rules: [
            {
                rule: ruleCode,
                name: ruleName,
                description: ruleDescription,
                id,
                url,
                total: total,
                count: 0,
                pass: 0,
                fail: 0,
                inapplicable: 0,
                warning: 0,
                missing: 0,
                questions: questions,
                selected: false,
                plusRule: manualRule && manualRule.plusRule ?
                    manualRule.plusRule : [],
                index: 0
            }
        ],
    });
```

Then the code searches for a match in the categories imported from `rules/categories.js`. If it finds a match, it adds a category object to the categories array. Each category object inside the categories array will contain information such as the `category.name`, the counters for that category (total, count, pass, fail, inapplicable, warning, missing) rules attribute array. The rules array will store the rules that will belong to that category. If the category already existed in the categories array, the rule will be added to that category's rules array attribute.

The rules object that is added to the rules array also stores similar information to the category object. It contains the `ruleCode`, `ruleName` and counters for that rule. The rules array will also contain an array that will store all the evaluations for that rule. These evaluations or tests are the smallest units in the structure. They will contain automatic and semi-automatic evaluations, besides the manual evaluations, which will be discussed next. The object also contains test results for each rule. These are inside the rules array with the array designated by "questions". When all the rules have been added, the categories are sorted alphabetically. The result of all these operations is stored on a global variable to be accessible throughout `result.js`.

With the main object created, the function `updateResults()` is then called. This function is called every time a change occurs on the main object. It contains many other functions that are required to update the `result.html`. This leads to `result.html` being updated continuously, as aforementioned; for example, a semi-automatic evaluation that just got completed is followed by updating the main object. The semi-automatic evaluations are the results of completed automatic tests deemed to require user input to deliver an accurate result. Concisely, any changes in the results outcomes can happen in two ways: the user completes an evaluation or can change the automatic result manually. These changes will affect the counters of the corresponding rules and categories. The previously mentioned actions will increment the counter on the rule, and in turn, it will increment the counter on the category to which the rule belongs.

Every time a change occurs, for example, the user manually changes the result of an automatic evaluation, the function `updateResults()` is called, that treats HTML like a blank slate, evaluates the main object, and reconstructs the HTML based on the updated main object. This also helps in making sure that a change that requires updates on multiple parts of the HTML happens with no side effects.

5.2.4 Semi-automatic Logic

The semi-automatic logic evaluations consist of a system created to check the result on an automatic test to see if it needs to be semi-automatic evaluated. As described in the previous chapter, every rule can be translated into an algorithm diagram flow, which takes into account decision points called the result codes. From these decision points, the sequential questions and their outcomes are included in the diagram.

The code implements these diagrams. The diagram flow implementation consists of an array of objects that describe the sequential questions and the order they should appear based on the user decision. The listing 5.7 presents the code for the diagram flow of the rule R8. The diagram flow can be consulted figure 4.2.

Listing 5.7: Diagram flow object for R8.

```
import categoryConst from "./const.js";

export default {
    code: 'QW-ACT-R8',
    category: categoryConst.IMAGE,
    tree: [{
        prerequisite: 'RC1',
        flow:[
            {
            key: '1A',
            title: 'Does accessible name #{a} describes purpose?',
            answerYes: 'Pass',
            answerNo: 'Fail',
            },
            {
            key: 'Pass',
            title: "The element's accessible name uses the filename
                which accurately describes the image and purpose",

            },
            {
            key: 'Fail',
            title: "The presence of the file extension in the
               accessible name does not accurately describe purpose
                of the image",
            }
        ]
    }]
}
```

Every rule flow is represented in a separate file that resides on a project folder called `rules`. The rules folder also contains a file called `index.js` that imports all the individual rules and exports them inside a single array. The `index.js` is imported unto `results.js`.

When checking the evaluations for their type, first is checked if an algorithm for the rule exists. If it exists, it means that every evaluation under this rule needs to be verified against the algorithm. The result code of the evaluation is used. If a match is found on the property prerequisite that resides inside the algorithm, the semi-automatic evaluation flow is retrieved.

A class was created to store all data information to help keep track of the flow's current state. The class name is called `DecisionTree`. The diagram flow array is used as an argument for the `DecisionTree` constructor.

The `DecisionTree` class implements methods such as `next()`, `prev()`, `current()` and `revert()` for controlling the state of the flow that is stored on the object instance of that class. All the interactions done by the user when providing information during a semi-automatic evaluation will be using this class.

5.2.5 Manual Logic

Suppose only manual tests were selected in the popup. In that case, AccessBot provides its own set of tests to present to the user and we can skip some of the steps common to automatic and semi-automatic logic previously mentioned. The code opens the popup (`result.html`) without the QualWeb evaluation to perform only manual tests. However, to simplify the code's execution, the manual evaluation results are also added to the main object obtained from the function `generateCategoriesData(result, options)`. The manual evaluations are done entirely on the side of AccessBot.

As mentioned previously, manual tests do not use the QualWeb result. The approach to generate the manual evaluation results added to the main object is slightly different from a semi-automatic. The manual diagram flow implementation (listing 5.8) is also different from the semi-automatic diagram flow implementation. The main difference is that it does not have a result code as a prerequisite from QualWeb and has more text properties. The diagram flow for the manual rule Keyboard can be consulted in figure 4.4.

Listing 5.8: Diagram flow code for keyboard rule (ACT ID 80af7b)..

```
import CategoryConst from "../const.js";

export default {
    code: '80af7b',
    url: 'https://act-rules.github.io/rules/80af7b',
    name: 'Are elements focusable with keyboard?',
    category: CategoryConst.KEYBOARD,
    whyImportant: 'Users must be able to access and interact with
       interface components using only the keyboard because using a
        mouse is not possible when the user has no vision or low
       vision or does not have the physical capability or dexterity
        to effectively control a pointing
    device.',
    descriptionTest:'Users must be able to navigate away from all
        components using a keyboard.',
    tree: [
        {
            key: '',
            title: 'Tab through content from start to finish by
                using standard keyboard commands (Tab key; Shift+
                Tab; Arrow keys; Esc key; Enter key; Space key) to
                navigate through all the interactive interface
                components in the target page. Check to see that
                keyboard focus is not trapped in any of the content.
                ',
            question: 'Can you navigate?',
            answerYes: 'Pass',
            answerNo: 'nextStep1'
        },
        {
            key: "nextStep1",
            title: 'If you can not navigate away from a component
                using standard keyboard commands (keyboard focus
                appears to be trapped in any of the content), check
                that help information is available explaining how to
                 exit the content and can be accessed via the
                keyboard. Examine the component`s accessible name
                and accessible description to determine whether they
                 describe an alternative keyboard command. If an
                alternative keyboard command is documented, test
                whether it works.',
            question: 'Could you find an alternative way to
                navigate?',
            answerYes: "Pass",
            answerNo: "Fail"
        },
        {
            key: 'Pass',
            title: 'No trap for keyboard navigation.',
        },
        {
            key: 'Fail',
            title: "There are traps in keyboard navigation",
        }
    ],
}
```

5.2.6 Features Logic

Other features implemented on AccessBot are: the possibility to highlight the tested element, remove the highlight selected, remove all highlights at once if more than one highlight option is selected, ability to write observations in each test, change the automatic result evaluation if the user disagrees with the result and the possibility to export the final evaluation in EARL and CSV format.

The code relies on the chrome event system to communicate between `result.js` and `contentScript.js` to implement the highlight feature. The API used is `chrome.runtime.sendMessage()` from `result.js` to a listener defined using `chrome.runtime.onMessage.addListener()` in `contentScript.js`. QualWeb stores a pointer reference string on its object for each evaluation that describes a path from the root DOM element to the evaluated element. The pointer string is sent on the event message to `background.js`, and when `background.js` receives this event, it sends it to the active tab on the user browser. `contentScript.js` will take this pointer string and invoke a function called `pointerToElement()`. The function will check if the path is for an element present in the body element of the HTML and if true, use `document.querySelector` function with the pointer. A thick red border style is added to the results of `querySelector`. The opposite result also exists that takes the highlighted element and removes the thick red border style.

AccessBot also allows the user to remove all highlights that are currently selected. An array that stores all currently selected highlights is updated every time a user checks or unchecks for a highlight. When the user clicks on the “Remove Highlights” button, the remove highlight event is triggered for each highlight on the array, and then the array gets cleared.

When an evaluation has a completed state, a text area element appears, allowing the user to write a text about that evaluation results. That text is stored on a property inside the evaluation object that resides on the main object. Every time a change is detected in the text area, that property gets updated with the text’s current value.

For the automatic tests, the user can modify the evaluation result if he believes it is not correct. The main object also contains a property called `manualAnswer`, which saves the modification to an automatic evaluation. A select HTML element is presented to the user, and if he selects an option other than an empty or the same result as evaluation, that option value will be stored on the property.

To create an EARL report for the user to export the AccessBot results, a file called `earl.js` contains all the logic necessary. This file exports a single function called `resultToEarl()` imported on `result.js` when the user makes a request to generate an EARL export. After the user inputs his name, this function is called. `resultToEarl()` takes care of the code required to return an EARL report object that will then be used to create the export files. It uses the same functionality developed for QualWeb to deliver an EARL report called function `generateEARLreport()`.

Nevertheless, because AccessBot also adds manual testing and modifies some automatic tests to become semi-automatic, it is necessary to expand upon the result the function `generateEARLReport()` delivers. The original QualWeb result object is used as an argument to `generateEARLreport()`. This ensures that the object the function receives is correct according to the function expectations. The function returns an object prepared to be interpreted by an EARL compatible application in JSON format. The function adds to the report the new information from AccessBot. The human assertor and the information of all the manual assertions that AccessBot introduced are inserted into this object. When this process is complete, it follows by determining the automatic assertions modified by AccessBot to become semi-automatic. All automatic assertions on the EARL object are evaluated and changed with the new semi-automatic type and evaluation result that AccessBot has stored.

Two properties that are not part of the EARL standard are added to each assertion (be it manual, semi-automatic, or automatic). These properties store information about user manually modified results, which will only be relevant for automatic tests and the user's notes. These additions will be relevant when exporting the result on the AccessBot side, especially for the CSV export.

The EARL object with all the additions and modifications is the result of the function `resultToEarl()`. With this object, depending on the user's choice in exporting to a JSON or a CSV, `result.js` processes it to create a downloaded file to the user's machine.

5.3 Updating the Rules on AccessBot

AccessBot has a determined number of rules. However, the ACT Community continues to develop more rules for implementation. Considering how AccessBot was implemented, it is easily updated to take into account the new rules that are being developed. This segment presents how to update automatic, semi-automatic, and manual rules.

The automatic and semi-automatic evaluations are based on the QualWeb engine that evaluates the target web page. At the moment, AccessBot "connects" to QualWeb through two libraries `act.js` and `qw-page.js`. Updating these two files will be enough to include the new ACT rules as they become supported by QualWeb.

5.3.1 Updating Automatic Rules

In this case, since evaluation, as its name says, is automatic, the only steps required to add a new automatic rule that the developer wants AccessBot to use and show are done on file `categories.js`. The developer should add the rule to the correspondent category array if the category is already on the default object. For example, to add a new rule R47 that belongs to category Spacing, first, the developer should verify if the category exists (listing 5.9). If not, on the file `const.js`, the developer should add the new category SPACING: "Spacing" to the object. The property value is the string that will appear on the new AccessBot category. After that, the developer should add the new category to the `categories.js` and add the rule to that category.

Listing 5.9: Defining the categories names in `index.js`.

```
export default {
    IMAGE: "Image",
    TITLE: "Title",
    KEYBOARD: "Keyboard",
    LANGUAGE: "Language",
    TIME: "Time",
    ORIENTATION: "Orientation",
    SENSORYVISUALCLUES: "Sensory and Visual Clue",
    AUDIOVIDEO: "Audio and Video",
    PARSING: "Parsing",
    ARIA: "ARIA",
    FORMS: "Form",
    HEADINGS: "Heading",
    TABLES: "Table",
    CONTRAST: "Contrast",
    LINKS: "Link",
    IFRAMES: "iFrame",
    BUTTONS: "Button",
    LABEL: "Label",
    OBJECT: "Object",
    TEXT: "Text",
    SPACING: "Spacing"
}
```

5.3.2 Updating Semi-Automatic Rules

The steps described for updating the automatic rules are also needed in the semi-automatic rules. Part of the semi-automatic evaluation is automatic since AccessBot checks the result code that needs to be complemented with user evaluation. To update a semi-automatic rule, the developer needs to create a new file with questions to ask the user. Listing 5.10 presents an example of the file `R44.js`. The developer needs to fill the object with the property values for the new rule. `code` is the QualWeb code, the `category` is the category of the rule, and `tree` is the array with the object with pre-requisite or result code from where the flow starts. `flow` is an array of steps objects; in this case, there are three steps. Each step is identified with a unique key. In this example, 1A is where the flow starts. From here, the first object is linked to the next using keys. If the user answers "Yes", the next key is "Pass", which is the second object. Nevertheless, if the user answers "No", the next object is the third with the key "Fail". The property `title` corresponds to the question to why it is a “Pass” or a “Fail”, depending on the step.

Listing 5.10: `R44.js` file with code for diagram flow of R44. Serves as an example for demonstrating how to update a simple semi-automatic rule. .

```
export default {
    code: 'QW-ACT-R44',
    category: CategoryConst.LINKS,
    tree: [{
        prerequisite: 'RC3',
        flow: [
            {
                key: '1A',
                title: 'Do the links have the same purpose?',
                answerYes: 'Pass',
                answerNo: 'Fail',
            },
            {
                key: 'Pass',
                title: "The \`links\` with the same accessible name
                    have equal purpose."

            },
            {
                key: "Fail",
                title: "`The \`links\` with the same accessible
                    name have different content."
            },
        ]
    }]
}
```

Some rules can be more complicated when they have more than one prerequisite, which means there may be more than one flow path for the rule. For example, rule R17 has two flows, as seen in listing 5.11, and the second flow starts with two prerequisites. The procedure is the same when it comes to constructing the object.

Listing 5.11: `R17.js` file with code for diagram flow of R17. Serves as an example for demonstrating how to update a complex semi-automatic rule. .

```
export default {
        code: 'QW-ACT-R17',
        category: CategoryConst.IMAGE,
        tree: [{
            prerequisite: 'RC1',
            flow: [
                {
                    key: '1A',
                    title: 'Is the image decorative?',
                    answerYes: 'Pass',
                    answerNo: 'Fail',
                },
                {
                    key: 'Pass',
                    title: "The test target is decorative.",
                },
                {
                    key: 'Fail',
                   title: "The presence of the file extension in
                      the accessible name does not accurately
                      describe purpose of the image",
                }
            ]
        },
        {
            prerequisite: 'RC3, RC6',
            flow: [
                {
                    key: '1B',
                    title: 'Is image a complex image (for example,
                   a graph)?',
                    answerYes: '2A',
                    answerNo: '2B',
                },
                {
                    key: '2A',
               title: "Does accessible name #{a} describe purpose?"
                  ,
                    answerYes:"3A",
                    answerNo: "2AFail"
                },
...
```

After the object is created, it needs to be imported on the file dedicated only to the semi-automatic rules with name `index.js` (listing 5.12).

Listing 5.12: Part of the `index.js` Demonstrating how to import a semi-automatic rule file with the diagram flow..

