A Declarative Model for Web Accessibility Requirements

—

Design and Implementation

by Jens Pelzetter

Dissertation
zur Erlangung des Grades eines Doktors der
Ingenieurwissenschaften
– Dr.-Ing. –

Vorgelegt im Fachbereich 3 (Mathematik und Informatik)
der Universität Bremen
im Oktober 2020

Datum des Promotionskolloquiums: 11. Dezember 2020

1. Gutachter:	Prof. Dr. Bernd Krieg Brückner (Universität Bremen)
2. Gutachter:	Prof. Dr. Carlos Duarte (Universidade de Lisboa)

Abstract

The web has become the primary source of information and the primary place for shopping for many people. Many services provided by companies and authorities can be used online. Despite extensive guidelines for accessible websites, many websites are not accessible. Therefore, many of these services are difficult to use for people with disabilities. Many tools are available that can help developers to create accessible websites. One problem with these tools is that different tools often produce different results for the same web page. Also, these tools are often difficult to maintain for their developers. In this thesis, a declarative model for accessibility requirements is presented, together with an example implementation. Using the model, developers of accessibility tools can focus on implementing the tests, instead of implementing and understanding all aspects of the guidelines.

Zusammenfassung

Das Internet ist für viele Menschen zu primären Quelle für Informationen geworden. Viele Dienstleistungen werden über das Internet angeboten. Für Menschen mit Behinderungen sind die im Internet angebotenen Dienste oftmals nur schwer nutzbar, da diese, trotz vorhandener Richtlinien, nicht barrierefrei gestaltet sind. Für die Prüfung der Barrierefreiheit von Webseiten existiert eine Vielzahl von Werkzeugen. Diese produzieren aber oftmals unterschiedliche Ergebnisse, und sind für die Entwickler schwierig zu warten. In dieser Dissertation wird ein deklaratives Modell für die Anforderungen an barrierefreie Webseiten vorgestellt. Dieses Modell kann von Entwicklern von Werkzeugen für die Barrierefreiheit von Webseiten genutzt werden. Diese können sich dadurch auf die Implementierung von Tests konzentrieren, und müssen nicht alle Aspekte der Richtlinien selbst implementieren.

Acknowledgments

First, I would like to thank my supervisors, Prof. Dr. Bernd Krieg-Brückner and Dr. Serge Autexier, for their support and valuable feedback in the last years. Also, I would like to thank all people from the CMS Garden, especially Meike Jung, Stephan Luckow, and Markus Wortmann, for their input and feedback, and the CMS Garden e.V. for letting me use their infrastructure for hosting the *web-a11y-auditor*. And, of course, I would like to thank my parents for supporting me during the last years and all people who provided feedback and support.

Contents

1. Introduction

1.1. Motivation

This thesis has been motivated by personal experience. In the past, I worked on several projects in which websites or applications have been developed. I also administered several websites and provided support for the authors of these sites. Also, I am a member of the CMS Garden e.V., a German association of people from communities of several Web Content Management Systems.

Accessibility for websites is an important topic that is often discussed. There are extensive guidelines for accessible web pages (see 2.1), but the implementation is not easy and often time-consuming. One problem I often faced is that checking a website for accessibility issues is a very time-consuming task. Even for a single page, this can take several hours, depending on the complexity of the page. Also, accessibility is not only a concern of the developer. Content for most sites is usually created by people without in-depth knowledge about information technology or the web. Tools that support people to make and keep a website accessible are required at all development stages of a website. A machine-readable representation of the guidelines for accessible web pages may make it easier to develop such tools.

1.2. Accessibility and the Web

The web has become the primary platform for getting information about almost all topics, interacting with other people, and shopping. These services can give people with impairments a large degree of participation and independence, at least in theory. In practice, this is not the case. Because of several limitations of the technologies used to create websites and web applications and especially how these technologies are often used, websites and web pages are often not as accessible as they could be.

Aside from the data transfer protocols, which are not the subject of this thesis, the web is built upon several core technologies. The Hyper Text Markup Language (HTML) [23] is used to describe the content of documents, Cascading Style Sheets (CSS) [26] are used to style the documents, and Java Script (standardized under the name ECMAScript [21]) is used to provide interactivity. In the past, the specifications for HTML and CSS were not developed as fast as the usage of the web grew.

HTML should only be used to describe the content of a document. Listing 1.2.1 shows an excerpt from an HTML document that uses HTML correctly. The heading is marked with an `h1` element as a heading of the first order. Paragraphs are marked using `p`. Some important phrases are emphasized using the `em` element.

```
<h1>This is an example test</h1>
<p>
This text is an example for the usage of HTML.
</p>
<p>
It does not have <em>any</em> useful content.
</p>
```

Listing 1.2.1: Correct, accessible HTML

The first versions of HTML also contained several elements that could be used to create some basic styling of documents. There were no dedicated options for creating layouts. For scientific documents – sharing those was the initial application for HTML – this was not a big problem. For other types of sites, people quickly began to use tables for creating layouts.

Later, the styling part was moved to a separate language. Modern web pages are styled using CSS. When the web started to become common, websites were often designed and created by people with a background in print design. These people often did not have a good understanding of HTML and its concepts. They often used generic elements like `div` and `span` almost exclusively. For example, especially in some older HTML document, a heading may look like a heading but is not marked as such.

Today most web developers and web designers have at least a basic knowledge about accessibility, but there are still many reasons why websites have accessibility problems. Many projects have a very limited budget or tight time constraints. Checking the accessibility is often the first thing left out to match the budget or project deadlines. Many websites are powered by Content Management Systems such as Drupal, Joomla, or WordPress. These applications make it possible for users with no background in web technologies to create content. Because these users do not have the background, they often do not think about the accessibility of the content they are creating. To some degree, this can be limited by technical means.

Some important types of assistive technology rely on exactly this information. The most prominent examples are screen readers. A screen reader is a program that creates a linear representation of the User Interface and outputs these data to a braille line or as speech. Screen readers also use the information extracted from correctly marked content of an HTML document to help users of screenreaders to navigate in an HTML document.

Modern websites and applications are often used on a variety of devices with different capabilities. Designing websites has become a separate profession. Nevertheless, often the visual design is regarded as more important than a correct markup. Designers also often create pitfalls for people with certain impairments. One example is the usage of color. The color red is usually used to indicate a problem or an error. Forms with some mandatory fields or fields that required data to be entered in a specific format are common examples. If the data entered by the user is not valid, the problematic fields

are often marked with a red border. Suppose the problematic fields are only marked using color. In that case, this can cause problems for many people because a surprisingly large number of people cannot distinguish the color red from other colors. These people may overlook the error message and think that the site or application to which the form belongs does not work correctly. Therefore, color should not be the only means to relay information to the user. Other examples of not so obvious accessibility problems are the size of clickable areas, the order of data in the document, or missing labeling of the document's primary language.

Another point of view is to look at the abilities people are using to interact with web pages and web applications. First, they perceive the site using their vision (seeing). Some content, for example, in videos or podcasts, may be provided as audio. For this content, they use their hearing. To operate the devices usually used to interact with computers (mouse, keyboard), they need their tactile sense to feel the device and the motoric abilities to operate the device. Moving the pointer to a specific element on the screen also requires hand-eye coordination. Also, the (textual) information provided by the site has to be perceived (cognitive abilities).

There are many possible causes for a certain degree of limitation of one or more of these abilities. Not all causes are medical. Temporal limitations are often caused by the situation rather than by medical conditions. For example, a web page that uses a color that has a low contrast to the background is difficult to read in bright sunlight. Some medical conditions are also temporary. For example, the loss of motoric abilities can be caused by a broken arm. This temporary loss of motoric abilities might limit the person's ability to operate a mouse or a keyboard or at least slow the person down when using a computer. Another example is a surgeon who wants to alter the displayed data on a screen in the operating room. The surgeon cannot use his hands to interact with the application presenting the data. Therefore, the surgeon needs another way to alter the data displayed on the screen.

Several aspects of accessibility overlap with the area of usability. In fact, various aspects of accessibility can be interpreted as extensions of the requirements for usable software. To provide developers with guidelines, the World Wide Consortium has published several guidelines for creating accessible web pages. The most important guidelines are the Web Content Accessibility Guidelines (WCAG) [9, 27], and the supplemental documents of the WCAG.

A common accessibility problem on web pages is insufficient contrast between the background color and the color of the text. In many cases, this problem could easily be fixed by client-side refactoring. The WCAG 2.0 [9] contains two success criteria for contrast. Success Criterion 1.4.3 specifies the minimal requirement for contrast, Success Criterion 1.4.6 specifies an enhanced requirement. In many cases, the only change necessary to match the requirements and make a web page better readable for people with sight problems is to make the darker color a bit darker and the lighter color a bit lighter.

Another common accessibility problem is that many websites do not specify their primary natural language (WCAG 2.0 Success Criterion 3.1.1) in which the content of the site is written. This information is needed by screen readers to choose the right pronunciation. A screen reader is a program that presents the information usually perceived

visually either as speech or as tactile output using a Braille output device. In HTML, it is also possible to specify the language of parts of a document by using an attribute (WCAG 2.0 Success Criterion 3.1.2). This information is also useful for screen readers. The screen reader can pronounce words or parts of a web page that are not written in the primary language of the page correctly if these parts are marked with the correct language.

A third example is the provision of alternative texts for images. These texts are often missing or applied incorrectly. The alternative text for an image is provided by the `alt` attribute of the `img` element. For decorative images, it is necessary to specify an *empty* `alt` attribute. Otherwise, the screen readers use the filename as an alternative text. More details about alternative texts for images on web pages can be found in the description of the img element in the HTML5 standard [23] and the description of technique H67 in [14].

1.3. Research Questions

Several authors (see section 2.3) have suggested that a machine-readable model of the guidelines for accessible web pages might help developers to create better tools for checking the accessibility of web pages. Until now, it was not clear if it is possible to create such a model based on the current guidelines. In this thesis, such a model, based on the current guidelines for accessible web pages (see section 2.1), will be developed. An additional goal for this model is to design the model in a way that allows it to use the model in software so that upgrades to the model do not require any changes in the code of the software. To investigate if it is possible to use such a model as base for accessibility checkers, a prototype of an application that uses the model will be developed as part of this thesis. For the development of the model, there are several design questions to be considered: There are several guidelines for accessibility concerning the accessibility of web pages. It is not clear which one of these recommendations is best suited for creating a machine-readable model. For practical use, the model has to be serialized in a suitable format. There are some formats, for example, the Web Ontology Language (OWL) [24], which allow not only to put data into such a model but also rules to infer new facts from the data in the model. Some parts of the logic for accessibility requirements might be put into the model itself, rather than relying on the developers using the model to implement this logic correctly.

Many tests for checking the accessibility of a web page can be automated, but some tests require human judgment. Some studies (see section 2.3) have shown that the results of manual evaluations often are not very reliable. To produce reliable results, the instructions for such tests have to be very precise to ensure comparable results. Therefore, the model and prototype of the tool have to be designed in a way that minimizes the ambiguity of the test instructions.

To summarize, two primary research questions will be investigated in this thesis:

- Can the requirements/guidelines for accessible web pages be normalized and put into a machine-readable model?
- Can such a model be used to create evaluation tools?

1.4. Outline

This thesis is structured as follows: in chapter 2, the background of this work and related work is discussed, including a description of the relevant standards for web accessibility and the current status of accessibility evaluation. In chapter 3, the declarative model itself is presented. An example of an evaluation tool using the model is presented in chapter 4. The results are discussed in chapter 5. Appendix B contains the publications this thesis is based on. These papers, presented at several conferences over the years, have provided continuous valuable feedback from the research community, particularly in regard to evaluating the prototype tool. A complete list of all tests used in the model described in this thesis is provided in appendix A.

2. Background and Related Work

2.1. Accessibility Standards for the Web

The primary (technical) guidelines for creating accessible web pages are the Web Content Accessibility Guidelines (WCAG) published by the W3C. The first version of the WCAG [12] was published in 1999 and has been superseded by the WCAG 2.0 [9] in 2008. After that, despite various technological changes, the WCAG have not been updated until 2018. One of the primary changes during this time is the increasing use of smartphones and tablets for accessing the web. In 2018 the WCAG 2.1 [9] have been published. WCAG 2.1 added several new requirements, especially for ensuring that web pages are accessible on mobile devices. Currently, another update, the WCAG 2.2 [4], is in its final steps. WCAG 2.2 contains additional updates to keep the WCAG in line with the devices and technologies for building websites and using the web.

Since version 2.0, the Web Content Accessibility Guidelines are organized by four principles:

- Perceivable
- Operable
- Understandable
- Robust

For each of these four principles, the WCAG contain several guidelines. Each guideline consists of several *Success Criteria* that describe the requirements for accessible web pages. For example, this includes providing alternatives for visual and auditory content, the contrast of text, and the use of color.

The WCAG 2.1 added 17 additional success criteria that address accessibility on mobile devices like smartphones and tablets. For example, this includes that a web page does not restrict its view and operation to a single display orientation (Success Criterion 1.3.4 Orientation) or that no scrolling in two dimensions is necessary (Success Criterion 1.4.10 Reflow). Also, the new Success Criterion 2.5.5 defines a minimum size for the target of pointer inputs. Version 2.2 of the WCAG will add success criteria to address some issues that have not been addressed fully in the WCAG yet. One example is the visibility of the keyboard focus. The W3C also provides supplemental documents for the Web Content Accessibility Guidelines. *Understanding WCAG 2.1* [11] explains the background of the guidelines and success criteria of the WCAG.

The Web Content Accessibility Guidelines are technology-agnostic. How the requirements defined by the success criteria of the WCAG can be implemented is explained in two

additional documents. *Techniques for WCAG 2.1* [10] provides a collection of techniques (best practices) for implementing accessible web pages using various approaches. In addition, *Techniques for WCAG 2.1* also contains a collection of *Failures* (Bad Practices). These failures describe common anti-patterns that make web content inaccessible. The techniques are organized into eleven groups: ARIA Techniques, Client-Side Script Techniques, CSS Techniques, Flash Techniques, General Techniques, HTML Techniques, PDF Techniques, Server-Side Script Techniques, Silverlight Techniques, SMIL Techniques, and Plain-Text Techniques. General Techniques are technology-agnostic. Most Server-Side techniques describe the usage of HTTP. All other techniques are tied to a specific technology. Flash Techniques and Silverlight Techniques are only of historical interest. Both Flash and Silverlight are more or less historic technologies that have been replaced with modern HTML, CSS, and JavaScript. How these techniques and failures relate to the success criteria of the WCAG is described in *How to meet WCAG (Quick Reference)* [17]. *How to meet WCAG* is an interactive document that maps the success criteria of the WCAG to the techniques and failures from *Techniques for WCAG*.

One goal of the Web Content Accessibility Guidelines is to enable assistive technology to process the information provided by a web page in a meaningful way. Assistive technologies like screen readers rely on information and interfaces provided by the operating system. For web pages, the browser has to provide this information to assistive technologies using the interfaces provided by the operating system. How browsers, or as they are called in most recommendations of the W3C, *User Agents*, should interact with these interfaces is described in the *User Agent Accessibility Guidelines* [5]. The UAAG uses the same structure as the WCAG 2 and define the accessibility requirements for user agents using guidelines, principles, and success criteria.

Most web content is created using tools like Web Content Management Systems. The *Authoring Tool Accessibility Guidelines (ATAG) 2.0* describe additional requirements for the accessibility of such tools. The ATAG have two goals: To ensure that authoring tools are accessible and that authoring tools support authors with the generation of accessible content. The ATAG are split into two parts: *Make the authoring tool user interface accessible* and *Support the production of accessible content*. Both parts use the same structure as the WCAG 2, using guidelines, principles, and success criteria to describe the requirements for accessible authoring tools.

In the beginning of the web, most websites were collections of static documents. This has changed with the wide adaption of JavaScript and related technologies. Modern websites are interactive applications that provide the same if not a better user experience as desktop applications. HTML alone can not provide all information required by assistive technologies to work with such interactive pages. To make it possible for developers to provide this information, the W3C has developed the *Accessible Rich Internet Applications Recommendation (ARIA)*. ARIA specifies additional attributes to provide the information required by assistive technologies.

HTML, at least since version 5, already provides authors with several elements for labeling the role of a section of a web page. For example, the element `nav` is used for sections that provide navigation. The main part of a web page can be labeled using the `main` element. However, for complex user interfaces, it is sometimes necessary to build

more complex widgets. ARIA defines several roles that can be used to describe the role of an HTML element. Most HTML elements have an implicit role [18]. For example, it is not necessary to add the role `main` to a `main` element because it already has this role. If necessary, the author can override this implicit role using the `role` attribute. For example, a tab sheet or tab panel is a common part of user interfaces. HTML has no special elements for building tab panels. They are usually built using `div` elements that are styled using CSS to look like a tab panel. ARIA provides the roles `tabpanel` and `tab`. These roles can be assigned to the HTML elements used to build the tab panel and the tab. For example, a `div` with the role `main` can be replaced with the `main` element. There are also several ARIA attributes to describe the state of a control or for linking elements of a web page with elements that provide additional descriptions.

There are some legal regulations concerning the accessibility of IT-Systems, such as the German "Barrierefreie Informationstechnikverordnung" (German Accessible Information Technology Ordinance, BITV 2.0) [40] or the European Web Accessibility directive [1]. Both mention the Web Content Accessibility Guidelines. Version 2.0 of the Web Content Accessibility Guidelines is also an ISO standard [25].

Unfortunately, it has become apparent that the Web Content Accessibility Guidelines and its supplemental documents are not that easy to use and to implement correctly. One problem is that the requirements defined by the W3C are often ambiguous. To make the interpretation less ambiguous, the W3C has developed the *ACT Rules Format* [19] (ACT stands for Accessibility Conformance Testing). The *ACT Rules Format* describes a structure for rules for testing the accessibility of web pages.

Two types of ACT Rules have been defined. Atomic rules define a specific requirement. Composite rules combine several other rules. Each ACT Rule consists of several sections. Both types of rules contain a unique ID, a description, a mapping to one or more success criteria of the WCAG, assumptions about the evaluated web page, or the elements for which the rule is applicable, possible limitations of assistive technology relevant for the rule, and test cases to check the implementation of a rule.

Moreover, atomic rules list the input aspects relevant to the rule, such as the DOM Tree or CSS Styling. The *Applicability* section of an atomic rule describes for which elements a rule is applicable. Each atomic rule defines at least one expectation that must be met by the elements for which the rule is applicable to pass the rule. If a rule has multiple expectations, each element for which the rule is applicable must pass all expectations. Composite rules may also have an Applicability section. The Expectation section of a composite rule lists all rules combined by the rule and defines whether all or at least one of the combined rules must pass. For evaluating the accessibility of a web page, the *Applicability* definition and *Expectations* are the most important sections. A community group has already created several rules using this format[1].

The practical application of the Web Content Accessibility Guidelines, as well as the development of web technologies, has lead to several accessibility challenges (see section 2.3). The W3C is currently investigating these challenges [35] and is working on version 3 of the Web Content Accessibility Guidelines [37].

[1] `https://act-rules.github.io/pages/about`

2.2. Accessibility Evaluation

Currently, the accessibility of web pages is usually evaluated manually. Several tools are available that can support the evaluator or can be used by developers during the development of a website or web application to check the web pages for accessibility issues. In this section, some of these tools are presented. The goal of this section is not to provide the reader with a full list of all available tools but to give the reader an impression of the problems with the current tools. The author of this thesis does not endorse any of these tools.

Some of the available tools focus on specific aspects of accessibility. One common accessibility issue is insufficient contrast between the background color and the color of the text. Such tools can be helpful when creating a new web design. One example is the WCAG Contrast Checker[2], which is provided as a browser extension for Firefox and Chrome. Figure 2.1 shows a screenshot of this tool.

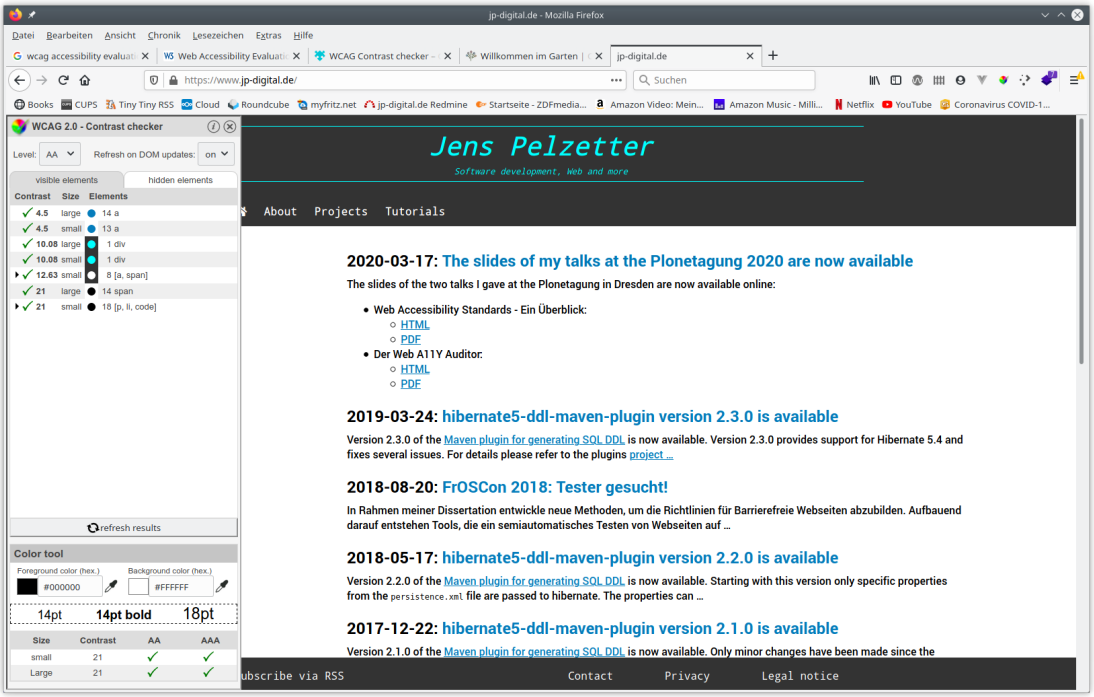

Figure 2.1.: Screenshot of the WCAG Contrast Checker

Many of the available tools are provided as extensions for browsers, in most cases, either Firefox or Chrome. Many extensions are available for both browsers. One well-known tool is *WAVE*, which is provided as a Web Service as well as an extension for Firefox and Chrome. WAVE primarily checks for DOM related issues. Also, in the option of the author of this thesis, the presentation of the results is not very user-friendly. The

[2] `https://addons.mozilla.org/de/firefox/addon/wcag-contrast-checker/`

output is cluttered with graphical symbols, and the web page itself is also obscured with graphical symbols. Because the symbols are inserted into the HTML of the web page, sometimes the layout of the web page is also affected by the output of the WAVE tool. Figure 2.2 shows a screenshot of the WAVE web application.

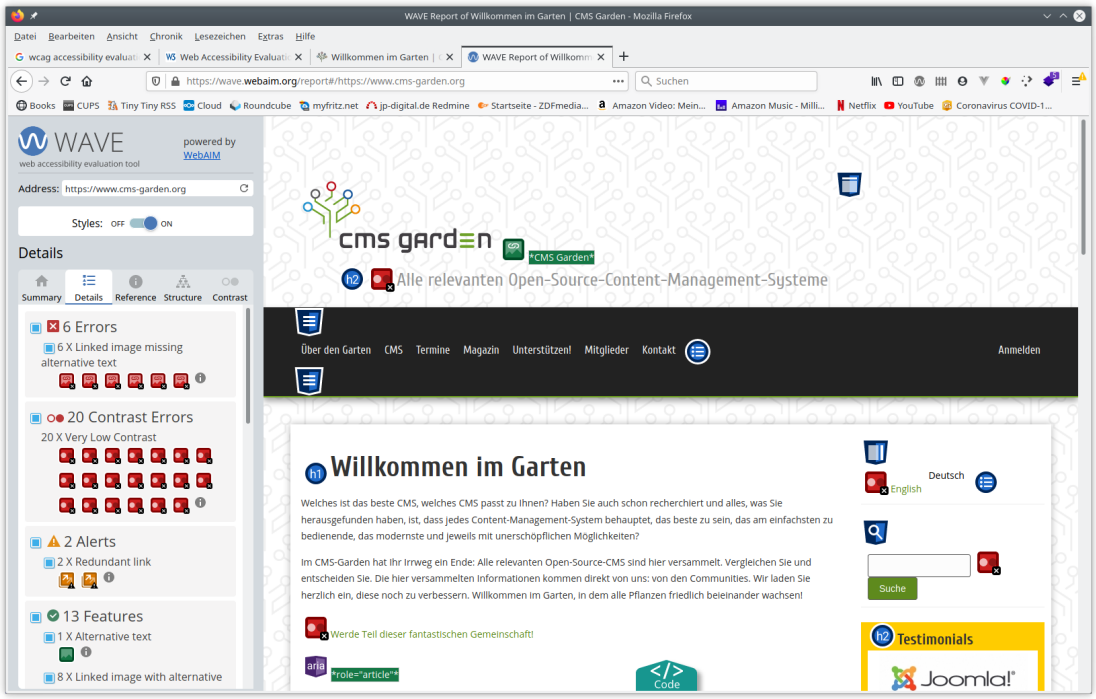

Figure 2.2.: Screenshot of the WAVE tool

Both of these tools only check those aspects of accessibility that can be automatically checked. For example, WAVE checks for the presence of headings but does not check whether the content of the heading is related to the content. There are tools developed in an academic background, which aim to check a wider range of accessibility, such as *MAUVE*. These tools are discussed in section 2.3.

Another approach uses manual test procedures. One prominent example in Germany is the BITV-Test[3]. The BITV-Test describes 60 steps for checking the accessibility of a web page. Most of these steps follow the same pattern: Open the page in a browser, check something manually, maybe check something else, record the result.

[3] `https://www.bitvtest.de/bitv_test/das_testverfahren_im_detail/pruefschritte.html`

2.3. Related Work

The accessibility of web pages can be evaluated using various methods. These methods can be grouped into three categories [2, 29]:

- Automatic Testing
- Expert Evaluation
- User Testing

A systematic review by Nuñez et al. [29] found very few studies about Expert Evaluation. In these studies, they found the experts evaluated the page via interaction.

User Testing is also a manual test methodology but with real users. Several tasks are developed beforehand by the evaluators. These tasks are executed by the participating users. User testing may be an effective method to find out how people with disabilities perform specific tasks on a web page [29].

According to Nuñez et al. [29], automated tools are mostly investigated by several studies. Most studies found that automated tools do not always find all existing accessibility issues on the evaluated page. The evaluation by experts is the best method to ensure compliance with standards. User testing is the best way to verify whether people with disabilities can use a web page. One problem with manual evaluation is that it is a very cumbersome task. Therefore tools that support the evaluation are crucial [2]. The available tools can be grouped into three categories [2]:

- Automatic evaluation tools
- Crowdsourcing tools
- Tools to support remote user evaluation

The effectiveness and correctness of automatic tools depend on the representation of the guidelines and on the ability of the developers of the tool to create a correct representation of the guidelines. Some guidelines require human judgment. For these guidelines, automatic tools can only give recommendations for manual evaluation [2].

One problem is the low coverage of the success criteria of the WCAG by automatic tools. The tools cover at most 50% of the WCAG 2.0 and often produce false negatives and false positives [2]. Many of these tools are available as plugins/extensions for web browsers like Chrome or Firefox. Frazão et al. [20] compared eight frequently used plugins for Google Chrome. Like Abascal et al.[2], they found that automated tools have poor coverage of the WCAG 2.1. Even when using multiple tools, they were only able to achieve a coverage of 40% of the WCAG 2.1. They also found several usability issues with the tools. Because of these usability issues and the variance of the results between the tools, they conclude that users may struggle with interpreting the results of these tools.

The correctness of automated tools depends on the ability of the developers of the tool to create a computational representation of the guidelines. The guidelines are written in natural language. Abascal et al. [2] suggest that there should be more consensus on how evaluation tools are developed and how they present their results. They also suggest that a simpler wording for the guidelines and unambiguous machine-readable specifications would help developers to create automated evaluation tools that produce better results.

Abascal et al. [2] also point out that accessibility is not yet fully integrated into the development process of web designers. Accessibility is often only considered when legal requirements demand it. According to Abascal et al., the reasons are that designers often lack training in accessible design and do not have the development tools that could help them. They suggest that better tools that are closely integrated into the workflows of designers and developers could help to improve the acceptance of accessibility as a quality factor.