```
import R1 from './R1.js';
import R2 from './R2.js';
import R6 from './R6.js';
import R8 from './R8.js';
import R9 from './R9.js';
import R10 from './R10.js';
import R11 from './R11.js';
(...)
```

5.3.3 Updating Manual Rules

Manual rules are distinct from automatic and semi-automatic since there is no involvement of QualWeb libraries in their evaluation. They only depend on the user. Considering this important point, manual rules are placed in a separate folder called `assessments`. The developer should create a file with the manual rule flow, as seen in listing 5.8. The manual object is also slightly different from the semi-automatic object since it has more properties with information: `code`, `url`, `name of the rule`, `why it is important` to test and the `description` of the test. The category is from the `const.js` file. The tree does not need to have the prerequisite code and has only the flow with its unique keys linking the steps. The file with the manual rule should then be imported in `index.js`, which is inside the folder `assessments`.

5.4 Difficulties encountered during development

Building a Chrome extension is different from building a web app since the extension runs along with the web page it evaluates; it requires learning core components and their relationships. Another difficulty encountered was bundling and debugging.

5.4.1 Bundling

One of the bundling issues was that the QualWeb libraries required had dependencies to `node.js` only libraries. This was one reason it was communicated to the QualWeb team that they needed to remove Puppeteer from the libraries that did not require us to use it directly. Puppeteer is a headless browser application meant for automation and testing. It is still used on the QualWeb core to retrieve and evaluate the user's web page when the QualWeb application is running on a server.

Initially, the QualWeb application used the resulting object from Puppeteer on all its libraries besides core. Now `qw-page` is used to wrap the retrieved document and window from Puppeteer, and `qw-page` does not have dependencies of node libraries. QualWeb application now uses the `qw-page` library for generating results. First, it is imported the `qw-page` library in `background.js`, since according to chrome extension logic, `background.js` should be responsible for the majority of the work of the extension. To communicate between `content.js` and `background.js`, it is necessary to use event systems that only allow sending data in JSON format. In order to send the document object and window object of the page (which are not in JSON) to the `background.js` (be used by `qw-page` to generate a `qw-page` object), before sending it, the HTML was processed to a string with JSON content and then it is sent through an event.

`background.js` received the string, then used a `DOMParser()` to recreate the page and use that page to reconstruct document and window object. However, this intermediate step caused inconsistencies in the page evaluation. The next alternative was to eliminate this step and work with `qw-page` and `act-rules` imports directly in `content.js` and remove them from `background.js`. In the content script, the document and window objects are manipulated, and the result object, which is already in JSON then sent to `background.js`. However, some problems arose during the bundling with Webpack with the surge of circular dependencies while importing `act.js` and `qw-page.js` (Webpack was importing twice). To resolve this situation, the imports were not made in `content.js` but added directly to the manifest as a dependency to the AccessBot Chrome extension.

5.4.2 Debugging

Extensions carry unique behavior properties. Extensions are just like web pages that can be debugged using the built-in tools of Google Chrome. In this case, the difficulties encountered are related to the location of the logs. Since extensions are made of different components, and each component has individual responsibilities, AccessBot logs could be localized in the `background.html`, on `popup.html`, on `result.html` and in the page under evaluation to access `content.js`. This is mentioned because debugging chrome extensions takes a longer time than typical web applications considering where the different log may appear.

5.5 AccessBot User Interaction

This section presents the user experience when interacting with the AccessBot user interface.

After the user clicks on the Chrome extension icon, a popup window appears and asks the user what the user wants to do, if it is a manual, semi-automatic, or automatic evaluation of the webpage, as seen in figure 5.1.

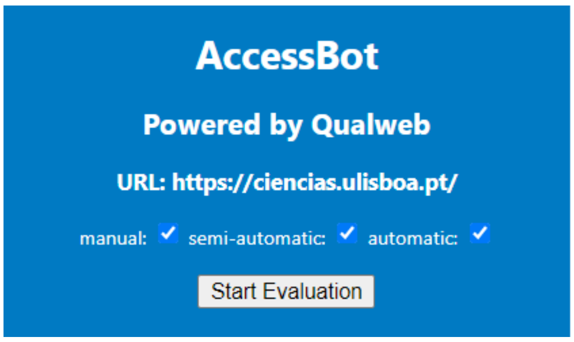
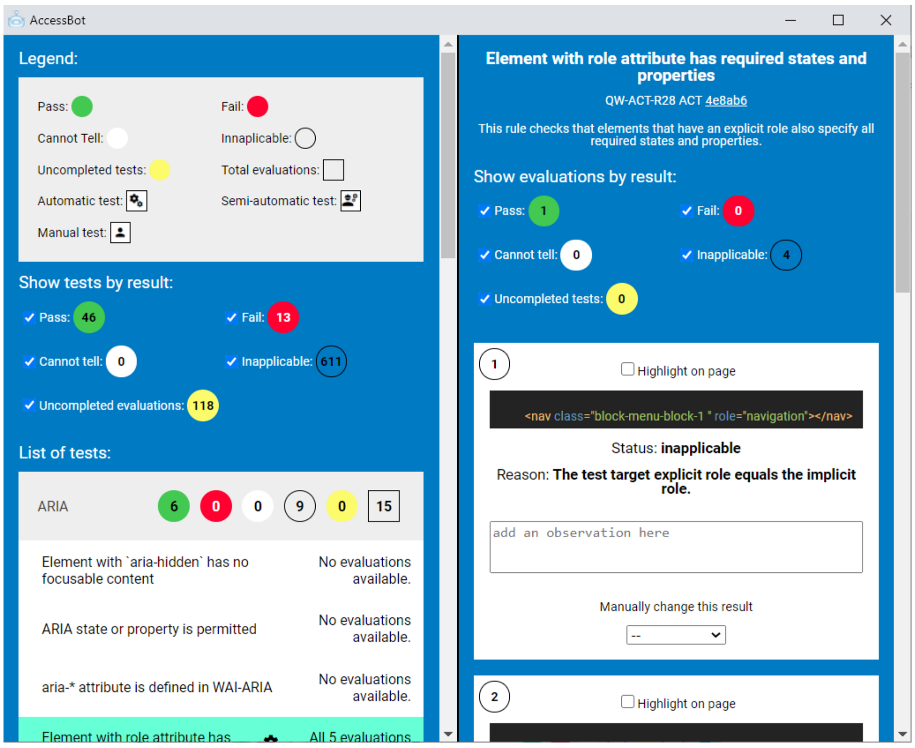

AccessBot
Powered by Qualweb
URL: https://ciencias.ulisboa.pt/
manual: ✓ semi-automatic: ✓ automatic: ✓
Start Evaluation

Figure 5.1: AccessBot popup that appears when user clicks on the extension icon.

After pressing the button “Start evaluation”, another popup window with the evaluation results appears (figure 5.2).



AccessBot

Legend:

Pass: ● Fail: ●
Cannot Tell: ○ Inapplicable: ○
Uncompleted tests: ● Total evaluations: □
Automatic test: Semi-automatic test:
Manual test:

Show tests by result:

✓ Pass: 46 ✓ Fail: 13
✓ Cannot tell: 0 ✓ Inapplicable: 611
✓ Uncompleted evaluations: 118

List of tests:

ARIA 6 0 0 9 0 15

Element with `aria-hidden` has no focusable content	No evaluations available.
ARIA state or property is permitted	No evaluations available.
aria-* attribute is defined in WAI-ARIA	No evaluations available.
Element with role attribute has	All 5 evaluations

Element with role attribute has required states and properties

QW-ACT-R28 ACT 4e8ab6

This rule checks that elements that have an explicit role also specify all required states and properties.

Show evaluations by result:

✓ Pass: 1 ✓ Fail: 0
✓ Cannot tell: 0 ✓ Inapplicable: 4
✓ Uncompleted tests: 0

1 Highlight on page

```
<nav class="block-menu-block-1 " role="navigation"></nav>
```

Status: **inapplicable**

Reason: **The test target explicit role equals the implicit role.**

add an observation here

Manually change this result

--

2 Highlight on page

Figure 5.2: AccessBot result window appears when the user selects what type of evaluation(s) wants to perform on the page.

On top, the user has the option to access the Legend and the Filter section (figure 5.3). The Legend section presents to the user a description of the icons that appear on AccessBot. The filter, which is below the label "Show tests by result", presents the user options to filter the test results according to the status: pass, fail, inapplicable, cannot tell, and uncompleted tests that need user input.

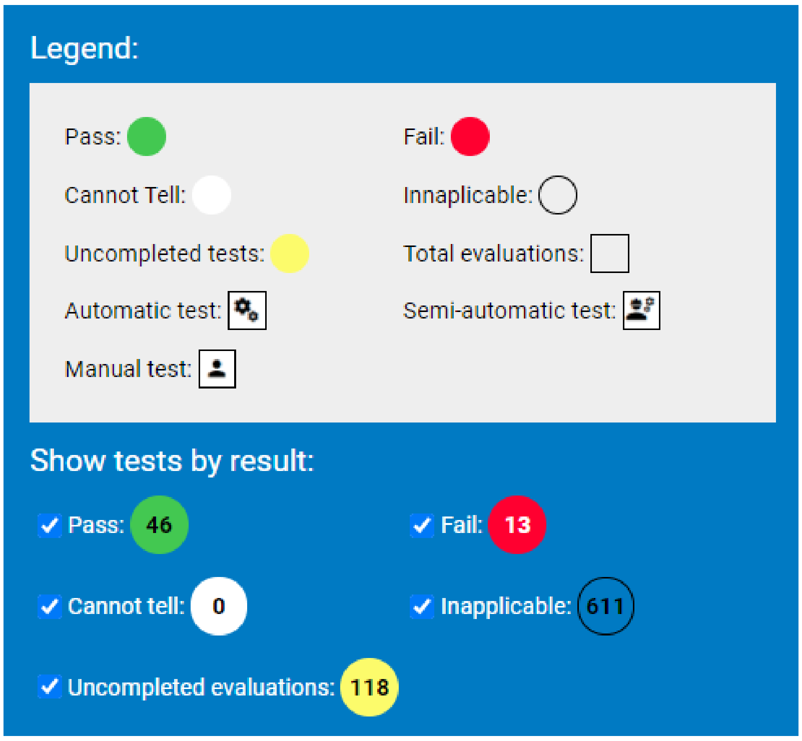
The test results are presented in accordion as collapsible content. There is a clear hierarchy presented. First, the user sees the different categories. When clicked, each category presents a set of rules, and each rule has a certain number of tests. Figure 5.4 shows the category *Form* selected exhibiting all the rules inside this category. In this case, the user selected the rule "Form control has an accessible name". According to the legend, it is a semi-automatic rule and has two evaluations that the user needs to complete.

Each evaluation, automatic, semi-automatic, and manual, is identified with specific symbols.

Each section in the hierarchy has individual counters beside the filter's global counters. The category counter gives the totals of tests of the rules of that category in the different states. The rule counter, which can be found on the right side of the window after the rule has been selected, gives information about the total tests and their states (figure 5.5).

When the user clicks in the rule, the elements that were evaluated are presented as enumerated card boxes (figure 5.6).

There is the name of the rule that was selected on the top of the right window, the

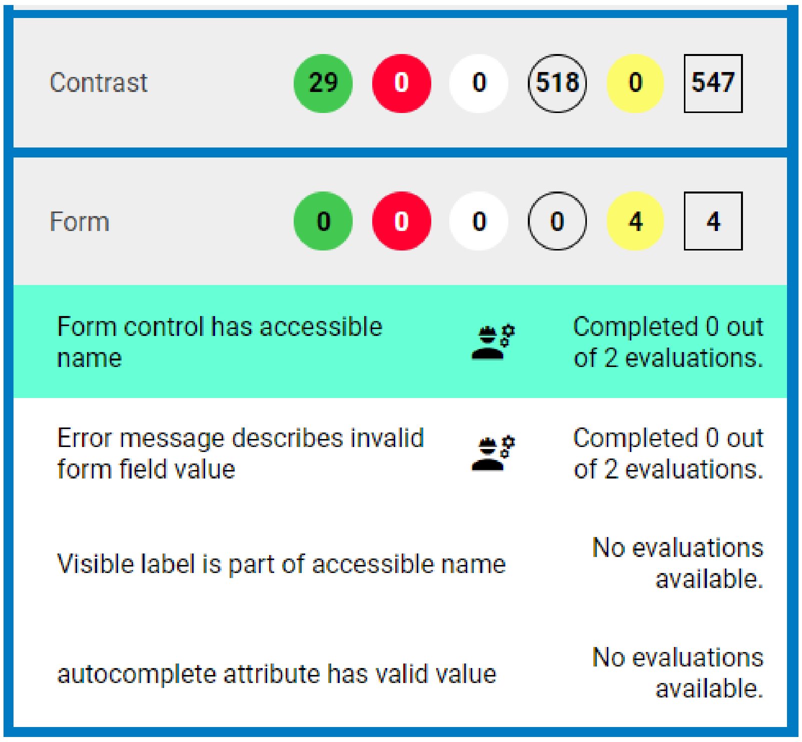

Legend:

Pass: 🟢	Fail: 🔴
Cannot Tell: ⚪	Innaplicable: ○
Uncompleted tests: 🟡	Total evaluations: □
Automatic test: ⚙	Semi-automatic test: 👤⚙
Manual test: 👤	

Show tests by result:

✓ Pass: 46	✓ Fail: 13
✓ Cannot tell: 0	✓ Inapplicable: 611
✓ Uncompleted evaluations: 118	

Figure 5.3: Legend and Filter section more detailed.

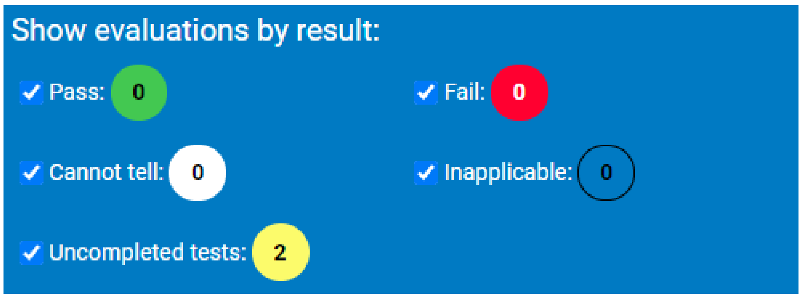


Contrast	29	0	0	518	0	547
Form	0	0	0	0	4	4

Form control has accessible name	👤⚙	Completed 0 out of 2 evaluations.
Error message describes invalid form field value	👤⚙	Completed 0 out of 2 evaluations.
Visible label is part of accessible name		No evaluations available.
autocomplete attribute has valid value		No evaluations available.

Figure 5.4: Form control accessible name rule selected (light blue) under the category Form. The other rules inside the category Form appear also.

QualWeb rule ID and ACT rule ID with its link to redirect the user to the rule's page on the ACT Rules-Community website (figure 5.7). The website has more detailed, easy to

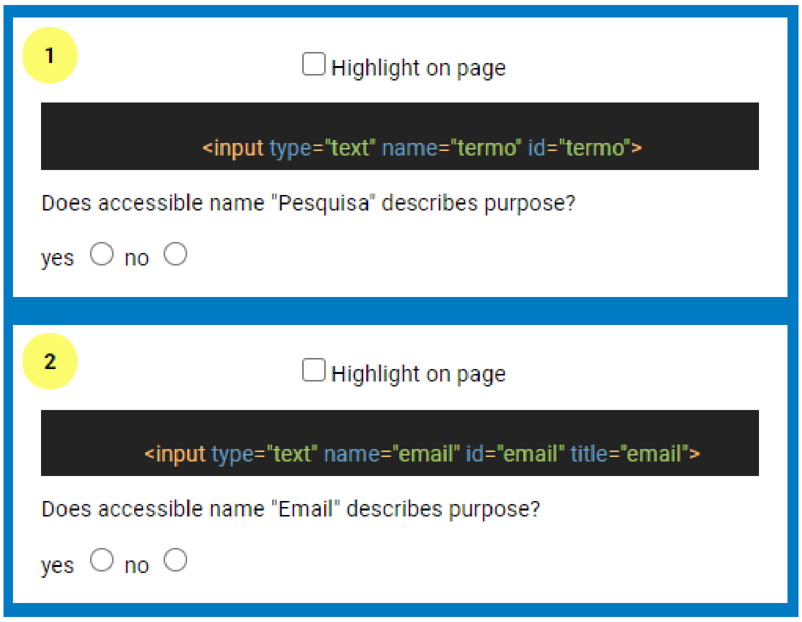

Show evaluations by result:

✓ Pass: 0 ✓ Fail: 0

✓ Cannot tell: 0 ✓ Inapplicable: 0

✓ Uncompleted tests: 2

Figure 5.5: When the rule is selected it appear the tests for the rule and the filter on the top of the window showing evaluations by result. In this case, it presents two uncompleted tests for the rule "Form control has accessible name".


1 Highlight on page

```
<input type="text" name="termo" id="termo">
```

Does accessible name "Pesquisa" describes purpose?

yes ◯ no ◯

2 Highlight on page

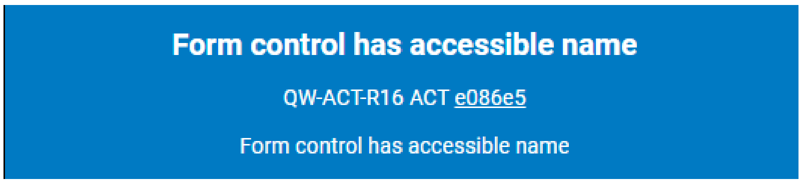
```
<input type="text" name="email" id="email" title="email">
```

Does accessible name "Email" describes purpose?

yes ◯ no ◯

Figure 5.6: White card box for each test result of a rule. In the case of the rule "Form has accessible name" there are two card boxes for uncompleted evaluations that require user input.

understand, and complete information about the rule. Immediately below, the user has a brief description text of what the rule checks.

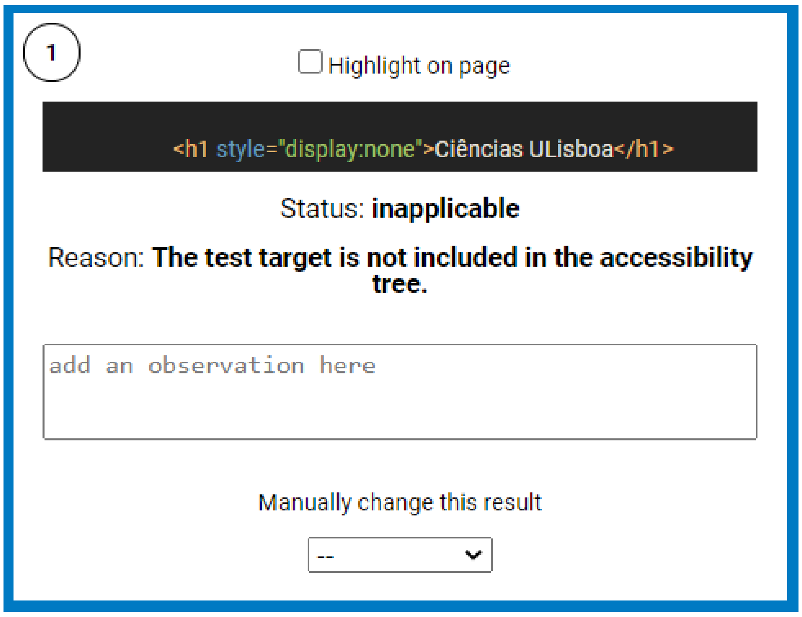

Form control has accessible name

QW-ACT-R16 ACT e086e5

Form control has accessible name

Figure 5.7: Information about the rule selected.

Each card box (figure 5.8) can highlight the element on the web page to be more easily identified by the user. It also has the HTML code for the evaluated element and the reason that justifies the evaluation's result state. For the automatic evaluations, the user can write notes that are automatically saved, and, in case of disagreeing with the result, the user can change the automatic result manually. Manually changing the result will alter the counters. Writing notes is also available in the semi-automatic and manual evaluations at the end of the evaluation.



```
<h1 style="display:none">Ciências ULisboa</h1>
```

Status: **inapplicable**

Reason: **The test target is not included in the accessibility tree.**

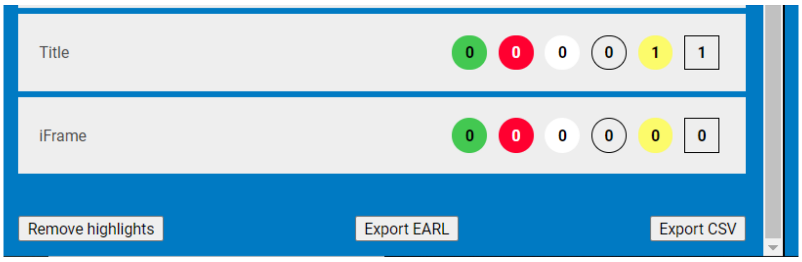
Figure 5.8: White card box features.

According to the algorithm defined for that test, semi-automatic and manual evaluations present questions to the user in the form of yes or no answers. At the end of the semi-automatic or manual evaluation appears a final result. This result is then updated in the counters. If the user makes a mistake, the answer can be reverted.

All the tests results for a particular rule on the right side also can be filtered according to the state selected by the user.

Suppose the user, during his evaluation of the web page, leaves numerous elements highlighted from different rules. In that case, there is an option to remove all highlights simultaneously using the button "Remove highlights" (figure 5.9).

After all the evaluations are performed, the user can export a report in CSV or EARL format (figure 5.9).



Title	0	0	0	0	1	1
iFrame	0	0	0	0	0	0

Figure 5.9: Remove highlights button, Export EARL button and Export CSV button.

Chapter 6

AccessBot Usability Testing

This chapter describes the usability testing of the AccessBot interface by users, specifically how it was performed, the behavior and reactions to the test, the results analysis, and the insights from users which lead to improvements to the interface. Usability testing is necessary to ensure that AccessBot is an effective, efficient, and enjoyable experience for users.

6.1 Usability Testing Method

The user testing method chosen was remote-moderated usability testing since I, the evaluator, cannot be present with the participants due to the COVID-19 pandemic. The user will share its screen while performing the test by using a video communication platform. It is a way to perform a qualitative "direct observation". Before the beginning of the test, users are asked to install the latest version of AccessBot. I introduce the test to participants, answer their queries, and ask them follow-up questions to perform specific prearranged tasks. The tests are recorded, under user authorization, to give the chance to review them later.

The process is standardized for all users in order to end up with consistent and reliable results. The answers to the questions asked, how they perform the tasks, and think aloud give a better understanding of their difficulties and allowed me to obtain the most information possible and include the user in the decision-making process.

The AccessBot user testing had two rounds. The first round had three participants, two developers, and one AMA accessibility evaluator. The second round had two participants, two AMA accessibility evaluators. After the first round, the problems users uncovered during testing were fixed and then revised. AccessBot interface is tested again in the second round. Not stopping after a single test, according to (Bevan et al., 2003; Nielsen et al., 1993), is a better use of limited resources than running a single usability test with the total number of users, in this case, five users.

This testing is one of the most thorough and in-depth methods for gaining user insights. It has some advantages, such as being less expensive, less time-consuming, and more convenient for the participants. The downside is the preparation of tasks to perform, questions to ask by being careful to not indirectly give the answers or influence results, setting up the tool used, and guide the users on how to install the application in the development phase.

6.2 Planning process for usability testing

Planning the details of the AccessBot usability testing is the most crucial part of the entire process. The purpose of the plan is to document what tools is going to use, the number of participants, and how the test will be administered. Each task is successfully completed when the participant indicates they have found the answer or completed the task goal.

The elements of the test plan are the following:

A The tools used for communication are the ZOOM[1] and Skype[2] communication platforms that allow the participants' screen and audio sharing. The participants should select a tool and be comfortable with the tool of choice.

B The tasks will be given to the participant by using the screen-sharing tool chat window during the session. Additionally, the evaluator reads the task out loud to the participant.

C The session is for one participant at a time, at a given hour and date. Since there is only one evaluator, this is a way to focus on one participant and focus on its behavior and thoughts.

D The participants should consent for their screen and audio to be recorded. The recording testing session gives a chance to review the participant's test later and capture data that the evaluator may miss.

E The participants are software developers or accessibility evaluators, considering the target users for AccessBot. The total number of participants is five, divided into two groups, one group per round. The total number of rounds is two. The first group has three participants and includes one accessibility evaluator but not a software developer. The other two participants do not have experience with accessibility but are software development students. The second group has two participants who are accessibility evaluators but not software developers. One of the participants in this group has a visual impairment expanding the testability of AccessBot by using a screen magnifier during the test. During the early stages of the design and

[1]The ZOOM application website: `https://zoom.us/`

[2]The Skype application website: `https://www.skype.com/pt/`

development process, a constructive approach is considered. This approach requires fewer participants than the summative, which requires a large number of users since the objective is to identify the main problems with the design, help figure out which features are useful and which are not, to improve the design. In the future, to collect an extensive amount of data and evaluate the design, a summative test will probably be required (Bennett, 2020).

F The website to be used during the testing is previously selected and the same for all five participants. The website URL is `https://ciencias.ulisboa.pt/`.

G The estimated total time is approximately one hour.

H Participants should perform the following tasks during the test:

1 Open the website URL to start to perform an accessibility evaluation;

2 Open AccessBot Chrome Extension;

3 Evaluate the webpage with all options selected;

4 Specify how many tests succeeded;

5 In the category Contrast indicate the number of total evaluations;

6 Select from the list the tests that failed;

7 In the category Language, observe the rules and point out the number of semi-automatic rules in that category;

8 Complete all the tests from the Language category;

9 Highlight the element on the uncompleted tests on the rule "Form control has an accessible name";

10 Look for the element that corresponds to the 13th fail result of the rule "Link has an accessible name";

11 Change the result of the element found in the last task to pass and add the observation "I changed the result to pass";

12 Remove the highlight of all elements highlighted;

13 Look up and specify one manual evaluation;

14 Export report CSV, open the file to check the content, and verify if the file contains the observation the tester has written and the changed result;

6.3 Usability Research Data Collection

The approach for organizing usability issues is to plot the data (Sauro et al., 2016) with issues in the rows and participants in the last columns. Table 6.1 represents the usability

Table 6.1: The table presents the users' failures in completing the predetermined tasks of the first round. Legend: ID - task identification; P(1-5) - Participants; x - failure to complete the task.

ID	WHERE	TASK	DES.	P1	P2	P3	P4	P5	OBS.
6	List of category of tests on the result window	Select from the test list which failed	Did not use the filter test option to select the tests that failed, instead used scroll	X	X	X			Filters not easily identified in the description, appears the word show instead of filters; Low contrast between the check boxes and the background colour
7	Category language tests on the result window	Indicate how many semi-automatic rules exist in the category language	The users did not give the correct answer. One user did not notice the semi-automatic icon on the legend while other noticed but did not make the association with the icon on the rule		X	X			Icons should be bigger in the legend or more distinct.
8	Category language tests on the result window	Complete all the tests from the Language category	Did not use the highlight option while completing the tests or used only for some tests.	X		X	X	X	Make the highlight option more visible.
10	List of test results of the rule "Link has an accessible name".	Identify the 13th element fail of the rule "Link has an accessible name"	Did not use the filter option to filter results that failed and gave wrong answer and scroll to search for the 13th of all results instead of the 13th of only the fails	X	X	X	X		Filters not easily identified in the description; appears the word show instead of filters; low contrast between the check boxes and the background colour.
12	Button "Remove highlights" on the left bottom of the results window.	Remove the highlight of all elements highlighted;	Did not use the button "Remove highlights" and opted to do it manually by searching for the elements that were previously highlighted	X	X			X	Change the location of the button to a more visible location.

research data collection from the first and second rounds of the usability AccessBot test. The first round includes participants P1, P2, P3. The second round includes participants, P4, and P5. Each issue presented in the tables has: the "ID" information column which corresponds to the number of the task in the planning process; the column "Where" locates where the task is performed in the AccessBot interface; the column "Task" is the task the participant should have completed successfully; the column "Description" (DES.), describes what the participant(s) did that lead to the failure to complete the task; the columns "P1", "P2", "P3", "P4", "P5" refer to the participants and if the column has a cross symbol (x), it means that particular participant failed to complete the task. The last column, "Observations" (OBS.), identifies the observations made by participants during the think-aloud test process.

6.4 Analysis of the First Round of User Testing

By analyzing the results representing the participants' responses, behavior, and observations, it is possible to verify that the tasks with the highest completion rates are the most straightforward ones, such as tasks 1, 2, 3, 4, 5, 9 and 11. These tasks briefly correspond to initiate AccessBot Chrome Extension and start evaluation and to specify, from direct observation of the counters, the totals of how many tests passed or the total evaluations performed for a specific category. Asking to highlight one element directly was another easy task performed by all users compared to not using the options highlight by themselves to help them perform other tasks. Writing observations in the text area was immediately performed by all the participants, although for two participants, there were some doubts if the observation written was saved. They were expecting to find, for example, a button that said "Submit" observation or the information that it had been saved.

The tasks that failed, meaning that one or more participants did not perform the task correctly, were 6, 7, 8, 10, and 12. Comparing these tasks, the tasks which all users failed were 6 and 10. The others had at least one participant that performed correctly. Task 6 and 10 are related to using the filters option to find the results to answer the tasks. The participants didn't use any of the filters available at all. By observing them and talking to the participants, they thought the filters were not there. Some possible reasons like the name "Filter" do not explicitly appear; low contrast with the background and input checkboxes that seemed like tick checkmarks which do not perform any action may have contributed to the participants not using the checkboxes. To perform these tasks, all participants chose to use the scrollbar.

By examining task 7, the icons corresponded to the automatic, semi-automatic, and manual rules in the legend, generated confusion for two participants. Somehow they could not correlate the legend to the icons in the rule. They noted that the icons were small and, because of that, seemed different.

Considering task 8, only one participant opted to use the highlight option to highlight the element being pointed for evaluation to answer the questions. The other participants, probably because they thought the answers to the category Language tests were more noticeable on the web page even without highlighting the element, they opted to infer the answers. The task to remove all previously highlighted elements was not done using the button "Remove Highlights". When the participants viewed the `result.html` window of AccessBot, they needed to scroll to the end to find the button; instead, they manually opted to turn the highlight off element by element. Only when they were asked to export the final evaluation report in CSV format (task 14), the participants explored better the interface to find the button to export. When they did that, they noticed other buttons such as "Remove Highlights" or "Export EARL".

Overall the participants concluded the usability test, on average, in one hour. They

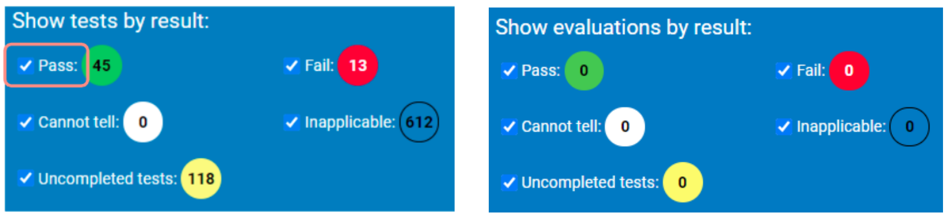


Figure 6.1: Left: Screenshot of AccessBot application showing the tests by result; Right: Screenshot of AccessBot application showing the evaluations by result.

were also generally satisfied with AccessBot functionalities. One of the participants said it was delighted and was going to use AccessBot in his work. The other said that although other evaluation tools pointed to the need to perform manual testing, the manual evaluations with questions to the user helped guide and save time on performing manual evaluations.

Taking these results into account and the recommendations for each issue presented, improvements to the AccessBot interface were implemented to make the user experience better and then re-tested.

6.5 Improvements after First Round of User Testing

After user testing and analysis, it is necessary to perform optimizations in user experience according to the observations findings. This section presents screenshots before AccessBot user testing and after to illustrate the improvements made.

Considering issues 6 and 10 of table 6.1, the users did not use the filter option to filter tests or obtain quick answers. Instead, the users opted to scroll and check item by item. The issue was potentially caused by the low contrast between the input checkboxes in the filters and the background, as seen in figure 6.1, which made it difficult for users to realize that there is a filter option they can use to filter tests and tests's results. This issue is resolved by increasing contrast with a white background box in the filters section, such as in figure 6.2.

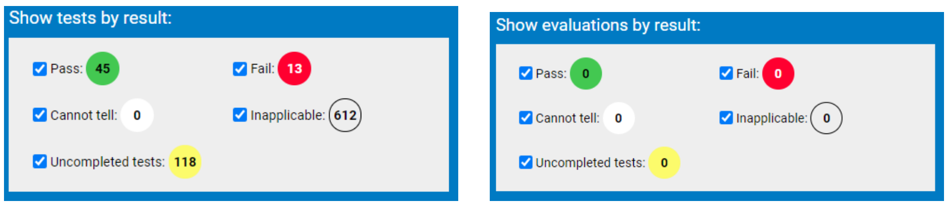
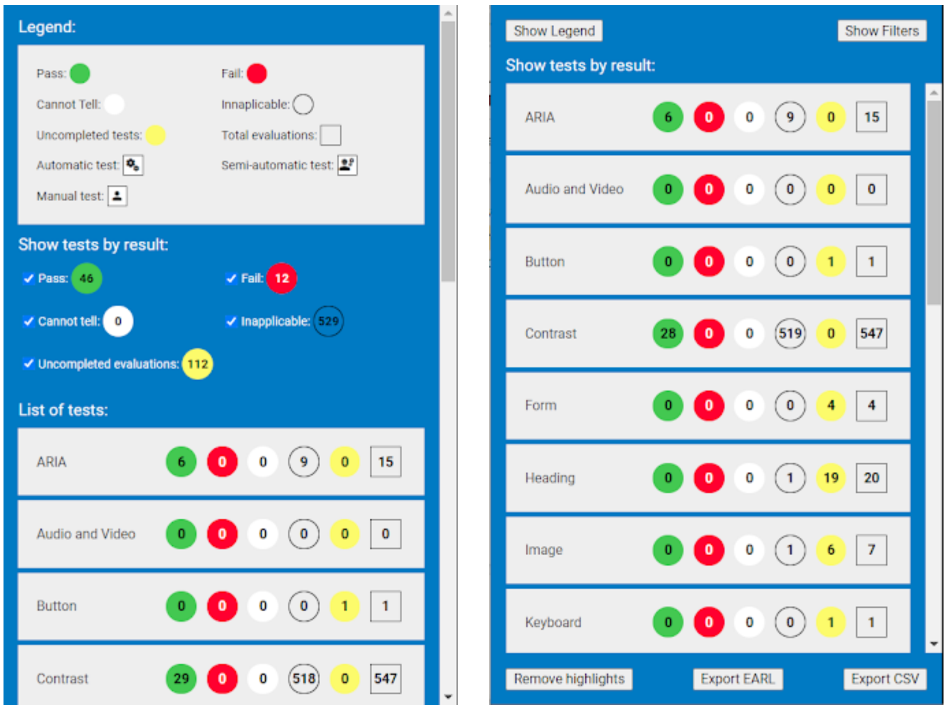


Figure 6.2: Left: Screenshot of AccessBot application showing the tests by result after modification; Right: Screenshot of AccessBot application showing the evaluations by result after modification. In both, the checkboxes are more visible

To expand the visual area of "List of tests", the Legend and Filter were collapsed to a button triggered accordion to hide them but also redirect the user more easily to Legend and Filter options. The scroll now also starts at the same level of the test results. For the same window size, figure 6.3 on the right shows a clear improvement, compared with the figure on the left, in user presentation of categories.



Legend:

Pass: 🟢	Fail: 🔴
Cannot Tell: ⚪	Innaplicable: ○
Uncompleted tests: 🟡	Total evaluations: □
Automatic test:	Semi-automatic test:
Manual test:	

Show tests by result:

✓ Pass: 46 ✓ Fail: 12
✓ Cannot tell: 0 ✓ Inapplicable: 529
✓ Uncompleted evaluations: 112

List of tests:

ARIA	6	0	0	9	0	15
Audio and Video	0	0	0	0	0	0
Button	0	0	0	0	1	1
Contrast	29	0	0	518	0	547

Show Legend Show Filters

Show tests by result:

ARIA	6	0	0	9	0	15
Audio and Video	0	0	0	0	0	0
Button	0	0	0	0	1	1
Contrast	28	0	0	519	0	547
Form	0	0	0	0	4	4
Heading	0	0	0	1	19	20
Image	0	0	0	1	6	7
Keyboard	0	0	0	0	1	1

Remove highlights Export EARL Export CSV

Figure 6.3: Left: Screenshot of AccessBot application showing Legend and Filter before modification; Right: Screenshot of AccessBot application showing two buttons "Show Legend" and "Show Filter". After collapsing the Legend and Filter section, the area showing the list of tests increased significantly.

For task 7, two of the three users had difficulty giving the correct answer besides checking Legend a few times. They pointed out that the Legend was too small to be perceptible. To change this, the border squares were removed, and the size increased by approximately 10 pixels (figure 6.4).

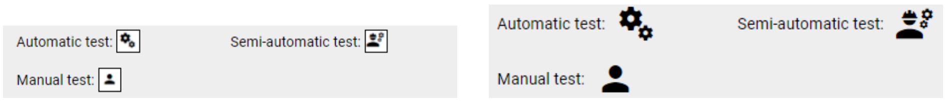
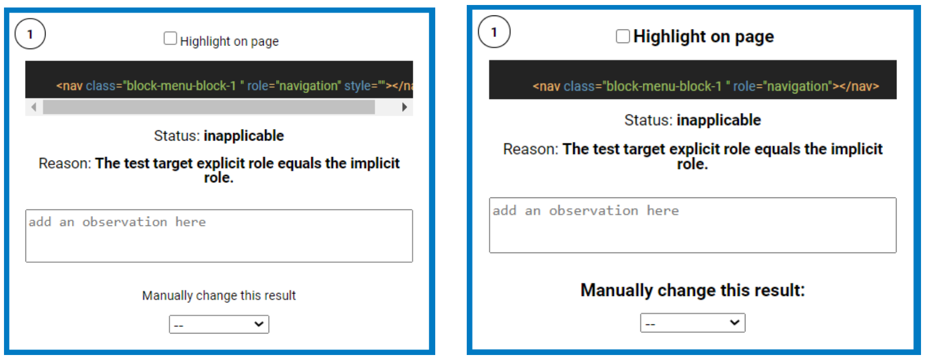


Figure 6.4: Left: Legend design before improvement; Right: Legend design after modification. Icons are bigger and squares around icons removed.

Considering task 8, most of the time, the users did not use the highlight option to identify the elements on the web page that is being evaluated. However, when asked to highlight in task 9, all the users completed the task successfully. When discussing this issue with the users, some pointed out that the highlight option could be more visible, but overall, they did not need it to perform task 8. Probably this happened since the semi-automatic evaluation on task 8 was considered manageable. Either way, the option to highlight the element was increased in size and font-weight. This modification was also done to the option to change the result manually (figure 6.5).



1 Highlight on page

```
<nav class="block-menu-block-1 " role="navigation" style=""></n
```

Status: **inapplicable**

Reason: **The test target explicit role equals the implicit role.**

add an observation here

Manually change this result

--

1 **Highlight on page**