The Evaluation and Report Language (EARL) [3, 39] has been proposed to make it easier to compare the results from different tools. EARL was developed by a working group of the W3C. Unfortunately, EARL has not reached the status of a W3C Technical Recommendation yet.

Several tools for user testing use crowdsourcing. These tools have two different approaches. Some of them are trying to improve the accessibility of a web page by adding metadata. The process of annotating a web page with such metadata is likely to be very time-consuming; crowd-based tools allow the distribution among many authors. Other crowd-based tools split the accessibility evaluation into small tasks for distribution, making the evaluation less expensive [2]. The effectiveness of such crowdsourcing-based tools has not yet been demonstrated.

Other crowdsourcing tools try to solve the problem of evaluating a web page of accessibility by dividing the evaluation process into small tasks. Each task is sent to several evaluators in parallel. The result is computed from the various results produced by the evaluators [2].

For evaluation by experts, a document from the W3C [13] defined that a technique reliable testable by humans if at least 80% of knowledgeable human evaluators agree on the conclusion. Several authors found that this level of agreement is not very likely. For example, Brajnik et al. [8] conducted a study with 25 experienced and 27 novices evaluators. They found that the experienced evaluators agree for 76% of the test cases. For novice evaluators, this rate drops by 10% for the same pages. Also, they found that expert evaluators are likely to produce 26-35% false positives and miss 26-35% of the real problems. Brajnik et al. also investigated whether a group of evaluators produces better results than individual evaluators [7]. To mimic a real-world scenario, the participants were novice evaluators with basic to intermediate knowledge of accessibility and the WCAG. The evaluations in the study were conducted in pairs. The results only differed marginally from the results of individual evaluators.

Accessibility is an important issue, especially for government sites. Some countries are already providing many services online. Due to the shortcomings of the existing tools, a custom tool has, for example, been created for testing the accessibility of Government Services in Singapore [28]. This tool combines several available tools: The Axe Evaluation

engine[4], the Chrome browser, and the Selenium framework[5]. The tool is integrated into their Continuous Integration pipeline.

Another approach for developing a test suite for web accessibility uses an XML-based language for specifying accessibility guidelines, the *Language for Web Guideline Definition* (LWGD), together with an environment called *MAUVE* [36]. The last paper about MAUVE is from 2015. Judging from their website[6], MAUVE has been updated to match the newest version of the WCAG. The checks in MAUVE are based on the techniques for implementing the WCAG [10].

The validation process of MAUVE is based on the DOM tree of the document. MAUVE downloads the web page to evaluate and creates the DOM tree itself. The validator module interprets the guidelines formalized in the XML language to check if the DOM tree passed the checks defined in XML. The LWGD language allows to define the element to check and several conditions to validate. The conditions can be combined using boolean operators.

The effectiveness of MAUVE was compared with the Total Validator[7], a commercial product. In comparison, MAUVE missed fewer problems than the Total Validator. For false positives, MAUVE reported more false positives than the Total Validator in some cases; in other cases, the Total Validator produced more. The paper about MAUVE [36] shows an example with two conditions: One to check whether an element is followed by another element, and one to check whether an element has a specific child element. The paper does not list all available conditions.

The W3C has also recognized the problems with the current guidelines and with the current accessibility tools. To summarize the requirements for a new version of the Web Content Accessibility Guidelines, the W3C is currently working on a document that summarizes the challenges with the current guidelines and possible approaches to mitigate them [35]. In the current working draft, several challenges are discussed. For example, how to scale conformance verification for large sites or the high number of possible permutations of dynamic sites. The working draft also describes several specific challenges to evaluating specific success criteria. For example, it is difficult to test a web page against success criterion *1.3.1 Info and Relationship* of the WCAG. Automated tools can only check the presence of structural markup but can not check if the markup is used correctly to represent the structure of the content.

The W3C has only recently started to work on the successor of the Web Content Accessibility Guidelines 2. Version 3 of the Web Content Accessibility Guidelines [37] will use a new structure and integrate some other recommendations like the *User Agent Guidelines* [5] and the *Authoring Tool Accessibility Guidelines* [34]. In the current draft, the WCAG 3 are structured into guidelines, which explain an accessibility requirement in plain language, outcomes that describe testable criteria for a guideline, and tests for validating the outcomes. In the current version, the draft distinguishes between two types of tests: *Atomic tests* are automated, semiautomatic, or manual tests. *Holistic*

[4] `https://www.deqeue.com/axe/`
[5] `https://www.selenium.dev/`
[6] `https://mauve.isti.cnr.it`
[7] `https://www.totalvalidator.com`

tests include testing by users and with assistive technology. The WCAG 3 will also include a scoring mechanism to communicate better how accessible a web page is.

3. A Declarative Model for Accessibility Requirements

3.1. Overview

Several authors (see section 2.3) have identified a machine-readable model of the requirements for accessible web pages as critical for a better adaption of the guidelines for accessible web pages. In this chapter, a possible approach for such a model is presented. The model has gone through several iterations, which are also described in this chapter.

3.2. Using OWL as a Rule Engine

The first version of the model developed in 2016/2017 (see section B.1) was based on the success criteria of the WCAG 2.0 [9] and the techniques and failures described in Techniques for WCAG 2.0 [14] and the sufficient techniques and failures as defined in the WCAG Quick Reference [41]. The goal was to put as much logic as possible into the ontology. A tool using this ontology would only be responsible for extracting information from the document to evaluate and adding this information to the ontology. With this approach, each evaluation would be stored as a separate ontology. This ontology containing the evaluation data imports the ontologies providing the definitions of the WCAG and the techniques.

Besides a model of the Web Content Accessibility Guidelines and the techniques and failures, this ontology also contained several rules for inferring whether a web page satisfies the success criteria of the WCAG and which conformance level a web page achieves. With this approach, an application using the model would only be responsible for executing some tests on the evaluated web page and write the results back to the ontology. This approach was presented at the PhD Symposium of the Extended Semantic Web Conference 2017 [30] (see also section B.1).

The ontology was created using the Web Ontology Language (OWL) [24]. For some more complex rules, the Semantic Web Rule Language (SWRL) [22] was used. The idea was that a tool using the ontology only has to check which techniques have been successfully implemented by the evaluated page and which failures can be found in the evaluated page. The results would be added to an ontology containing the data of the evaluation. Using several SWRL rules, the reasoner could infer which success criteria are satisfied by the page and which conformance level of the WCAG the page achieves. One characteristic of OWL, the Open World Assumption, turned out to be a problem. Due to this assumption, a reasoner may assume that there might be other individuals with similar characteristics, even if there are none defined in the ontology. For example, it

is not possible to simply write a rule that states that a web page meets a conformance level when all success criteria required by the conformance level are satisfied. Instead, it is necessary to use cardinality statements:

```
WebPage(?p), (meetsSuccessCriterion min 25
SuccessCriterionForConformanceLevel-A)(?p)
-> compliesToConformanceLevel(?p, ConformanceLevel-A)
```

The SWRL rule is used to infer if the page complies to conformance level A of the WCAG. To allow the reasoner to infer the conformance level, a cardinality statement is used, which contains the exact number of success criteria required by conformance level A of the WCAG. To make it easier to write these rules, a subclass of the class `SuccessCriterion` has been added to the ontology. This subclass contains all success criteria required by a conformance level. With OWL, it is possible to infer if an individual is an instance of a specific class. In this case, all individuals of the class `SuccessCriterion` associated with the individual representing the conformance level A by the object property `requiredBySuccessCriterion` are instances of this class.

Unfortunately, this approach did not scale well. For example, multiple axioms are needed for each success criterion to state how many situations a success criterion has. Most of these axioms are required to tell the reasoner that there are no more individuals, for example, success criteria or sufficient techniques, to be considered. This large number makes it difficult to maintain the ontology. To make this task easier, a small application was developed that scraped the web pages of the W3C and created the axioms.

Nevertheless, the ontology needed additional manual refinement. Another problem was that for each success criterion, multiple axioms have to be added to the ontology. For example, each association between a success criterion (or one of its situations) and a sufficient technique is an axiom. An additional axiom is needed to tell the reasoner that there are only these techniques that are sufficient and no more. This leads to an increasing number of axioms, which causes a low performance of the reasoner. In some cases, the reasoning takes several hours and sometimes even causes out of memory errors. Due to these problems, the idea to put most logic into the ontology was abandoned, and a simpler approach has been implemented which uses an ontology only as a knowledge base.

3.3. WCAG Sufficient Techniques and Failures vs ACT Rules

For the second version, several options were considered as a starting point. During the development of the first approach (see section 3.2), it became already obvious that the structure in the WCAG Quick Reference [41, 17] is difficult to translate to a formal model. For each success criterion, the WCAG Quick Reference provides a number of techniques that are sufficient for satisfying the success criterion. Also, several failures are listed for each success criterion. If one of these failures is present, a web page fails the success criterion. For which elements a success criterion is applicable is described in natural language. For many success criteria, the sufficient techniques differ depending

on the context of the element. The WCAG Quick Reference uses the term *Situation* to describe these contexts. For example, the sufficient techniques for Success Criterion *1.4.1 Contrast (Minimum)* depend on the size of the text and its font-weight:

- Situation A: text is less than 18 point if not bold and less than 14 point if bold
- Situation B: text is at least 18 point if not bold and at least 14 point if bold

For other success criteria, for example, *1.4.8 Visual Presentation*, the structure is different. For these success criteria, a number of requirements are described. For each requirement, several sufficient techniques are listed. To satisfy these success criteria, all of the requirements must be satisfied. For some of the success criteria added in version 2.1 of the WCAG, yet another structure is used. For instance, the success criterion *1.4.11 Non-text Contrast* covers two different types of objects inside a web page: User Interface Components and Graphical Objects. For this success criterion, the sufficient techniques are organized into three groups: User Interface Component Contrast, Graphics with sufficient contrast, and Text in or over graphics. For some success criteria, the list of sufficient techniques is not a flat list. Instead, a technique from the group of general techniques is listed, and several technology-specific techniques for implementing the general techniques are listed for each of the general techniques.

Some of the techniques described in *Techniques for WCAG* are written in a very general way to cover as many use cases as possible. Therefore, it is possible to apply these techniques to most web pages. However, these often unspecific descriptions make it sometimes difficult to test if a technique has been used correctly. Also, this makes it difficult to write automated or semi-automated tests for the techniques.

To provide a less ambiguous set of rules, the W3C has published the ACT Rules Format recommendation [19] (see section 2.1). Because of the clearer and less ambiguous structure, the ACT Rules have been used as the foundation for building the declarative model described here. Each ACT Rule uses the same basic structure. For atomic rules, two sections are most interesting for creating a declarative model: The applicability definition describes for which elements the rule is applicable. The expectations define the requirements which an element for which the rule is applicable must meet.

3.4. Atomic Tests as Building Blocks of the Declarative Model

For designing the model that will be described in this section, it was decided to go a step back and try to design the model in a way that is independent of a particular implementation and serialization. As described in the previous section, it was decided to base the model described in this section on the ACT Rules.

Among the available rules, both the applicability definitions as well as the expectations contain many repeating phrases such as "... is included in accessibility tree ...". These repeating phrases are used as the starting point for developing the declarative model described here. Based on these phrases, a number of *atomic tests* have been defined,

which can be used to describe the expectations and the applicability requirements of a rule in a formal way. How these tests are implemented is not part of the model but the responsibility of the developers of the application that uses the model.

Each of the tests can have several outcomes. The ACT Rules Format recommendation [19] specifies three possible outcomes for a rule: `Inapplicable`, `Passed`, and `Failed`. For the model described here and with an implementation in mind, the possible outcomes for tests have been extended. If the tested element passes the test, the outcome of the test is `PASSED`. If the element fails the test, the outcome is `FAILED`. If the test has not yet been executed, the outcome of the test should be recorded as `UNKNOWN`. If a test is executed on an element for which the test is not applicable, the outcome of the test is `INAPPLICABLE`. An atomic rule in the ACT Rules Format – composite rules do not contain any tests – uses these tests in the applicability definition and the expectations. The applicability definition only uses tests that are applicable to all elements. In the expectations, some tests might only work for specific tests. Therefore the outcome `INAPPLICABLE` should occur very rarely and only during the development of new rules or tests and would suggest a wrong usage of a test. If an implementation supports manual tests, the implementation can use the outcome `WAITING_FOR_MANUAL_EVALUATION` to indicate that a test has to be performed by a human. If an implementation does not support manual tests or cannot determine the outcome of a test for other reasons, the outcome of the test is `CANT_TELL`.

For example, the rule *Button has an accessible name*[1] checks if a button has an accessible name. The applicability definition for this rule is:

> The rule applies to elements that are included in the accessibility tree with the semantic role of *button*, except for *input* elements of *type="image"*.

This applicability definition can be broken down into three tests:

- Checking whether the element is included in the accessibility tree.
- Checking whether the element has the semantic role of `button`.
- Checking whether the element is an image button. This can be done using the CSS selector `input[type=image]`.

If written as pseudo-code, the rule may look like this:

- `isIncludedInAccessibilityTree()`
- `hasRole("button")`
- `matchesCssSelector("input[type=image]")`

Three "meta"-tests have been defined to combine the outcomes of multiple tests, resembling the usual combinations *and*, *or* and *not*. To avoid conflicts with languages like OWL or programming languages, the tests are named `allOf`, `oneOf`, and `negate`. `allOf`

[1] `https://act-rules.github.io/rules/97a4e1`

and `oneOf` can be used to combine multiple tests and reduce the outcome of these tests to a single outcome. `allOf` will return `PASSED` as outcome if all combined tests have the outcome `PASSED`. If one of the combined tests has the outcome `FAILED`, the outcome of `allOf` will be `FAILED`. If one of the combined outcomes is either `INAPPLICABLE`, `UNNKOWN`, `WAITING_FOR_MANUAL_EVALUATION` or `CANT_TELL`, the outcome of `allOf` is this value.

`oneOf` only requires one of the combined tests to pass. The outcome of `oneOf` is `FAILED` if the outcome of all combined tests is `FAILED`. If one of the combined outcomes is `INAPPLICABLE`, `UNKNOWN`, `WAITING_FOR_MANUAL_EVALUATION`, or `CANT_TELL`, the outcome of `oneOf` is this value. The negation of outcomes works in a similar way. If the outcome of the test to negate is `PASSED`, the outcome of `negate` is `FAILED` and vice versa. All other outcomes (`INAPPLICABLE`, `UNKNOWN`, `WAITING_FOR_MANUAL_EVALUATION`, or `CANT_TELL`) are not changed.

With these three additional tests, it is possible to write down the applicability definition of a rule in a more formal way. For image buttons, the rule *Button has an accessible name* is not applicable. Therefore the test has to be negated:

```
not(matchesCssSelector("input[type=image]"))
```

The rule is only applicable for elements that pass all three tests. Therefore the outcomes are combined using `allOf`, which results in the following definition in pseudo-code:

```
allOf(
    isIncludedInAccessibilityTree(),
    hasRole("button"),
    not(matchesCssSelector("input[type=image]"))
)
```

The expectation of this rule requires that every button has an accessible name:

> Each target element has an accessible name that is not empty (`""`).

To check whether an accessible name is available for the element, only one test is necessary:

```
hasAccessibleName()
```

Another example: Images can be used for different purposes on a web page: Either as pure decoration or to support the textual content of the web page. If an image is not used as decoration, it needs an accessible name that describes the content of the image. Decorative images have to be marked correctly using an empty `alt` attribute or with an appropriate semantic role. The rule *Image has accessible name*[2] checks whether an image is marked as decorative or has an accessible name.

The applicability definition of the rule is:

[2] `https://act-rules.github.io/rules/23a2a8`

> The rule applies to HTML `img` elements or any HTML element with the semantic role of `img` that is included in the accessibility tree.

This applicability definition can be broken down into three tests:

- The element has the semantic role of `img`. Sometimes the equivalent role `image` is used for images. This role is also provided as a parameter for the `hasRole` test.
- All `img` elements are applicable, regardless of the role assigned to them.
- An applicable element is included in the accessibility tree.

Or if written as pseudo-code:

- `hasRole("img", "image")`
- `matchesCssSelector("img")`
- `isIncludedInAccessibilityTree()`

The test `hasRole` accepts multiple parameters. In this case, the test has two parameters. Because there are often multiple options for the correct semantic role, the test accepts an unlimited number of role names as parameters. To pass the test, the tested element must have one of these semantic roles.

In this case, the rule is applicable for elements with the semantic role `image` or `img` or for `img` elements, regardless of their semantic role. Therefore these two tests are combined using `oneOf`. In addition, the tested element must also be included in the accessibility tree. Therefore the outcome of the test `isIncludedInAccessibilityTree` is combined with the outcome of `oneOf` using `allOf`. Combined, the applicability definition of the rule *Image has accessible name* can be written as follows (pseudo-code):

```
allOf(
    oneOf(
        hasRole("img", "image"),
        matchesCssSelector("img")
    ),
    isIncludedInAccessibilityTree()
)
```

The expectation of the rule is:

> Each target element has an accessible name that is not empty (`""`), or is marked as decorative.

This expectation defines two options for passing the test:

- the image has an accessible name

- the image is marked as decorative

The target element has only to pass one of these tests to pass the rule. Therefore the two tests `hasAccessibleName` and `isDecorative` are combined using `oneOf`:

```
oneOf(
    hasAccessibleName()
    isDecorative()
)
```

Based on the ACT Rules already available from the ACT Rules Community Group, 48 atomic tests[3] have been defined. Some of these tests accept parameters. Other tests do not require any parameters. A list of the tests can be found in appendix A. To create a model of the ACT Rules, the tests have been combined to describe the applicability definitions and the expectations of the rules. In addition, a model might also contain additional information, for example, the descriptions of the rules. For example, the complete rule *Image has accessible name* could like this:

```
[
    ...,
    {
        "id": "23a2a8",
        "name": "Image has accessible name",
        "description": "...",
        "applicability": allOf(
            oneOf(
                hasRole("img", "image"),
                matchesCssSelector("img")
            ),
            isIncludedInAccessibilityTree()
        ),
        "expectations": [
            oneOf(
                hasAccessibleName()
                isDeorative()
            )
        ]
    },
    ...
]
```

This example uses a JSON-like notation for showing the structure of a rule in the model. In addition to the applicability definition and the expectations – in this case, the

[3]Note: The current working draft of the WCAG 3 also uses the term atomic tests, but with another meaning

rule has only one expectation – the rule has some additional properties: A unique ID, a name, and a description of the rule.

3.5. An OWL Ontology of the Model

The model described in the previous section is not tailored for a specific serialization. There are several requirements for a serialization format: The format must provide some option to describe object-list structures and support properties (for descriptions, names, etc.). It would also be helpful that the format supports some method to define a schema to ensure the consistency of the model. For this thesis, a serialization using OWL has been developed. The model could also be expressed using RDF or a custom JSON format. OWL has several advantages compared with other options like RDF. One advantage is the possibility to check the consistency of an OWL ontology. OWL has a clearly defined semantics, and there are several semantic reasoners available that allow it to reason on an ontology. When used as a knowledge base only, the number of axioms is manageable, and there are no performance problems like those encountered with the first version of the ontology that contained the evaluation data, the model, and the rules to infer which success criteria the evaluated page passes or fails. Another advantage is the availability of a graphical editor (Protégé[4]).

The ontology also contains additional data, for example, the success criteria from the WCAG [9, 27] and how they are related to the ACT Rules, and the description of the techniques and failures from *Techniques for WCAG* [10]. These data can be used by applications using the ontology to provide the users of an application using the model with background information about the rules and the results. The ontology has been split into smaller modules to make maintenance of the ontology easier. Each of these modules is a separate OWL file that imports some of the other files. In addition, the structure of the model – classes and the definitions of object and data properties – and the data – the individuals describing the rules, etc. – are kept in separate files.

This separation makes it easier to maintain the ontology. The structure will not change, but the data might change. For example, the structure of the WCAG 2.0, 2.1, and 2.2 is the same. Only the success criteria have been changed. To create an ontology for a newer version of the WCAG 2, only the data module has to be edited. The structure module is not changed.

The structure modules are:

- `tests.owl`
- `wcag2x.owl`
- `wcag2x-techniques.owl`
- `act-rules.owl`

[4] `https://protege.stanford.edu/`

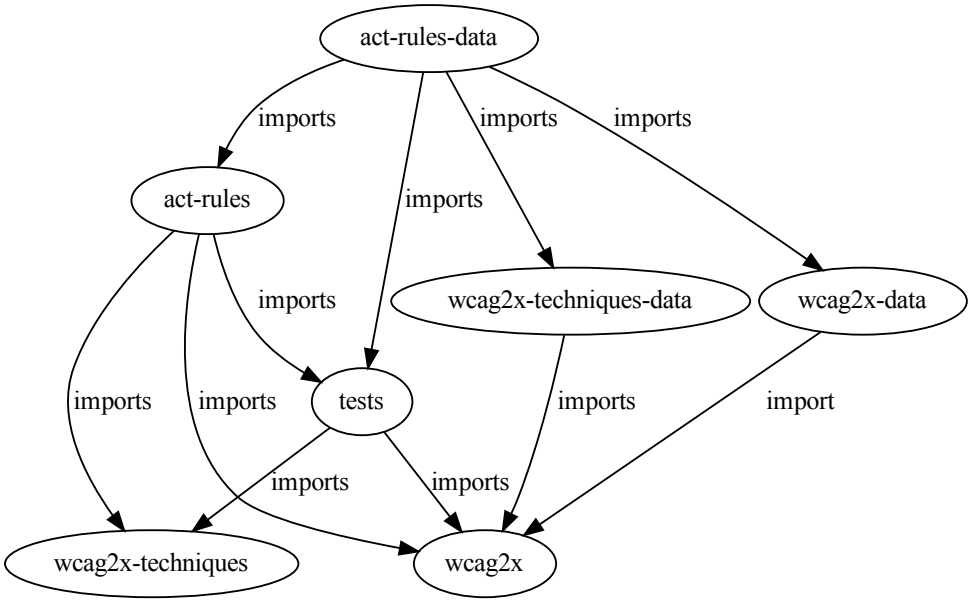

Figure 3.1.: Ontology modules and imports

These four OWL files only contain classes and definitions for object and data properties. The entities in the `tests.owl` file represent the tests and their parameters. The `wcag2x.owl` file contains classes, object, and data properties for modeling the concepts of the Web Content Accessibility Guidelines like success criteria, guidelines, and principles. `wcag2x-techniques` contains the classes, object properties, and data properties for describing the techniques described in the *Techniques for WCAG 2.1* [10] document. The `act-rules.owl` file provides the classes, object properties, and data properties needed for modeling ACT Rules as an ontology.

The data modules are

- `wcag21-data.owl`
- `wcag21-techniques.owl`
- `act-rules-data.owl`

The `wcag21-data.owl` file contains the instances for the principles, guidelines, and success criteria defined in the Web Content Accessibility Guidelines 2.1 [27], the file `wcag21-techniques-data.owl` contains instances for the techniques described in the *Techniques for WCAG* 2.1 [10]. The ACT Rules themselves are provided by the file `act-rules-data.owl`. The entities from the files `act-rules-data.owl`. `act-rules.owl`

and `tests.owl` are used in the file `act-rules-data.owl` for modeling the ACT Rules and the test procedures for the applicability and the expectations of the rules. The data from the WCAG modules is only used to provide the user with background information about the rules.

Each test is represented by an OWL class in the ontology. A specific instance of a test is represented by an OWL individual. Some tests have parameters that are represented as data properties or object properties. For example, the test `hasAttribute` checks if the target element has a specific attribute. The name of the attribute is provided by the `attributeName` data property. The "meta"-tests `allOf`, `oneOf` and `negate` are also represented by OWL classes. To associate instances of these classes with the tests, the object properties `combines` and `negates` are used.

The ACT Rules Format recommendation [19] defines two types of ACT Rules. Atomic Rules describe specific requirements. Composite Rules combine the outcome of several other rules. Both types of rules are represented by OWL classes in the ontology. The applicability definitions and expectations are built using additional classes and object properties.

The UML diagram in figure 3.2 shows the primary classes and properties of the ontology. The classes for the individual tests as well as several properties – most of them are inverse properties of the properties in the diagram – have been omitted for brevity. For subclass relationships and object properties, the usual UML notation is used. Data properties are shown in their domain class. For the tests, only the base class `Test` and the classes for the three "meta" tests are shown. Each of the tests listed in appendix A is represented by a class. For tests that have parameters, there are also data properties that are not shown in the diagram. These data properties are used to describe the parameters of the tests.

Atomic and Composite ACT Rules are represented by the `AtomicACTRule` and the `CompositeACTRule` classes. Both types of rules share several properties. The common base class `BaseACTRule` is used to collect these properties. The input rules of a Composite ACT Rule are provided by the `hasInputRule` object property. Because a composite rule can use both Atomic Rules as well as Composite Rule as input rules, the range of this object property is `BaseACTRule`. For the applicability definition and the expectations, separate classes (`ACTRuleApplicability` and `ACTRuleExpectation`) are used.

For the expectations, this was necessary because a rule can have multiple expectations. To keep the structure of expectations and applicability definitions as similar as possible, the applicability definition is also represented by a separate class. To provide the test for the applicability definition, the object property `isSelector` is used. The test of an expectation is provided by the `hasTest` object property. Both properties are functional object properties. Therefore, an individual of the domain class can only be associated with exactly one individual of the range class.

To provide some background information, the ontology also contains classes for the concepts of the Web Content Accessibility Guidelines (`Principle`, `Guideline`, `Success-Criterion`), and the properties for describing their relationships with each other as well as with the ACT Rules. The ontology also contains classes for the concepts *Technique* and *Failure* from *Techniques for WCAG* [14, 10] and properties for relating them to ACT

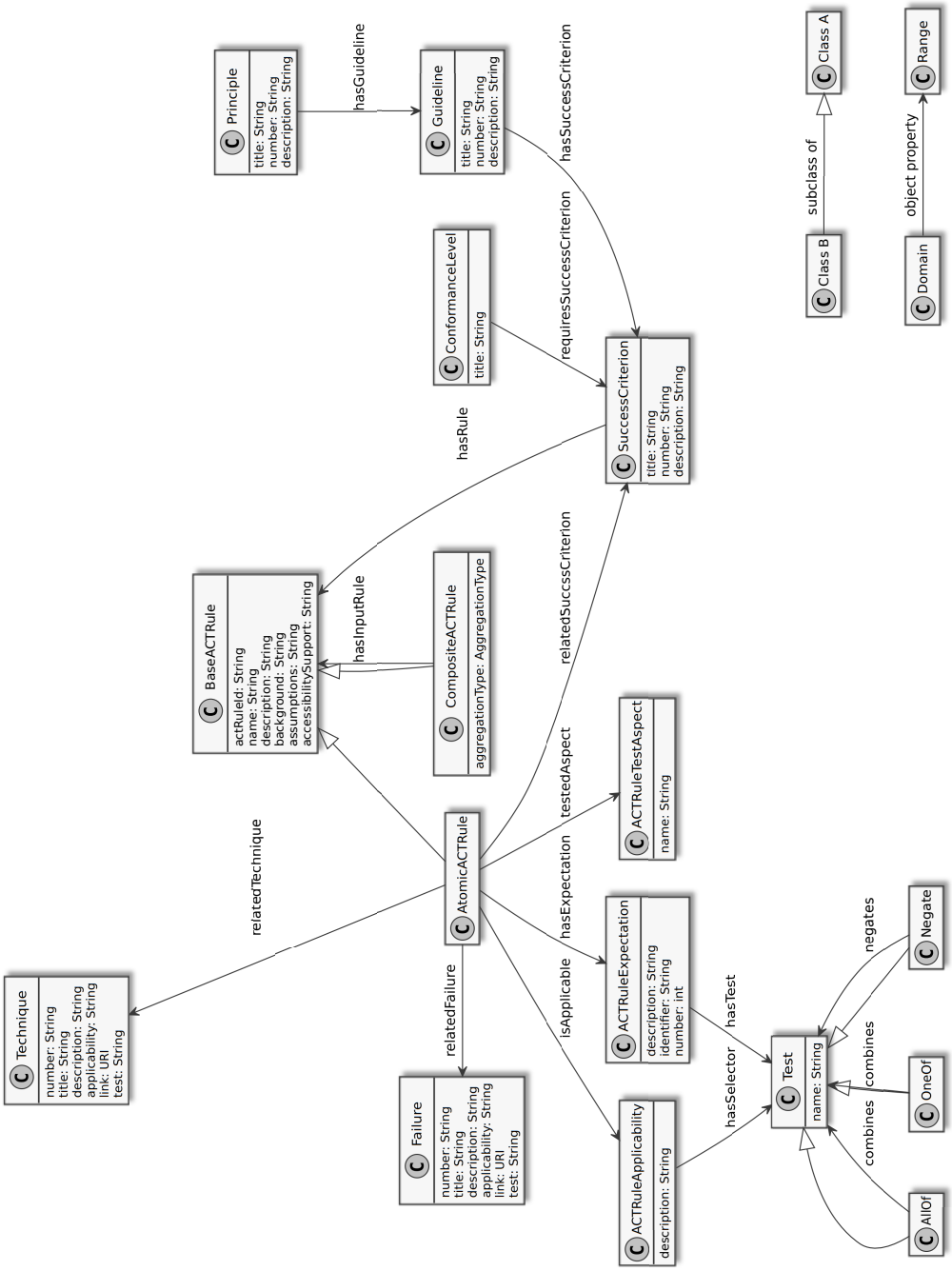

Figure 3.2.: Primary classes and properties of the ontology.