```
<nav class="block-menu-block-1 " role="navigation"></nav>
```

Status: **inapplicable**

Reason: **The test target explicit role equals the implicit role.**

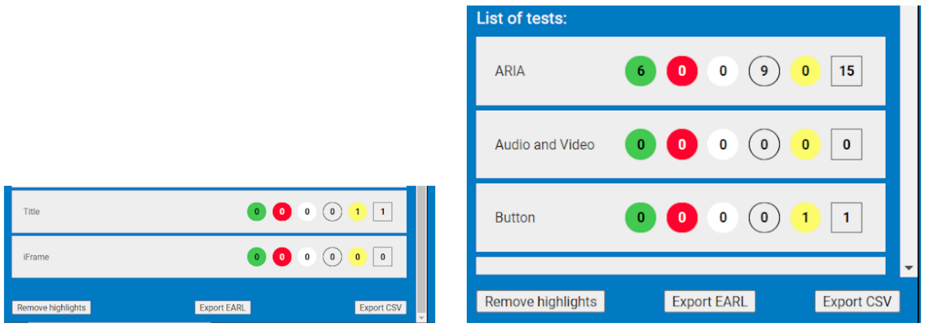
add an observation here

Manually change this result:

--

Figure 6.5: Left: Legend design before improvement; Right: Legend design after improvement. The "Highlight on page" together with "Manually change this result" is more noticeable.

Considering task 12, the "Remove highlights" button and other buttons at the end of the scroll on the left side of the window were not easily accessible. To change this situation, the buttons are now fixed to be readily available to the user when the result window pops up, so the user does not need to scroll to the end of the window to see the buttons (figure 6.6).



Title	0	0	0	0	1	1
iFrame	0	0	0	0	0	0

Remove highlights Export EARL Export CSV

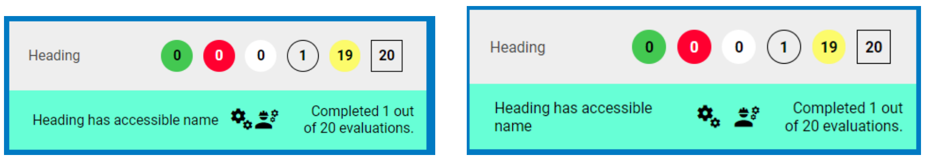
List of tests:

ARIA	6	0	0	9	0	15
Audio and Video	0	0	0	0	0	0
Button	0	0	0	0	1	1

Remove highlights Export EARL Export CSV

Figure 6.6: Left: The list of tests scroll reaches the end of the window. The user had to scroll to the end to visualize the buttons; Right: Scroll fixed on the window. When the window appears, the buttons are immediately visible.

Other usability suggestions by the participants that was implemented was the broader separation of automatic and semi-automatic icons when a rule has automatic and semi-automatic tests (figure 6.7).



Heading 0 0 0 1 19 20

Heading has accessible name Completed 1 out of 20 evaluations.

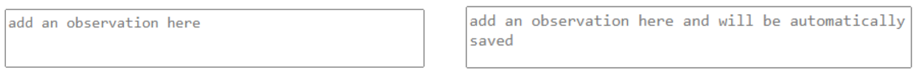
Heading 0 0 0 1 19 20

Heading has accessible name Completed 1 out of 20 evaluations.

Figure 6.7: Left: icons showed, for example, rule "Heading has an accessible name" before improvement; Right: After modification, the icons are separated, increasing perception.

The text written in the input field for observations, although automatically saved, left the users wondering if they needed to perform an action after writing their observation.

To change this, the input field gives information that the observation will be automatically saved (figure 6.8).


add an observation here

add an observation here and will be automatically saved

Figure 6.8: Left:Input field for writing observations before improvement; Right: Input field informing the user the observation will be automatically saved.

6.6 Analysis of the Second Round of User Testing

The second round included two experienced accessibility evaluators. The evaluators had a few minutes (approximately five minutes) to engage with the interface, which seemed to decrease the distress in completing the tasks and increased confidence and calmness. Time for engagement with the interface did not happen in the first round, where evaluators received the tasks soon after starting the evaluation. They generally were more anxious looking for the answers to the tasks, and that is why in the second round, I made this adjustment to get a more in-depth level of insight on how to improve the content. The problems identified in the first round were mostly solved as shown by the analysis of the second-round results after implementing the improvements described.

To complete task 8, the evaluators used the highlight option to highlight some tests but not all. The occasional use of the highlight option probably happened because they could infer the semi-automatic evaluation's answers by only reading the element HTML code.

Task 10 failed to one participant on the second round while the other completed successfully. The earlier participant did not use the filter option and instead used the scroll to find the result that failed between all the total results instead of using the filter; the other participant used the filter to select the results that failed and found the required element.

Both of the users completed task 13, but there is still some frustration and much time spent searching for a manual evaluation icon when all the evaluation types are preselected. This means that probably some improvement concerning this aspect could also be made.

For participant 4, there was disorientation regarding the result window's left side and the tests shown on the window's right side. Although on top of the right side of the window there is the description of the rule which the results are being presented and the rule selected is in different color light blue, the participant sometimes felt lost. Probably there should be a stronger view association between the rule that is being evaluated and the right side of the window.

All the participants in the first round completed the tasks on Windows. One of the participants of the second round tested AccessBot on Mac. The possibility to use Mac is significant since browser Chrome may behave differently depending on the operating system. The behavior of AccessBot during user testing was stable without any differences from Windows participants.

In the second round, one participant has visual impairment which provided another critical perspective of the AccessBot interface. A screen magnifier was used during testing. The main difficulty was when the participant's mouse cursor clicked on the button "Show Legend" or "Show Filters", a toolbar of the screen magnifier appeared, and the participant was unable to click the buttons. To increase the readability of the interface elements was suggested to add tooltips to the interface.

Overall, the tests performed were satisfactory. According to the evaluator's feedback, they are pleased with AccessBot functionalities and features, including the manual and semi-automatic evaluations and how the data is presented to the user. The less positive aspects were mainly cosmetic, and the future changes will increase the usability of using AccessBot to reach a more significant number of users with or without disabilities. Some minor cosmetic improvements will still be made before the completion of the thesis. However, additional design changes and new suggested features will be considered in the future due to time constraints. The interface is a continuous work in progress, and more usability tests are necessary to keep up the development.

6.7 Improvements after Second Round of User Testing

The improvements made after the second round are mainly cosmetic to increase the visibility and usability of features.

Besides optimizing color contrast, the participants noted the font size was small. In the context of accessibility, to improve it for those who experience a visual impairment that is not color blindness, optimizing font sizes also benefits users. Although there is no official WCAG font size standard, the font size 15px was suggested through all AccessBot application.

The `popup.html` also improved the checkboxes to be according to the other checkboxes in AccessBot by putting the checkbox before text and increasing the contrast with the background's gray square box (figure 6.9).

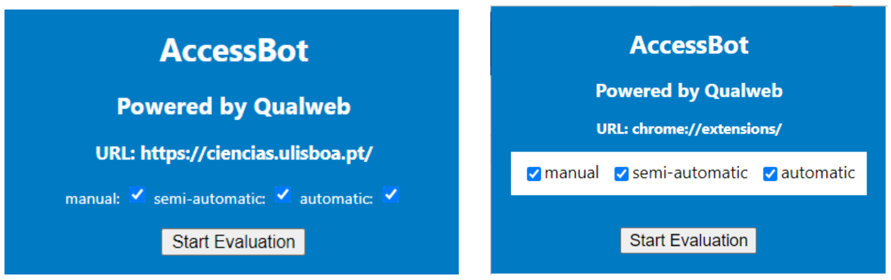

AccessBot
Powered by Qualweb
URL: https://ciencias.ulisboa.pt/
manual: ✓ semi-automatic: ✓ automatic: ✓
Start Evaluation
AccessBot
Powered by Qualweb
URL: chrome://extensions/
✓ manual ✓ semi-automatic ✓ automatic
Start Evaluation

Figure 6.9: Left: Popup window before improvement. Right: Popup window with modifications in the checkboxes to increase contrast and visibility.

When the `result.html` appeared to participants, the buttons “Show Legend” and “Show Filters” were collapsed and the content was hidden by default. The user needed to click buttons in order to see the content. However, it was a suggestion by one of the participants to have the “Show Legend” automatically visible when the user clicks in “Start Evaluation” in order to be readily visible to the user. If the user wanted to hide information, he only needed to click the button. The button “Show Filters” remained collapsed by default to reduce information clutter and allow good visibility of the Accessbot interface.

It was suggested to change the counters’ visual appearance in Accessbot to increase the visual difference between success, alert, and fail. To make it more intuitive and improve user experience, it was decided to use the traffic light colors order: green for a pass, yellow as an alert of uncompleted evaluations, and red for fail, followed by cannot tell, inapplicable and total evaluations. The space between the circles and square shapes was increased to distance the total evaluations from the individual evaluations visually. These changes are shown in figure 6.10.

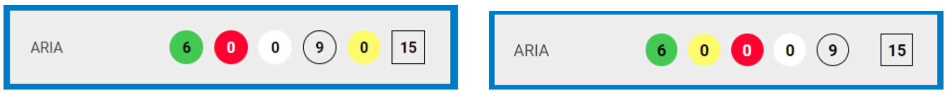


Figure 6.10: Left: Category counters before improvement. Right: Category counters organized using the traffic light system to be more intuitive for users.

To note that the global counters, although located inside the filters option, are also visible outside this option, so the user can immediately observe the counters without the need to click the button “Show Filters” (figure 6.11).

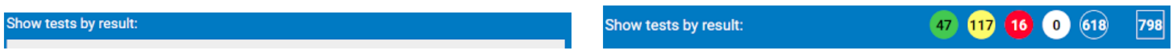


Figure 6.11: Left: No global counters shown outside ”Show Filters” section. Right: Global counters shown outside ”Show Filters” section and organized also by using the traffic lights system.

When changing manually the result of the automatic evaluations, one of the participants was blocked after getting the alert yellow sign, thinking that he should do something. To make it more straightforward, the alert sign was changed to a tick sign to denote choice and indicate the concept “yes, it is done” (figure 6.12).

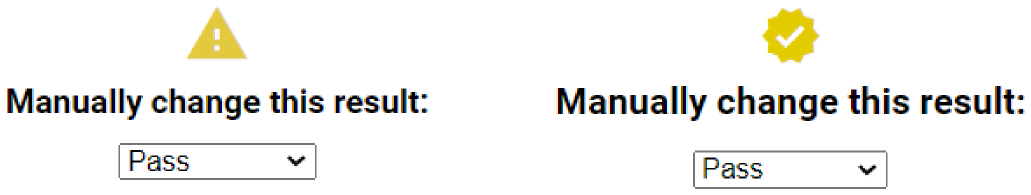


Figure 6.12: Left: Alert icon when user changed the result of an automatic evaluation; Right: modification of the icon to tick sign instead of an alert sign.

Contrast helps the design to be organized and to emphasize a focal point. In this case, although the links of AccessBot, which redirect the user to ACT-Rules Community, have not been evaluated directly in user testing, it was proposed to change the color of links in order to be more noticeable. This change is indicated in figure 6.13.

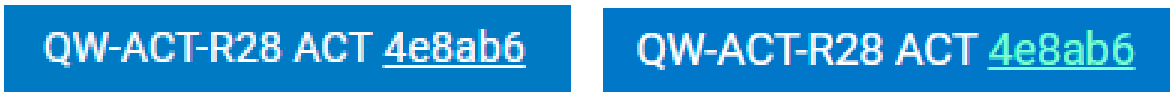


Figure 6.13: Left: Link for the ACT Rule website in white; Right: Link with different color, in light blue, to help users distinguish better the link and the Rule ID.

Tooltips were added to better identify the counter elements when the user hovers over or focuses on, as suggested by one of the participants. They contain the text information matching the circular counter symbol to the status they represent, as seen in figure 6.14, without the need to consult the legend.

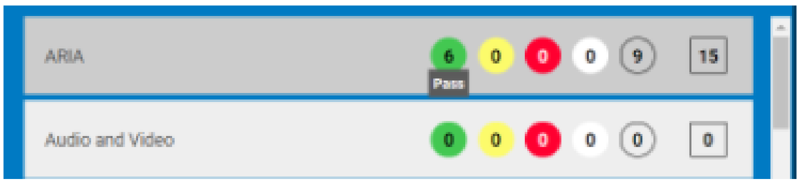


Figure 6.14: Example of tooltip showing that the green circular counter corresponds to the pass test results. Note that the pointer is not visible when taking the screenshot.

Chapter 7

Conclusion

The web is an essential-good for all people with or without disabilities and it should be accessible to everyone. Different forms of disabilities should be taken into account such as visual, hearing, motor, and cognitive disabilities. To know if a website is accessible to everyone it needs to be evaluated or tested. The principal objective in accessibility testing is to identify errors and provide information to developers. This will promote future design and implementation changes that improve the site's usability and accessibility.