Rules. The ontology does not yet provide properties to describe the sufficient techniques for a success criterion. The ontology described in this section is also described in some more details in [32].

4. Using the Model

4.1. Overview

In this chapter, the prototype of an evaluation tool – the *web-a11y-auditor* – that uses the declarative model described in chapter 3 is presented. The *web-a11y-auditor* uses the OWL-based serialization of the model described in section 3.5. The general architecture of the *web-a11y-auditor*, as well as the challenges encountered during the implementation of the *web-a11y-auditor*, are discussed.

4.1.1. Design Considerations

Several authors describe automation as a key factor for making accessibility evaluations for web pages easier and less work-intensive [2, 29]. Not all tests can be done automatically, but a significant number can be automated. There are several options for doing automated accessibility tests. All approaches have to inspect the HTML of the web page to evaluate. More exactly, they have the inspect the so-called DOM tree created from the HTML source. *DOM* stands for *Document Object Model*, a standardized API [42] for interacting with the element tree of an HTML document. There are libraries like JSoup[1] that allow it to parse and interact with an HTML document in an application. Using such libraries can lead to wrong results because these libraries do not behave in the same way as browsers. For some tests, for example, checking if an element can be focused using keyboard interaction, it is necessary to simulate these interactions. Also, several rules require to analyze the styling of the element. The styling of HTML documents is done using Cascading Style Sheets (CSS) [6, 26]. Libraries like JSoup usually do not implement a CSS parser. Therefore it is necessary to run these tests in a browser.

Running the automatic tests in a browser is already used to test the user interface of web applications. To control browsers from another application, a standardized interface, the Web Driver API [38], has been developed. The Web Driver API is implemented by all major browsers, such as Firefox or Chrome. Several test frameworks are available which use this API for running tests in a browser. Because of its maturity and the support for various browsers using the standardized Web Driver API, the Selenium framework was selected. Another important aspect to consider is the usability of the application. Usability aspects include, for example, the installation of the app – ideally, the user should not have to worry about this – or an easy-to-use user interface.

[1] `https://jsoup.org/`

4.1.2. First Attempts

In the course of this dissertation, several implementations have been created, alongside the different versions of the model. The first attempt was a desktop application implemented using Java Swing. This application required the user to install the Web Driver for their operating system manually and configure the browser to use accordingly. Initial tests showed that this was too complicated even for developers. Therefore the application was reimplemented as a web service. The first version of this web service was a single application. This version had serious performance problems. The current version has been split into several independent services. The architecture of this application – the *web-a11y-auditor* – is discussed in the remainder of this chapter.

4.2. Architecture

There are several general functions that have to be provided by the application:

- Provide a UI for the users
- Create the necessary entities in the database
- Save the evaluation results in the database
- Analyze the document
- Execute the tests for applicability and expectations

The user interface of the web-a11y-auditor is provided by a JavaScript application implemented using the Nuxt.js framework[2]. The application uses a RESTful API provided by the Web module of the *web-a11y-auditor*. This module is only responsible for creating the basic entities for storing the data of the evaluation and for providing read access to the database.

The central module is the Job Manager, which receives the results from the worker instances, creates new jobs, and stores the results in the database. The worker module is responsible for executing the tests and is the only module that communicates with the browser used for testing using Selenium. This separation makes it easy to replace Selenium with another test framework, if necessary. Multiple worker instances may be started to speed up the evaluation process. The Job Manager is also responsible for reading the information about the rules from the OWL ontology and for creating the entities for tests, rules, and other objects based on these data.

Working with OWL ontologies in an application turned out the be challenging. The standard library for working with OWL ontologies is the OWL API[3]. One particular problem with this library is that even simple tasks like getting the value of a data property require much boilerplate code. Especially for developers who are used to other

[2] `https://nuxtjs.org/`
[3] `http://owlcs.github.io/owlapi/`

common APIs like the Java Persistence API, the OWL API feels cumbersome to use. To reduce the amount of boilerplate code to be written for the applications developed while working on this thesis, it was decided to move the repeating parts to a separate library. Constants for the IRIs of OWL classes and properties proved to be very useful. Therefore, a code generator for these constants was added to the library. The result is called *owlapi-simplex* and is available at Github[4]. *owlapi-simplex* is a code generator that generates most of the boilerplate code for a given ontology automatically. More precisely, *owlapi-simplex* generates the following things:

- constants for the IRIs of all OWL entities (Classes, Data Properties, Object Properties) in the given ontology
- Repository style classes for all OWL classes in the ontology
- A loader class that takes care of loading the ontology.

The repository classes provide methods for retrieving all instances of an OWL class with a single-line method call. Another part of *owlapi-simplex* is a collection of utility classes that provide simpler methods than the OWL API for retrieving the values of object and data properties.

Communication between the modules of the *web-a11y-auditor* is done using Apache ActiveMQ Artemis[5]. The messages are managed using several queues and topics. For storing the data, a PostgreSQL database is used. The diagram in figure 4.1 shows an overview of the architecture.

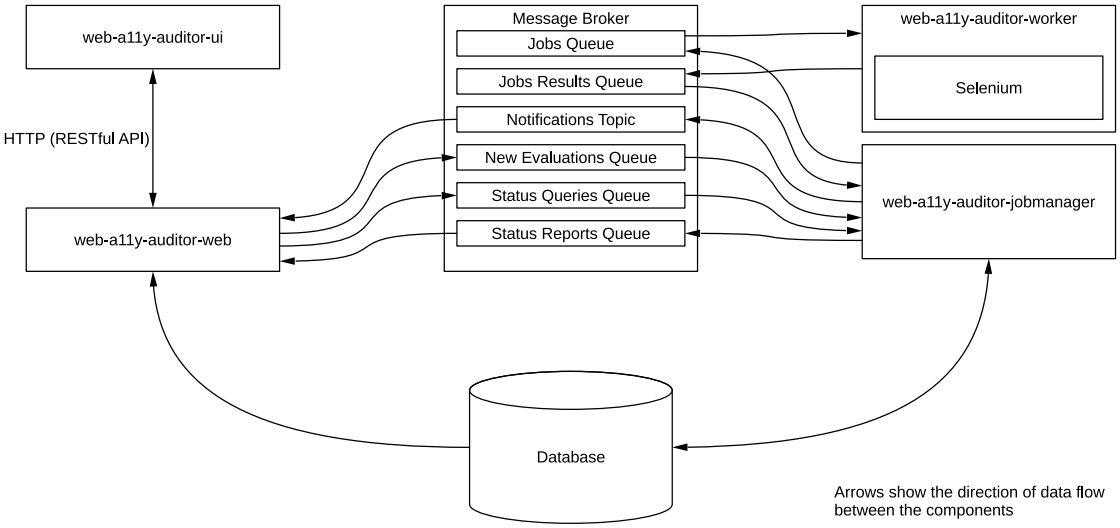

Figure 4.1.: Overview of the architecture of the web-a11y-auditor.

The evaluation of a web page is done in five steps:

[4] `https://jpdigital.github.io/owlapi-simplex/`
[5] `https://activemq.apache.org/components/artemis/`

1. Analyze the document and store a screenshot of the document and the CSS selectors and coordinates of each element. The screenshot and the element coordinates are used by the web application to show which element is related to a result. Instead of storing a screenshot for each element, only a screenshot of the document and the coordinates of each element are stored to reduce the required storage space. It is not unusual for a web page to consist of 500 or more individual elements. Even if each screenshot only has 1 MB, an evaluation would require 500 MB only to store the screenshots.

2. Generate the tasks for checking which rules are applicable for which element of the evaluated document.

3. Check which rules are applicable for which element.

4. Generate the tasks for checking the expectation for the applicable rules.

5. Check if the elements match the expectations of the applicable rules.

For each of the atomic tests to execute, the Job Manager creates a task and puts it into a message queue. The templates for these tasks are read from the ontology. If a worker is idle, the worker regularly checks the task queue for new tasks. If the task queue contains a task, the task is retrieved by the worker instance and processed. The result is sent back to the Job Manager using a result queue. The result queue is regularly checked by the Job Manager. Aside from the specification of the tests, all other data provided by the ontology is only used by the web application for providing the user with additional information about the rules.

Some tests can not be done automatically yet. These tests are added to a list of tests that have to be performed by the user. The atomic tests allow it to ask the evaluator very specific questions, which makes it possible even for inexperienced users to do an evaluation. For example, checking for the existence of headings can be done automatically. For checking if the content of a section is sufficiently described by its heading, human judgment is required. In this case, the *web-a11y-auditor* will create a manual evaluation task for each heading, with the question: *Does the highlighted heading "example heading" describe the content of section associated with the heading?* (*example heading* is replaced with heading text). The Job Manager adds these questions to the list of manual tests.

4.3. User Interface

One problem of existing tools for evaluating the accessibility of web pages is the user interface, which is often not very intuitive to use. Especially for people with little experience in accessibility, these tools are often difficult to use. The user interface of the *web-a11y-auditor* is intended to be as simple as possible without losing important aspects.

To start an evaluation with the *web-a11y-auditor* (see figure 4.2), the users only have to enter the URL of the page to evaluate.

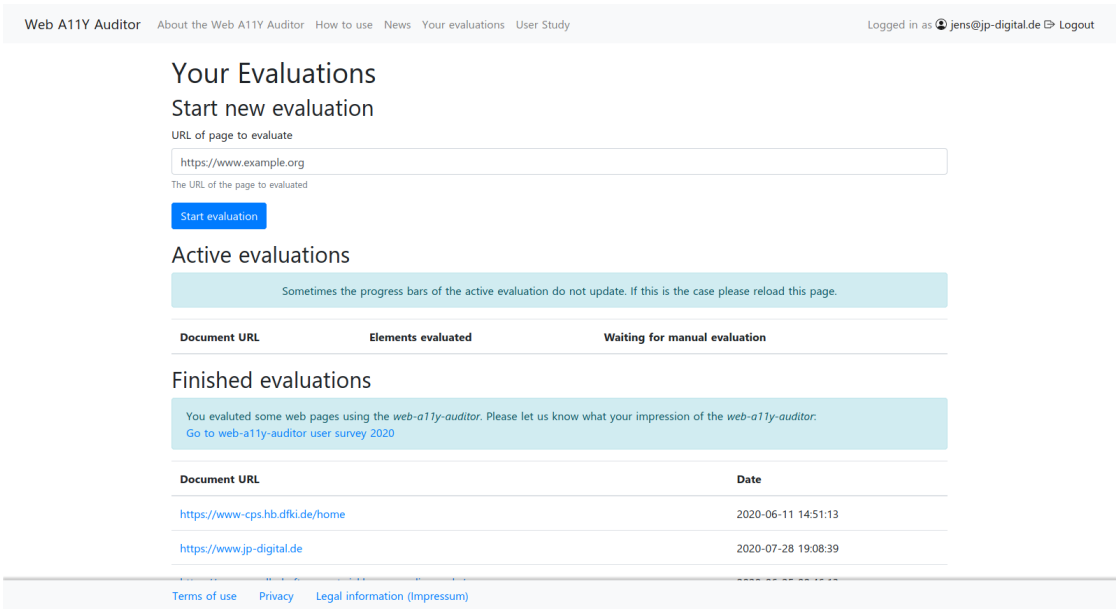

Figure 4.2.: Start an evaluation

After starting the evaluation, the input field is replaced by some progress bars showing the progress of the evaluation (see figure 4.3).

For manual tests, the user is presented with a simple question and a screenshot of the evaluated page. The element to be evaluated is highlighted if the element is visible (see figure 4.4). The user is only given two options to answer the question: *Passed* and *Failed*.

After all automatic and manual tests have been executed, the user can browse the results for each rule and element (see figure 4.5). For each rule, the overall result is shown by the badges beside the name of the rule. Selecting a rule leads the user to a screen that provides more details about the results of the rule for the tested document. For each tested element, the result is shown. The user can also view a screenshot in which the element is highlighted.

4.4. Adding New Rules

One goal of using a declarative model for the accessibility rules was to avoid code changes when a rule is updated or when new rules are added. The rules developed by the ACT Rules Community Group, which have been used for the creation of the model, have been updated regularly while this dissertation was in progress. Integrating them worked as expected. It was only necessary to integrate the changes into the declarative model (the OWL ontology). After replacing the OWL files used by the *web-a11y-auditor*, the *web-a11y-auditor* used the updated rules without any changes in the code. It was only necessary to change the code of the *web-a11y-auditor* when an updated rule used a test that was not already part of the model. The methods for executing the tests are isolated

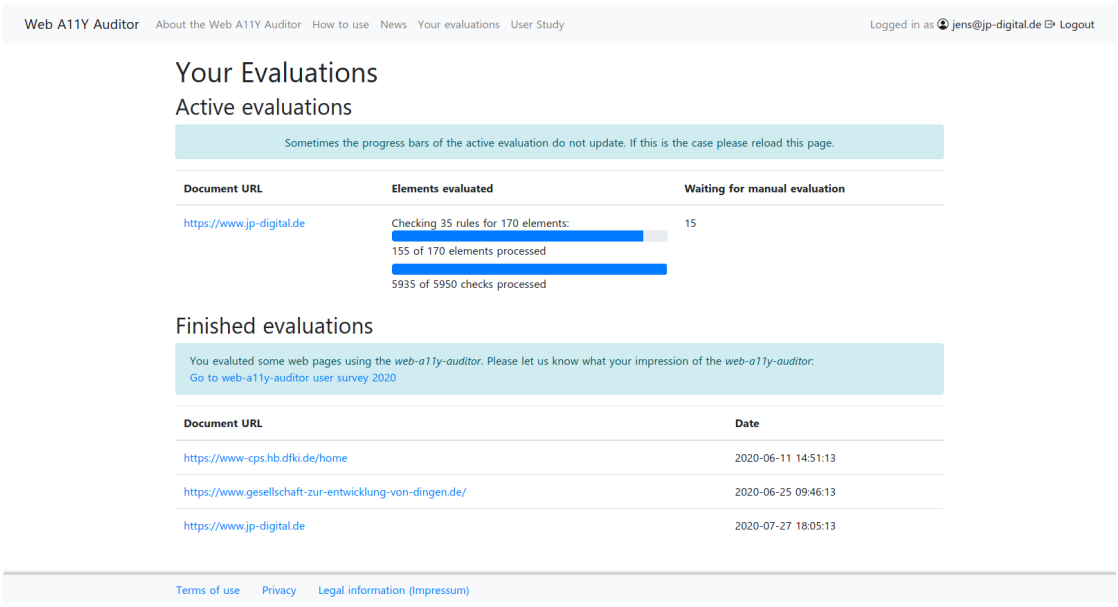

Figure 4.3.: Active evaluation

from the rest of the code, making it easy to add new tests. Code changes have only been necessary when a rule required a test that was not implemented before.

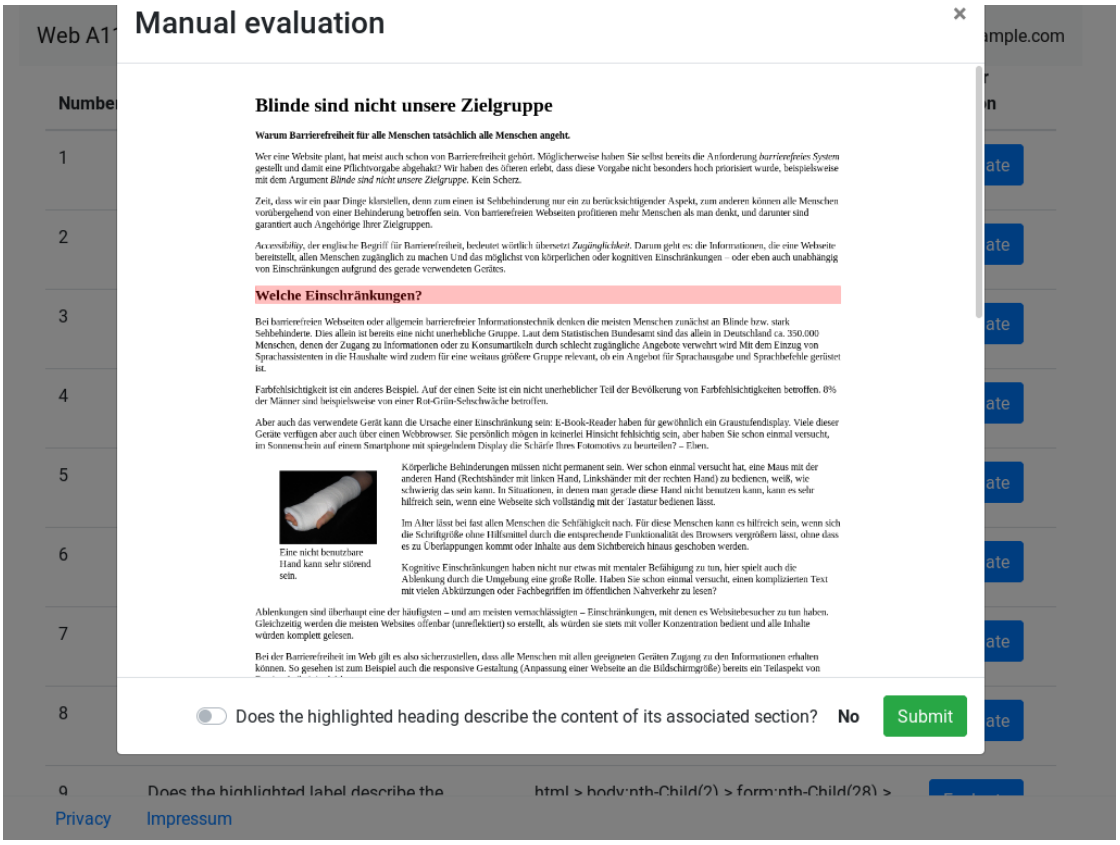

Figure 4.4.: Manual evaluation

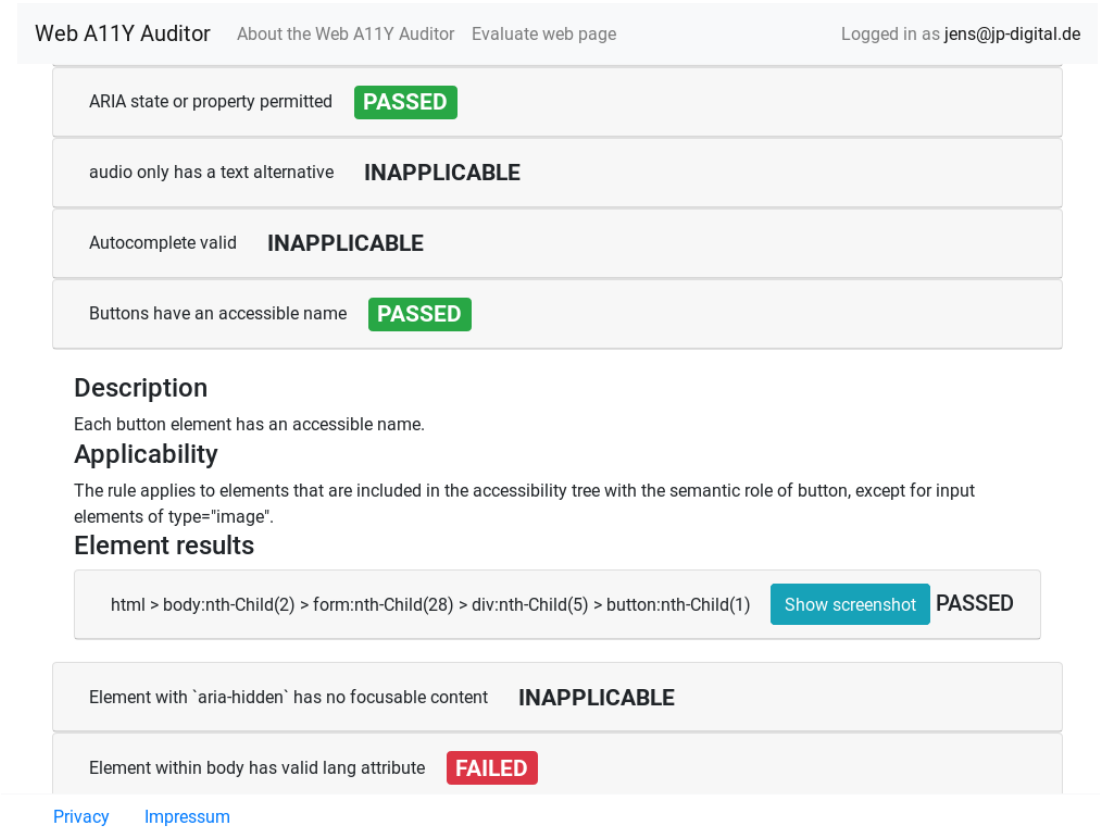

Figure 4.5.: Manual execution of a test.

5. Discussion and Conclusions

5.1. Discussion

The primary contribution of this thesis is the foundation for a declarative model for accessibility requirements, together with an example serialization of the model using OWL ontologies. The basic building blocks of this model, the atomic tests, can be used to create different types of rules. As an example, a machine-readable version of the rules developed by the ACT Rules Community Group has been presented in this thesis. As an example of a possible serialization, an OWL ontology is presented. This serialization uses the atomic test structure to build a machine-readable model of rules written in the ACT Rules Format [19]. This ontology may be used by accessibility checkers as a knowledge base and provides a harmonized model of the requirements for accessible websites.

Using OWL ontologies in an application turned out to be not as simple as expected. The current standard API for working with OWL ontologies, the OWL API, is very low level. For developers familiar with modern APIs like the Java Persistence API, the OWL API feels uncomfortable and cumbersome to use. One problem is that the OWL API requires much boilerplate code, even for simple tasks. For example, retrieving all instances of a specific OWL class from an ontology is a common task. One would expect that this could be done with one or two method calls. Unfortunately, this is not the case with the OWL API. Using the OWL API, this task requires several method calls and the creation of several objects. OWL API Simplex, a set of utility functions and a code generator for the OWL API, has been developed as a side product of this thesis to minimize the boilerplate code. OWL API Simplex is available on Github[1]. OWL API Simplex was very helpful and significantly reduced the time necessary for developing the web-a11y-auditor.

During the development of the example implementation, *web-a11y-auditor* (see chapter 4), the rules published by the ACT Rules Community Group have been updated several times. In most cases, it was only necessary to update the model to include these changes. Code changes have only been necessary if an update included a test not already covered by the model. Besides rules for accessible web pages, the structure of the model could also be useful for testing other aspects of web pages, for example, conformance with design guidelines.

One goal of the ACT Rules Format is to make it easier for developers to create automatic or semiautomatic test tools. From the point of view of the author of this thesis, this goal has been achieved. Problems with the translation of rules into the declarative model described in this thesis, or other issues when implementing tests, were caused by

[1] `https://github.com/jpdigital/owlapi-simplex`

issues in the description of the rules, not by issues with the structure defined by the ACT Rules Format[19]. Often there was already a discussion about these rules on the relevant pages.

At first glance, the approach presented in this paper looks very similar to the approach used by the MAUVE project [36]. However, the approaches differ in many ways. In LGWD, the definitions of the conditions and checks are much more oriented towards a DOM API (at least in the examples shown in the paper). This focus on the DOM API makes it difficult, if not impossible, to add checks that can not be done using the DOM API, for example, checking for keyboard traps. Another difference is the model itself. LGWD is a custom XML language. The model presented in this paper uses an ontology that allows combining the knowledge represented in the ontology with other ontologies.

In the initial concept for the ontology, it was planned to integrate more logic into the ontology in the form of SWRL rules. This approach did only work for very small sets of data. Even small web pages often contain several hundred HTML elements, which lead to a large number of axioms in the ontology. This large number of axioms makes the reasoning process extremely slow. Also, it often requires more memory than available. For example, for a web page with 500 elements – which is not an unusual number – the evaluation process would produce a result for each rule and element. With the 35 rules currently supported by the web-a11y-auditor, this would produce 17500 results for rules. Most applicability definitions also contain more than one test. Therefore the number of test results is even larger. Each of these results would be an individual in the ontology, with several properties. Each of these properties is an axiom in the ontology and has to be processed by the reasoner.

The example implementation, the *web-a11y-auditor* (see chapter 4) is operational and has received some positive feedback from web developers. Due to the COVID-19 pandemic, it was unfortunately not possible to conduct a complete study with users. Of course, such a study could have been done remotely, but for several reasons, it would have been better to do the experiments in a live setting, for example, to react quickly to problems with the software. While the model (the ontology) itself is not that complex, implementing a performant application that uses the model proved to be challenging. Currently, the demo implementation works, but the evaluation process can take several hours for larger web pages. This unsatisfactory performance is caused by two issues. The first issue is in the current implementation. At the moment, the Worker module of the *web-a11y-auditor* executes each test by a separate call of the appropriate methods of the Selenium framework. These invocations take some time because Selenium has to send instructions to the browser and process the results for each invocation. The second issue is the current structure of the ontology. Currently, equivalent tests for different rules are represented by different instances of the test. Some of these tests are used frequently. Some of these tests, for example, the test for checking if an element is included in the accessibility tree, are used frequently. These tests are currently executed several times for each element. Nevertheless, the *web-a11y-auditor* has shown that the declarative model described in this thesis works and can be used as intended for creating tools for checking the accessibility of web pages.

5.2. Conclusions

In this thesis, a declarative model for accessibility requirements for web pages was presented. The foundations of this model are so-called atomic tests, which are small, easy to implement tests. Each of these tests only checks a specific aspect. These tests are combined to formulate rules for testing the accessibility of web pages. Based on the rules developed by the ACT Rules Community Group, several atomic tests have been developed. This approach could be a possible option for creating a machine-readable model for rules in the ACT Rules Format or other similar rules.

A possible serialization using the *Web Ontology Language* (OWL) of the model was also presented. In this serialization, classes are used to model the concept of the model, and individuals are used to describe specific applications of the tests. For this example, the ACT Rules Format [19] and the rules based on this format developed by the ACT Rules Community Group have been used. In addition, the ontology also contains additional classes, properties, and individuals to provide a user with background information, for example, how a rule is related to the success criteria of the *Web Content Accessibility Guidelines (WCAG)*.

An example of the use of the model in an accessibility evaluation tool was presented in chapter 4. This implementation, the *web-a11-auditor*, uses the model as a knowledge base to create the test tasks and uses the background information from the ontology to display the results to the user. The *web-a11y-auditor* worked reasonably well for the purpose of showing a working implementation using the declarative model. Regarding the performance, the current implementation leaves some room for improvements.

5.3. Future Work

In the current implementation (see chapter 4), the evaluation process is still slow. One option for optimizing the evaluation process is to optimize the model, or more exactly, by "normalizing" the ontology. In the current version of the ontology, each usage of a test is modeled using a separate individual. Each test individual is either associated directly or indirectly (by one of the "meta"-tests) with only one applicability definition or expectation. Several tests, for example, `isIncludedInAccessiblityTree`, are used in several rules. At the moment, these tests are repeated for each rule and executed several times for the same element. These tests should only run once to speed up the evaluation process.

Another cause for the improvable performance is that each test is executed in a distinct invocation of the Selenium framework. To improve the performance, all tests that could be executed in one turn should be collected first and executed in a single invocation. A different architecture could be helpful to make it easier to collect the tests for a single invocation. In this revised architecture, the Worker module would take over most of the tasks currently done by the Job Manager module. Instead of individual tests, Selenium would be used to execute collections of tests. For example, all tests for checking the applicability of the available rules could be executed in a single invocation of Selenium.

To make it easier to work with ontologies, *owlapi-simplex* has been developed during this thesis as a side product. *owlapi-simplex* has worked reasonably well and made using ontologies in the *web-a11y-auditor* a lot easier in comparison with the OWL API. To make *owlapi-simplex* production-ready, more testing is necessary, especially with very large ontologies with large numbers (several hundred) of classes and properties.

Currently, several tests require human judgment. Using techniques like Deep Learning or other methods from the field of artificial intelligence, it might be possible to automate more of these tests. One potential candidate is the check if an image is decorative. Another test that should be easy to automatize is checking whether the natural language provided by the `lang` attributes matches the language of the text of the evaluated web page.

In its current version, the web-a11y-auditor can only test static web pages. Modern web pages are often dynamic. The DOM tree of these pages is changed by scripts embedded in the page. Also, web pages are often created dynamically on a server by an application or service. The content of these pages may depend on various variables. Several modern web frameworks that are commonly used to build dynamic web pages provide the option to manage the state of the user interface (the web page) with a central store. In most implementations of such a store, the data in the store is immutable and can only be changed using mutator functions. This pattern can be interpreted as a type of finite-state machine in which the mutator functions are the state transitions.

A similar approach might be useful to evaluate dynamic web pages. Before testing the web page, a test plan consisting of the possible states of a web page and the possible transitions from each state is created. Ideally, this plan should be created automatically. In its current version, the web-a11y-auditor only tests single pages, but not complete sites. The state model approach described for testing dynamic pages might also be useful for testing sites. In this case, the states are the individual pages (identified by a unique URL) of the site. The links for changing to another page are the state transitions.

Web pages that are part of a website often contain repeating components. One possible approach for minimizing the work necessary for evaluating large websites is not to evaluate full pages but the components from which these pages are built. An open question is if the accessibility of components may depend on the context in which the components are used or if the accessibility of a component is independent of its context.

It would be useful to integrate accessibility checks into the workflows of web developers to support them with the creation of accessible web pages and applications. One possible approach for the integration of accessibility checks into the workflow is the integration of accessibility checks into a continuous integration pipeline. One option for this integration could be a tight integration with Selenium. Selenium and other similar frameworks are already used by many projects for automatically testing the user interface. A possible integration of accessibility checks would provide a single method or function that can be called inside a test case. This function checks the accessibility of the current state of the evaluated page. If possible, this function should only check the changed regions of the evaluated page. The results are recorded. For tests that can not be done automatically, the test tool would add a note to the report indicating that a manual test is necessary.

Another option for integrating accessibility checks into the workflow of web develop-

ers is an integration into the developer tools of the browsers. All major browsers have already started to integrate tools for checking the accessibility of web pages into their developer tools, for example, a view for inspecting the accessibility tree or contrast checkers. A possible integration of a future version of the *web-a11y-auditor* may look like the following: A web extension provides a user interface, either integrated into the developer tools or as a sidebar. When a page is reloaded, all tests which can be done inside the browser are executed directly by the web extension. Tests that can not be executed inside the browser are sent to the *web-a11y-auditor* and executed on the server. The Web Extension provides a user interface for inspecting the results. These results would provide a near-live status of the accessibility status of the web page. For manual tests, the extension could notify the developer that a manual test is necessary and provide the developer with instructions about the tests. To make testing dynamic web pages easier, the Web Extension could also monitor the DOM tree for changes and reevaluate the changed areas.

Another option for improving the accessibility of websites is to help authors to create accessible content. Most web content is created by users without a technical background using a Web Content Management System. These users require detailed guidance to create accessible content. A tool using an extended version of the ontology described in this article that is integrated directly into a Content Management System might help the authors to produce accessible content, for example, by evaluating the content on-the-fly while the author is working on the content.

A. Atomic Tests

This chapter provides a reference of all atomic tests available. The tests are shown in a function-like style. The type of parameters is shown as suffix, separated from the name of the parameter by a colon. Possible types are: Another test (shown as `Test` if all tests can be used or as the name of the test if only a specific type of test can be used), `string`, `number`, and `boolean`. Optional parameters are enclosed in square brackets. If the type is followed by a pair of square brackets, the parameter is an array of this type. Some parameters also support a variable number of values, indicated by three dots following the parameter. Some parameters have default values. The default value is separated from the parameter definition by an equal sign.

`accessibleNameIsEquivalentToFilename()`
This test checks if the accessible name of the target element is equivalent to the filename of the resource referenced by the target element. Casing, leading, and trailing whitespaces are ignored. For passing the tesst, only a partial match is required. For example, if the accessible name contains only the last part of the path, the test will pass.

This test is only applicable for elements that reference a file, for example, the `img` element that references an image using the `src` attribute.

`ariaAttributeIsDefined()`
If the target element has any attributes with a name that starts with `aria-` this test checks if the attribute is defined in the ARIA specification [15]. If all applicable attributes are defined in the ARIA specification, the outcome of the test is `PASSED`, otherwise `FAILED`. If the target element has not applicable attributes, the outcome of the test is `PASSED`.

`allOf(test1: Test, test2: Test, test3: Test, ...)`
A meta test that combines the results of other tests. To pass, all of the combined tests must pass. If at least one of the combined tests has the outcome `FAILED`, the test will fail. If one of the combined tests is `UNKNOWN`, the outcome of `allOf` will be `UNKNOWN`. Likewise, if one of the tests has the outcome `WAITING_FOR_MANUAL_EVALUATION`, the outcome will be `WAITING_FOR_MANUAL_EVALUATION`.

`ariaStateOrPropertyIsPermitted()`
Checks if the ARIA states and properties present on the target element are permitted for the target element. The test passes if all ARIA states and properties of the target element are defined in the ARIA specification [15]. If no ARIA states or properties are assigned to the target element, the test should also pass. This test does *not* check if the value of the state or property is valid.

`ariaStateOrPropertyIsValid()`
Checks if the value of an ARIA state or property is valid. The valid values are defined in the ARIA specification [15]. The test passes if the values of ARIA states and properties assigned to the target element are valid. If no ARIA states or properties are assigned to the target element, the test should also pass.

`attributeValueIsNotGreaterThan(attributeName: string, value: number)`
Checks if the value of the attribute given by the `attributeName` parameter is not greater than the value provided by the `value` parameter.

`attributeValueIsSingleTermOrList(attributeName: string)`
A test that checks if the value of the attribute with the name provided by the `attributeName` parameter is either a single term (word) or a space-separated list.

`attributeValueIsUnique(attributeName: string)`
For some attributes, the value of the attribute must be unique for the document in which the value is defined. One example of such an attribute is the `id` attribute. This test checks if the value of the attribute given by the `attributeName` parameter is unique for the document.

`auditoryInformationAvailableAsText()`
Checks if the auditory information from the media provided by the current media element is also available as text on the page. The test is only useful for media elements like `audio` and `video`. There are several options for providing the text: Using a description track on the media element, or by referencing a text bock via a suitable ARIA attribute like `aria-labelledby` or `aria-describedby`.

`containsElement(elementName: string)`
Checks if the target element contains at least one element of the type provided by the data property `element name`.

`containsNoFocusableElement()`
Users who cannot use a pointing device (mouse) operate a web page using the keyboard by stepping through all focusable elements. This test checks if the target element contains *no* focusable elements. If one of the descendants of the target element is focusable, the outcome of the test is `FAILED`.

`containsNoneEmptyText()`
Checks if the target element contains at least one none empty text node. Text nodes only containing whitespace characters (spaces, linebreaks, tabs) are considered empty by this test.

`contentMatchesAccessibleName()`
Checks if the accessible name and the content of the target element match.

`contentProvidesInformation(expectation: string)`
Checks if the content of the target element provides the information described by the `expectation` parameter. This test will usually be evaluated manually.

`elementIsNotEmpty(elementName: string, firstElementOnly: boolean = false)`
Checks if the target element has at least one descendant element of the type specified by `elementName`, which is not empty. If the parameter `firstElementOnly` is set to true, only the first element of the type provided by `elementName` is checked.

`evaluteManually(expectation: string)`
This test can be used to describe expectations that can not be described using other test functions and which have to be evaluated manually. An implementation must replace the following placeholders in the provided text with the correct values:

- `$textContent`: Text content of the target element.
- `$accessibleName`: The accessible name of the target element.

`explicitRoleIsImplicitRole()`
All HTML elements have an implicit ARIA role [18]. This test checks if the role explicitly assigned to the target element using the `role` attribute is the same as the implicit role. If this is the case, the outcome of the test is `PASSED`, otherwise `FAILED`. If the target element has no explicit role, the outcome of the test is `FAILED`.

`hasAccessibleName()`
In parallel to the DOM tree used to create the view of a web page, browsers manage an additional tree structure used by assistive technologies to present the web page to users of such technologies. Elements are represented in this tree by an accessible name[16]. For example, a screen reader uses the accessible name to present an element to the user.

This test checks if the target element has an accessible name. If the target element has an accessible name, the outcome of the test is `PASSED`, otherwise `FAILED`.

```
hasAncestorWithRoleInAccessibilityTree(
    roleName: string[, roleName: string...])
```
Checks if the target element has an ancestor element with one of the roles specified by the parameters. The role can either be the implicit role of the element or an explicitly assigned role.

`hasAriaAttribute([emptyAttributePermitted: boolean = false]`
`[,whitespaceOnlyPermitted: boolean = false])`
Checks if the target element has an ARIA attribute (an attribute which name starts with `aria-`). The parameter `emptyAttributePermitted` controls if empty attributes (attributes without a value) are permitted. If `false` (the default), such attributes will be threaded as not existing. Likewise, the parameter `whitespaceOnlyPermitted` controls if the value of the attribute can be whitespace only. If set to `false` (the default), attributes with a value that is whitespace only are considered as not existing.

`hasAriaState(stateName: string, stateValue: string)`
Checks if the target element has a specific ARIA state. The name of the `state` is provided by the `stateName` parameter, the expected value by the `stateValue` parameter.

`hasAriaStateOrProperty()`
Checks if an ARIA state or property is assigned to the target element.

`hasAttribute(attributeName: string`
`[, emptyAttributePermitted: boolean = false]`
`[, whitespaceOnlyPermitted = false])`
Checks if the attributes with the names provided by the `attributeName` parameter(s) are present on the target element. The optional data property `emptyAttributePermitted` can be used to allow empty attributes. The parameter `whitespaceOnlyPermitted` can be used to control if the attribute can have a value that consists only of whitespace.

`hasChildElement()`
Checks if the target element has any child elements.

`hasDescriptionTrack()`
Checks if an audio or video element contains a description track.

`hasExplicitRole()`
Checks if a role is explicitly assigned to an element using the `role` attribute.

`hasExplicitRoleWithRequiredOwnedElements()`
This test checks if an explicit role has been assigned to the target element, which requires that the target element contains certain other elements.

`hasLabel()`
Checks if the current (media-) element has a label.

`hasPlayButton()`
Checks if the current media element has a play button.

`hasRole(role1, ...)`
Checks if the target element has one of the provided roles, either as an implicit or explicit role.

`hasStatesAndPropertiesRequiredByRole()`
Checks if all ARIA states and properties required by the semantic role of the target element are present.

`hasValidRole()`
Checks if the role assigned to the element is specified in the Accessible Rich Internet Applications recommendation [15].

`hasVisibleTextContent()`
Checks if the target element has any visible text content.

`informationAvailableAsAudio(expectation: string)`
Checks if the information that is described by the data property expectation and provided by the target element is also available as audio.

`informationAvailableAsText(expectation: string)`
Checks if the information that is described by the data property expectation and provided by the target element is also available as text.

`informationAvailableInDescriptionTrack(expectation: string)`
Checks if the information of the media provided by the current media element is available as description track.

`isAssignedToElementWithRole(roleName: string[,roleName: string...])`
Checks if the target element is assigned to an element with a one the semantic roles specified by the `roleName` parameters.

`isAssociatedWithElementWithRole(roleName: string[,roleName: string...])`
This test checks if the target element is somehow associated with an element with a specific role. The following attributes are tested:

- `for`
- `aria-labelledby`
- `aria-describedby`

`isFocusable()`
Users who cannot use a pointing device (mouse) operate a web page using the keyboard by stepping through all focusable elements. This test checks if the target element is focusable and is therefore operable for keyboard users. If the element is focusable, the outcome of the test is `PASSED`, otherwise `FAILED`.

`isIncludedInAccessibilityTree()`
Checks if the target element is part of the accessibility tree.

`isMarkedAsDecorative()`
This test mostly targets images. Images on web pages are often used for decorative purposes. Such elements have to be marked as decorative so that assistive technology can exclude them. There several ways to do that, for example, by setting the alternative name to an empty string (`alt=""`) or by assigning the role `presentation` to the element.

If the target element is marked as decorative, the outcome of this test is `PASSED`, otherwise `FAILED`.

`isMediaElement()`
Audio tracks and videos can be included in HTML by the `audio` and `video` elements. This test checks if the target element is one of these two elements. If the target element is an `audio` or a `video` element, the outcome of this test is `PASSED`. Otherwise, the outcome of the test is FAILED.

This test is primarily used for checking the applicability of rules which check the accessibility of media.

`isOwnedByElementWithRequiredContext()`
Checks if the target element is owned (is a descendant) of an element providing the required context for an element with the semantic role of the target element.

`isPlaying()`
Checks if the current media element is playing.

`isReferencedByAttribute(attributeName: string)`
Checks if the target element is referenced from another element by one of the attributes provided by the `attributeName` data property.

`isStreaming()`
Checks if the current media element provides streaming media.

`isValidAutoFill(attributeName: string)`
Checks if the value of the attribute with the name provided by the data property `attributeName` is a valid autofill value. If the data property is not present, the value of the attribute `autocomplete` is tested.

`isValidLanguage(attributeName: string)`
Checks if the value of the attribute with the name provided by the data property `attributeName` is a valid language tag.

`isVisible()`
Checks if the target element is visible and in the viewport.

`langAttributesAreEqual()`
Checks if the values of the `lang` and the `xml:lang` attribute of an element are the same if both attributes are present.

`matchesCssSelector(selector: string[, exceptions: MatchesCssSelector[]])`
Checks if the target element matches the CSS selector provided by the `selector` parameter. It is possible to provide expectations (other selectors) to exclude elements.

`matchesXpathSelector(selector: string`
`[, exceptions: MatchesXpathSelector[]])`
Checks if the target element matches the XPath selector provided by the parameter `selector`. It is possible to provide expectations (other selectors) to exclude elements.

`negate(test: Test)`
The `negate` meta-test negates the outcome of another test. `PASSED` becomes `FAILED`, `FAILED` becomes `PASSED`. `CANNOT_TELL` and other outcomes are not changed.

`oneOf(test1: Test, test2: Test, test3: Test, ...)`
A meta test that combines the results of other tests. To pass, at least one of the combined tests must pass. If all of the combined tests have the outcome `FAILED`, the test will fail. If the outcome of one of the combined tests is `UNKNOWN` the outcome of `oneOf` will be `UNKNOWN`. Likewise, if one of the tests has the outcome `WAITING_FOR_MANUAL_EVALUATION` the outcome will be `WAITING_FOR_MANUAL_EVALUATION`.

`onlyOwnsElementsWithRequiredRole()`
Checks if the target element only has descendants with the required roles.

`textTransscriptContainsAllInformation()`
Checks if the available text transcript of the media provides all information rovided by the media.

`textTransscriptIsAvailable()`
A text transcript of the media is available.

`videoContainsAudio()`
The video provided by the current video element contains audio.

B. Publications

A list of all publications of the author is available via ORCID: `https://orcid.org/0000-0002-9872-5944`

B.1. Extended Semantic Web Conference (ESWC) 2017

The first paper published about the subject of this thesis was part of the PhD Symposium of the 14 Extended Semantic Web Conference (ESWC) in 2017 [30]. The paper describes the first version of the ontology developed.

A Knowledge-Based Framework for Improving Accessibility of Web Sites

Jens Pelzetter(✉)

FB3 Informatik, Universität Bremen, Bremen, Germany
jens.pelzetter@uni-bremen.de

Abstract. Many sites in the World Wide Web are, unfortunately, not accessible or usable for people with impairments despite several existing guidelines. This paper describes an approach for improving the accessibility of web sites using ontologies as the foundation for several tools. The approach is investigated as a PhD thesis as part of other research that uses ontologies to provide disabled or elderly people with assistance for several everyday tasks.

Keywords: Accessibility · World Wide Web · Ontologies

1 Introduction

For many people the World Wide Web has become their primary source of information. The technologies developed for the Web are used in many other areas. Many organizations have intranets to share informations with their employees. Many applications are provided as Web Applications. These only require a web browser as client.

Developing good web sites or web applications is a quite challenging task. Applications must be usable with a variety of devices (smartphones, tablets and PCs). On top of this, the web sites or web applications should be *accessible*. Accessibility of web sites or web applications is a wide field. Many people only think about blind people when they hear the term *accessibility* in association with web sites or web applications. Accessibility for the web includes much more. Impairments that can affect the way how people may interact with web sites and web applications include problems with all senses and also motoric impairments.

However there are not many tools that support developers to create accessible web sites. Many sites currently available on the web are not accessible and will not become accessible very soon. In this paper, an approach is proposed to use ontologies as the foundation of several tools to improve web accessibility.

The terms *web site* and *web application* will be used interchangeably in this paper since the difference has become very small in the last years. The term *web site* will be used for a collection of web pages, the term *web page* refers to a single page inside a web site.

E. Blomqvist et al. (Eds.): ESWC 2017, Part II, LNCS 10250, pp. 226–235, 2017.
DOI: 10.1007/978-3-319-58451-5_17

2 State of the Art

There are several guidelines for accessible web sites. The most current and widely adopted standard are the *Web Content Accessibility Guidelines 2.0* (WCAG 2.0) [5] created by the W3 Consortium. Most of the other standards for accessible web sites are based on it, for instance the German BITV [22].

Nevertheless many web sites either ignore these standards completely or do not implement them correctly. One reason might be that there are no good, simple to use tools to test a web site for accessibility. All test procedures for accessibility we are aware of either only check some of the very basic requirements or require manual testing.

For the WCAG 2.0, there are several studies which examine, how reliable the results of these tools are [1,3,4]. These studies discovered that the reliability of the results depends on the experience of the tester. Unexperienced testers either find very few or too many problems.

Garrido et al. propose the use of *client side refactorings* to make web pages accessible [8]. Such refactorings use small pieces of JavaScript altering a web pages to make it more accessible. However, the user has to know which refactorings must be applied to a certain web page to make the site accessible for him or her.

The Cloud4all project [18] was a broad attempt funded by the European Union to improve the accessibility of IT technology. In this project an infrastructure was developed that should allow users to store a profile with their preferred settings in the Cloud. Applications can retrieve that profile from the Cloud. The appropriate settings from these profiles are applied to a specific environment using so called *matchmakers*[24]. The primary focus of the project was on traditional, native applications and the usage of native accessibility functions of the operating systems and desktop environments [9]. Besides implementations for Windows and Gnome [2], a proof of concept implementation for web sites was created [19].

Other researchers have tried to use semantic web technologies to enhance the accessibility of web pages and applications. Kouroupetroglou et al. [15] developed a framework which uses annotations in the pages. These annotations can be used by a user agent to provide a better user experience for users of assistive technology. Their research was focused on visually impaired people.

A similar approach is described by Semaan et al. [21], but with a stronger focus on describing the relationship between the several blocks of information on a web page. Their approach was to transform a web page into an RDF document which could then be viewed in the special browser. This special browser uses the additional informations about the document structure to enhance the user experience for users with special needs.

In 2014 the W3C has published ARIA 1.0 [7] (**A**ccessible **R**ich **I**nternet **A**pplications). ARIA uses an approach similar to the approaches described in [15,21]. A web page is annotated with special attributes. The informations provided by these annotations are used by the browser to provide assistive technology with additional information about the elements of a web page.

In other areas, such as mobility assistance, formal modeling approaches have been used with some success [16,20]. Our research group at Universität Bremen and DFKI Bremen has already created a large ontology in OWL-DL [14] describing illnesses, impairments and how they effect abilities such as sight. This ontology describes what of mobility assistance a person with certain impairments requires.

3 Problem Statement and Contributions

Despite the various approaches described in Sect. 2 and the availability of standards like the WCAG 2.0 [5] many web sites are still not accessible. There is also a lack of good tools for checking the accessibility of web sites. All tools which do an automatic check of the accessibility of a web page only check a limited range of requirements. An example for such an tool is the WAVE tool[1]. Test procedures which check a larger range of accessibility requirements require extensive manual work. An example is the BITV-Test[2].

But even if we get good tools for evaluating the accessibility of web pages there will still be many non accessible pages. Therefore it is also necessary to provide tools for disabled users to provide them with a better user experience when accessing non accessible web pages.

The primary research question of this work is whether ontologies can be used to model the knowledge about accessible web pages in a formal way *and* whether they can be used to automatically infer knowledge about accessible web page. One of the possible use cases is a tool which analyses a web page and then uses the knowledge from an ontology about accessibility for web pages to automatically apply refactorings as described in [8].

A common accessibility problem on web pages is an insufficient contrast between the background color and the color of the text. In many cases, this problem could easily be fixed by a client side refactoring. The WCAG 2.0 [5] contains two Success Criteria for contrast. Success Criterion 1.4.3 specifies the minimal requirement for contrast, Success Criterion 1.4.6 specifies an enhanced requirement. Often the only thing necessary to match the requirements and make a web page better readable for people with sight problems is to make the darker color a bit darker and the lighter color a bit lighter.

Another common accessibility problem is that many web sites do not specify their primary language (WCAG 2.0 Success Criterion 3.1.1). This information is needed by screen readers to choose the right pronunciation. A screen reader is a program, which presents the informations normally perceived visually either as speech or as tactile output using a Braille output device. In HTML it is also possible to specify the language of parts of a document by using an attribute (WCAG 2.0 Success Criterion 3.1.2). This information is also useful for screen readers. If the language is provided for a word or part of a web page, which is

[1] WAVE Tool: http://wave.webaim.org/.
[2] BITV-Test http://www.bitvtest.eu/.

not in the primary language of the web page, the screen readers can pronounce this word or part correctly.

A third example is the provision of alternative texts for images. These texts are often missing or applied incorrectly. The alternative text for an image is provided by the `alt` attribute of the `img` Element. For decorative images it is necessary to specify an *empty* `alt` attribute. Otherwise the screen readers use the filename as an alternative text. More details about alternative texts for images on web pages can be found in the description of the HTML element in the HTML5 standard [12] and in the description of technique H67 in [6].

There are several challenges along the way to accessible web pages. The first one is to translate the Web Content Accessibility Guidelines 2.0, a semi-formal specification written in natural language, to a formal description (an ontology).

The WCAG 2.0 consists of several documents. The primary one is written in a technology neutral manner. This document has not been updated since 2008. It describes several Success Criteria for accessible web sites, grouped into guidelines and principles. How these Success Criteria can be implemented is described in separate documents. The document describing the possible techniques [6] to implement the Success Criteria is regularly updated (last updated in October 2016) to include new technologies and other developments.

The W3 provides a tool [23] to connect the Success Criteria and the techniques. The challenging part for modeling the ontology is the connection between the Success Criteria and the techniques (which also describe test procedures). For some Success Criteria, the applicable techniques depend on certain conditions (called situations). For others this is not the case. Some techniques are meta techniques, which can be implemented by several other techniques. For some Success Criteria, two technologies are combined into a new one in the descriptions provided by the tool.

A second challenge lies in the nature of web sites. Web sites are written in HTML. The HTML standard has seen many different versions in the last 25 years, the current one is HTML 5 [12]. For several reasons, some web sites have been written in a very sloppy manner. Even today many web sites are not completely valid when checked with a validation tool for HTML. Users usually don't notice this because the user agents (browsers) have been become very good in making sense of defect HTML documents. Thus there are many invalid HTML documents out there. Analyzing them using formal methods should be quite challenging.

4 Research Methodology and Approach

To achieve the goals described above, the first step was to identify the relevant standards for accessible web sites including literature research about current approaches for test tools and methods for making inaccessible web sites accessible. To learn how users with impairments use the web sites, several afflicted users have been interviewed.

The next step is to create an ontology describing the relevant standards and methods to represent the properties of the web site under test. This also involves

combining the ontology describing the WCAG 2.0 with the existing ontology of impairments and abilities.

Using the ontologies, some tools will be created as "proof of concept" and tested with users and web developers. The tools for web developers will use the ontologies to guide web developers through an accessibility test of a web site. The accessibility test itself will be semi-automatic. Some requirements can be checked without human interaction. Some requirements can not be check automatically, for example if the alternative text for an image is sufficient. For these requirements the test tool will guide the tester through the test procedure using structured questions.

The tools for users will include a browser plugin using the ontologies and automatic test procedures to automatically apply refactorings to web sites, depending on the abilities of the user and the properties of a web site. An example of an accessibility problem that can thus be fixed is insufficient contrast between foreground and background colors (cf. Sect. 3).

There are several different groups of impairments that affect how users can or can not use web sites. The most well-known are of course blindness or the inability to use standard input devices. However, there are many other forms and degrees of impairment that are relevant for accessible web sites, for example color blindness or a reduced field of vision. Moreover, people with cognitive impairments (caused for example by a head injury) might have problems using web sites. A test setup will be developed to evaluate how helpful the tools developed are for users with different kind of impairments. These relationships between the impairments of person how they effect the abilities of person and how they can compensated will be modelled in several interlinked ontologies.

There are several standards, which can be used by web sites to provide a formal description about their content. These include Microformats [17], Microdata [13], and RDFa [11]. If a web site provides such information, it should be possible to use this information to provide some kind of navigation assistance for the web site. This could be very useful for users with cognitive impairments who have difficulty finding information in a complex web site.

The ontologies developed as the foundation of the tools for web accessibility will be used to verify and test the pattern-based ontology tools developed by our research group. One focus of the ontology tools is to support ontology designers with safe maintenance support for ontologies.

5 Preliminary Results and Current Work

It has been more difficult than expected to translate the WCAG 2.0 into an ontology. Several versions of an ontology describing the WCAG 2.0 had to be developed to test different modeling approaches. Some relations between the concepts of the WCAG 2.0 and their instances could not be expressed with OWL 2 DL alone. To express these relations some SWRL rules [10] are used. Current work is focused on the ontology representing the WCAG 2.0 and the supporting documents using OWL 2 as well as a first simple tool.

The ontology describing the WCAG 2.0 has been divided into three parts representing the concepts and relations in these documents. They do not contain any of the Success Criteria, Techniques etc. Due to the amount of data – the *Techniques for WCAG 2.0* document for example contains several hundred techniques – a web scrapping tool has been developed that extracts the data from the web site of the W3C using the jsoup library[3]. The extracted data is used to create the OWL objects representing the Success Criteria etc. using the OWL API[4]. Fortunately this was quite easy thanks to the well-structured HTML format of the relevant documents. In addition to the six ontology documents for the WCAG 2.0, an additional ontology has been created containing some SWRL rules used to infer whether a web page is satisfying a conformance level.

The WCAG 2.0 defines several conformance levels for the accessibility of web pages. To achieve a conformance level, a web page has to meet several success criteria. This relation is represented by the `requiresSuccessCriterion` object property and the inverse object property `requiredByConformanceLevel`.

For each success criterion, several test cases are provided in the supporting documents, grouped into two major categories: *Techniques* describe an approach for meeting a success criterion in a specific situation; *Failures* describe the conditions under which a success criterion cannot be met by a web page.

The ontology contains classes for situations, techniques and failures, and the web pages under evaluation. Success criteria and situations are related by the object properties/inverses `hasSituation`/`isSituationForSuccessCriterion`, a success criterion and a failure by `hasFailure`/`isFailureForSuccessCriterion`, and `hasSufficientTechnique`/`isSufficientTechniqueForSituation` relates, which techniques are sufficient for a specific situation.

To achieve a particular conformance level, a web page must meet all its success criteria. A success criterion is met by a web page, if the web page does not contain any of the failures and meets the requirements for all situations of the success criterion. To meet the requirements of a situation, the web page has to implement at least one of the sufficient techniques for the situation successfully. The requirements for a situation to be met, if the situation is not applicable for a webpage, are also considered. Whether a web page meets a success criterion, contains a failure, etc., is represented by several object properties in the ontology. The foundation are the object properties `successfullyImplementsTechnique`, `containsFailure` and `notContainsFailure`.

To infer that a web page does not match a success criterion, if one of the failures is present on the web page, the following rule is used:

```
WebPage(?p), SuccessCriterion(?sc), Failure(?f),
isFailureForSuccessCriterion(?f, ?sc), containsFailure(?p, ?f)
-> notMeetsSuccessCriterion(?p, ?sc)
```

Whether a web page matches the requirements for a specific situation is inferred using two rules:

[3] jsoup library: https://jsoup.org.
[4] OWLAPI: https://github.com/owlcs/owlapi.

```
WebPage(?p), Situation(?s), notAppliesToSituation(?p, ?s)
-> matchesRequirementsForSituation(?p, ?s)

WebPage(?p), Situation(?s), Technique(?t),
isSufficientTechniqueForSituation(?t, ?s),
successfullyImplementsTechnique(?p, ?t)
-> matchesRequirementsForSituation(?p, ?s)
```

The first rule simply states that a web page matches the requirements for a situation, if the situation is not applicable for the web page, even if the web page does not implement any of the sufficient techniques for the situation. If the web page implements at least one of the sufficient techniques for a situation successfully, the web page matches requirements for that situation.