Nowadays, the monitoring of public websites accessibility is mandatory. It guarantees that the Government and similar entities should fulfill the rules demanded by decrees of law.

Accessibility testing relies on standards W3C. The standards provide success criteria and techniques that support several components of web development. Some countries also have specific government guidelines that complement the W3C standards and the websites should fulfill both of these standards.

The implementation of these guidelines on identifying web accessibility issues can be done using automatic testing, however, it can only found about 30% of the problems. The rest of the problems must be located using manual testing which can detect a bigger number of accessibility problems. AccessBot was created to have a broader coverage of success criteria combining automatic testing with manual testing. It complements automatic evaluations from the QualWeb engine, in a Chrome extension environment.

The assisted evaluation by AccessBot can be supported in different ways. It can identify all the elements affected by the automatic evaluation from QualWeb and the user can alter its results manual, or presents a procedure (list of steps) for a technique, guiding the user during the process. The process of guiding the user can have a semi-automatic component if AccessBot relies on automatic evaluation by QualWeb but needs the input of the user to reach a final result or manual, where there is no interference from QualWeb and the manual evaluation is intrinsic to AccessBot.

Besides providing an assisted evaluation, another objective of the AccessBot is to enable unequivocal interpretation and implementation of testing methods by using the rules defined according to ACT Rules. This enables transparency and harmonization of testing methods and resolves some of the problems with the interpretation of the WCAG success criteria and techniques.

Preliminary to the development of AccessBot, a study was done to obtain an overall perspective of what Chrome extensions exist at the present moment, which ones are most used by developers, their strengths and weaknesses. Overall, the tools analyzed were easy to install and use but not flawless, since during evaluations, for example, they crashed and it was necessary to refresh the browser. The results obtained between tools varied in what success criteria they evaluate (for example, WAVE does not evaluate level AAA conformance criteria), varied in classifying the results according to its impact on the user, and in classifying the result to be an error or a warning. All these variances are a consequence of how specifications and heuristics are implemented for each tool. The analysis showed that individual tools have poor coverage of the WCAG 2.1 success criteria. If more than one tool is used together the coverage increases, affording approximately 10% to 40% more coverage than if using only one tool. Given that the best option seems to be to use more than one tool, to improve accessibility and adherence to developing accessible websites.

Although not perfect and with limitations, the automatic tools remain important to help users evaluate websites to find most violations of success criteria that can be automated. However, they should always be complemented with manual testing procedures and the results are analyzed objectively with reasoned judgment.

The planning execution of design, implementation, and user testing of AccessBot were performed in the time frame planned, although there were some challenges.

The implementation logic for AccessBot has three fundamental aspects. The connection to QualWeb, the implementation of the semi-automatic and manual algorithms for the assisted evaluations, and the development of features to improve users' usability. The features developed are, for example, the possibility for the user to choose which type of evaluations he wants to perform such as manual, semi-automatic, and automatic; filter options to hide unwanted result states on tests and tests results; counters to store the different state values; the possibility to change the result of automatic evaluations; highlighting of elements and to export the final evaluation report in CSV and EARL format.

The AccessBot user testing consisted of two rounds with a total of five participants. It gave a deep insight into improvements that needed to be done to learn about users' behaviors and preferences while using AccessBot. After the first round, cosmetic and functionality improvements were made. These were tested in the second round and the difficulties of the first round were overtaken and new suggested cosmetics and features were also made.

In general, the development of the thesis contributed to the accessibility field in informatics, since AccessBot received positive feedback from AMA, the public institution responsible to guarantee that the public government websites and similar entities will improve their accessibility by respecting the accessibility rules demanded. It also became apparent that assisted evaluation is of major importance; it helps overtake a considerable part of the accessibility problems that automatic tools have and should complement automatic evaluations.

The project AccessBot intends to be continually updated to have the more recent ACT rules and to be continually developed to reach a greater number of users.

7.1 Future Work

After user testing additional changes were made. However, other user interface and features suggestions were not considered high priority but are presented here and may be considered future work.

- AccessBot revealed to have contrast problems. Some of the problems were corrected, but others still need to be addressed, such as modifying background color to increase contrast. An example given was to change the background to the dark blue color; another example was to have an icon to increase contrast if needed. AccessBot contrast is vital since it is meant to be used by many users or accessibility evaluators that may have low vision, low contrast vision, or color vision deficiency that require sufficiently-contrasting colors.

- Improvement of visual information content, for example, apply a filter icon. Visual communication is about sharing information in an exact accurate way that involves visual elements. Nowadays, since users are used to much visual stimulation, some users may prefer icons and skip text information, which may cause loss of content.

- Although AccessBot is designed to be easy to understand, one of the user testing participants referred that a popup tip was useful to provide helpful and additional content on how to use AccessBot. This feature will help users eventually not used to accessibility evaluations since it has the main objective of thoroughly explaining how the evaluation is done together with accessibility definitions.

- Create a Portuguese version of AccessBot; as said, AccessBot intends to reach a broader audience, and in the future, a translation to Portuguese will be considered;

- Export button feature in HTML besides CSV and EARL format. The structure would be different from the CSV and EARL that present results according to the rules. The HTML structure would aggregate results according to their status; for example, for all the tests that passed or failed independently of the rules, the user would have access to the list.

- During web page testing, users cannot interact with the web page because they can change the web page and need to restart the AccessBot evaluation. To avoid this, it is necessary to implement an event with the information “Assessed page has changed. The results are not up to date.” in order to inform users.

- An option to “check/uncheck all” options if the user needs to select or deselect various result states in the filters section.
- Ability to use CSV upload in order to pre-fill AccessBot results evaluation. If necessary, this step would be useful to divide the evaluation between different evaluators and continue a previous evaluation session.
- Unite AccessBot with machine learning by helping AccessBot finding patterns in results. This solution is useful in case of having more than hundreds of test results for one rule and the users, instead of evaluating all of them, start by evaluating a few. AccessBot recognizes there is a pattern and concludes all the rest of the evaluations.

Bibliography

Accessibility Guidelines Working Group (2017). *Techniques for WCAG 2.1*. Updated 10 July 2020. URL: `https://www.w3.org/WAI/WCAG21/Techniques/`. (accessed: 15.08.2020).

ACT-Rules Community Group (Apr. 2019). *About Us — ACT-Rules Community*. URL: `https://act-rules.github.io/pages/about/`. (accessed: 30.12.2019).

Bennett, Jessica (Jan. 2020). *Formative Vs Summative : The User Testing Battle*. URL: `https://usabilitygeek.com/formative-vs-summative-the-user-testing-battle/`. (accessed: 20.09.2020).

Bevan, Nigel et al. (2003). "The "Magic Number 5": Is It Enough for Web Testing?" In: *CHI '03 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '03. Ft. Lauderdale, Florida, USA: Association for Computing Machinery, pp. 698–699. ISBN: 1581136374. DOI: `10.1145/765891.765936`. URL: `https://doi.org/10.1145/765891.765936`.

Brajnik, Giorgio (2008). "Beyond Conformance: The Role of Accessibility Evaluation Methods". In: *Web Information Systems Engineering – WISE 2008 Workshops*. Ed. by Sven Hartmann, Xiaofang Zhou, and Markus Kirchberg. Lecture Notes in Computer Science. Berlin, Heidelberg, pp. 63–80. URL: `https://doi.org/10.1007/978-3-540-85200-1_9`.

Brajnik, Giorgio, Andrea Mulas, and Claudia Pitton (2007). "Effects of sampling methods on web accessibility evaluations". In: *Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility - Assets '07*. DOI: `10.1145/1296843.1296855`. URL: `https://dl.acm.org/doi/10.1145/1296843.1296855`.

Carvalho, Michael Crystian Nepomuceno et al. (2018). "Accessibility and Usability Problems Encountered on Websites and Applications in Mobile Devices by Blind and Normal-Vision Users". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: Association for Computing Machinery, pp. 2022–2029. ISBN: 9781450351911. DOI: `10.1145/3167132.3167349`. URL: `https://doi.org/10.1145/3167132.3167349`.

Duarte, Carlos Miguel Ribeiro (2018). "ISIAAW - Interpretação Semântica de Imagens na Avaliação da Acessibilidade Web". MA thesis. Lisboa: Faculdade Ciências, Universidade Lisboa.

Fernandes, Nádia et al. (2012). "Evaluating the Accessibility of Rich Internet Applications". In: *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*. W4A '12. Lyon, France: Association for Computing Machinery. ISBN: 9781450310192. DOI: 10.1145/2207016.2207019. URL: https://doi.org/10.1145/2207016.2207019.

Foley, E (July 2019). *W3C Publishes Draft of Standard Format for Writing Accessibility Test Rules - Level Access*. URL: https://www.levelaccess.com/w3c-publishes-draft-of-standard-format-for-writing-accessibility-test-act-rules/. (accessed: 30.12.2019).

Frazão, Tânia and Carlos Duarte (2020). "Comparing Accessibility Evaluation Plug-Ins". In: *Proceedings of the 17th International Web for All Conference*. W4A '20. Taipei, Taiwan: Association for Computing Machinery. ISBN: 9781450370561. DOI: 10.1145/3371300.3383346. URL: https://doi.org/10.1145/3371300.3383346.