Now we need to define an rule to infer that a web page meets a success criterion, if the web page matches the requirements for situations of the success criterion and does not contain any of the failures for the success criterion. But due to the Open World Assumption of OWL 2, we cannot simply assert this. Unless stated otherwise, there may be failures or situations that are not described in the ontology. Therefore it is necessary to explicitly assert that there are no other situations or techniques. For this purpose, two classes are added for each success criterion, which contain only the situations and failures for this success criterion.

An example for failures of Success Criterion 1.1.1 in OWL functional syntax is

```
Declaration(Class(wcag20tf:FailureForSuccessCriterion-1-1-1))
EquivalentClasses(wcag20tf:FailureForSuccessCriterion-1-1-1
    ObjectOneOf(<wcag20-techniques#F13> <wcag20-techniques#F20>
                <wcag20-techniques#F72>))
SubClassOf(wcag20tf:FailureForSuccessCriterion-1-1-1
           wcag20-techniques:Failure)
SubClassOf(wcag20tf:FailureForSuccessCriterion-1-1-1
           ObjectHasValue(wcag20tf:isFailureForSuccessCriterion
               <wcag20#successCriterion-1-1-1>))
```

An earlier version of the ontology used cardinality assertions to achieve the same effect, but these made reasoning extremely slow. Using these classes, the following rule states that a web page meets a specific success criterion:

```
WebPage(?p), (matchesRequirementsForSituation
    min 6 SituationForSuccessCriterion-1-1-1)(?p),
(containsFailure max 0 FailureForSuccessCriterion-1-1-1)(?p)
-> meetsSuccessCriterion(?p, successCriterion-1-1-1)
```

This rule uses a class expression with a cardinality requirement. It requires that the web page `?p` have a associated to at least six instances of the class `SituationForSucessCriterion-1-1-1` by the object property `matchesRequirementsForSituation`.

The same approach is used to infer that a web page achieves a conformance level. Which success criteria are required by a conformance level is asserted using a class to infer whether a web page achieves a particular conformance level:

```
WebPage(?p), (meetsSuccessCriterion min 25
    SuccessCriterionForConformanceLevel-A)(?p)
-> compliesToConformanceLevel(?p, ConformanceLevel-A)
```

The next step will be to develop ontologies for the test procedures for the techniques, the refactorings and the requirements of users with impairments.

6 Evaluation Plan

When the first tools are ready to use the tools will be tested with various users. The first group of testers will be users with impairments, who will test the tools that automatically apply refactorings to a web site. The sites used in these tests will be evaluated with the tools developed before for testing web sites for accessibility problems to find out, which accessibility problems they might have. The users will have to execute several tasks, such as finding a specific information, on each site. For these tests the users will be split into two groups. The first group will execute the tasks without the support of the tools developed. The second group will use the tools developed for automatically applying client side refactorings and execute the tasks with the support of these tools. The results of the two groups will be compared to find out whether the tools improve the usability of the web sites for these users. To ensure that both groups are balanced regarding their abilities we will do interviews with each participant before they execute the tasks.

The second group of testers will consist of several web developers, who will use the tools in their daily work. This group will include web developers in larger companies with a solid background in programming and web design, but also developers and designers from small companies, who only occasionally develop web sites and have little programming background. Before the testers will start to use the tools, they will be asked to fill in a questionnaire with some questions estimating their experience in the field of accessibility. After about four to eight weeks, the testers will interviewed about their experience with the tools. The web sites that have been created with the help of the tools developed will be analyzed to investigate whether they have less accessibility problems than average sites.

7 Conclusions

The primary goal of the PhD thesis outlined in this paper is to develop a solid foundation for tools to improve the accessibility of web applications and web sites. This will allow developers to provide tools and better web applications and web sites. Users, especially those who rely on assistive technologies, will

get web sites and web applications that are hopefully more accessible and thus better usable.

Accessibility for web sites and web applications is a complex domain. The lessons learned while developing the ontologies describing the knowledge about this domain will be helpful for other developers, who create ontologies in other similarly complex domains.

Acknowledgments. I would like to thank my supervisor Bernd Krieg-Brückner and my mentor Vojtěch Svátek for their valuable feedback.

References

1. Alonso, F., Fuertes, J.L., González, Á.L., Martínez, L.: On the testability of WCAG 2.0 for beginners. In: Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A). p. 9. ACM (2010)
2. Antúnez, J.H., Clark, C., Markus, K.: The GPII on desktops in PCs OSs: Windows and GNOME. In: Stephanidis, C., Antona, M. (eds.) UAHCI 2014. LNCS, vol. 8516, pp. 390–400. Springer, Cham (2014). doi:10.1007/978-3-319-07509-9_37
3. Brajnik, G., Yesilada, Y., Harper, S.: Testability and validity of WCAG 2.0: the expertise effect. In: Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility, pp. 43–50. ACM (2010)
4. Brajnik, G., Yesilada, Y., Harper, S.: Is accessibility conformance an elusive property? A study of validity and reliability of WCAG 2.0. ACM Trans. Accessible Comput. (TACCESS) **4**(2), 8 (2012)
5. Caldwell, B., Cooper, M., Reid, L.G., Vanderheiden, G.: Web Content Accessibility Guidelines (WCAG) 2.0, December 2008. http://www.w3.org/TR/WCAG20/
6. Cooper, M., Kirkpatrick, A., Connor, J.O.: Techniques for WCAG 2.0 (2016). https://www.w3.org/TR/2016/NOTE-WCAG20-TECHS-20161007/
7. Craig, J., Cooper, M., Pappas, L., Schwerdtfeger, R., Seeman, L.: Accessible rich internet applications (WAI-ARIA) 1.0. Technical report, W3C, March 2014. https://www.w3.org/TR/wai-aria/
8. Garrido, A., Firmenich, S., Rossi, G., Grigera, J.: Personalized web accessibility using client-side refactoring. IEEE Internet Comput. **17**(4), 58–66 (2013)
9. Gemou, M., Bekiaris, E., Vanderheiden, G.: Auto-configuration through cloud: initial case studies for universal and personalised access for all. In: 2013 IST-Africa Conference and Exhibition (IST-Africa), pp. 1–8. IEEE (2013)
10. Harrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML (2004). https://www.w3.org/Submission/SWRL/
11. Hermann, I., Adida, B., Sporny, M., Birbeck, M.: RDFa 1.1 Primer - Third edition. Rich Structured Data Markup for Web Documents (2015)
12. Hickson, I.: HTML5. A vocabulary and associated APIs for HTML and XHTML, May 2011. http://www.w3.org/TR/html5/
13. Hickson, I.: HTML microdata (2013). https://www.w3.org/TR/microdata/
14. Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: OWL 2 Web Ontology Language Primer, December 2012. https://www.w3.org/TR/owl2-primer/

15. Kouroupetroglou, C., Salampasis, M., Manitsaris, A.: A semantic-web based framework for developing applications to improve accessibility in the WWW. In: Proceedings of the 2006 International Cross-disciplinary Workshop on Web Accessibility (W4A): Building the Mobile Web: Rediscovering Accessibility? W4A 2006, NY, USA, pp. 98–108 (2006). doi:10.1145/1133219.1133238
16. Krieg-Brückner, B.: Generic ontology design patterns: qualitatively graded configuration. In: Lehner, F., Fteimi, N. (eds.) KSEM 2016. LNCS (LNAI), vol. 9983, pp. 580–595. Springer, Cham (2016). doi:10.1007/978-3-319-47650-6_46
17. Microformats 2. http://microformats.org/wiki/microformats2
18. Ortega-Moral, M., Peinado, I., Vanderheiden, G.C.: Cloud4all: scope, evolution and challenges. In: Stephanidis, C., Antona, M. (eds.) UAHCI 2014. LNCS, vol. 8516, pp. 421–430. Springer, Cham (2014). doi:10.1007/978-3-319-07509-9_40
19. Peinado, I., Ortega-Moral, M.: Making web pages and applications accessible automatically using browser extensions and apps. In: Stephanidis, C., Antona, M. (eds.) UAHCI 2014. LNCS, vol. 8516, pp. 58–69. Springer, Cham (2014). doi:10.1007/978-3-319-07509-9_6
20. Rink, M., Krieg-Brückner, B.: Wissensbasierte Konfiguration von Mobilitäts-Assistenten. In: VDE e.V. (ed.) Zukunft Lebensräume Kongress 2016 (ZL 2016), pp. 201–206. VDE Verlag, April 2016
21. Semaan, B., Tekli, J., Issa, Y.B., Tekli, G., Chbeir, R.: Toward enhancing web accessibility for blind users through the semantic web. In: 2013 International Conference on Signal-Image Technology Internet-Based Systems, pp. 247–256, December 2013
22. Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz, Barrierefreie Informationstechnik-Verordnung - BITV 2.0 (2002)
23. How to Meet WCAG 2.0. A customizable quick reference to Web Content Accessibility Guidelines (WCAG) 2.0 requirements (success criteria) and techniques. https://www.w3.org/WAI/WCAG20/quickref/
24. Zimmermann, G., Strobbe, C., Stiegler, A., Loitsch, C.: Global public inclusive infrastructure (GPII)-personalisierte benutzerschnittstellen/global public inclusive infrastructure (GPII)-towards personal user interfaces. i-com **13**(3), 29–35 (2014)

B.2. Web for All Conference (W4A) 2018

This paper[31], submitted and accepted for the Doctoral Consortium of the 15th International Cross-Disciplinary Conference on Web Accessibility (W4A) in 2017, describes the possible usage of ontologies as a foundation for Web Accessibility Tools. The version of the ontology described in this paper is an intermediate state between the ontology described in the paper for the ESWC 2017 and the final version.

Using Ontologies as a Foundation for Web Accessibility Tools

Jens Pelzetter
Universität Bremen
Bibliothekstraße 1
28359 Bremen
jens.pelzetter@uni-bremen.de

ABSTRACT

Creating web sites has become quite a complex task. One of most important aspects of a modern web site is accessibility. However, despite extensive standards many web sites have accessibility issues. This paper presents a new approach for creating tools to improve the accessibility of web sites using ontologies.

CCS Concepts

•Information systems → Ontologies; •Human-centered computing → Accessibility systems and tools;

Keywords

Accessibility Web Ontologies

1. INTRODUCTION

Several guidelines exist to make web sites accessible. The most current and widely adopted standards are the *Web Content Accessibility Guidelines 2.0* (WCAG 2.0) [3] created by the W3 Consortium. Most of the other standards for accessible web sites are based on it, for instance the Barrierefreie Informationstechnik-Verordnung (BITV) [11], a German regulation about accessible information technology that is mandatory for all web sites of federal institutions in Germany.

However, most web sites either ignore such standards completely or do not implement them correctly. One reason might be that the available tools to evaluate the accessibility of a web site are not as easy to use as they should be. Test procedures for accessibility either only check part of the requirements or require manual testing. Many aspects can be tested automatically but some require human judgment. An important question in this context is, how rules for accessible web pages can be provided in a machine-readable form.

W4A '18 April 23–25, 2018, Lyon, France

DOI: https://doi.org/10.1145/3192714.3196316

2. RELATED WORK

Research about improving the accessibility of websites is very diverse. Some of the research investigates the evaluation process and how to improve it. Some other research is focused on enriching web pages with additional informations, or to transform web pages into a more accessible form.

For the WCAG 2.0 standard, several studies examine, how reliable the results of accessibility evaluations are [1, 2]. These studies discovered that the reliability of the results depends on the experience of the tester. Unexperienced testers either find very few or too many problems.

Garrido et al. propose the use of *client side refactorings* to make web pages accessible [7]. Such refactorings use small pieces of JavaScript altering a web page to make it more accessible. However, the user has to know, which refactorings should be applied to a certain web page to make the site accessible. Another approach to enhance the accessibility of web sites is to annotate the HTML with additional information. The Accessible Rich Internet Applications (ARIA) standard [5], first published by the W3C in 2014, and updated in December 2017, uses this approach.The information provided by the annotations is made available to assistive technologies by the user agent (browser).

Ontologies have also been used in some research to improve the accessibility of web pages. The ABBA project used ontologies to create a model of a web page, which is used by a special browser which provides extensive navigation assistance for visually impaired users [6]. An accessibility assessment environment that uses ontologies has been described in [12]. The combination of an ontology of the WCAG and disabilities has been described in [9]. The goal of this project was to provide developers not only with guidelines how to make web pages accessible, but also why this is necessary.

3. APPROACH

There are two major problems regarding accessible web sites. The first one is to support web developers with creating accessible web sites. The second problem is the large number of existing web sites that are often not accessible. To address both problems, it is proposed here to create, in particular, a machine-readable version of the WCAG 2.0 standard and its companion documents, potentially also of other accessibility guidelines with a similar approach.

In other areas of assistance for impaired people, formal modeling in the form of ontologies has been used successfully. One example is mobility assistance [10]. Ontology Languages like the Web Ontology Language (OWL) provide a

formal foundation for modeling knowledge. One important difference to databases is that is possible to infer implicit knowledge from an ontology. Ontologies can also contain complex rules for inferring facts. This is used in the approach described here to decouple the rules for conformance satisfaction from the implementation of the tool(s) used to evaluated the accessibility of a web page.

4. PRELIMINARY RESULTS

An ontology has been created that models the WCAG 2.0 specification [3], the Techniques for WCAG 2.0 [4] and the information from the WCAG Quick Reference Tool[1]. To describe how a web page can be tested for conformance with the WCAG, the rules developed by the Auto WCAG Community Group[2] have been integrated into the ontology. An early version of this ontology was presented at the ESWC 2017 conference [8].

The ontology, created using OWL 2, describes the various concepts, e.g. Success Criteria, Conformance Levels and Techniques from the WCAG 2.0. It is planned to update the ontology to describe the WCAG 2.1 when the WCAG 2.1 and the supporting documents are finally published.

A prototype of an evaluation tool using the ontology is currently under development. This evaluation tool is implemented as a web extension that runs side by side in a browser. The current test environment is Firefox, but the Web Extension API is also supported by Chrome and Edge.

Tests that do not require human judgment are run without user intervention. If one of the steps of the test procedure requires human judgment the tool will mark the elements under evaluation and present the user is simple question to evaluate if the element satisfies the test. Test results are added to the ontology. The tool uses the ontology to infer, which conformance levels the evaluated web page satisfies.

5. CONCLUSIONS AND FUTURE WORK

Preliminary results indicate that ontologies can be used to provide a knowledge base about the requirements for accessible web sites. It has still to be verified whether an evaluation tool that uses the ontology is an improvement compared to existing tools. This will be investigated by a series of user tests with several web developers. The test persons will use the tool to evaluate several pages and then be asked to judge the tool.

As a second application, it is planned to combine the WCAG ontology with ontologies describing diseases and impairments. The combined ontology will be used by a web extension (browser plugin) that improves the accessibility of existing websites. This web extension will also have information about the abilities of the user and will use the information in the ontology to apply custom refactorings to a website to make the site better accessible for the user.

The work on the ontology and the software using this ontology is also helpful for improving the methodology for ontology development. The ontology created implicitly contains several design patterns, which could be useful for ontologies in other application domains. These patterns will be investigated in further research.

[1]https://www.w3.org/WAI/WCAG20/quickref/
[2]https://auto-wcag.github.io/auto-wcag/

6. ACKNOWLEDGMENTS

I would like to thank my supervisor Bernd Krieg-Brückner for his valuable feedback.

7. REFERENCES

[1] F. Alonso, J. L. Fuertes, Á. L. González, and L. Martínez. On the testability of WCAG 2.0 for beginners. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, page 9. ACM, 2010.

[2] G. Brajnik, Y. Yesilada, and S. Harper. Testability and validity of WCAG 2.0: the expertise effect. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, pages 43–50. ACM, 2010.

[3] B. Caldwell, M. Cooper, L. G. Reid, and G. Vanderheiden. Web Content Accessibility Guidelines (WCAG) 2.0, 12 2008.

[4] M. Cooper, A. Kirkpatrick, and J. O. Connor. Techniques for WCAG 2.0, 2016.

[5] J. Diggs, J. Diggs, M. Cooper, R. Schwerdtfeger, and J. Craig. Accessible Rich Internet Applications (WAI-ARIA) 1.1. Technical report, W3C, 2017.

[6] R. Fayzrakhmanov, M. Göbel, W. Holzinger, B. Krüpl, and R. Baumgartner. A unified ontology-based web page model for improving accessibility. In *In Proc. of the 19th international conference on World Wide Web (WWW'2010*, pages 1087–1088. ACM, 2010.

[7] A. Garrido, S. Firmenich, G. Rossi, and J. Grigera. Personalized web accessibility using client-side refactoring. *IEEE Internet Computing*, 17(4):58–66, July 2013.

[8] J. Pelzetter. A knowledge-based framework for improving accessibility of web sites. In E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler, and O. Hartig, editors, *The Semantic Web*, pages 226–235, Cham, 2017. Springer International Publishing.

[9] C. Ponsard, P. Beaujeant, and J. Vanderdonckt. Augmenting accessibility guidelines with user ability rationales. In P. Kotzé, G. Marsden, G. Lindgaard, J. Wesson, and M. Winckler, editors, *Human-Computer Interaction – INTERACT 2013*, pages 579–586, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[10] M. Rink and B. Krieg-Brückner. Wissensbasierte Konfiguration von Mobilitäts-Assistenten. In VDE e.V., editor, *Zukunft Lebensräume Kongress 2016 (ZL 2016)*, pages 201–206. VDE Verlag, Apr. 2016.

[11] Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung - BITV), 2002.

[12] K. Votis, R. Lopes, D. Tzovaras, L. Carriço, and S. Likothanassis. A semantic accessibility assessment environment for design and development for the web. In C. Stephanidis, editor, *Universal Access in Human-Computer Interaction. Applications and Services*, pages 803–813, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

B.3. Web for All Conference (W4A) 2020

The paper *A declarative model for accessibility requirements*[32] describes the final structure of the model and the ontology based on the model as well as an early version of the *web-a11y-auditor*. The paper was nominated as best technical paper.

A Declarative Model for Accessibility Requirements

Jens Pelzetter
jens.pelzetter@uni-bremen.de
University of Bremen
Bremen, Bremen

ABSTRACT

The web has become the primary source of information for many people. Many services are provided are on the web. Despite extensive guidelines for the accessibility of web pages, many web sites are not accessible making these web sites difficult or impossible to use for people with disabilities. Evaluating the accessibility of web pages can either be done manually, which is a very laborious task or using automated tools. Unfortunately, the results from different tools are often inconsistent because of the ambiguity of the current guidelines. In this paper, a declarative approach for describing the requirements for accessible web pages is presented. This declarative model will help developers of accessibility evaluation tools to create tools that produce more consistent results and are easier to maintain.

KEYWORDS

accessibility, WCAG, ACT Rules

ACM Reference Format:
Jens Pelzetter. 2020. A Declarative Model for Accessibility Requirements. In *17th Web for All Conference (W4A '20), April 20–21, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3371300.3383339

1 INTRODUCTION

The web has not only become the primary source of information for many people in the last two decades. Many companies sell their products online. In many countries, government services are available on the web. But despite the availability of extensive guidelines for accessible web pages and web applications many web sites are not accessible. On the other hand, a significant number of people have slight impairments or develop slight impairments when they become older. In the next years, the first people who have grown up with the internet will reach an age in which age-related impairments become imminent. Therefore, accessibility will become even more relevant for web pages in the next years.

Accessible web pages can also be helpful for other people, for example for people with temporary impairments. For instance, the ability to use a pointing device (mouse) can be limited due to an injury of the dominant hand. In this case, it can be helpful for a user if a web page can be operated using the keyboard. Environmental conditions like bright sunlight are another example. Under such conditions, a web page with insufficient contrast can become very difficult to read.

Most of the guidelines for accessible web pages are based on the Web Content Accessibility Guidelines [13] published by the W3C, an ISO standard since 2012 [12]. The importance of accessible web pages has also been recognized by many legislative bodies. For example, the European Union issued the European Web Accessibility directive [1]. "Barrierefreie Informationstechnikverordnung" (German Accessible Information Technology Ordinance, BITV 2.0) [19] implements the European Accessible Web Directive as national legislation. Both cite the WCAG. Still, many web pages remain inaccessible or difficult to use for people with impairments.

The manual evaluation of the accessibility of a web site is a time-consuming task that requires good expertise in web technologies *and* accessibility. There are tools alleviating this task. The Web Accessibility Evaluation Tools List[1] published by the Web Accessibility Initiative of the W3C contains 136 entries.

The completeness and implementation approaches of these tools vary significantly. Some tools only check a certain aspect of web accessibility, for example, whether the text on a web page has sufficient contrast. Other tools check a variety of criteria.

Another difference is the user interface: Some tools integrate their user interface into the browser of the user as browser extensions. Some are implemented as stand-alone applications or web services.

Only 28 of these tools are categorized as supporting the most recent version 2.1 of the WCAG. Many tools have not been updated to implement the most recent guidelines and to support the most recent web technologies. One reason for this problem is that the implementation of the guidelines defined by the WCAG and its supplemental documents like the *Techniques for WCAG 2.1* [6] is quite complex. From the user's perspective, these tools are often cumbersome to use. Different tools often produce inconsistent results and still require intensive manual work.

This paper presents a declarative model for describing accessibility requirements for web pages according to the Web Content Accessibility Guidelines by W3C. Rules for checking the accessibility of a web page have been broken down into small tests that can be implemented independently.

The declarative model describing accessibility requirements presented in this paper allows developers to focus on the implementation of tests and on an easy to use interface for the user. Different Tools using this declarative model should produce more consistent results. Adding new rules to a tool using the declarative model or improving existing rules should be much easier since only the declarative model has to be updated instead of changing the code of the tool.

[1] https://www.w3.org/WAI/ER/tools/, retrieved Dec 14 2019

W4A '20, April 20–21, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-7056-1/20/04…$15.00
https://doi.org/10.1145/3371300.3383339

The rest of the paper is structured as follows: In section 2 related work is discussed. Section 3 gives a brief overview of the current accessibility standards. In section 4 the declarative model and its representation using an ontology are discussed. A prototype of a tool using the declarative model is presented in section 5. Section 6 discusses the results and future work.

2 RELATED WORK

The different approaches for evaluating the accessibility of a web page can be categorized into three categories [2, 14]:

- Automatic Testing
- Manual Inspection (by experts)
- User Testing

Automated testing is mostly used [14], but does not always find all existing problems. Testing by experts in the most effective way and user testing works most effectively to verify how people with disabilities perform certain tasks on a web page [2, 14].

One important difference between user testing and expert evaluation is that user testing is more focused on the usability of web pages for users with disabilities while an expert evaluation is more focused on the technical side [2].

Despite their limitations, automated tools play an important role in the process of developing accessible web sites because they significantly reduce the time and effort required to conduct an evaluation [2]. The currently available automated tools can produce different results, false negatives or false positives because of different implementations of the guidelines. To avoid missing problems multiple different tools are used for an accessibility evaluation [2].

Automated tools can not check all requirements. For example, to check if a heading is sufficient for its associated section human judgment is required. The effectiveness of automated tools varies depending on the number of tests implemented. Another factor that affects the effectiveness of a tool is the ability of the developers of the tool to translate the guidelines for accessible web pages which are expressed in natural language into a computational representation [2].

To make comparing the results from different tools easier the Evaluation and Report Language (EARL)[3, 18] has been proposed. Unfortunately, EARL has not reached the status of a W3C Technical Recommendation yet.

Several tools for user testing use crowdsourcing. These tools have two different approaches. Some of them are trying to improve the accessibility of a web page by adding metadata. The process of annotating a web page with such metadata is likely to be very time-consuming; crowd-based tools allow the distribution among many authors. Other crowd-based tools split the accessibility evaluation into small tasks for distribution, making the evaluation less expensive [2]. The effectiveness of crowdsourcing-based tools has not yet been demonstrated.

Manual inspection by experts also raises problematic issues. Even experts do not find all accessibility problems. In some cases, a manual inspection may also produce false positives (problems that do not exist). In a study based on the WCAG 2.0 by Brajnik et al. in 2012 [5], experts and novice evaluators evaluated several web pages for accessibility problems. Experts were only correct in 76% of all cases. This rate dropped by about 10% for novice evaluators. Expert users produced 26% to 35% false positives and missed 26% to 35% of the real problems. Novice evaluators without much experience produced much more false positives than experienced evaluators and found less real problems than experienced experts. Due to a large variety in the results of novice evaluators, no conclusions for that group were drawn in the study [4, 5].

It has been suggested to develop unambiguous, machine-readable specifications to facilitate a better application of the accessibility guidelines and to produce more uniform results, and to develop tools, which seamlessly integrate into the development process of web pages, to increase the adoption of accessibility guidelines [2].

An XML based language for specifying accessibility guidelines, the *Language for Web Guideline Definition* (LWGD), has been proposed together with an environment called *MAUVE* for evaluating accessibility [15]. The last paper about MAUVE is from 2015. Judging from the web site[2], the application has been updated to match the newest version of the WCAG. The checks in MAUVE are based on the techniques for implementing the WCAG [6].

The validation process of MAUVE is based on the DOM tree of the document. MAUVE downloads the web page to evaluate and creates the DOM tree itself. The validator module interprets the guidelines formalized in the XML language to checks if the DOM tree passed the checks defined in XML. The LWGD language allows it to define the element to check and several conditions to validate. The conditions can be combined using boolean operators.

The effectiveness of MAUVE was compared with the Total Validator[3], a commercial product. In comparison, MAUVE missed fewer problems than the Total Validator. For false positives, MAUVE reported more false positives than the Total Validator in some cases; in other cases, the Total Validator produced more. The paper about MAUVE [15] shows an example with two conditions: One to check whether an element is followed by another element, and one to check whether an element has a specific child element. The paper does not list all available conditions.

3 ACCESSIBILITY STANDARDS

The primary (technical) guidelines for creating accessible web pages are the Web Content Accessibility Guidelines (WCAG) [13] published by the W3C. The WCAG are a technology-neutral description of the requirements for accessible web pages. Possible techniques for implementing the WCAG are in *Techniques for WCAG 2.1* [6], published separately from the WCAG. The *Techniques for WCAG* also describe several common failures, i.e. common bad practices in web development, which make web pages and web applications difficult to use for people with impairments.

Each description of a technique or failure in the *Techniques for WCAG* document contains a test procedure to check whether the technique has been successfully implemented or not. For each failure, a test procedure is provided for checking that the failure is not present on a web page.

Neither the Web Content Accessibility Guidelines nor the Techniques for WCAG defines, which technique can be used to satisfy a success criterion of the WCAG or which failures cause a web page to fail a success criterion. For this purpose, the W3C provides

[2] https://mauve.isti.cnr.it
[3] https://www.totalvalidator.com

an interactive web page[4]. The web page shows, for each success criterion, which techniques are sufficient for implementing the requirements, and lists relevant failures. For some success criteria, different techniques can be sufficient depending on the characteristics of the document. For example, the sufficient techniques for success criterion *1.4.3 Contrast (Minimum)* depend on the font size of the text. Some success criteria can only be satisfied by the combination of two techniques.

Understanding these requirements requires time, a good understanding of web technologies and accessibility requirements, and careful reading. To make the technical requirements for accessible web pages easier to understand and less ambiguous, and to harmonize the interpretation of the requirements defined by the WCAG, the W3C has published a new recommendation, the Accessibility Conformance Testing (ACT) Rules Format [10]. This recommendation defines a structure for writing rules to test accessibility.

Two types of ACT Rules have been defined. Atomic rules define a specific requirement, composite rules combine several other rules. Each ACT Rule consists of several sections. Both types of rules contain a unique ID, a description, a mapping to one or more success criteria of the WCAG, assumptions about the evaluated web page or the applicable elements, possible limitations of assistive technology relevant for the rule, and test cases to check the implementation of a rule.

Moreover, atomic rules list the input aspects relevant for the rule, for example, the DOM Tree or CSS Styling. The *Applicability* section of an atomic rule describes for which elements a rule is applicable. Each atomic rule defines at least one expectation that must be met by applicable elements to pass the rule. If a rule has multiple expectations each applicable element must pass all expectations. Composite rules may also have an Applicability section. The Expectation section of a composite rule lists all rules which are combined by the rule and defines whether all or at least one the combined rules have to pass. For evaluating the accessibility of a web page the *Applicability* definition and *Expectations* are the most important sections.

A community group has already created several rules using this format[5].

For example, the rule *Button has an accessible name* [6] describes how to check whether a button has an accessible name. Buttons are used frequently in modern web design for all kinds of interactions like opening a menu. To be usable with assistive technology like screen readers a button must have an accessible name that is provided to the screen reader by the browser. The applicability section describes how to find all elements for which this rule is applicable:

> The rule applies to elements that are included in the accessibility tree with the semantic role of *button*, except for *input* elements of *type="image"*.

To be applicable for the rule, elements must match several conditions:

- The element must be included in the accessibility tree. Apart from the DOM tree which is used to create the visual output of an HTML document a browser also manages a second tree of elements, the accessibility tree. This tree is provided to assistive technologies by the browser using the accessibility API of the operating system. The DOM tree and the accessibility tree are not the same. An element that is present in the DOM tree may not be part of the accessibility tree.
- The second requirement is that an element has the role of a button. The term *role* originates from the ARIA [7] recommendation. Roles are used in ARIA (among other extensions to HTML) to provide the accessibility API with more information about the semantics of an HTML document. Many HTML elements have implicit roles [9]. For example, the role `button` is implicitly assigned to the HTML elements for creating buttons (`<input type="button">` and `button`). It is also possible to create a widget that looks like a button using other HTML elements like a `div` container. For this widget, the role `button` must be provided explicitly by the author. In both cases the rule *Button has accessible name* is applicable.
- The third requirement is that the element is not an image button. With `<input type="image">` it is possible to use an image as a button. This type of buttons is excluded from this rule because image buttons are checked by other rules.

The rule has a single expectation:

> Each target element has an accessible name that is not empty ("").

The expectation states that each applicable element must have an accessible name and that the accessible name cannot be an empty string. The accessible name is used by assistive technology like screen readers to disclose the button to the user. To be useful, the accessible name should contain a brief description of the purpose of the button. The algorithm that should be used by browsers to compute the accessible name of an element is described in a technical recommendation [8] published by the W3C.

4 THE DECLARATIVE MODEL

4.1 General structure

With the exception of MAUVE all other tools for checking web pages for accessibility problems known to the author implement tests for checking the requirements for accessible web pages directly as code. This makes maintenance or customization of tools difficult. The approach described in this paper uses a declarative model of the accessibility rules to describe *what* to check, not *how* to perform tests.

The key components of the declarative model are clearly specified atomic tests that can be combined to build complex rules. The implementation of the tests is the responsibility of the developers of the tools that use the declarative model.

Using this declarative model to implement evaluation tools has several advantages. Developers can focus on a reliable implementation of the tests and an easy-to-use user interface.

Different tools may implement the tests in different ways. A tool implemented as a browser extension may use the APIs provided by the browser for analyzing the evaluated document. Another tool with a stand-alone implementation may use a browser automation framework such as Selenium for accessibility evaluation.

[4] https://www.w3.org/WAI/WCAG21/quickref
[5] https://act-rules.github.io/pages/about
[6] https://act-rules.github.io/rules/97a4e1 accessed 2019-12-07

The results produced by the tools may still differ in detail, but this can only be caused by the limitations of the implementation of the tests and not by different interpretations of the rules. Since the code of the implementation of each check is only small, independent units of code can be tested much better than larger complex pieces.

Another advantage is that changes to rules or new rules do not require any code changes (if all required tests are already implemented). Only the declarative model has to be updated.

The ACT Rules developed by the ACT Rule Community Group have been chosen as a starting point. They were chosen because they provide the best available (if incomplete) summary of the requirements for accessible web pages. The ACT Rules also contain fewer special cases than the description of sufficient technologies provided by the WCAG QuickRef. Therefore, the assumption was, that the ACT Rules are easier to model. For the ACT Rules developed by the ACT Rules community, there are also test cases available for each rule, which allows checking the implementation of a rule.

The first step in developing the model was the definition of the atomic tests which are needed to build a model of the ACT Rules. In the current version, the ontology contains 48 tests that are used to build a declarative, machine-readable model the applicability definition and expectations of ACT Rules. Some of these tests require additional parameters. These parameters are also described in the model.

The tests can be grouped into two categories: Element Filters and Expectation Tests. Element Filters are used in the applicability definitions to describe for which elements the rule is applicable. Expectation Tests are used to model the expectations of the rules which must be matched by the applicable elements to be considered accessible.

An ACT Rule may have several *outcomes*. The outcome `Passed` indicates that an element has passed a rule, the outcome `Failed` indicates that an element has failed a rule. If one applicable element fails a rule, the complete document fails the rule. If no elements to which a rule is applicable are found, the outcome of the rule is `Inapplicable`. An ACT Rule tested by an automatic tool may also have the outcome `cannot tell` if one of the tests of the rule cannot be done automatically.

4.2 Examples for Tests

The following examples for checks are shown in a function-like notation. The checked element is not shown as a parameter but has, of course, to be provided to an implementation.

`matchesCssSelector(selector)`

Checks if an element matches the specified CSS selector. This test is used in many applicability definitions to find the elements for which the rule is applicable. Usually, this test is combined with other tests to find the elements for which an ACT Rule is applicable.

`hasRole(roleName, ...)`

This test checks if an element has a specific semantic role (cf. [7]), either implicitly or explicitly assigned. Most HTML elements have an implicit role. The author of an HTML document can change this role using the `role` attribute.

This test is often used to filter out form controls or buttons. Several roles may be passed as parameters to the test. The test passes if at least one of the provided roles is assigned to the tested element.

`hasExplicitRole()`

This test checks if a role was explicitly assigned to an element, for example, the role `button` to a `div` element used as a button.

`hasAriaAttribute()`

This test checks if any ARIA attributes are assigned to an element. This test is usually used in rules that check the correct use of ARIA to find elements annotated with ARIA attributes.

`hasAccessibleName()`

Using the algorithm for computing the accessible name of an element(see [8]) this test checks if an element has an accessible name. The accessible name is important for assistive technology like screen readers. For example, a screen reader can not present a button without an accessible name properly.

The implementation of these tests may vary. Some require access to the DOM tree and are only checking the presence or absence of attributes or elements. A tool implementing the tests should not use the HTML document received from the server directly. On modern web pages, the initial HTML code is often altered by scripts that are running when the browser has loaded the page. If an HTML document is syntactically incorrect browsers try to fix the errors by altering the DOM tree. Therefore the DOM tree generated by the browser should be used for accessibility evaluations and not the raw source code of the document.

4.3 Combination of Tests

For most applicability definitions and expectations, it is necessary to combine several tests. For some applicability definitions and expectations, some form of negation is required, for example, to express that an element should not have a specific role.

The model provides three options for this purpose. `negate` is used to express negation, `allOf` and `oneOf` are used to combine the outcomes of multiple tests. The names `negate`, `allOf` and `oneOf` were chosen instead of the simpler names `no`, `and` and `or` to avoid conflicts with ontology languages or programming languages in which *not*, *and* and *or* are often reserved words.

An implementation of `negate` should change the outcome of the associated test from `passed` to `failed` and vice verse. Other outcomes like `cannot tell` should not be changed by the implementation of `Negate`.

Two options are available for combining multiple tests: `allOf` requires that all combined tests pass, `oneOf` requires only one of the combined tests to pass. An implementation also has to handle special cases, for example, whether one test has the outcome `cannot tell`. In this case, the implementation of `allOf` as well as of `oneOf` should return the outcome `cannot tell`.

4.4 Examples for ACT Rules in the model

The following examples of ACT Rules are given in the same notation as the examples of individual tests.

4.4.1 Button has an accessible name. The rule *Button has an accessible name*[7] checks if a button has an accessible name. The applicability definition for this rule is:

> The rule applies to elements that are included in the accessibility tree with the semantic role of *button*, except for *input* elements of *type="image"*.

This applicability definition can be broken down into three tests:

- Checking whether the element is included in the accessibility tree.
- Checking whether the element has the semantic role of `button`.
- Checking whether the element is an image button. This can be done using the CSS selector `input[type=image]`.

Using the declarative model described here, the applicability definition of this rule can be expressed as:

```
allOf(
    isIncludedInAccessibilityTree(),
    hasRole("button"),
    not(matchesCssSelector("input[type=image]"))
)
```

The applicability definition requires that applicable elements are *not* image buttons. Therefore, the outcome of the test for the CSS selector is negated. The accessibility definition also requires that all requirements are met by applicable elements. Therefore, `allOf` is used to combine the tests.

The expectation of this rule requires that every button has an accessible name:

> Each target element has an accessible name that is not empty ("").

To check whether an accessible name is available for the element, only one test is necessary:

```
hasAccessibleName()
```

4.4.2 Image has accessible name. Images can be used in different ways on a web page. One purpose is decoration, the other is to support the textual content. If an image is not used as decoration it needs an accessible name that describes the content of the image. Decorative images have to be marked correctly using an empty `alt` attribute. The rule *Image has accessible name*[8] checks whether an image is either marked as decorative or has an accessible name.

The applicability definition of the rule is:

> The rule applies to HTML `img` elements or any HTML element with the semantic role of `img` that is included in the accessibility tree.

This applicability definition can be broken down into three tests:

- The element has the semantic role of `img`. Sometimes the equivalent role `image` is used for images. This role is also provided as a parameter for the `hasRole` test.
- All `img` elements are applicable, regardless of the role assigned to them.
- An applicable element is included in the accessibility tree.

These tests can be expressed in their combination as

```
allOf(
    oneOf(
        hasRole("img", "image"),
        matchesCssSelector("img")
    ),
    isIncludedInAccessibilityTree()
)
```

The tests for the role and the CSS selector are combined with `oneOf` to express that only one of these tests has to pass. The result of `isIncludedInAccessibilityTree` is combined with the result of `oneOf` to express that an element for which this rule is applicable has the be included in the accessibility tree.

The expectation of the rule is:

> Each target element has an accessible name that is not empty ("), or is marked as decorative.

The expectation can be broken down into two tests:

- Checking whether the image has an accessible name
- Checking whether the image is marked as decorative

This can be expressed as

```
oneOf(
    hasAccessibleName()
    isDecorative()
)
```

An applicable element has only to pass one of the two tests. Therefore, both tests are combined using `oneOf`.

4.5 ACT Rules as an Ontology

There are several options for creating a machine-readable representation of models like the one described in this paper. One possible option is the development of a custom XML language or a JSON data model. Another option is to use linked data (RDF) or an ontology. For the model described here an ontology was used. There are several different ontology languages. One of the most common ones is the Web Ontology Language (OWL)[11], which as used for the ontology described here.

The ontology[9] contains the tests described in the previous sections, the ACT Rules (as of November 2019) as well as the Success Criteria of the WCGA 2.1.00

Using an ontology has several advantages compared to a custom XML language such as LGWD used by the MAUVE project (see section 2). Ontologies have well-defined semantics which allows it to validate their consistency. Existing ontologies can easily be reused. Ontologies providing knowledge for different domains can be combined to a larger ontology It is also possible to define complex rules in an ontology that can be used by a reasoner to infer knowledge based on the data in the ontology. A very early version of the ontology and the software used this approach to infer the results of an evaluation. Unfortunately, this approach did not scale well (see section 6).

The tests described in section 4.1 are modeled as classes. Individual applications of the tests for a rule are modeled as individuals together with the required parameters. The parameters of the tests are provided using data properties. The UML diagram in figure 1 shows the primary classes and properties of this ontology.

[7] https://act-rules.github.io/rules/97a4e1
[8] https://act-rules.github.io/rules/23a2a8
[9] https://ontology.web-a11y-auditor.net

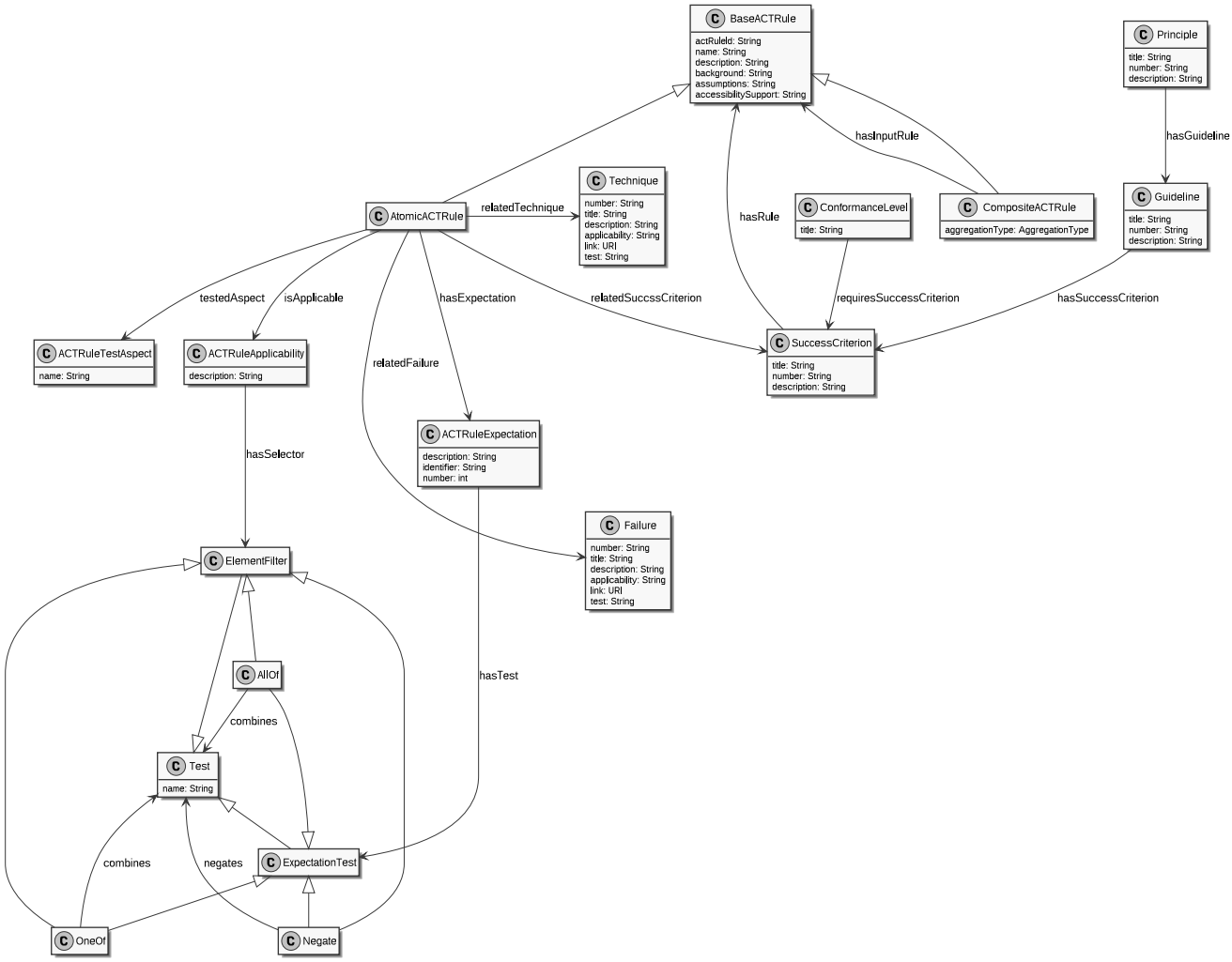

Figure 1: Primary classes in the ontology

Rules are represented as individuals of one of these two classes `AtomicACTRule` or `CompositeACTRule`. Both classes are subclasses of the `BaseACTRule` class which contains the properties shared by atomic and composite rules: for example the rule ID which provides a unique identifier for the rule, the name of the rule and the description of the rule. These data can be used by an application using the ontology to display the rule to a user.

The ontology also contains classes and properties for describing the success criteria, guidelines, principles, and conformance levels of the WCAG, and user with background information about rules.

The applicability definitions and expectations are modeled using separate classes. In addition to the property for associating the test(s) for the applicability definition or the expectation, the `Applicability` and `Expectation` classes also provide a property for the textual description of the applicability definition or the expectation. These data can be used by a tool using the ontology to display information about the rule to a user.

The `Expectation` class has two properties which are not part of the ACT Rules format: The `number` property is used for ordering the expectations of a rule. The `identifier` property provides a unique identifier for the expectation.

The tests are modeled using subclasses of the `Test` class. For each of the atomic tests found in ACT Rules, the ontology contains a subclass of the `Test` class. For brevity, these classes are not shown in figure 1. The `hasSelector` property is used to associate an individual of the `Applicability` class with an individual of the test class `ElementFilter`. For expectations, the association between the individual of the `Expectation` class and the `ExpectationTest` class is expressed using the `hasTest` property.

The operators for combining the results of other tests (`allOf`, `oneOf`, and `negate`) are also modeled as subclasses of the `Test` class. This allows using these operators in the same places as atomic tests. The associations between an individual of the classes `AllOf` or `OneOf` and the combined tests are expressed using the `combines`

property. For the association between an individual of the `Negate` class and the negation test the `negates` property is used.

In both cases an individual of the `Applicability` or the `Expectation` class can only be associated with one individual of the `Test` class. This can either be a real test or one of the combining tests `oneOf` or `allOf`.

Composite rules do not have an applicability section or expectations. Instead, they combine the outcomes of multiple rules into a single result. A composite rule either requires that all combined rules pass or that at least one of the combined rules passes.

Composite rules are represented using the `CompositeRule` class in the ontology. The aggregation type is represented using the `aggregationType` property, the input rules are associated with a composite rule using the `hasInputRule` property.

Figure 2 shows how the rule *Button has accessible name* is represented in the ontology. The rule itself is represented by an individual of the `AtomicACTRule` class. Applicability definition and expectation are modeled using individuals of the classes `Applicability` and `Expectation`. The tests are represented by individuals of the appropriate subclasses of the `Test` class. In this example„ these are the classes `IsIncludedInAccessibilityTree`, `HasRole`, `MatchesCssSelector` and `HasAccessibleName`.

The individual for the `HasRole` test has an additional data property providing the role to check for. Likewise, the `MatchesCssSelector` test used for image buttons has an additional property providing the CSS selector to check for. The tests without parameters d not have any additional properties.

An individual of the `Applicability` or the `Expectation` class can only be associated with one individual of the `Test` class. If an applicability definition or an expectation is consists of multiple tests these tests have to be combined using an individual of the `oneOf` or `allOf` classes. In the example, the applicability definition contains three tests which are combined using `AllOf`. The test `MatchesCssSelector` is also negated to exclude all image buttons.

5 PROTOTYPE IMPLEMENTATION

A prototype of an application that uses the ontology described in section 4, the *web-a11y-auditor* [10], has been implemented. This application consists of several modules: The Web Application, which also provides a RESTful API, a worker application, responsible for executing the tests, and a browser extension for Firefox providing a user interface for the web-a11y-auditor embedded into the developer tools of Firefox. The browser extension uses the RESTful API provided by the web application for interacting with the application. Figure 3 shows an overview of the architecture of the web-a11y-auditor.

The workers only execute the test tasks. The combination of the results to the result of a rule is done by the web application when the results are accessed. The tests to execute - each element and test a task is generated - can be processed by multiple worker instances. The worker application is also multi-threaded. Each instance of the worker application can process several tasks simultaneously.

A user uses either the Web application or the Browser extension for starting the evaluation of a web page by providing the URL to evaluate. The evaluation is done in five steps:

- Analyze the document and store a screenshot of the document and the CSS selectors and coordinates of each element. The screenshot and the element coordinates are used by the web application to show which element is related to a result.
- Generate the tasks for checking which rules are applicable for which element of the evaluated document.
- Check which rules are applicable for which element
- Generate the tasks for checking the expectation for the applicable rules.
- Check if the elements match the expectations of the applicable rules.

For each of the atomic tests to execute a task is created and stored in the database. The ontology provides the template for these tasks. One of the Worker instances retrieves the task, executes it and stores the result in the database. Aside from the specification of the tests, all other data provided by the ontology are only used by the web application.

The execute the tests the web-a11y-auditor uses the Selenium framework[11]. Selenium uses the Web Driver API [16, 17] to control a browser instance. The tests are executed in this browser instance using Selenium. This approach has some advantages compared with APIs like JSoup[12] which are implementing only a DOM parser. DOM parsers like JSoup only implement parts of a browser rendering engine. For example, elements generated using JavaScript could not be checked using a DOM parser which does not execute JavaScript. For some tests, for example, checking if a web page is operable using the keyboard can not be done using a DOM parser. Selenium allows it to simulate all kinds of user interactions with a web page and check the results.

Some tests cannot be executed automatically yet. For these tests, the web-a11y-auditor guides the user through the tests. If a test can not be executed automatically the test is added to a list of tasks requiring manual evaluation. The user can choose any of these tasks from the list. The tests are presented to the user in the form of a simple question like

> Does the highlighted heading describe the content of its associated section?

The questions shown in this dialog are yes/no questions. The answer corresponds with one of the outcomes `passed` and `failed` (please note: the web application currently uses a switch control for selecting the result).

In the web application, the dialog contains a screenshot of the web page. The element under evaluation is highlighted in the screenshot. In the browser extension, the element is highlighted directly in the browser window. Figure 5 shows an example of the dialog in the web application. When all manual evaluations are done the results of the evaluation are presented to the user.

A user can either use the web application or the browser extension for interacting with the application. Both variants provide the same options to the user. Figure 4 shows a screenshot of the results of a finished evaluation.

[10] https://www.web-a11y-auditor.net

[11] https://selenium.dev/

[12] https://jsoup.org/

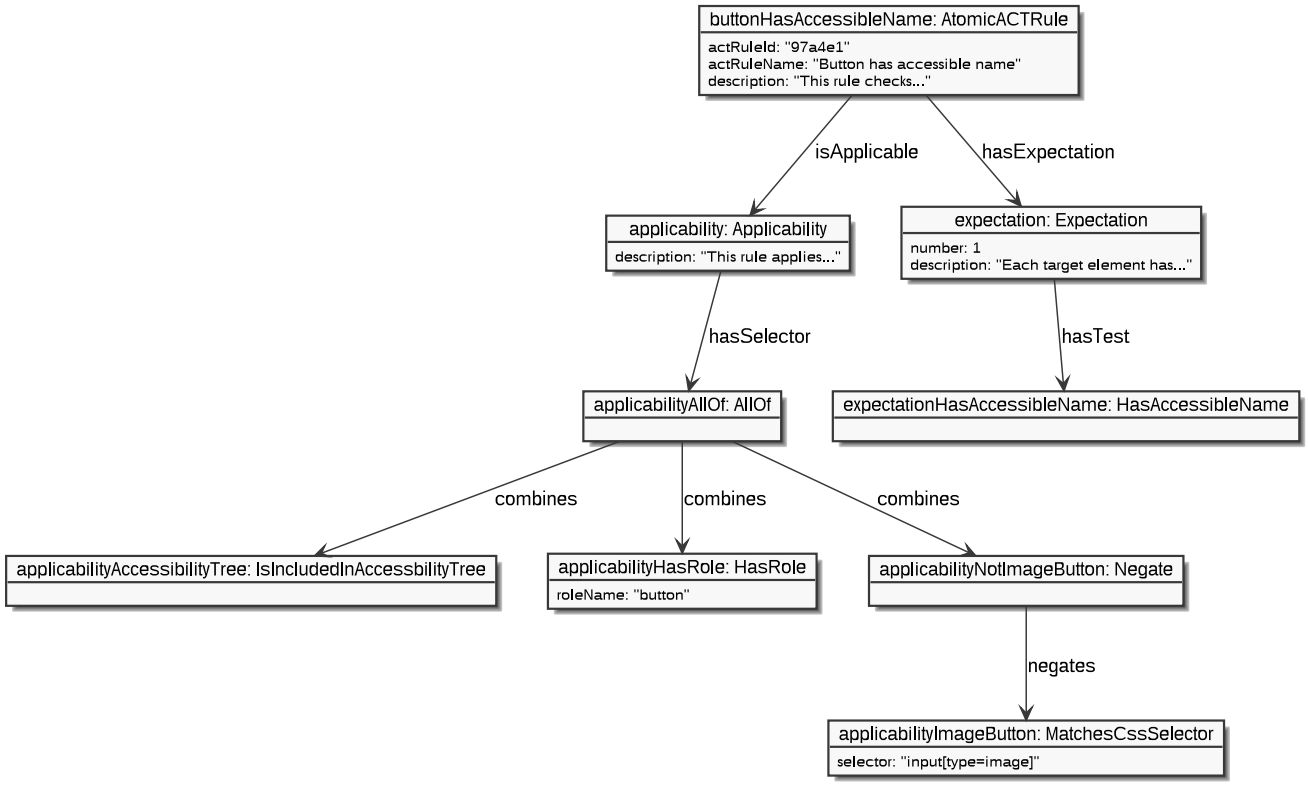

Figure 2: Representation of a rule in the ontology

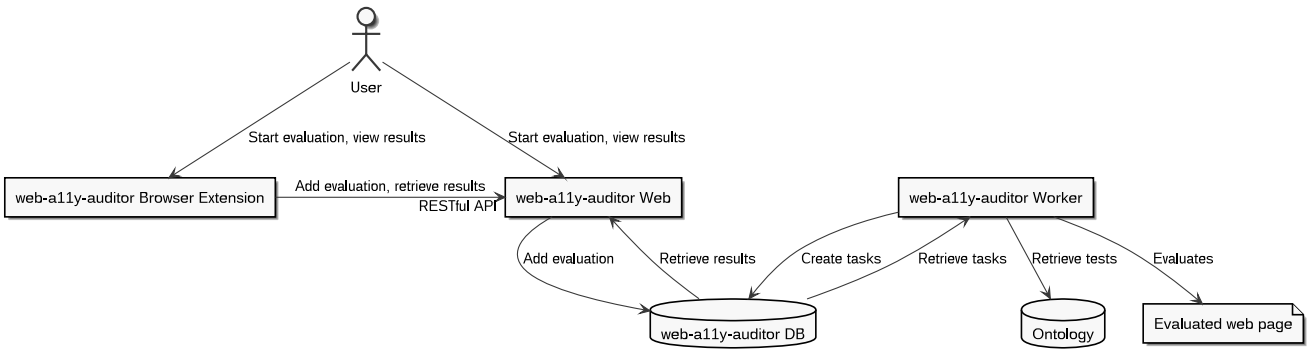

Figure 3: Overview of the architecture of the web-a11y-auditor

6 CONCLUSIONS AND FUTURE WORK

In this paper, an ontology providing a declarative model for accessibility requirements of web pages has been presented. The current ontology is based on the structure defined in the ACT Rules technical recommendation [10]. This ontology may be used by accessibility checkers as a knowledge base and provides a harmonized model of the requirements for accessible web sites. Moreover, this model is easy to extend.

The easy extensibility was tested during the development of the prototype application. During the development of the prototype described in section 5, the ACT Rules published by the ACT Rules Community Group have been updated several times. These changes included the addition of new rules, the removal of some rules as well as changes to the applicability definitions and expectations of some rules. For some rules, requirements were added to the applicability definitions and expectations or removed from them.

As expected no changes in the code of the prototype were necessary to integrate the updated rules into the prototype. Only the ontology was edited to match the updated rules. After replacing the ontology the web-a11y-auditor used the updated rules.

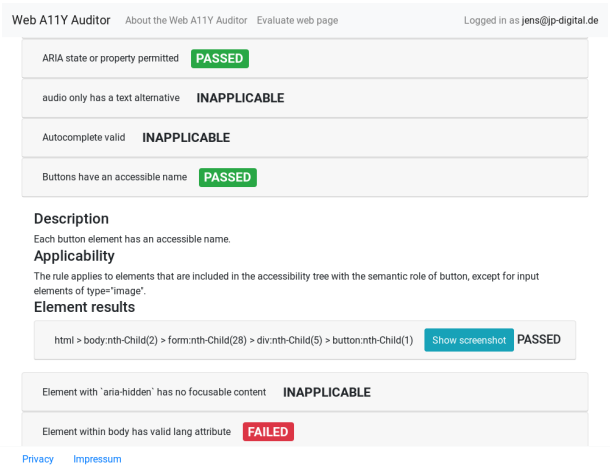

Figure 4: Results of evaluation in the web-a11y-auditor

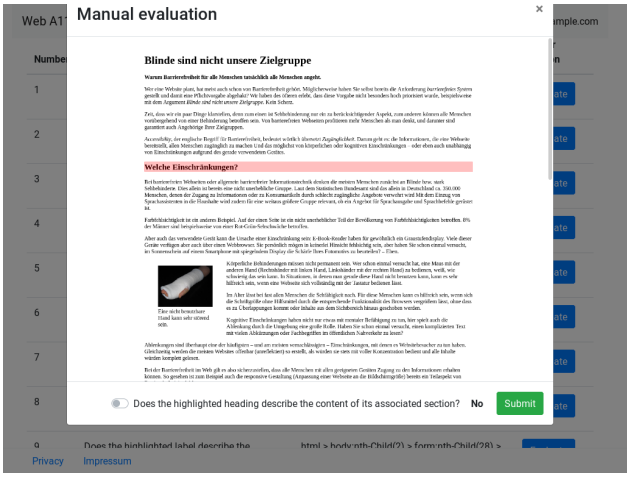

Figure 5: Dialog for manual evaluation in the web-a11y-auditor web application

At first glance, the approach presented in this paper looks very similar to the approach used by the MAUWE project [15]. However, the approaches differ in many ways. In LGWD the definitions of the conditions and checks are much more oriented towards a DOM API (at least in the examples shown in the paper). This makes it difficult if not impossible to add checks which can not be done using the DOM API, for example checking for keyboard traps. Another difference is the model itself. LGWD is a custom XML language. The model presented in this paper uses ontologies that provide a clearly defined semantics and basic structure.