Google (2020a). *chrome.browserAction*. URL: https://developer.chrome.com/extensions/browserAction. (accessed: 16.03.2020).

— (2020b). *Design User Interface*. URL: https://developer.chrome.com/extensions/user_interface. (accessed: 16.03.2020).

— (2020c). *Getting Started Tutorial*. URL: https://developer.chrome.com/extensions. (accessed: 15.03.2020).

— (2020d). *Manage Events with Background Scripts*. URL: https://developer.chrome.com/extensions/background_pages. (accessed: 16.03.2020).

— (2020e). *Manage Events with Background Scripts*. URL: https://developer.chrome.com/extensions/background_pages. (accessed: 16.03.2020).

— (2020f). *Overview*. URL: https://developer.chrome.com/extensions/overview. (accessed: 20.03.2020).

— (2020g). *What are extensions?* URL: https://developer.chrome.com/extensions. (accessed: 15.03.2020).

Government Digital Service, Uk's Cabinet Office (Apr. 2018). *Accessibility tools audit results - Overview - GDS accessibility team*. URL: https://alphagov.github.io/accessibility-tool-audit/. (accessed: 10.12.2019).

Hudson, Roger (Nov. 2011). *Measuring accessibility*. URL: https://usability.com.au/2011/11/measuring-accessibility/. (accessed: 30.12.2019).

Kirkpatrick, Andrew et al. (June 2018). *Web Content Accessibility Guidelines (WCAG) 2.1*. W3C Recommendation. URL: https://www.w3.org/TR/WCAG21/. (accessed: 30.12.2019).

Matos, Inês (2017). "SCREW - Semantic Content Analysis for Repair and Evaluation of Web Accessibility". MA thesis. Lisboa: Faculdade Ciências, Universidade Lisboa.

Mehta, Prateek (2016). *Creating Google Chrome extensions*. Apress.

Ministério da Ciência e da Tecnologia e Ensino Superior (2018). “Decreto-Lei n.º 83/2018”. In: *Diário da República n.º 202/2018, Série I de 2018-10-19*.

Nielsen, Jakob and Thomas K. Landauer (1993). “A Mathematical Model of the Finding of Usability Problems”. In: *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*. CHI ’93. Amsterdam, The Netherlands: Association for Computing Machinery, pp. 206–213. ISBN: 0897915755. DOI: 10.1145/169059.169166. URL: https://doi.org/10.1145/169059.169166.

Petrie, Helen, Ra Harrison, and Sundeep Dev (2005). “Describing images on the web: a survey of current practice and prospects for the future”. In: *Proceedings of 3rd International Conference on Universal Access in Human-Computer Interaction*.

Rømen, Dagfinn and Dag Svanæs (2012). “Validating WCAG versions 1.0 and 2.0 through usability testing with disabled users”. In: *Universal Access in the Information Society* 11.4, pp. 375–385. DOI: 10.1007/s10209-011-0259-3. URL: https://doi.org/10.1007/s10209-011-0259-3.

Santos Vicente, João Afonso Leal dos (2018). “Migração do Observatório Português de Acessibilidade Web”. MA thesis. Lisboa: Faculdade Ciências, Universidade Lisboa.

Sauro, Jeff and James R. Lewis (2016). “Chapter 2 - Quantifying user research”. In: *Quantifying the User Experience (Second Edition)*. Ed. by Jeff Sauro and James R. Lewis. Second Edition. Boston: Morgan Kaufmann, pp. 9–18. ISBN: 978-0-12-802308-2. DOI: https://doi.org/10.1016/B978-0-12-802308-2.00002-3. URL: http://www.sciencedirect.com/science/article/pii/B9780128023082000023.

Schmutz, Sven, Andreas Sonderegger, and Juergen Sauer (2016). “Implementing Recommendations From Web Accessibility Guidelines”. In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 58.4, pp. 611–629. DOI: 10.1177/0018720816640962. URL: https://www.ncbi.nlm.nih.gov/pubmed/27044605.

Stephanidis, Constantine, Helen Petrie, and Nigel Bevan (2009). “The Evaluation of Accessibility, Usability, and User Experience”. In: *The universal access handbook*. Vol. 20091047. Human Factors and Ergonomics. Taylor & Francis Group, pp. 1–16. URL: http://www.crcnetbase.com/doi/abs/10.1201/9781420064995-c20%20http://dx.doi.org/10.1201/9781420064995-c20.

UXPA International, Eduardo Meza-Etienne, and Kara Zirkle (June 2019). *WCAG 2.1: What You Need to Know About the Most Recent Accessibility Standards*. UXPA International Conference. URL: https://pt.slideshare.net/UXPA/wcag-21-what-you-need-to-know-about-the-most-recent-accessibility-standards. (accessed: 30.12.2019).

Vigo, Markel, 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*. W4A ’13. Rio de Janeiro, Brazil: Association for Computing Machinery. ISBN: 9781450318440. DOI: 10.1145/2461121.2461124. URL: https://doi.org/10.1145/2461121.2461124.

W3C (Oct. 2019a). *Accessibility Conformance Testing (ACT), Rules Format 1.0*. URL: https://www.w3.org/TR/act-rules-format/. (accessed: 30.12.2019).

— (May 2019b). *Accessibility Testing*. URL: https://www.w3.org/wiki/Accessibility_testing. (accessed: 30.12.2019).

W3C Web Accessibility Initiative (Feb. 2005). *Introduction to Web Accessibility*. URL: https://www.w3.org/WAI/fundamentals/accessibility-intro/. (accessed: 18.11.2019).

— (Mar. 2006). *W3C Web Accessibility Initiative Web Accessibility Evaluation Tools List*. Information on specific evaluation tools is updated frequently. URL: https://www.w3.org/WAI/ER/tools/. (accessed: 05.12.2019).

— (2016). *Accessibility, Usability, and Inclusion*. URL: https://www.w3.org/WAI/fundamentals/accessibility-usability-inclusion. (accessed: 05.01.2021).

— (Dec. 2017a). *Selecting Web Accessibility Evaluation Tools*. Ed. by Shadi Abou-Zahra, Nicolas Steenhout, and LauraEditors Keen. URL: https://www.w3.org/WAI/test-evaluate/tools/selecting/. (accessed: 30.12.2019).

— (2017b). *Understanding Conformance*. Updated 10 July 2020. URL: https://www.w3.org/WAI/WCAG21/Understanding/conformance. (accessed: 15.08.2020).

— (2017c). *Understanding Techniques for WCAG Success Criteria*. Updated 10 July 2020. URL: https://www.w3.org/WAI/WCAG21/Understanding/understanding-techniques.html. (accessed: 15.08.2020).

— (2019a). *Evaluating Web Accessibility Overview*. Ed. by Shawn LawtonEditor Henry. Updated 19 October 2020. URL: https://www.w3.org/WAI/test-evaluate/. (accessed: 25.11.2019).

— (Oct. 2019b). *How to Meet WCAG (Quick Reference)*. URL: https://www.w3.org/WAI/WCAG21/quickref/. (accessed: 25.11.2019).

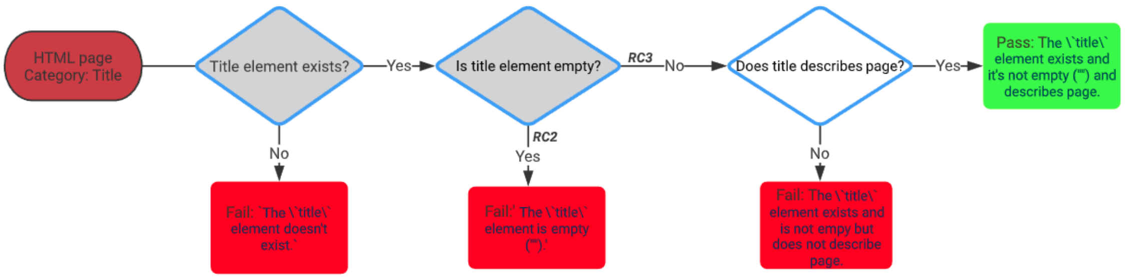
— (Mar. 2019c). *W3C Accessibility Standards Overview*. URL: https://www.w3.org/WAI/standards-guidelines/. (accessed: 24.11.2019).

Appendix A

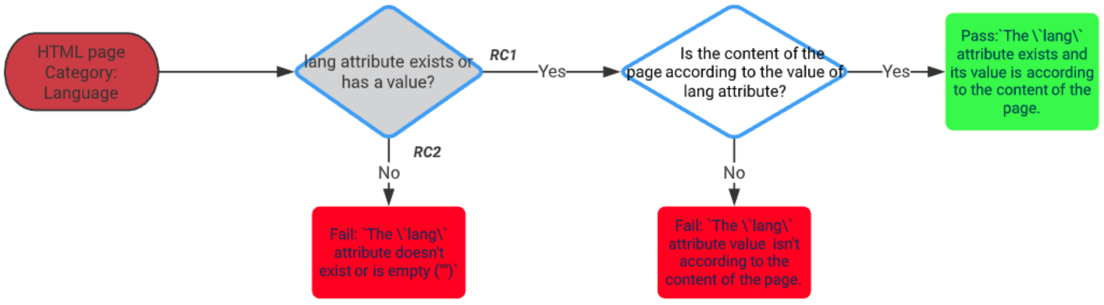
Semi-automatic Test Algorithms

This section presents the semi-automatic test algorithms for the rules defined as semi-automatic. The QualWeb Rule ID (R#) is presented together with the ACT Rule Name. The representation of the rules is according to the ACT rules published on 31th of July 2020.

A.1 R1 - HTML page has title.


HTML page Category: Title
Title element exists?
Yes
Is title element empty?
RC3
No
Does title describes page?
Yes
Pass: The \`title\` element exists and it's not empty ("") and describes page.
No
Fail: `The \`title\` element doesn't exist.`
RC2
Yes
Fail:' The \`title\` element is empty ("").'
No
Fail: The \`title\` element exists and is not empy but does not describe page.

A.2 R2 - HTML page has a lang attribute


HTML page Category: Language
lang attribute exists or has a value?
RC1
Yes
Is the content of the page according to the value of lang attribute?
Yes
Pass:`The \`lang\` attribute exists and its value is according to the content of the page.
RC2
No
Fail: `The \`lang\` attribute doesn't exist or is empty ("")`
No
Fail: `The \`lang\` attribute value isn't according to the content of the page.

A.3 R6 - Image button has an accessible name.

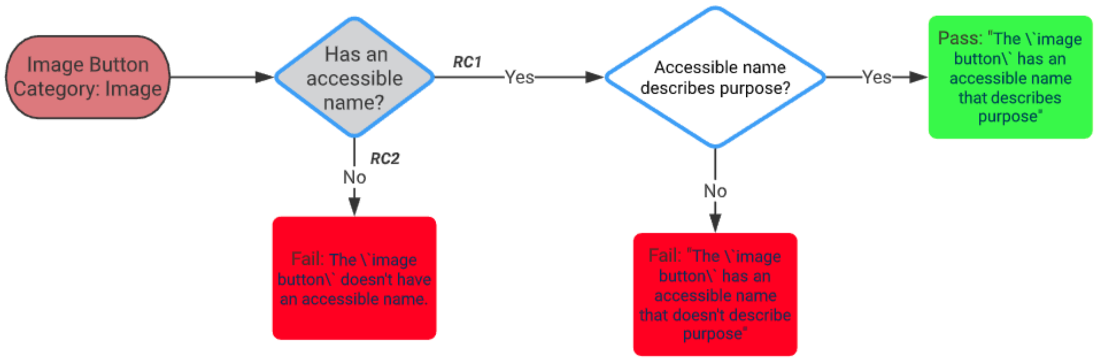


Image Button
Category: Image
Has an accessible name?
RC1
Yes
RC2
No
Fail: The \`image button\` doesn't have an accessible name.
Accessible name describes purpose?
Yes
Pass: "The \`image button\` has an accessible name that describes purpose"
No
Fail: "The \`image button\` has an accessible name that doesn't describe purpose"

A.4 R8 - Image filename is accessible name for image

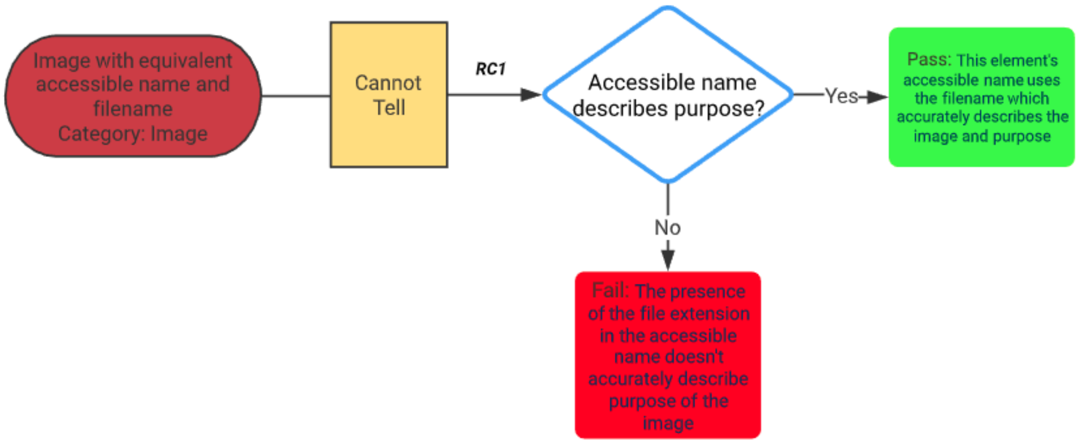
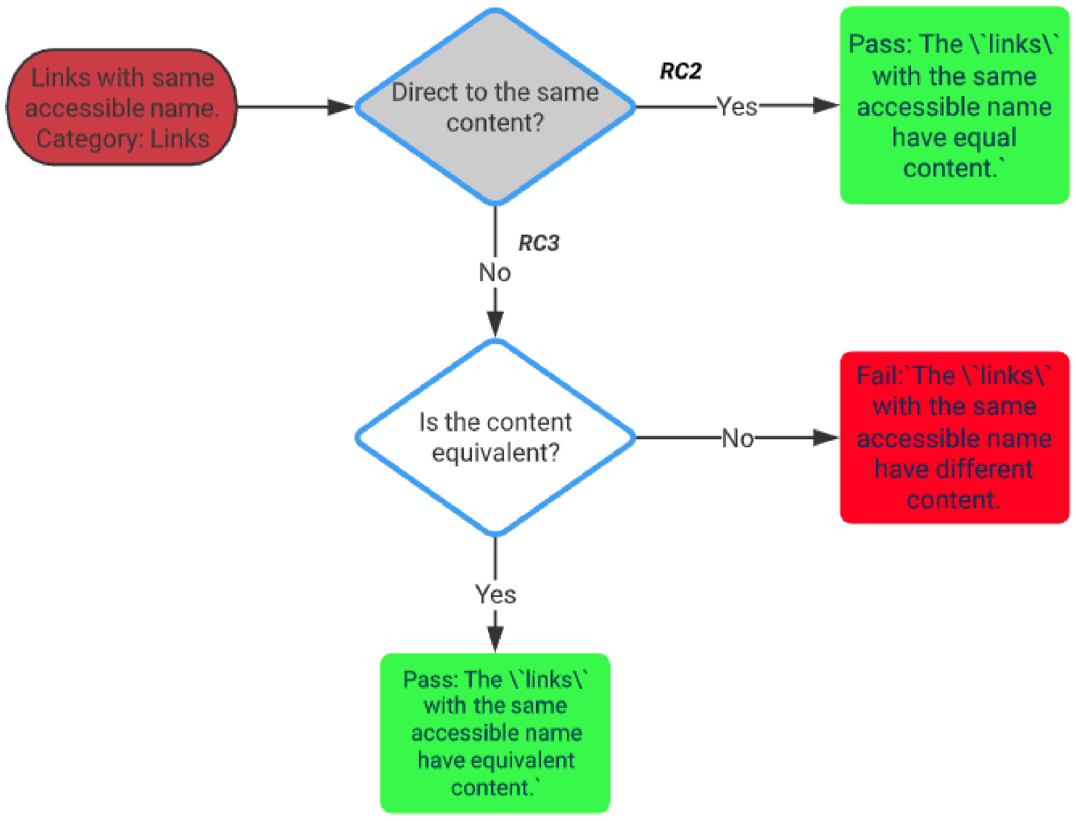
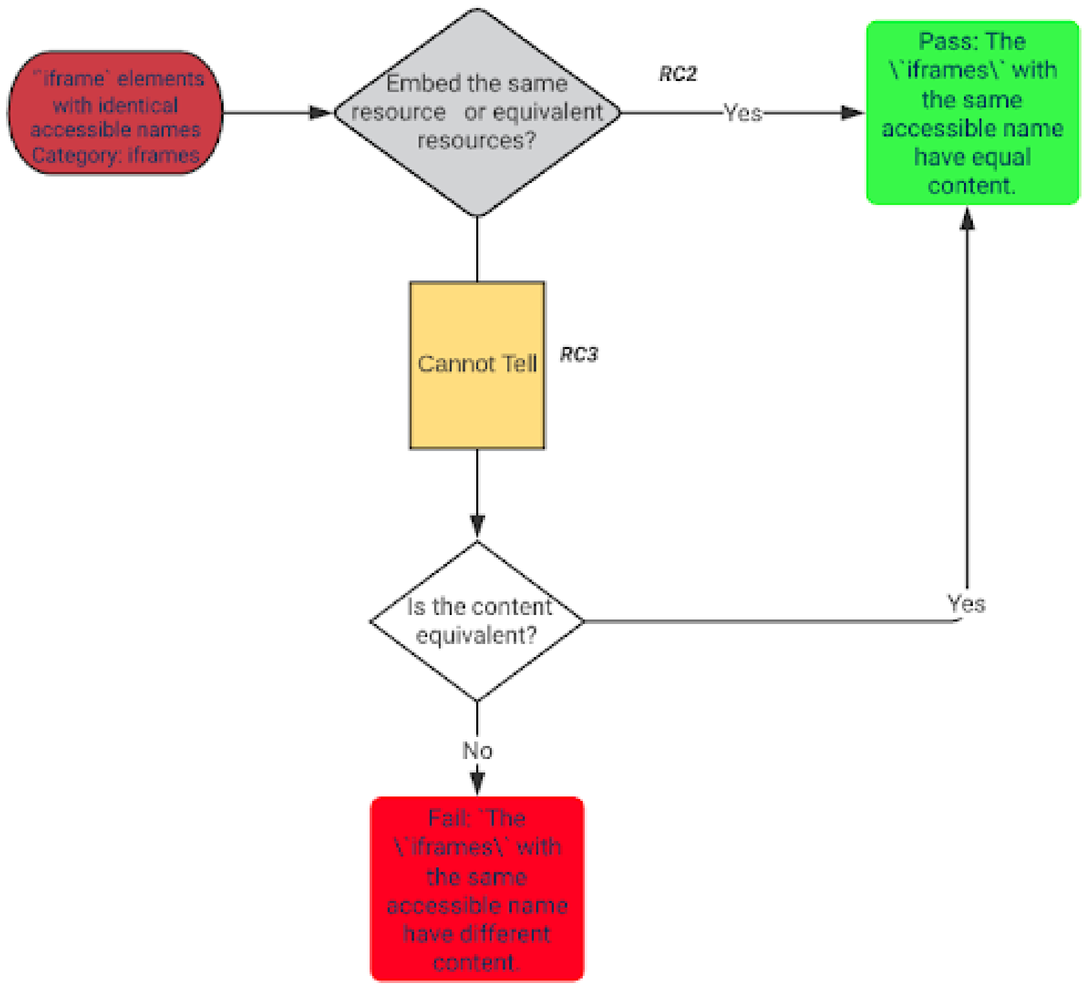


Image with equivalent accessible name and filename
Category: Image
Cannot Tell
RC1
Accessible name describes purpose?
Yes
Pass: This element's accessible name uses the filename which accurately describes the image and purpose
No
Fail: The presence of the file extension in the accessible name doesn't accurately describe purpose of the image

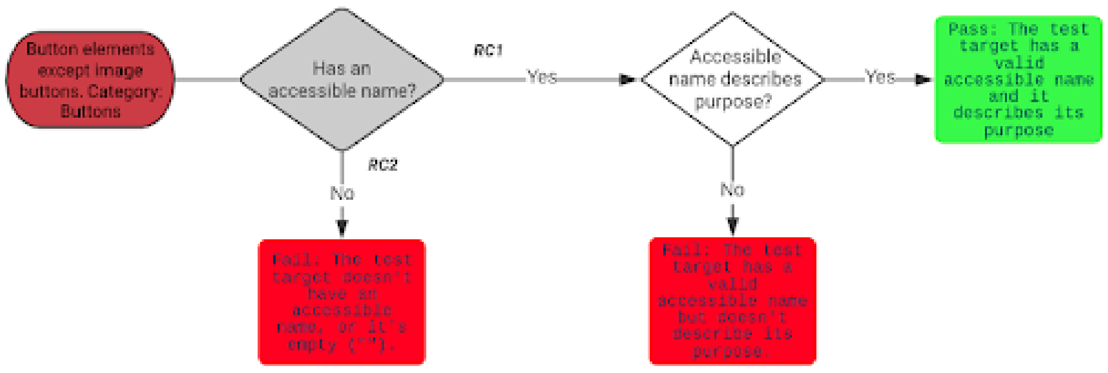
A.5 R9 - Links with identical accessible names have equivalent purpose.


Links with same accessible name. Category: Links
Direct to the same content?
RC2
Yes
Pass: The \`links\` with the same accessible name have equal content.`
RC3
No
Is the content equivalent?
No
Fail:`The \`links\` with the same accessible name have different content.
Yes
Pass: The \`links\` with the same accessible name have equivalent content.`

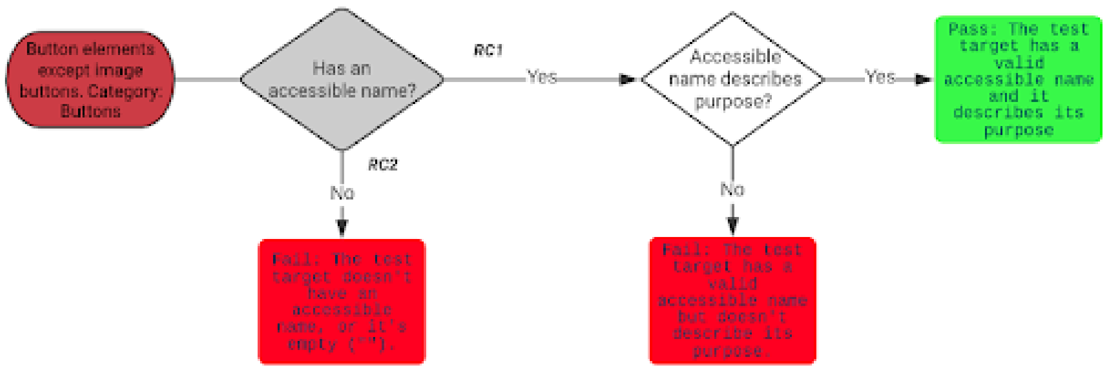
A.6 R10 - iframe elements with identical accessible names have equivalent purpose.


`iframe` elements with identical accessible names Category: iframes
Embed the same resource or equivalent resources?
RC2
Yes
Pass: The \`iframes\` with the same accessible name have equal content.
Cannot Tell
RC3
Is the content equivalent?
Yes
No
Fail: `The \`iframes\` with the same accessible name have different content.

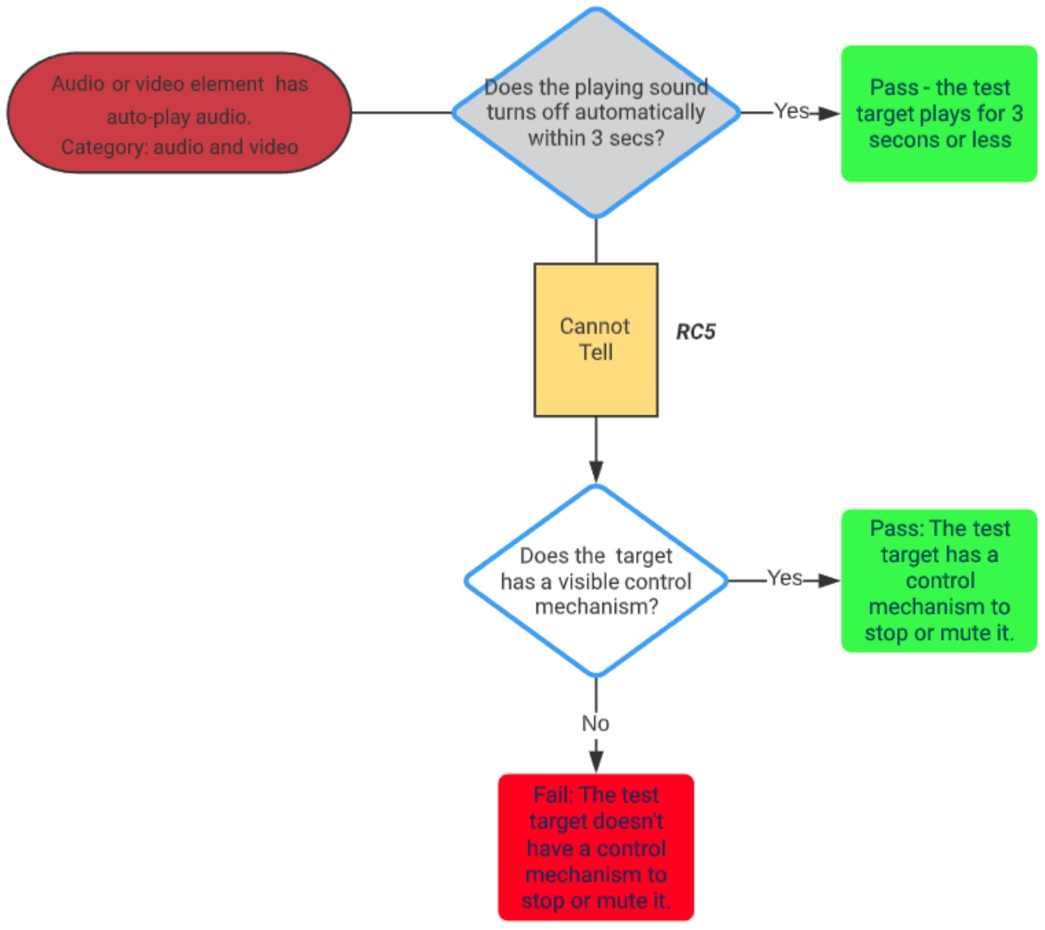
A.7 R11 - Button has an accessible name.


Button elements except image buttons. Category: Buttons
Has an accessible name?
RC1
Yes
Accessible name describes purpose?
Yes
Pass: The test target has a valid accessible name and it describes its purpose
RC2
No
Fail: The test target doesn't have an accessible name, or it's empty ("").
No
Fail: The test target has a valid accessible name but doesn't describe its purpose.

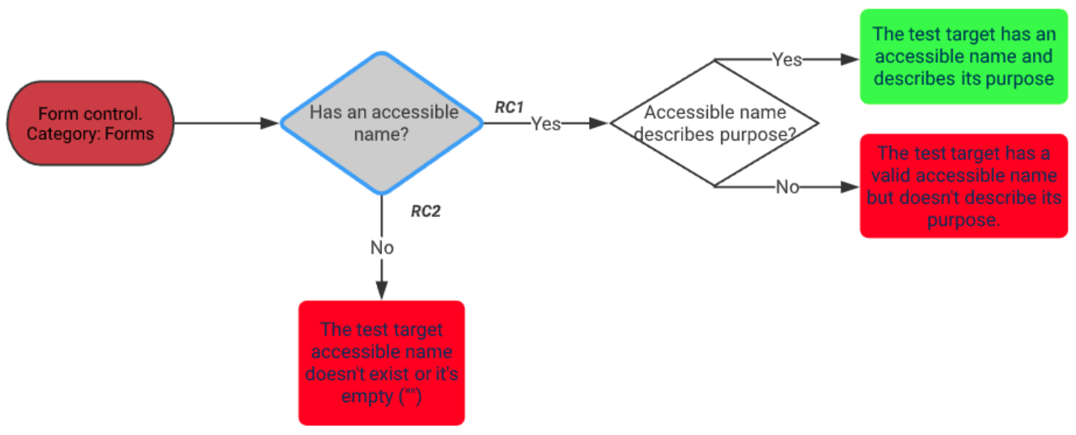
A.8 R12 - Link has an accessible name.


Button elements except image buttons. Category: Buttons
Has an accessible name?
RC1
Yes
Accessible name describes purpose?
Yes
Pass: The test target has a valid accessible name and it describes its purpose
RC2
No
Fail: The test target doesn't have an accessible name, or it's empty ("").
No
Fail: The test target has a valid accessible name but doesn't describe its purpose.

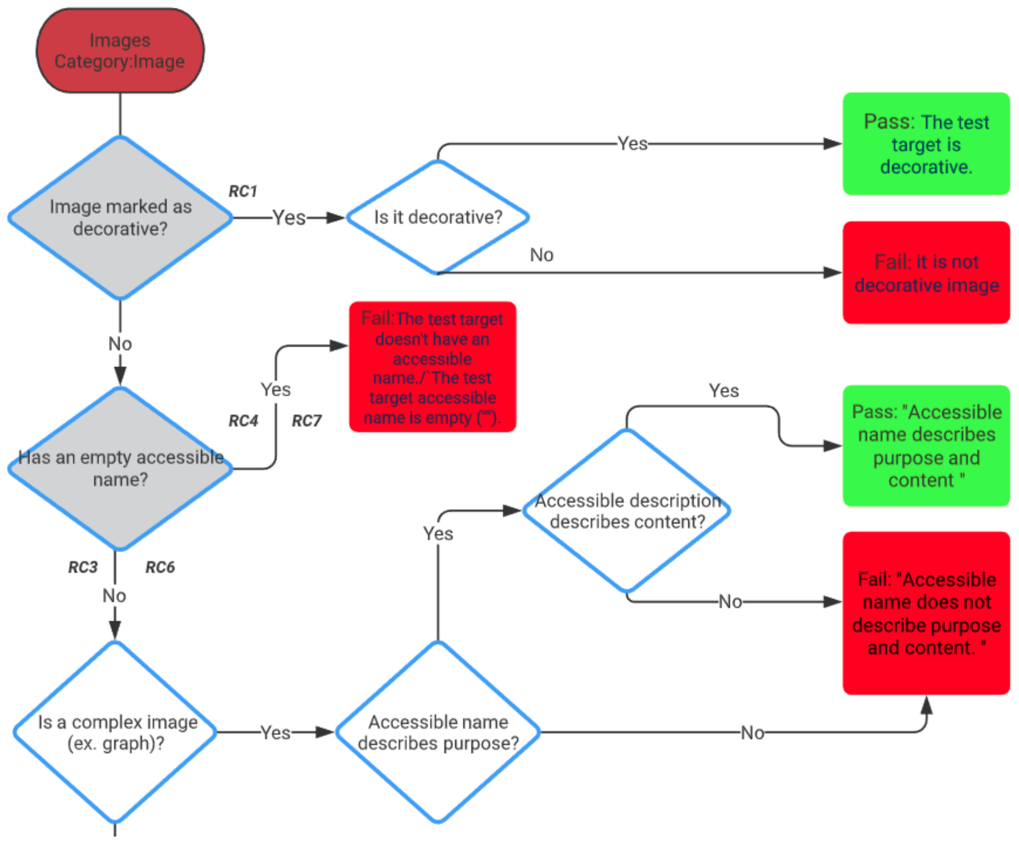
A.9 R15 - Audio or video has no audio that plays automatically.


Audio or video element has auto-play audio.
Category: audio and video
Does the playing sound turns off automatically within 3 secs?
Yes
Pass - the test target plays for 3 secons or less
Cannot Tell
RC5
Does the target has a visible control mechanism?
Yes
Pass: The test target has a control mechanism to stop or mute it.
No
Fail: The test target doesn't have a control mechanism to stop or mute it.

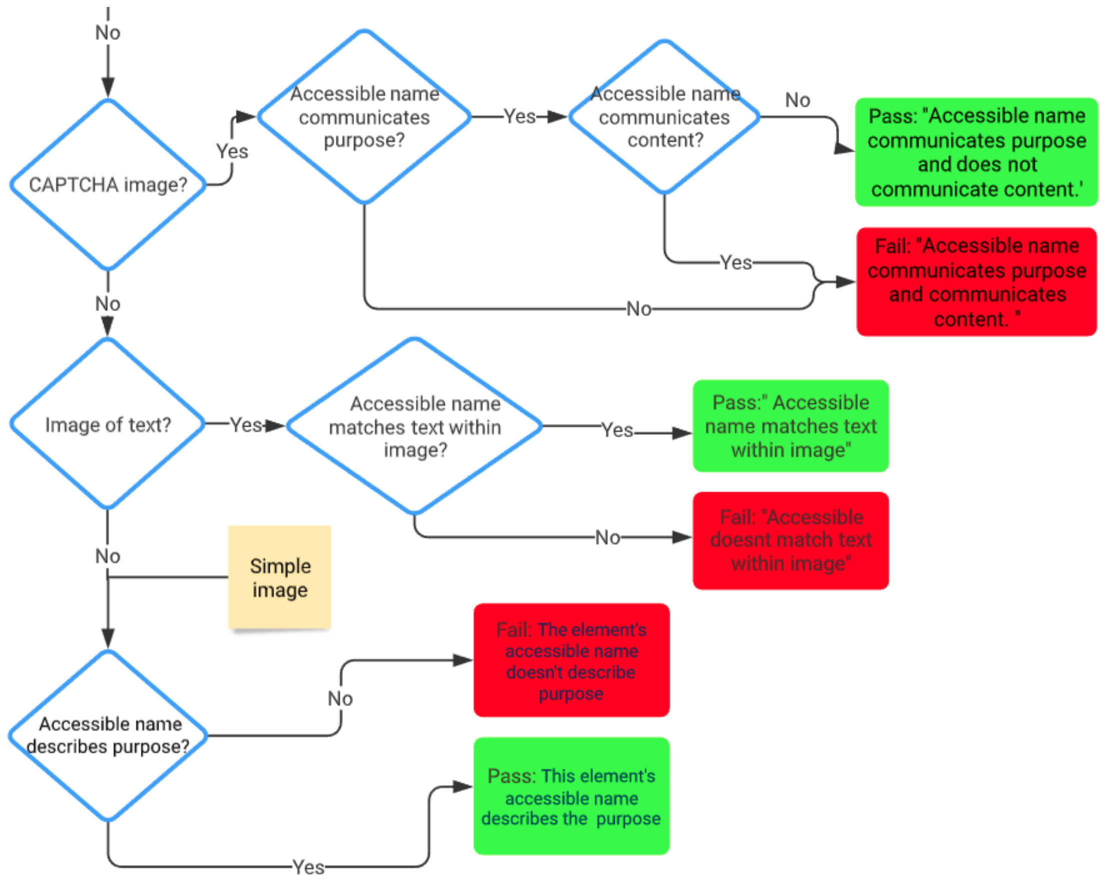
A.10 R16 - Form control has an accessible name.


Form control.
Category: Forms
Has an accessible name?
RC1
Yes
RC2
No
Accessible name describes purpose?
Yes
No
The test target has an accessible name and describes its purpose
The test target has a valid accessible name but doesn't describe its purpose.
The test target accessible name doesn't exist or it's empty ("")

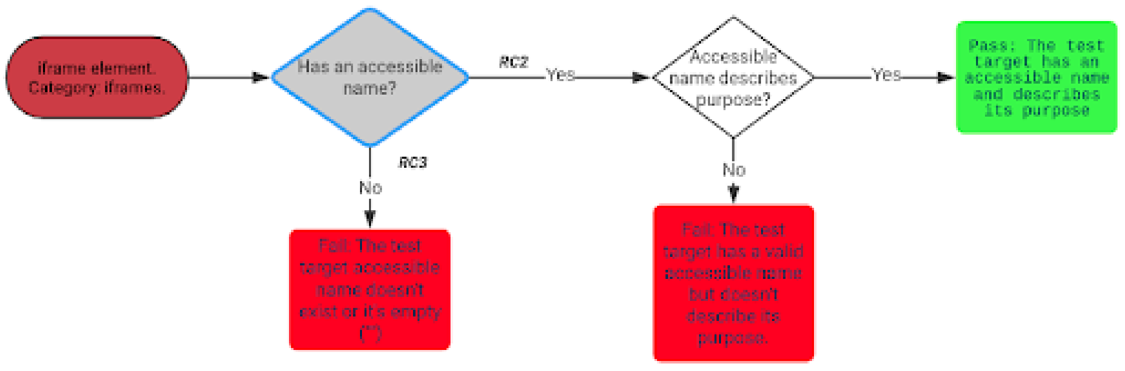
A.11 R17- Image has an accessible name.


Images
Category:Image
Image marked as decorative?
RC1
Yes
Is it decorative?
Yes
Pass: The test target is decorative.