The W3C itself uses a simpler form of ontologies - taxonomies - in various specifications. The taxonomies are usually available as RDF files. For example, the roles defined in the ARIA technical recommendation are available as RDF. There are also more libraries in different programming languages for processing RDF than for processing OWL ontologies. An RDF variant of the model will be made available soon.

Currently the usability and effectiveness of the prototype described in section 5 is evaluated in a study with potential users. First tests revealed some issues in the user interface and the implementation itself which will be addressed in a new version of the web-a11y-auditor which should be available in February 2020.

It is also planned to refine the ontology further and to model other rules like the tests described in the Techniques for WCAG document or the test procedures described by the German BITV-Selbsttest[13].

Originally it was indented to put much more logic into the ontology, for example inferring if a document passed a rule. The first experiments showed several problems with that approach. One problem already emerged during the development of the first version of the ontology itself. OWL uses an Open World Assumption. Expressing that there are no more instances of a class than those specified in the ontology required complex additional modeling. The second problem was performance. With only a few (less than 40 elements) the reasoning worked as expected. But even small web pages usually contain several hundred HTML elements. With that number of individuals, reasoning on OWL ontologies with the available reasoners like Openllet[14] become extremely slow and required several Gigabytes of RAM. Sometimes this even caused out of memory errors. Therefore, it was decided to use the ontology only to model the ACT Rules and to implement all other things in code. There are some newer reasoners like Konclude[15] which may have better performance. But these reasoners are not production-ready yet and have therefore not been tested yet.

Editing the ontology is currently done using the Protégé ontology editor[16]. Using editors like Protégé requires knowledge about the concepts behind OWL. To make editing the ontology and especially the rules in the ontology easier a graphical editor will be developed as a next step. Another possibility to make editing the rules easier might be to use Natural language processing to generate the rules directly from the specifications.

To automate more of the tests technologies like machine learning or natural language processing might be useful. For example, checking if the text of the web page or a section matches the

Another possible extension is the combination of the rules ontology with an ontology of diseases, impairments, and how they affect the abilities to use web pages and an ontology with possible refactorings for web pages. A tool implemented as a browser extension could use these combined ontologies to check for accessibility problems relevant for the specific user, and then apply refactorings to make a web page more usable for this specific user.

Modern web sites often use JavaScript to dynamically add content to a page. Some web sites are already more like applications and consist only of a single page (Single Page Applications, or SPAs). Evaluating such sites may require other methods than those used at the moment.

Web Content is often created by users without a technical background using Content Management Systems such as Wordpress

[13] https://www.bitvtest.de
[14] https://github.com/Galigator/openllet
[15] https://www.derivo.de/en/produkte/konclude.html
[16] https://protege.stanford.edu/

or Joomla. At the moment most Content Management Systems do not assist authors in creating accessible content. A tool integrated into the CMS using the ontology described in this paper might help authors to create accessible content.

ACKNOWLEDGMENTS

I would like to thank my Ph.D. supervisors Prof. Dr. Bernd Krieg-Brückner and Dr. Serge Autexier for their valuable feedback.

REFERENCES

[1] 2016. Directive (EU) 2016/2102 of the European Parliament and of the Council of 26 October 2016'on the accessibility of the websites and mobile applications of public sector bodies. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016L2102

[2] Julio Abascal, Myriam Arrue, and Xabier Valencia. 2019. Tools for Web Accessibility Evaluation. In *Web Accessibility: A Foundation for Research*, Yeliz Yesilada and Simon Harper (Eds.). Springer London, London, 479–503. https://doi.org/10.1007/978-1-4471-7440-0_26

[3] Shadi Abou-Zahra and Michael Squillace. 2017. Evaluation and Report Language (EARL) 1.0 Schema. https://www.w3.org/TR/EARL10-Schema/

[4] Giorgio Brajnik, Yeliz Yesilada, and Simon Harper. 2010. Testability and validity of WCAG 2.0: the expertise effect. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*. ACM, 43–50.

[5] Giorgio Brajnik, Yeliz Yesilada, and Simon Harper. 2012. Is accessibility conformance an elusive property? A study of validity and reliability of WCAG 2.0. *ACM Transactions on Accessible Computing (TACCESS)* 4, 2 (2012), 8.

[6] Alastair Campbell, Michael Cooper, and Andrew Kirkpatrick. 2019. Techniques for WCAG 2.1. https://www.w3.org/WAI/WCAG21/Techniques/

[7] Joanmarie Diggs, James Craig, Shane McCarron, and Michael Cooper. 2017. *Accessible Rich Internet Applications (WAI-ARIA) 1.1.* Technical Report. W3C. https://www.w3.org/TR/wai-aria-1.1/

[8] Joanmarie Diggs, Bryan Garaventa, and Michael Cooper. 2018. *Accessible Name and Description Computation 1.1.* Technical Report. W3C. https://www.w3.org/TR/accname-1.1/

[9] Steve Faulkner and Scott O'Hara. 2019. *ARIA in HTML W3C Working Draft 06 December 2019.* Technical Report. W3C. https://www.w3.org/TR/html-aria/

[10] Wilco Fiers, Maureen Kraft, Mary Jo Mueller, and Shadi Abou-Zahra. 2019. *Acessibility Conformance Testing (ACT) Rules Format 1.0.* Technical Report. W3C. https://www.w3.org/TR/act-rules-format/

[11] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. 2012. OWL 2 Web Ontology Language Primer. https://www.w3.org/TR/owl2-primer/

[12] ISO/IEC 40500:2012 Information technology — W3C Web Content Accessibility Guidelines (WCAG) 2.0 2012. ISO/IEC 40500:2012 Information technology — W3C Web Content Accessibility Guidelines (WCAG) 2.0. https://www.iso.org/standard/58625.html

[13] Andrew Kirkpatrick, Joshue O. Connor, Alastair Campbell, and Michael Cooper. 2018. Web Content Accessibility Guidelines (WCAG) 2.1. https://www.w3.org/TR/WCAG21/

[14] Almendra Nuñez, Arturo Moquillaza, and Freddy Paz. 2019. Web Accessibility Evaluation Methods: A Systematic Review. In *Design, User Experience, and Usability. Practice and Case Studies*, Aaron Marcus and Wentao Wang (Eds.). Springer International Publishing, Cham, 226–237.

[15] Antonio Giovanni Schiavone and Fabio Paternò. 2015. An extensible environment for guideline-based accessibility evaluation of dynamic Web applications. *Universal Access in the Information Society* 14, 1 (March 2015), 111–132. https://doi.org/10.1007/s10209-014-0399-3

[16] Simon Stewart and David Burns. 2018. *WebDriver W3C Recommendation 05 June 2018.* Technical Report. W3C. https://www.w3.org/TR/webdriver1/

[17] Simon Stewart and David Burns. 2019. *WebDriver Level 2 W3C Working Draft 24 November 2019.* Technical Report. W3C. https://www.w3.org/TR/webdriver/

[18] Carlos A Velasco, Shadi Abou-Zahra, and Johannes Koch. 2017. Developer Guide for Evaluation and Report Language (EARL) 1.0. https://www.w3.org/TR/EARL10-Guide/

[19] Verordnung 2011. Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung - BITV 2.0. https://www.gesetze-im-internet.de/bitv_2_0/BJNR184300011.html

B.4. Frontiers in Computer Science 2020 (under review)

This paper[33] is a submission for a special issue of Frontiers in Computer Science about accessibility, edited by some of the chairs of the W4A 2020. This article has been accepted at January 6th, 2021 and will published soon.

frontiers

A declarative model for web accessibility requirements and its implementation

Jens Pelzetter [1,*]

[1] *Faculty 3 - Mathematics and Computer Science, University of Bremen, Germany*

Correspondence*:
Jens Pelzetter
jens.pelzetter@uni-bremen.de

8267 words, 5 figures

ABSTRACT

The web has become the primary source of information for many people. Many services are provided on the web. Despite extensive guidelines for the accessibility of web pages, many websites are not accessible, making these websites difficult or impossible to use for people with disabilities. Evaluating the accessibility of web pages can either be done manually, which is a very laborious task, or by using automated tools. Unfortunately, the results from different tools are often inconsistent because of the ambiguity of the current guidelines. In this paper, a declarative approach for describing the requirements for accessible web pages is presented. This declarative model can help developers of accessibility evaluation tools to create tools that produce more consistent results and are easier to maintain.

Keywords: Web, Accessibility, WCAG, ACT Rules, Accessibility Evaluation

1 INTRODUCTION

The web has become the primary source of information for many people in the last two decades. Many companies sell their products online. In many countries, government services are available on the web. However, despite the availability of extensive guidelines for accessible web pages and web applications, many websites are not accessible yet. On the other hand, a significant number of people have slight impairments or develop slight impairments when growing older. Before long, the first people who have grown up with the internet will reach an age in which age-related impairments become imminent. Therefore, accessibility will become even more relevant for web pages in the next years.

Accessible web pages may also be helpful for other people, like people with temporary impairments. For instance, the ability to use a pointing device (mouse) may be limited due to an injury of the dominant hand. In this case, it may be helpful for a user if a web page can be operated using the keyboard. Environmental conditions like bright sunlight are another example. Under such conditions, a web page with insufficient contrast can become very difficult to read.

Most of the guidelines for accessible web pages are based on the Web Content Accessibility Guidelines (Kirkpatrick et al., 2018) published by the W3C. The previous version 2.0 of the WCAG (Caldwell et al., 2008) became an ISO standard in 2012 (ISO, 2012). Many legislative bodies have also recognized the importance of accessible web pages. For example, the European Union issued the European Web Accessibility directive (The European Parliament and the Council of the European Union, 2016). The

”Barrierefreie Informationstechnikverordnung” (German Accessible Information Technology Ordinance, BITV 2.0) (Verordnung, 2011) implements this directive as national legislation. Both cite the WCAG.

The manual evaluation of the accessibility of a web site is a time-consuming task that requires good expertise in web technologies *and* accessibility. Several tools are available that provide some support for evaluating the accessibility of a web page. The Web Accessibility Evaluation Tools List[1] published by the Web Accessibility Initiative of the W3C contains 136 entries. The completeness and implementation approaches of these tools vary significantly. Some tools only check a specific aspect of web accessibility, for example, whether the text on a web page has sufficient contrast. Other tools check a variety of criteria. Another difference is the user interface: Some tools integrate their user interface into the user's browser as a browser extension. Some are implemented as stand-alone applications or web services.

Only 28 of these tools are categorized as supporting the most recent version 2.1 of the WCAG. Many tools have not been updated to implement the most recent guidelines and support the most recent web technologies. One reason for this problem is that the implementation of the guidelines defined by the WCAG and its supplemental documents like the *Techniques for WCAG 2.1* (Campbell et al., 2019) is quite complex. From the user's perspective, these tools are often cumbersome to use. Different tools often produce inconsistent results and still require intensive manual work.

This paper presents a declarative model for describing accessibility requirements for web pages according to the Web Content Accessibility Guidelines by W3C. Rules for checking the accessibility of a web page have been broken down into small tests that can be implemented independently. The declarative model describing accessibility requirements presented in this paper allows developers to focus on the implementation of tests and an easy-to-use interface for the user. Different Tools using this declarative model should produce more consistent results. Adding new rules to a tool using the declarative model or improving existing rules should be much easier since only the declarative model has to be updated instead of changing the code of the tool.

The rest of the paper is structured as follows: In section 2, related work is discussed. Also, a brief overview of the current accessibility standards is provided. In section 4.1, the declarative model and its representation using an ontology and a prototype of a tool using the declarative model are presented. Section 5 discusses the results and future work.

This paper is an extended version of a previously published paper (Pelzetter, 2020) presented at the Web for All conference in April 2020. The declarative model is described in more detail. A more detailed description of the *web-a11y-auditor*, a prototype of a tool that uses the declarative model, has also been added.

2 BACKGROUND AND RELATED WORK

2.1 Background

The primary (technical) guidelines for creating accessible web pages are the Web Content Accessibility Guidelines (WCAG) (Kirkpatrick et al., 2018) published by the W3C. The Web Content Accessibility Guidelines are a technology-agnostic description of the requirements for accessible web pages. Possible techniques for implementing the WCAG are described in *Techniques for WCAG 2.1* (Campbell et al., 2019). *Techniques for WCAG* also describes several common failures. These failures describe common bad practices in web development. Web pages on which these bad practices can be found are often difficult to use for people with impairments. Each description of a technique or failure contains a test procedure to

[1] `https://www.w3.org/WAI/ER/tools/`, retrieved Dec 14th, 2019

check whether the technique has been successfully implemented or not. For each failure, a test procedure is provided for checking that the failure is not present on a web page.

Neither the Web Content Accessibility Guidelines nor the Techniques for WCAG document define which technique may be used to satisfy a success criterion of the WCAG or which failures cause a web page to fail a success criterion. *How to meet WCAG (Quick Reference)* (Eggert and Abou-Zahra, 2019) describes which techniques can be used to satisfy each success criterion of the WCAG. It also lists the failures described in the *Techniques for WCAG* that are relevant for each success criterion. For some success criteria, different techniques may be sufficient depending on the characteristics of the document. For example, the sufficient techniques for success criterion *1.4.3 Contrast (Minimum)* depend on the font size and the font-weight of the text. Some success criteria can only be satisfied by the combination of multiple techniques.

Understanding these requirements requires time, a good understanding of web technologies and accessibility requirements, and careful reading. To make the technical requirements for accessible web pages easier to understand and less ambiguous, and to harmonize the interpretation of the requirements defined by the WCAG, the W3C has published a new recommendation, the Accessibility Conformance Testing (ACT) Rules Format (Fiers et al., 2019). This recommendation defines a structure for writing rules to test accessibility.

Two types of ACT Rules have been defined: Atomic rules define a specific requirement, composite rules combine several other rules. Each ACT Rule consists of several sections. Both types of rules contain a unique ID, a description, a mapping to one or more success criteria of the WCAG, assumptions about the evaluated web page or the elements for the rule is applicable, possible limitations of assistive technology relevant for the rule, and test cases to check the implementation of a rule.

Moreover, atomic rules list the input aspects relevant to the rule, such as the DOM Tree or CSS Styling. The *Applicability* section of an atomic rule describes for which elements a rule is applicable. Each atomic rule defines at least one expectation that must be met by the elements for which the rule is applicable. If a rule has multiple expectations, each element for which the rule is applicable must satisfy all expectations. Composite rules may also have an Applicability section. The Expectation section of a composite rule lists all rules combined by the rule and defines whether all or at least one of the combined rules must be passed by the elements for which the rule is applicable. For evaluating the accessibility of a web page, the *Applicability* definition and *Expectations* are the most relevant sections. A community group has already created several rules using this format[2].

For example, the rule *Button has an accessible name* [3] describes how to check whether a button has an accessible name. Buttons are used frequently in modern web design for all kinds of interactions, like opening a menu. Such a button is only usable with assistive technology like screen readers if the button has an accessible name. The accessible name is provided to the screen reader by the browser. The applicability section describes how to find all elements for which this rule is applicable:

> The rule applies to elements that are included in the accessibility tree with the semantic role of `button`, except for `input` elements of `type="image"`.

This definition describes several conditions that have to be met by the elements for which the rule is applicable:

[2] `https://act-rules.github.io/pages/about`

[3] `https://act-rules.github.io/rules/97a4e1` accessed 2019-12-07

- The element must be included in the *accessibility tree*. Apart from the DOM tree used to create the visual output of an HTML document, a browser also manages a second tree of elements, the accessibility tree. This tree is provided to assistive technologies by the browser using the accessibility API of the operating system. The DOM tree and the accessibility tree are not the same. An element that is present in the DOM tree may not be part of the accessibility tree.
- The second requirement is that an element has the semantic role *button*. The term *role* originates from the ARIA (Diggs et al., 2017) recommendation. Roles are used in ARIA (among other extensions to HTML) to provide the accessibility API of the operating system with more information about the semantics of an HTML document. Many HTML elements have implicit roles (Faulkner and O'Hara, 2020). For example, the role `button` is implicitly assigned to the HTML elements for creating buttons (`<input type="button">` and `button`).

  It is also possible to create a widget that looks like a button using other HTML elements like a `div` container. For this widget, the role `button` must be provided explicitly by the author. In both cases, the rule *Button has accessible name* is applicable.
- The third requirement is that the element is *not* an *image button*. With `<input type="image">` it is possible to use an image as a button. This type of buttons is excluded from this rule because image buttons are checked by other rules.

The rule has a single expectation:

> Each target element has an accessible name that is not empty (`""`).

The expectation states that each element for which the rule is applicable must have an accessible name and that the accessible name cannot be an empty string. The accessible name is used by assistive technology like screen readers to disclose the button to the user. The accessible name should contain a brief description of the purpose of the button. The algorithm that browsers should use to compute the accessible name of an element is described in a technical recommendation (Diggs et al., 2018) published by the W3C.

2.2 Related Work

The different approaches for evaluating the accessibility of a web page may be categorized into three categories (Abascal et al., 2019; Nuñez et al., 2019):

- Automatic testing
- Manual inspection (by experts)
- User testing

Automated testing is mostly used according to Nuñez et al. (2019) but does not always find all existing problems. Testing by experts is the most effective way, and user testing works most effectively to verify how people with disabilities perform specific tasks on a web page (Abascal et al., 2019; Nuñez et al., 2019). One important difference between user testing and expert evaluation is that user testing is more focused on the usability of web pages for users with disabilities. In contrast, an expert evaluation is more focused on the technical side (Abascal et al., 2019).

Despite their limitations, automated tools play an important role in the process of developing accessible websites because they significantly reduce the time and effort required to conduct an evaluation (Abascal et al., 2019). The currently available automated tools may produce different results, false negatives, or false positives because of different implementations of the guidelines. Multiple tools may be combined for an accessibility evaluation to avoid missing potential problems(Abascal et al., 2019).

Automated tools can not check all requirements. For example, to check whether a heading is sufficient for its associated section, human judgment is required. The effectiveness of automated tools varies depending on the number of tests implemented. Another factor that affects the effectiveness of a tool is the ability of the developers of the tool to translate the guidelines for accessible web pages, which are expressed in natural language, into a computational representation (Abascal et al., 2019). The *Evaluation and Report Language (EARL)*(Abou-Zahra and Squillace, 2017; Velasco et al., 2017) has been proposed to facilitate the comparison of results from different tools. Unfortunately, EARL has not reached the status of a W3C Technical Recommendation yet.

Several tools for user testing use crowdsourcing. These tools have two different approaches. Some of them are trying to improve the accessibility of a web page by adding metadata. The process of annotating a web page with such metadata is likely to be very time-consuming; crowd-based tools allow the distribution among many authors. Other crowd-based tools split the accessibility evaluation into small tasks for distribution, making the evaluation less expensive (Abascal et al., 2019). The effectiveness of crowdsourcing-based tools has not yet been demonstrated.

Manual inspection by experts also has its issues. Even experts do not find all accessibility problems. In some cases, a manual inspection may also produce false positives (problems that do not exist). In a study based on the WCAG 2.0 (Brajnik et al., 2012), experts and novice evaluators evaluated several web pages for accessibility problems. Experts were only correct in 76% of all cases. This rate dropped by about 10% for novice evaluators. Expert users produced 26% to 35% false positives and missed 26% to 35% of the real problems. Novice evaluators without much experience produced much more false positives than experienced evaluators and found less real problems than experienced experts. Due to a large variety in the results of novice evaluators, no conclusions for that group were drawn in the study (Brajnik et al., 2010, 2012).

It has been suggested to develop unambiguous, machine-readable specifications to facilitate a better application of the accessibility guidelines, to produce more consistent results, and to develop tools that seamlessly integrate into the development process of web pages to increase the adoption of accessibility guidelines (Abascal et al., 2019).

An XML based language for specifying accessibility guidelines, the *Language for Web Guideline Definition* (LWGD), has been proposed together with an environment called *MAUVE* for evaluating accessibility (Schiavone and Paternò, 2015). The validation process of MAUVE is based on the DOM tree of the document. MAUVE downloads the web page to evaluate and creates the DOM tree itself. The validator module interprets the guidelines formalized in the XML language to checks whether the DOM tree passes the checks defined in XML. The LWGD language allows it to define the element to check the conditions to validate. The conditions may be combined using boolean operators.

The effectiveness of MAUVE was compared with the Total Validator[4], a commercial product. In comparison, MAUVE missed fewer problems than the Total Validator. For false positives, MAUVE reported more false positives than the Total Validator in some cases; in other cases, the Total Validator produced more. The paper about MAUVE (Schiavone and Paternò, 2015) shows an example with two conditions: One to check whether an element is followed by another element, and one to check whether an element has a specific child element. The paper does not list all available conditions.

[4] `https://www.totalvalidator.com`

A newer version of MAUVE, called MAUVE++, has also been extended to support the WCAG 2.1 and was compared to the WAVE tool Broccia et al. (2020). MAUVE++ still uses the LWGD for specifying the rules to check. The main improvements are on the user site, including a better presentation of the result and the option to validate complete websites.

3 METHODS

The declarative model described in this article was developed in three steps. The key components of the model are atomic tests that can be combined to express more complex rules. Two options were considered as a starting point: The tests described in the Techniques for WCAG (Campbell et al., 2019), and the rules developed by the ACT Rule Community Group[5] based on the new ACT Rules format (Fiers et al., 2019).

The description of the techniques for WCAG (Campbell et al., 2019) does not contain information when a technique is applicable. This information is provided in an additional document, *How to meet WCAG* (Eggert and Abou-Zahra, 2019). It turned out that the descriptions of the applicability of the techniques from this document are sometimes ambiguous. Also, the description often contained conditions that turned out to be difficult the translate into a formal, machine-readable form.

The second option that was considered as a starting point was the ACT Rules Format (Fiers et al., 2019), which has been developed with the goal of creating unambiguous rules that can be implemented more easily. These rules turned out to be much easier to translate into a formal model. An ACT Rule consists of several sections, which differ depending on the type of the rule. Atomic rules describe a specific requirement that a web page has to satisfy to be accessible. Composite rules combine the outcome of other rules to a single outcome and are used to describe complex requirements.

The two sections of an atomic ACT Rule that are most important for creating the model are the applicability definition and the expectations. The applicability definition describes for which elements a rule is applicable. The expectations describe the requirements that each element for which the rule is applicable must satisfy. The applicability sections and the expectation sections contain many repeating phrases such as "...is included in Accessibility Tree...".

In the first step, these phrases have been collected and used to define the atomic tests described in chapter 4.1. Some of these tests require parameters, for example, the name of an attribute. The definitions developed in this step are not tailored to a specific serialization. One possible serialization is RDF. Other possible serializations are a custom XML or JSON format or an ontology.

As a second step, the model had to be put into a machine-readable form. In this case, the Web Ontology Language (OWL) (Hitzler et al., 2012) was chosen. OWL has several advantages. OWL has clearly defined semantics, allowing to verify the consistency of an ontology using a semantic reasoner. Ontologies providing knowledge for different domains can be combined to a larger ontology. It is also possible to define complex rules in an ontology that can be used by a reasoner to infer knowledge based on the data in the ontology. A very early version of the ontology and the software used this approach to infer the results of an evaluation. Unfortunately, this approach did not scale well (see section 5). The ontology also contains classes for the concepts of the WCAG, such as principles, guidelines, success criteria, and the techniques and failures described in the Techniques for WCAG. This information is used to provide context about the rules for the user. The ontology itself has been split into individual modules that may be reused independently from each other. The ontology has been created using the Protégé editor [6].

[5] https://act-rules.github.io/pages/about

[6] https://protege.stanford.edu/

The third step was the development of a prototype application that uses the ontology. The application is described in detail in section 4.2. In addition to showing that the model may be used to create an evaluation tool, the second goal of the prototype was to test how manual evaluation steps can be simplified so that even inexperienced users can perform them and produce reliable results. The architecture of the *web-a11y-auditor* is described in section 4.2.

4 RESULTS

4.1 Declarative model

Except for MAUVE, all other tools for checking web pages for accessibility problems known to the author implement tests for checking the requirements for accessible web pages directly in code. Implementing the checks directly as code makes maintenance or customization of tools difficult. The approach described in uses a declarative model of the accessibility rules to describe *what* to check, not *how* to perform tests. The key components of the declarative model are clearly specified atomic tests that are combined to build complex rules. The implementation of the tests is the responsibility of the developers of the tools that use the declarative model.

Using this declarative model to implement evaluation tools has several advantages. Developers may focus on a reliable implementation of the tests and an easy-to-use user interface. Different tools may implement the tests in different ways. A tool implemented as a browser extension may use the APIs provided by the browser for analyzing the evaluated document. Another tool with a stand-alone implementation may use a browser automation framework such as Selenium for accessibility evaluation.

The results produced by the tools may still differ in detail, but this can only be caused by the limitations of the implementation of the tests and not by different interpretations of the rules. Also, because each implementation of a test is only a small, independent unit of code, the implementations of the test tasks are easy to test.

For the development of test tools, the usage of the declarative model has an additional advantage. To adopt the requirements for accessible web pages for new technologies or changing usage patterns, the guidelines and rules for accessible web pages have to be updated with an increasing pace. Using the declarative model, it is not necessary to change the code of a test tool to integrate new rules. Instead, only the model has the be updated. Code changes are only necessary if a rule requires an atomic test that was not implemented before.

The ACT Rules developed by the ACT Rule Community Group have been chosen as a starting point because they provide the best available (if incomplete) summary of the requirements for accessible web pages. The ACT Rules also contain fewer special cases than the description of sufficient technologies provided by the WCAG Quick Reference. Therefore, the assumption was that the ACT Rules are easier to model. For the ACT Rules developed by the ACT Rules community, there are also test cases available for each rule, which allows checking the implementation of a rule.

The first step in developing the model was the definition of the atomic tests needed to build a model of the ACT Rules. In the current version, the ontology contains 48 tests used to build a declarative, machine-readable model of the applicability definition and expectations of ACT Rules. Some of these tests require additional parameters. These parameters are also described in the model.

An ACT Rule may have several *outcomes*. The outcome `Passed` indicates that an element has passed a rule and the outcome `Failed` indicates that an element has failed a rule. If one element for that the rule is applicable does meet the expectations of the rule, the complete document fails the rule. If no elements for

which the rule is applicable are found, the outcome of the rule is `Inapplicable`. An ACT Rule tested by an automatic tool may also have the outcome `cannot tell` if one of the tests of the rule cannot be done automatically.

4.1.1 Examples for Tests

The following examples for checks are shown in a function-like notation. The tests described here can be interpreted as an extension of the `Element` interface of the DOM API (WHATWG, 2020). Therefore, the tested element is not explicitly specified as a parameter. How exactly these tests are implemented depends on the design of the implementation. The test `matchesCssSelector(selector)` checks if an element matches the specified CSS selector. This test is used in many applicability definitions to find the elements for which the rule is applicable. Usually, this test is combined with other tests to find the elements for which an ACT Rule is applicable.

Accessibility APIs require information about the role of an HTML element, for example, if the element is a button. For many HTML elements, implicit roles have been defined (Faulkner and O'Hara, 2020). If necessary, authors can change the role of an HTML element using the `role` attribute. The test `hasRole(roleName, ..)` checks if an element has one of the roles provided in the parameters, either as an implicit role and explicitly assigned. This test is often used to filter out form controls or buttons. The test passes if the tested element has at least one of the roles provided in the parameters.