No
Fail: it is not decorative image
No
Has an empty accessible name?
Yes
RC4
RC7
Fail:The test target doesn't have an accessible name./"The test target accessible name is empty ("").
RC3
RC6
No
Is a complex image (ex. graph)?
Yes
Accessible name describes purpose?
Yes
Accessible description describes content?
Yes
Pass: "Accessible name describes purpose and content "
No
Fail: "Accessible name does not describe purpose and content. "
No

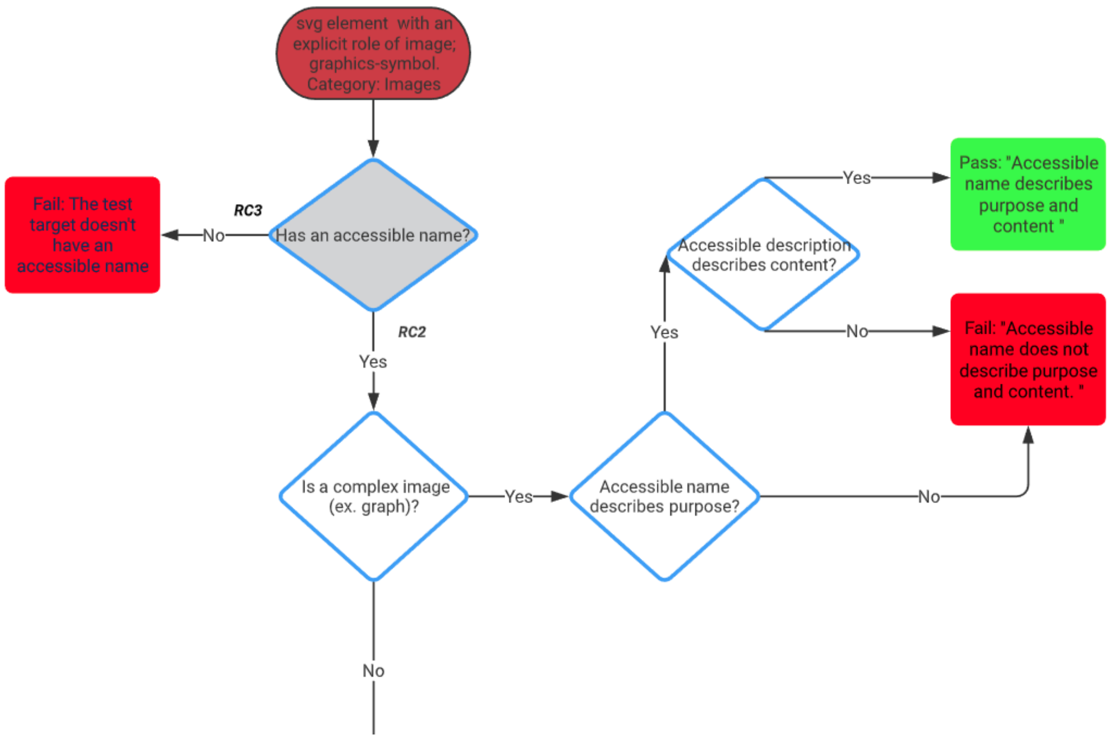
Continues next page.


No
CAPTCHA image?
Yes
Accessible name communicates purpose?
Yes
Accessible name communicates content?
No
Pass: "Accessible name communicates purpose and does not communicate content.'
Yes
Fail: "Accessible name communicates purpose and communicates content. "
No
No
Image of text?
Yes
Accessible name matches text within image?
Yes
Pass:" Accessible name matches text within image"
No
Fail: "Accessible doesnt match text within image"
No
Simple image
Accessible name describes purpose?
No
Fail: The element's accessible name doesn't describe purpose
Yes
Pass: This element's accessible name describes the purpose

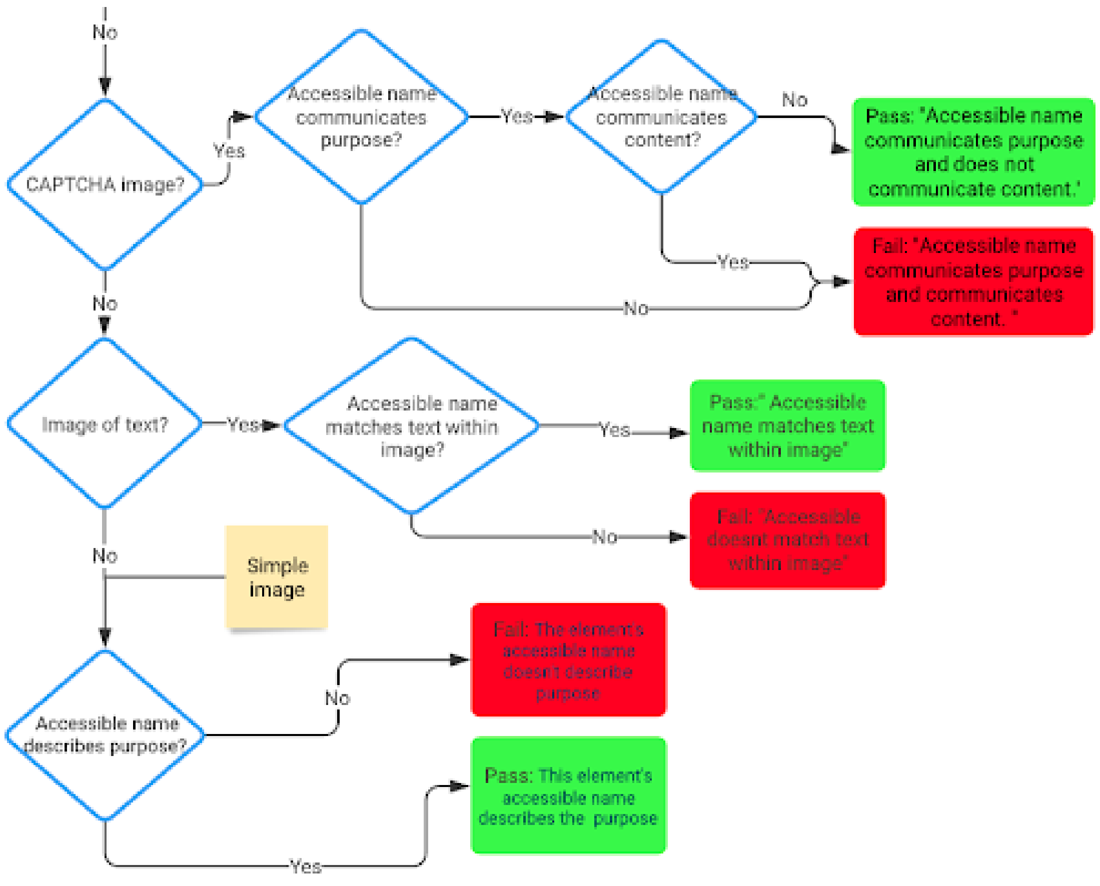
A.12 R19 - iframe element has an accessible name.


iframe element. Category: iframes.
Has an accessible name?
RC2
Yes
Accessible name describes purpose?
Yes
Pass: The test target has an accessible name and describes its purpose
RC3
No
Fail: The test target accessible name doesn't exist or it's empty ("")
No
Fail: The test target has a valid accessible name but doesn't describe its purpose.

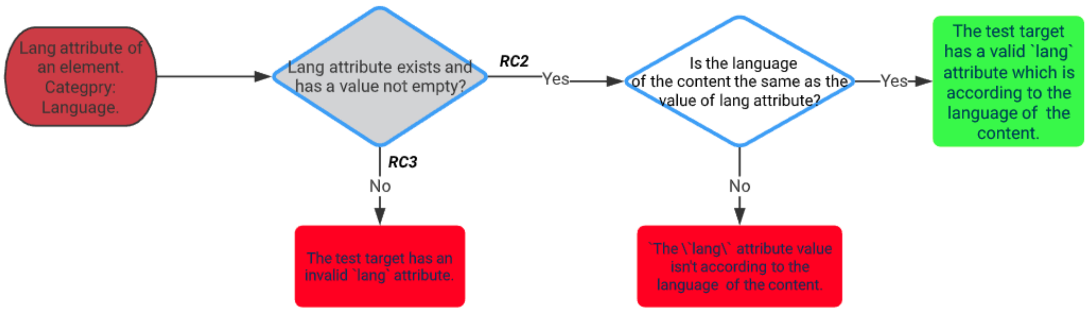
A.13 R21 - svg element with explicit role has an accessible name.


svg element with an explicit role of image; graphics-symbol. Category: Images
Has an accessible name?
RC3
No
Fail: The test target doesn't have an accessible name
RC2
Yes
Is a complex image (ex. graph)?
Yes
Accessible name describes purpose?
Yes
Accessible description describes content?
Yes
Pass: "Accessible name describes purpose and content "
No
Fail: "Accessible name does not describe purpose and content. "
No
No

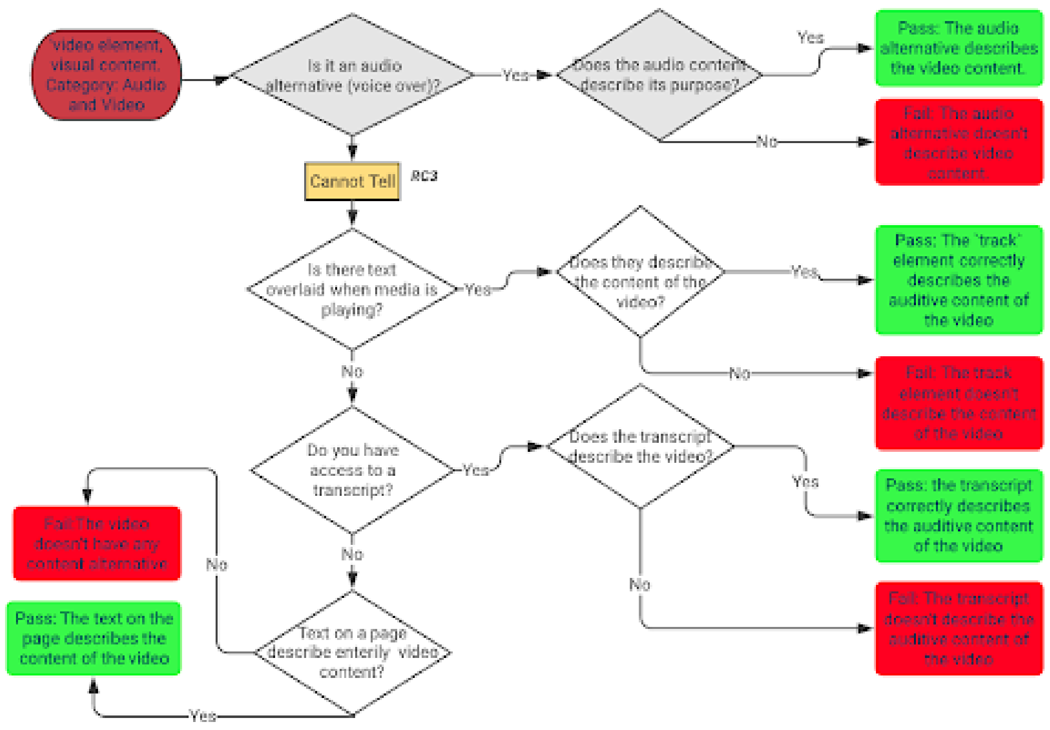
Continues next page.


No
CAPTCHA image?
Yes
Accessible name communicates purpose?
Yes
Accessible name communicates content?
No
Pass: "Accessible name communicates purpose and does not communicate content.'
Yes
No
Fail: "Accessible name communicates purpose and communicates content. "
No
Image of text?
Yes
Accessible name matches text within image?
Yes
Pass:" Accessible name matches text within image"
No
Fail: "Accessible doesnt match text within image"
No
Simple image
Fail: The element's accessible name doesn't describe purpose
No
Accessible name describes purpose?
Pass: This element's accessible name describes the purpose
Yes

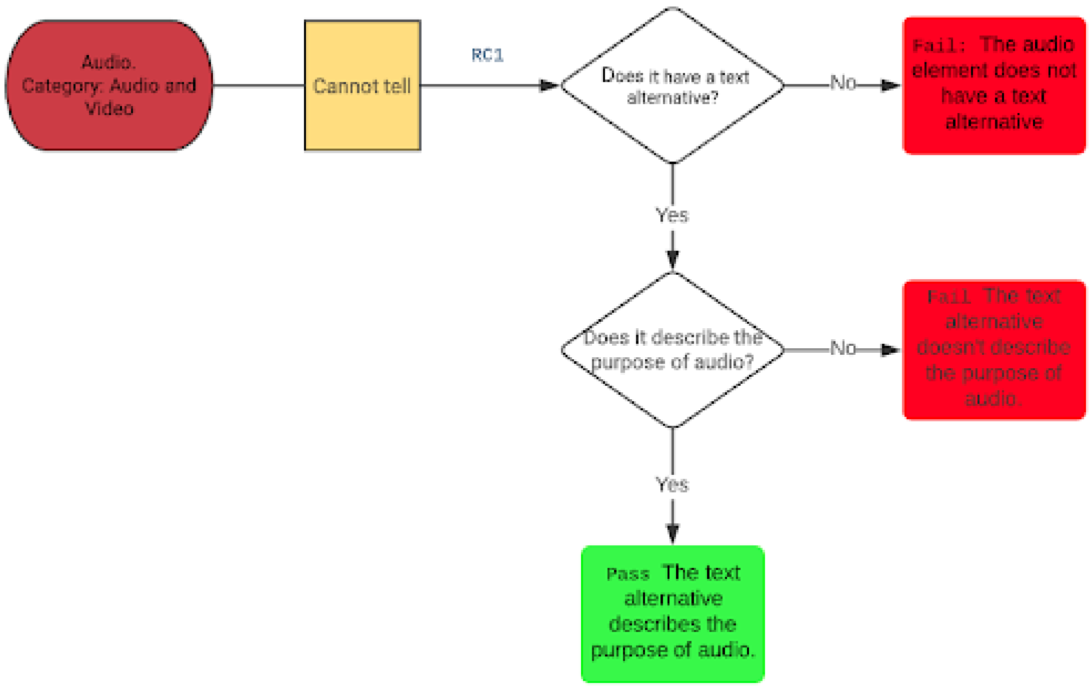
A.14 R22 - Element within body has valid lang attribute.


Lang attribute of an element. Categpry: Language.
Lang attribute exists and has a value not empty?
RC2
Yes
Is the language of the content the same as the value of lang attribute?
Yes
The test target has a valid `lang` attribute which is according to the language of the content.
RC3
No
The test target has an invalid `lang` attribute.
No
`The \`lang\` attribute value isn't according to the language of the content.

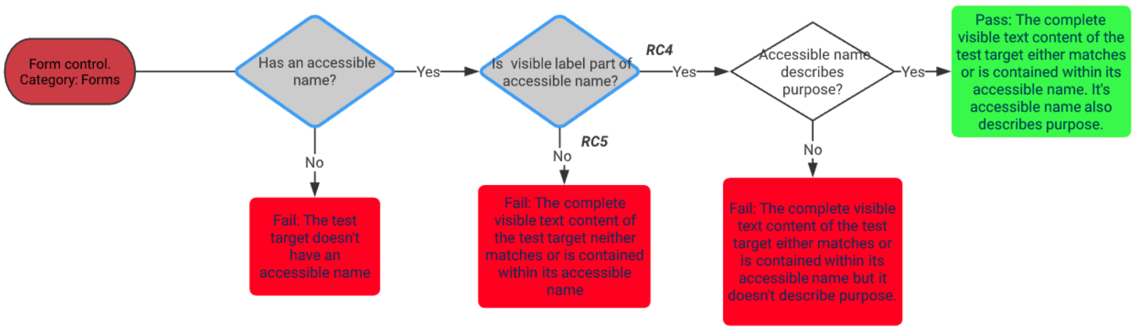
A.15 R23 - Video element visual content has an accessible alternative.


video element, visual content. Category: Audio and Video
Is it an audio alternative (voice over)?
Yes
Does the audio content describe its purpose?
Yes
Pass: The audio alternative describes the video content.
No
Fail: The audio alternative doesn't describe video content.
Cannot Tell
RC3
Is there text overlaid when media is playing?
Yes
Does they describe the content of the video?
Yes
Pass: The `track` element correctly describes the auditive content of the video
No
Fail: The track element doesn't describe the content of the video
No
Do you have access to a transcript?
Yes
Does the transcript describe the video?
Yes
Pass: the transcript correctly describes the auditive content of the video
No
Fail: The transcript doesn't describe the auditive content of the video
No
Text on a page describe enterily video content?
No
Fail:The video doesn't have any content alternative
Yes
Pass: The text on the page describes the content of the video

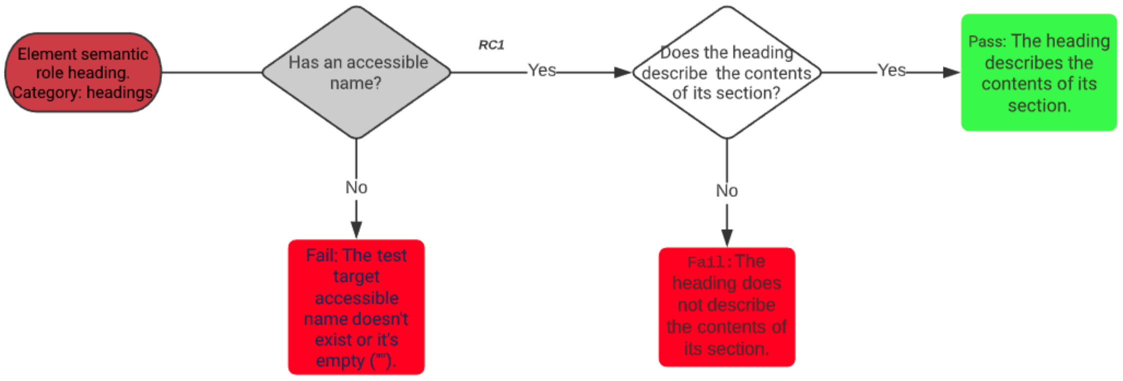
A.16 R29 - Audio element content has text alternative.


Audio.
Category: Audio and Video
Cannot tell
RC1
Does it have a text alternative?
No
Fail: The audio element does not have a text alternative
Yes
Does it describe the purpose of audio?
No
Fail: The text alternative doesn't describe the purpose of audio.
Yes
Pass The text alternative describes the purpose of audio.

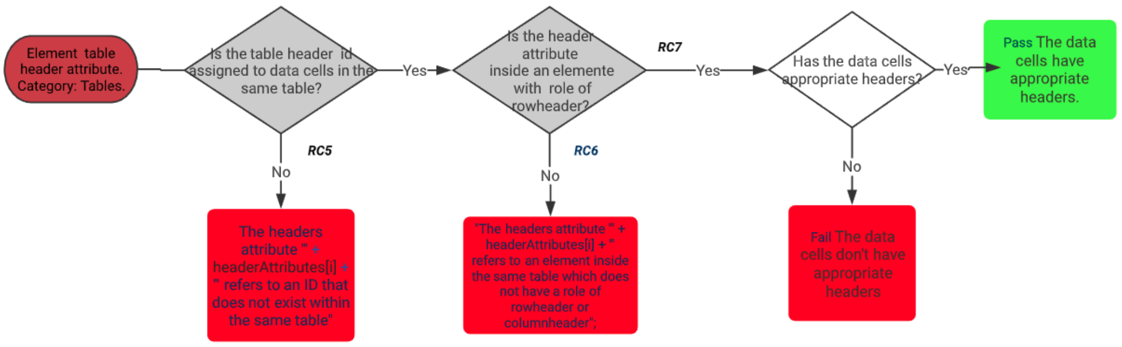
A.17 R30 - Visible label is part of accessible name.


Form control.
Category: Forms
Has an accessible name?
Yes
No
Fail: The test target doesn't have an accessible name
Is visible label part of accessible name?
RC4
Yes
RC5
No
Fail: The complete visible text content of the test target neither matches or is contained within its accessible name
Accessible name describes purpose?
Yes
No
Pass: The complete visible text content of the test target either matches or is contained within its accessible name. It's accessible name also describes purpose.
Fail: The complete visible text content of the test target either matches or is contained within its accessible name but it doesn't describe purpose.

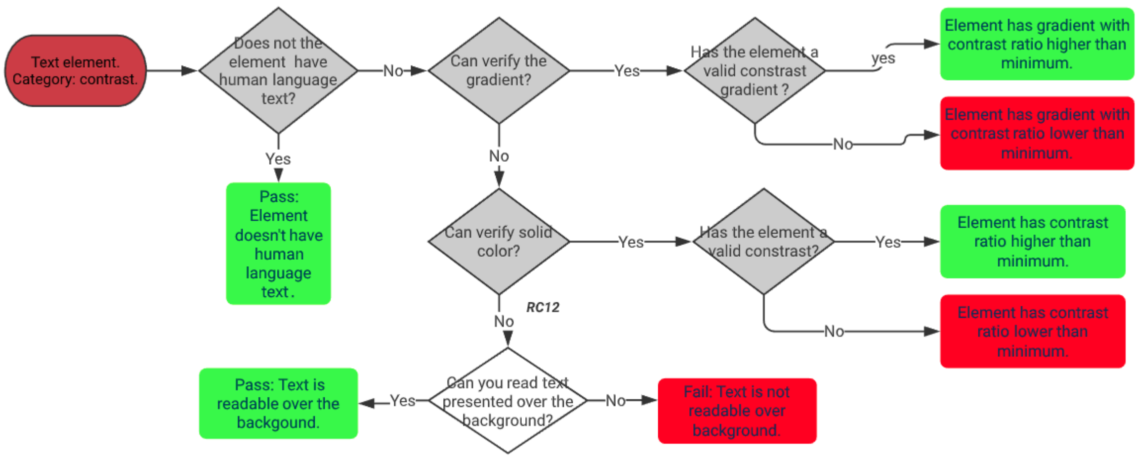
A.18 R35 - Heading has non-empty accessible name.


Element semantic role heading. Category: headings
Has an accessible name?
RC1
Yes
Does the heading describe the contents of its section?
Yes
Pass: The heading describes the contents of its section.
No
Fail: The test target accessible name doesn't exist or it's empty ("").
No
Fail: The heading does not describe the contents of its section.

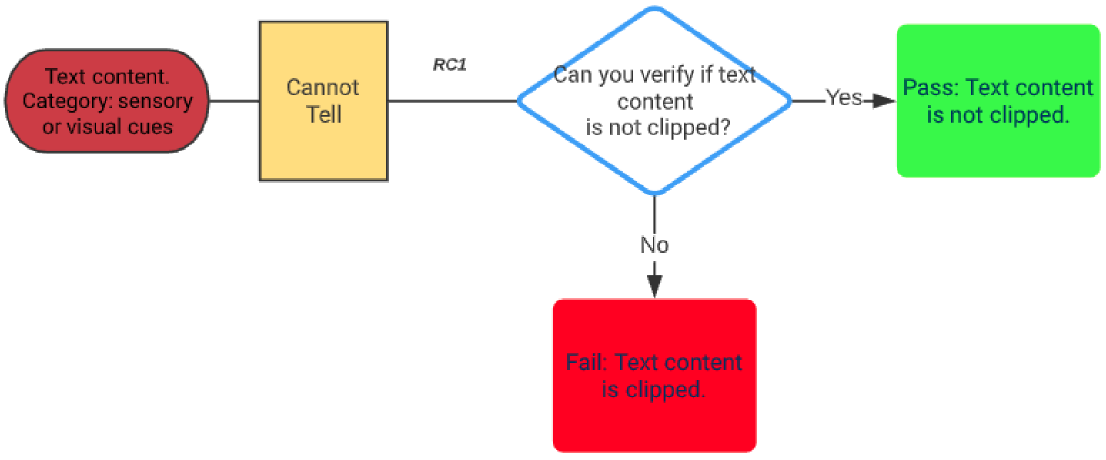
A.19 R36 - Headers attribute specified on a cell refers to cells in the same table element.


Element table header attribute. Category: Tables.
Is the table header id assigned to data cells in the same table?
Yes
Is the header attribute inside an elemente with role of rowheader?
RC7
Yes
Has the data cells appropriate headers?
Yes
Pass The data cells have appropriate headers.
RC5
No
The headers attribute '" + headerAttributes[i] + "' refers to an ID that does not exist within the same table"
RC6
No
"The headers attribute '" + headerAttributes[i] + "' refers to an element inside the same table which does not have a role of rowheader or columnheader";
No
Fail The data cells don't have appropriate headers

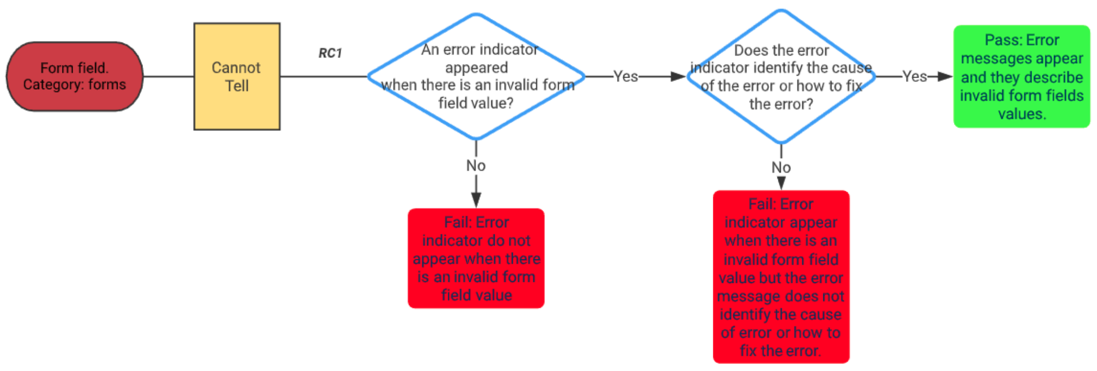
A.20 R37 - Text has minimum contrast


Text element. Category: contrast.
Does not the element have human language text?
No
Yes
Pass: Element doesn't have human language text.
Can verify the gradient?
Yes
No
Has the element a valid constrast gradient ?
yes
Element has gradient with contrast ratio higher than minimum.
No
Element has gradient with contrast ratio lower than minimum.
Can verify solid color?
Yes
Has the element a valid constrast?
Yes
Element has contrast ratio higher than minimum.
No
Element has contrast ratio lower than minimum.
RC12
No
Can you read text presented over the background?
Yes
Pass: Text is readable over the backgound.
No
Fail: Text is not readable over background.

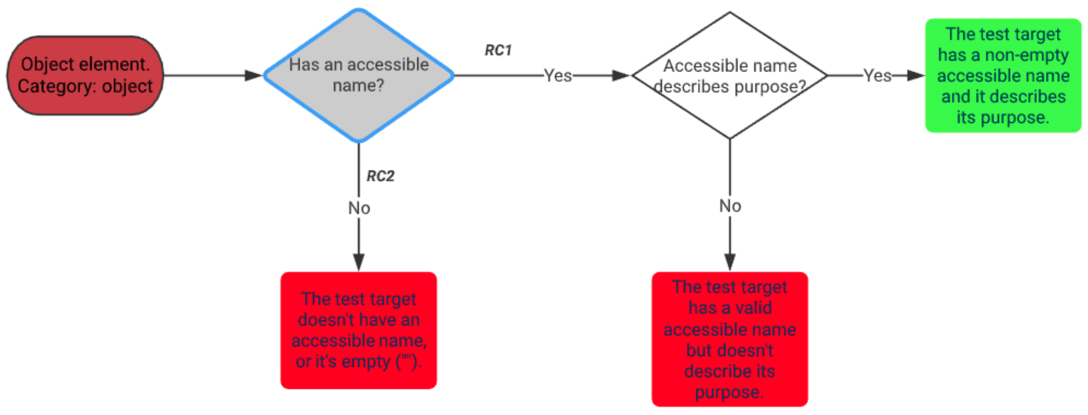
A.21 R40 - Zoomed text node is not clipped with CSS overflow.


Text content. Category: sensory or visual cues
Cannot Tell
RC1
Can you verify if text content is not clipped?
Yes
Pass: Text content is not clipped.
No
Fail: Text content is clipped.

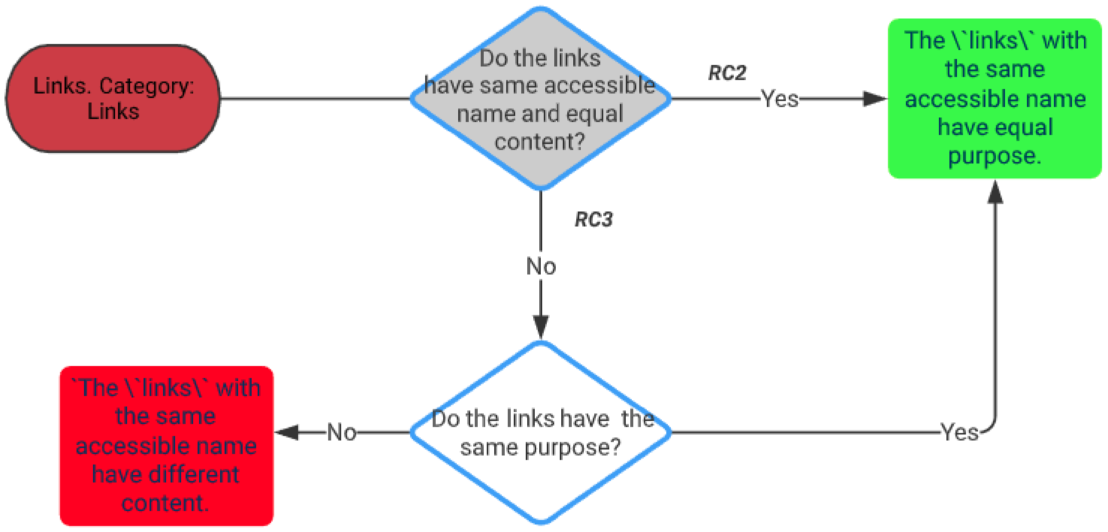
A.22 R41 - Error message describes invalid form field value.


Form field.
Category: forms
Cannot Tell
RC1
An error indicator appeared when there is an invalid form field value?
Yes
No
Does the error indicator identify the cause of the error or how to fix the error?
Yes
No
Pass: Error messages appear and they describe invalid form fields values.
Fail: Error indicator do not appear when there is an invalid form field value
Fail: Error indicator appear when there is an invalid form field value but the error message does not identify the cause of error or how to fix the error.

A.23 R42 - Object element rendering non-text content has non-empty accessible name.


Object element.
Category: object
Has an accessible name?
RC1
Yes
RC2
No
Accessible name describes purpose?
Yes
No
The test target has a non-empty accessible name and it describes its purpose.
The test target doesn't have an accessible name, or it's empty ("").
The test target has a valid accessible name but doesn't describe its purpose.

A.24 R44 - Links with identical accessible names and context serve equivalent purpose.

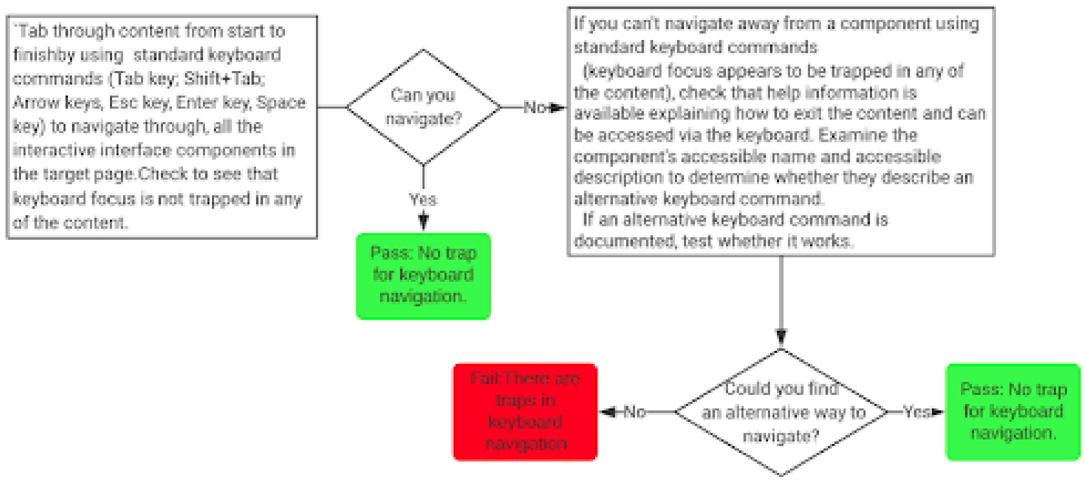

Links. Category: Links
Do the links have same accessible name and equal content?
RC2
Yes
The \`links\` with the same accessible name have equal purpose.
RC3
No
Do the links have the same purpose?
Yes
No
`The \`links\` with the same accessible name have different content.

Appendix B

Manual Test Algorithms

This section presents the manual test algorithms for the rules defined as manual. The rule id from ACT-Rules is presented. The representation of the rules is according to the ACT rules published on 31th of July 2020.

B.1 Rule id: 80af7b - Focusable element has no keyboard trap.

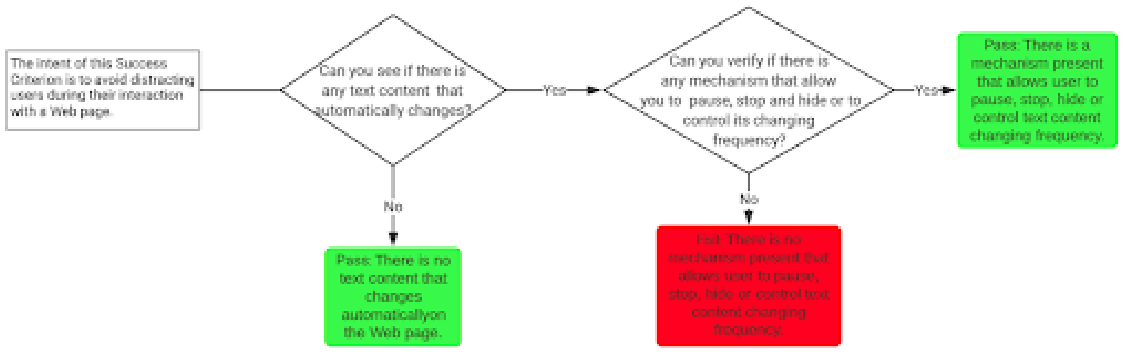


`Tab through content from start to finishby using standard keyboard commands (Tab key; Shift+Tab; Arrow keys, Esc key, Enter key, Space key) to navigate through, all the interactive interface components in the target page.Check to see that keyboard focus is not trapped in any of the content.

If you can't navigate away from a component using standard keyboard commands (keyboard focus appears to be trapped in any of the content), check that help information is available explaining how to exit the content and can be accessed via the keyboard. Examine the component's accessible name and accessible description to determine whether they describe an alternative keyboard command. If an alternative keyboard command is documented, test whether it works.

Can you navigate?
No
Yes
Pass: No trap for keyboard navigation.
Could you find an alternative way to navigate?
No
Yes
Fail:There are traps in keyboard navigation
Pass: No trap for keyboard navigation.

B.2 Rule id: efbfc7 - Text content that changes automatically can be paused, stopped or hidden


The intent of this Success Criterion is to avoid distracting users during their interaction with a Web page.
Can you see if there is any text content that automatically changes?
Yes
No
Pass: There is no text content that changes automaticallyon the Web page.
Can you verify if there is any mechanism that allow you to pause, stop and hide or to control its changing frequency?
Yes
No
Pass: There is a mechanism present that allows user to pause, stop, hide or control text content changing frequency.
Fail: There is no mechanism present that allows user to pause, stop, hide or control text content changing frequency.

Appendix C

AccessBot Main Object

An example of a main object `console.log` of the webpage https://ciencias.ulisboa.pt/ accessibility evaluation is presented:

```
Object
    categories: Array(18)
        0:
            count: 15
            fail: 0
            fixedName: ”ARIA”
            inapplicable: 9
            index: 0
            missing: 0
            name: ”ARIA”
            pass: 6
            rules: Array(8)
            0: rule: ”QW-ACT-R13”, name: ”Element with
                ‘aria-hidden‘ has no focusable content”,
                description: ”This rule checks that elements with
                an aria-hidden attribute do not contain focusable
                elements.”, id: ”6cfa84”, url:
                ”https://act-rules.github.io/rules/6cfa84”, . . .
                1: rule: ”QW-ACT-R25”, name: ”ARIA state or
                property is permitted”, description: ”This rule
                checks that WAI-ARIA states or propertie. . . re
                allowed for the element they are specified on.”,
                id: ”5c01ea”, url:
                ”https://act-rules.github.io/rules/5c01ea”, . . .
                2: rule: ”QW-ACT-R27”, name: ”aria-* attribute is
                defined in WAI-ARIA”, description: ”This rule
```

```
checks that each aria- attribute specified is
defined in ARIA 1.1.”, id: ”5f99a7”, url:
”https://act-rules.github.io/rules/5f99a7”, . . .
3:
    count: 5
    description: ”This rule checks that
    elements that have an explicit role also
    specify all required states and
    properties.”
    fail: 0
    id: ”4e8ab6”
    inapplicable: 4
    index: 3
    missing: 0
    name: ”Element with role attribute has
    required states and properties”
    pass: 1
    plusRule: []
    questions: Array(5)
        0:
            complete: true
            description: ”The test target
            explicit role equals the
            implicit role.”
            elements: [. . . ]
            index: 0
            manualAnswer: ””
            note: ””
            resultCode: ”RC2”
            selected: false
            type: ”auto”
            verdict: ”inapplicable”
            __proto__: Object
        1: verdict: ”inapplicable”,
        description: ”The test target
        explicit role equals the implicit
        role.”, resultCode: ”RC2”, elements:
        Array(1), selected: false, . . .
        2: verdict: ”passed”, description:
        ”The test target ‘role‘ doesn’t have
        required state or property”,
        resultCode: ”RC5”, elements:
        Array(1), selected: false, . . .
        3: verdict: ”inapplicable”,
        description: ”The test target
        explicit role equals the implicit
        role.”, resultCode: ”RC2”, elements:
        Array(1), selected: false, . . .
        4: verdict: ”inapplicable”,
        description: ”The test target
        explicit role equals the implicit
        role.”, resultCode: ”RC2”, elements:
        Array(1), selected: false, . . .
        length: 5
        __proto__: Array(0)
    rule: ”QW-ACT-R28”
    selected: false
    total: 5
    url:
    ”https://act-rules.github.io/rules/4e8ab6
    warning: 0
    __proto__: Object
4: rule: ”QW-ACT-R34”, name: ”ARIA state or
property has valid value”, description: ”This rule
checks that each ARIA state or property has a
valid value.”, id: ”6a7281”, url: ”https://ac (...)
```