For some rules, it is also necessary to check if a role has been explicitly assigned to an element, such as the role `button` to a `div` element used as a button. The test `hasExplicitRole()` checks if a role has been explicitly assigned to the tested element. The test only checks if the role of the element has been explicitly assigned, not if the element has a specific role. A combination of `hasExplicitRole` and `hasRole` is used to test if a specific role has been explicitly assigned to an element.

The Accessible Rich Internet Applications recommendation (ARIA) (Diggs et al., 2017) defines several attributes that can be used to provide the accessibility API of the operating system with additional information. Several rules are only applicable for an element if they have an ARIA attribute. The test `hasAriaAttribute()` is used to check if at least one ARIA attribute is assigned to the tested element.

One important property for the accessibility of web pages is the accessible name of an element. The accessible name is part of the accessibility tree and is, for example, used by assistive technology like screen readers to present the element to the user. For example, if a button has no accessible name, a screen reader cannot properly announce the button to the user, and the user has no clue about the function of the button. The accessible name is computed from several properties (Diggs et al., 2018). The test `hasAccessibleName()` used the algorithm for computing the accessible name to check if the tested element has a none empty accessible name.

The implementation of these tests may vary. Some require access to the DOM tree and are only checking the presence or absence of attributes or elements. A tool implementing the tests should not use the HTML document received from the server directly. On modern web pages, the initial HTML code is often altered by scripts running after the browser has loaded the page. If an HTML document is syntactically incorrect, browsers try to fix the errors by altering the DOM tree. Therefore, the DOM tree generated by the browser should be used for accessibility evaluations and not the raw source code of the document.

4.1.2 Combination of Tests

For most applicability definitions and expectations, it is necessary to combine several tests. For some applicability definitions and expectations, some form of negation is required, for example, to express that an element should not have a specific role.

The model provides three options for this purpose. `negate` is used to express negation, `allOf` and `oneOf` are used to combine the outcomes of multiple tests. The names `negate`, `allOf` and `oneOf` were chosen instead of the simpler names `not`, `and` and `or` to avoid conflicts with ontology languages or programming languages in which *not*, *and* and *or* are often reserved identifiers.

An implementation of `negate` should change the outcome of the associated test from `passed` to `failed` and vice versa. Other outcomes like `cannot tell` should not be changed by the implementation of `Negate`.

Two options are available for combining multiple tests: `allOf` requires that all combined tests pass, `oneOf` requires only one of the combined tests to pass. An implementation also has to handle special cases, for example, whether one test has the outcome `cannot tell`. In this case, the implementation of `allOf` as well as of `oneOf` should return the outcome `cannot tell`.

4.1.3 Expressing ACT Rules using the model

The following examples of ACT Rules are given in the same notation as the examples of individual tests.

The rule *Button has an accessible name*[7] checks if a button has an accessible name. The applicability definition for this rule is:

> The rule applies to elements that are included in the accessibility tree with the semantic role of *button*, except for *input* elements of *type="image"*.

This applicability definition can be broken down into three tests:

- Checking whether the element is included in the accessibility tree.
- Checking whether the element has the semantic role of `button`.
- Checking whether the element is an image button. This can be done using the CSS selector `input[type=image]`.

Using the declarative model, the applicability definition of this rule can be expressed as pseudo code:

```
allOf(
    isIncludedInAccessibilityTree(),
    hasRole("button"),
    not(matchesCssSelector("input[type=image]"))
)
```

The applicability definition requires that elements for which the rule is applicable are *not* buttons of the type image. Therefore, the outcome of the test for the CSS selector is negated. The applicability definition also requires that all requirements are met by elements for which the rule is applicable. Therefore, `allOf` is used to combine the tests.

The expectation of this rule requires that every button has an accessible name:

> Each target element has an accessible name that is not empty (`""`).

Only one test is necessary the check whether an accessible name is available for an element:

[7] https://act-rules.github.io/rules/97a4e1

```
hasAccessibleName()
```

Images can be used in different ways on a web page. One purpose is decoration; another purpose is to support the textual content. If an image is not used as decoration, it needs an accessible name that describes the content of the image. Decorative images have to be marked correctly using an empty `alt` attribute. The rule *Image has accessible name*[8] checks whether an image is either marked as decorative or has an accessible name.

The applicability definition of the rule is:

> The rule applies to HTML `img` elements or any HTML element with the semantic role of `img` that is included in the accessibility tree.

This applicability definition can be broken down into three tests:

- The element has the semantic role of `img`. Sometimes the equivalent role `image` is used for images. This role is also provided as a parameter for the `hasRole` test.
- All `img` elements are applicable, regardless of the role assigned to them.
- The element is included in the accessibility tree.

These tests can be expressed in their combination as

```
allOf(
    oneOf(
        hasRole("img", "image"),
        matchesCssSelector("img")
    ),
    isIncludedInAccessibilityTree()
)
```

The tests for the role and the CSS selector are combined with `oneOf` to express that only one of these tests has to pass. The result of `isIncludedInAccessibilityTree` is combined with the result of `oneOf` to express that an element for which this rule is applicable has to be included in the accessibility tree.

The expectation of the rule is:

> Each target element has an accessible name that is not empty (`""`) or is marked as decorative.

The expectation can be broken down into two conditions:

- Checking whether the image has an accessible name
- Checking whether the image is marked as decorative

These conditions can be expressed as

```
oneOf(
    hasAccessibleName()
```

[8] https://act-rules.github.io/rules/23a2a8

```
    isDecorative()
)
```

An element for which the rule is applicable has only to pass one of the two tests. Therefore, both tests are combined using `oneOf`.

4.1.4 ACT Rules as an Ontology

There are several options for creating a machine-readable representation of models like the one described in this paper. One possible option is the development of a custom XML language or a JSON data model. Another option is to use linked data (RDF) or an ontology. For the model described here, an ontology was used. There are several different ontology languages. One of the most common ones is the Web Ontology Language (OWL) (Hitzler et al., 2012), which is used for ontology described in this article.

The ontology[9] contains the tests described in the previous sections, the ACT Rules (as of November 2019), as well as the Success Criteria of the WCGA 2.1. The tests described in section 4.1 are modeled as classes. Individual applications of the tests for a rule are modeled as individuals together with the required parameters. The values of the parameters are provided using data properties. The UML diagram in figure 1 shows the primary classes and properties of this ontology. The diagram uses the UML notation. The generalization relationship is used to show a subclass relationship between two classes. Object properties are shown using the usual UML elements for properties. Data properties are shown as properties inside their domain class. Please note that the diagram does not show all properties. For example, most of the object properties have an inverse counterpart to make it easier to retrieve related individuals from both ends of a relationship. These properties are not shown in the diagram.

Rules are represented as individuals of one of these two classes `AtomicACTRule` or `CompositeACTRule`. Both classes are subclasses of the `BaseACTRule` class. The `BaseActRule` class contains the properties shared by atomic and composite rules: The rule ID, which provides a unique identifier for the rule, the name of the rule, and the description of the rule. These data can be used by an application using the ontology to display the rule to a user.

The ontology also contains classes and properties for describing the success criteria, guidelines, principles, and conformance levels of the WCAG. Additional properties for describing the relations between ACT Rules and the Success Criteria of the WCAG are also provided. This information can be used by tools using the ontology to provide the user with background information about the rules.

The applicability definitions and expectations are modeled using separate classes. To provide a human-readable description of an applicability definition or an exception, the `Applicability` and `Expectation` classes also provide a property for the textual description. The textual description can be used by a tool using the ontology to display information about the rule to a user. The `Expectation` class has two properties that are not part of the ACT Rules format: The property `number` is used to order the expectations of a rule. For some use cases, it is also useful to have a unique identifier for an expectation. The `identifier` property provides such an identifier.

The tests are modeled using subclasses of the `Test` class. For each of the atomic tests found in ACT Rules, the ontology contains a subclass of the `Test` class. For brevity, these classes are not shown in figure 1. The `hasSelector` property is used to associate an individual of the `Applicability` class with an individual of the test class `ElementFilter`. For expectations, the association between the individual of the `Expectation` class and the `ExpectationTest` class is expressed using the `hasTest` property.

[9] `https://ontologies.web-a11y-auditor.net`

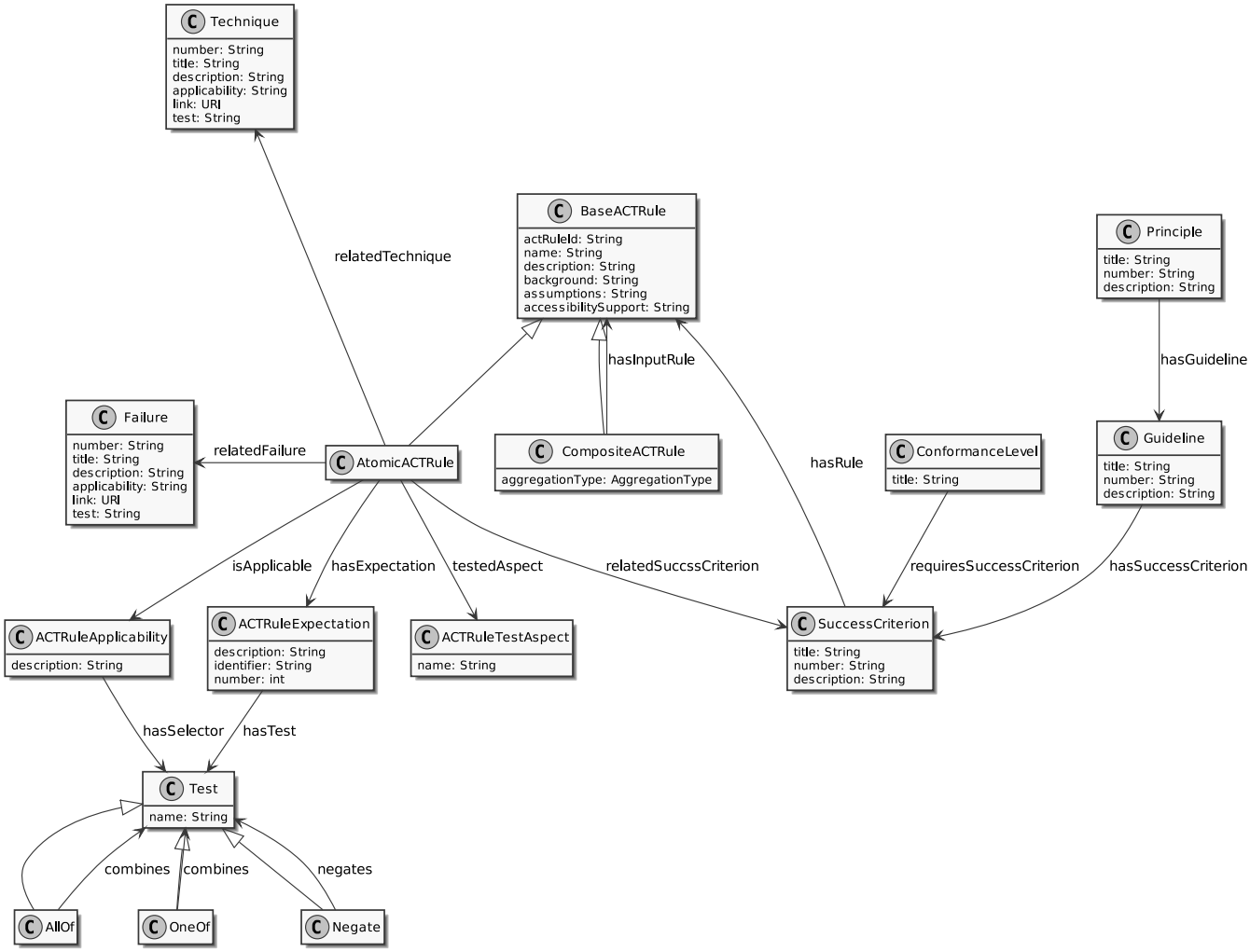

Figure 1. UML diagram showing the primary classes of the *web-a11y-auditor* ontology

The operators for combining the results of other tests (`allOf`, `oneOf`, and `negate`) are also modeled as subclasses of the `Test` class. Treating these operators as tests makes it possible to use them in the same places as atomic tests. The associations between an individual of the classes `AllOf` or `OneOf` and the combined tests are expressed using the `combines` property. For the association between an individual of the `Negate` class and the negated test, the `negates` property is used. In both cases, an individual of the `Applicability` class or the `Expectation` class can only be associated with one instance of the `Test` class. This instance can either be a real test or one of the combining tests `oneOf` or `allOf`.

Composite rules do not have an applicability definition or expectations. Instead, they combine the outcomes of multiple rules into a single result. A composite rule either requires that all combined rules pass or that at least one of the combined rules passes. Composite rules are represented in the Ontology by instances of the `CompositeRule` class. The aggregation type is represented using the `aggregationType` property. The input rules are associated with a composite rule using the `hasInputRule` property.

Figure 2 shows a UML object diagram of the individuals used to represent the rule *Button has accessible name* in the ontology. The type of the individual is provided after the colon on the top of the rectangle representing the individual. The rule itself is represented by an individual of the `AtomicACTRule` class.

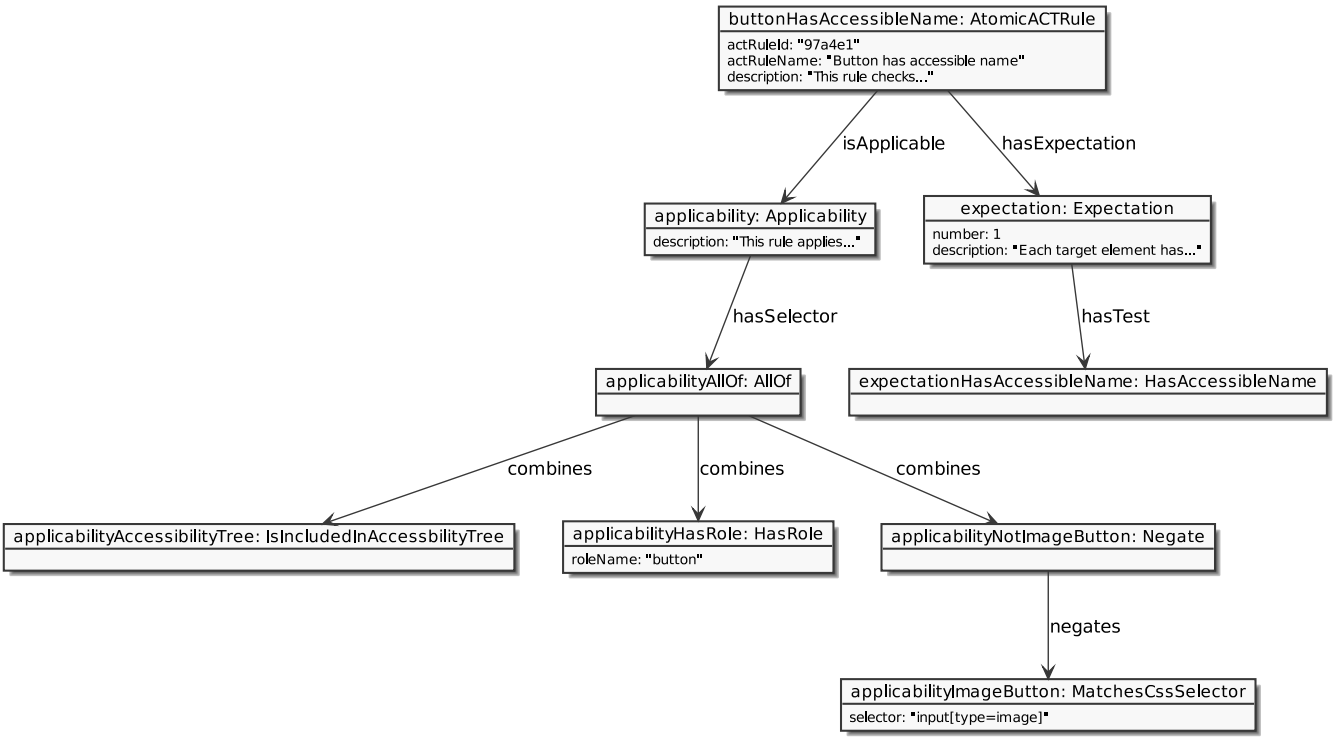

Figure 2. UML object diagram showing the rule *Button has accessible name* as an example of a rule in the ontology

Applicability definition and expectations are modeled using individuals of the classes `Applicability` and `Expectation`. The tests are represented by individuals of several subclasses of the `Test` class. In this example, these are the classes `IsIncludedInAccessibilityTree`, `HasRole`, `MatchesCssSelector` and `HasAccessibleName`.

The test `HasRole` requires an additional parameter to provide the role(s) for which the test will check. This information is provided by an additional data property. Likewise, the CSS selector used by the `MatchesCssSelector` test is provided by an additional data property. The tests without parameters do not have any additional properties.

An individual of the `Applicability` class or the `Expectation` class can only be associated with one instance of the `Test` class. If an applicability definition or an expectation consists of multiple tests, these tests have to be combined using an individual of the `oneOf` or `allOf` classes. In the example, the applicability definition contains three tests which are combined using `AllOf`. The test `MatchesCssSelector` is also negated to exclude all image buttons.

A complete list of supported ACT Rules and atomic tests is available as supplementary material.

4.2 The web-a11y-auditor — A Prototype Implementation

A prototype of an application that uses the ontology described in section 4.1, the *web-a11y-auditor*, has been implemented. The application is split into several modules, which are all run as independent services. 3 shows an overview of the architecture of the *web-a11y-auditor*.

The user interface is provided by the *web-a11y-auditor-ui* service. This module is implemented using the Nuxt framework[10]. The web-a11y-auditor-ui service interacts with a RESTful API provided by the

[10] `https://nuxtjs.org/`

web-a11y-auditor-web module. This module retrieves the information for showing results from the database. If a new evaluation is created, the *web-a11y-auditor-web* module sends a message to the *web-a11y-auditor-jobmanager-module*. The job manager module receives this message and creates a task for analyzing the document to evaluate. This task is sent to an instance of the *web-a11y-auditor-worker* module. This module is responsible for all interaction between the web page to evaluate and the *web-a11y-auditor*. The worker modules use the Selenium framework [11] to analyze the web page to evaluate. A message broker is used to manage the communication between the worker instances, the job manager, and the web module. The *web-a11y-auditor* uses Apache ActiveMQ Artemis[12] as message broker. Several different message queues are used to organize the communication between the modules of the *web-a11y-auditor*. All message queues are FIFO queues. The *New Evaluations Queue* is used by the *web-a11y-auditor-web* module to notify the *web-a11y-auditor-jobmanager* about new evaluations. The *Jobs Queue* is used by the job manager to send jobs to the worker instances. The first free worker instance will pick up the next task from the queue and execute it. Results for the test jobs are sent back to the job manager using the *Job Results Queue*. The job manager processes the messages, stores the results in the database, and creates additional tasks if necessary. For example, if an applicability test task is finished and a rule is applicable for an element, the job manager will create the tasks for checking if the element passes all expectations of the rule. The *Notifications queue* is used by the job manager to notify the *web-a11y-auditor-web* module about status changes. Using the *Status Queries Queue*, the *web-a11y-auditor-web* module can query the job manager about the status of an evaluation. The status reports are generated asynchronously and sent back to the *web-a11y-auditor-web* module using the *Status Reports Queue*.

The evaluations and the result are currently stored in a relational database (PostgreSQL). The Job Manager is the only module that can write to the database. All other modules are only reading from the database. The workers only execute the test tasks. The overall outcome of a rule for an element is computed from the results of the tests by the web application when test results are accessed. To process the tests, multiple worker instances are used to speed up the evaluation process.

To start an evaluation, the user enters the URL of the page to evaluate. The evaluation is done in five steps:

1. Analyze the document and store a screenshot of the document and the CSS selectors and coordinates of each element. The screenshot and the element coordinates are used by the web application to show which element is related to a result.
2. Generate the tasks for checking which rules are applicable for which element of the evaluated document.
3. Check which rules are applicable for which element.
4. Generate the tasks for checking the expectation for the applicable rules.
5. Check if the elements match the expectations of the applicable rules.

In the first phase, one of the workers analyzes the web page to evaluate and extracts a unique CSS selector for the element and the coordinates and the size of the bounding box of each element. Also, a screenshot of the web page to evaluate is generated in this first phase. All this information is stored in the database. In the subsequent phases, the results for each supported rule are added.

For each of the atomic tests to execute, a task is created and sent to the workers. The job manager gets the information about which atomic tests have to be executed for an applicability definition or an expectation

[11] `https://selenium.dev/`

[12] `https://activemq.apache.org/components/artemis/`

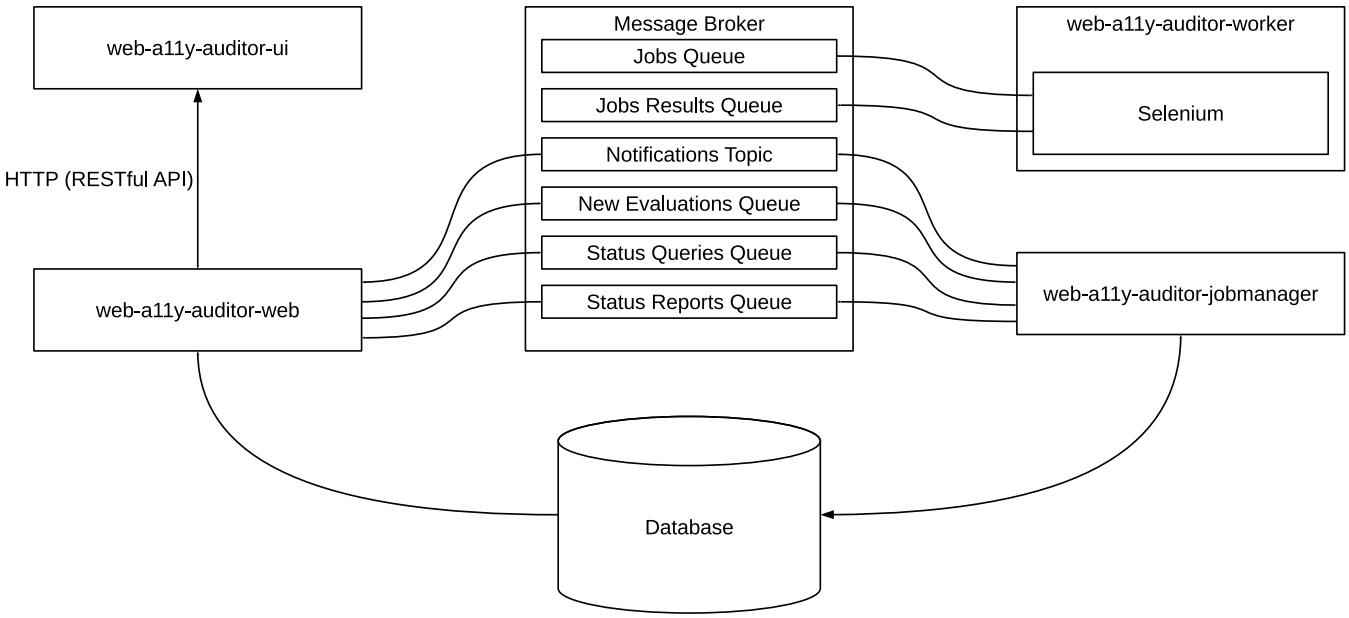

Figure 3. Diagram showing the architecture of the web-a11y-auditor

of a specific rule from the ontology. Based on this information, which includes the name of the test to execute and, in some cases, additional parameters, the job manager creates the tasks for the workers. How a specific atomic test is executed depends on the implementation of the worker.

One of the worker instances retrieves the task, executes it, and sends the result back to the Job Manager. The Job Manager stores the result in the database. Aside from the specification of the tests, all other data provided by the ontology are only used by the web application.

To execute the tests, the *web-a11y-auditor* uses the Selenium framework[13]. Selenium uses the Web Driver API (Stewart and Burns, 2018, 2019) to control a browser instance. The tests are executed in this browser instance using Selenium. This approach has some advantages compared with APIs like JSoup[14], which are implementing only a DOM parser. DOM parsers like JSoup only implement parts of a browser rendering engine. For example, elements generated using JavaScript could not be checked using a DOM parser that does not execute JavaScript. Also, some tests cannot be done by a simple inspection of the DOM tree. For example, for a test that checks if an element is focusable using the keyboard, it is necessary to simulate keyboard interaction.

Some tests cannot be executed automatically yet. For these tests, the *web-a11y-auditor* guides the user through the tests. If a test can not be executed automatically, the test is added to a list of tasks that require manual evaluation. This list is presented to the user. The user can process this list in any order. The tests are presented to the user in the form of a simple question like:

> Does the highlighted heading describe the content of its associated section?

The dialog shows a screenshot of the evaluated web page. The element under evaluation is highlighted in the screenshot. To create the image with the highlighted element, the screenshot created in the first phase of the evaluation is used. The highlighted section is added using the coordinates and the size of the bounding

[13] `https://selenium.dev/`

[14] `https://jsoup.org/`

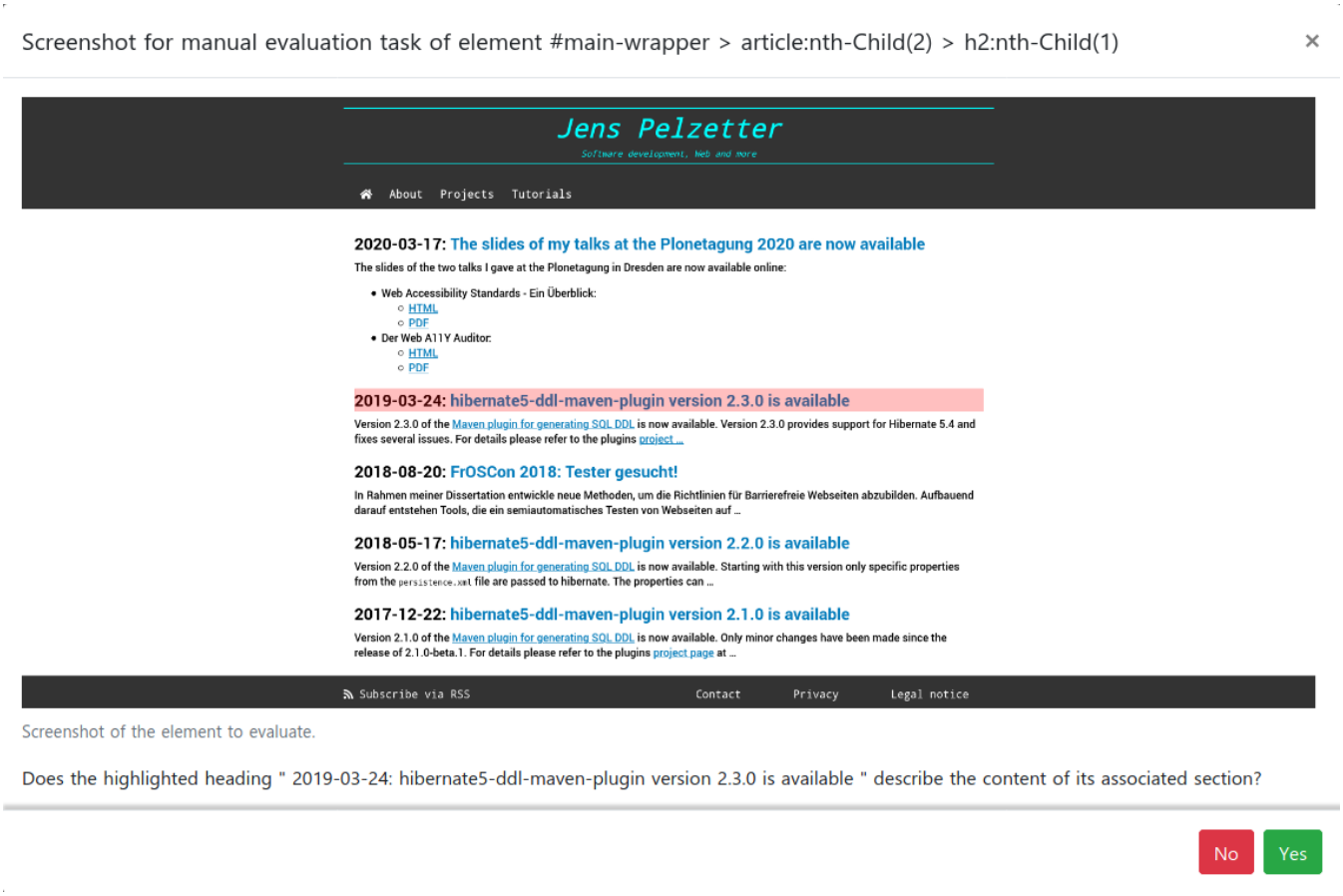

Figure 4. Dialog for manual evaluation in the *web-a11y-auditor* web application

box of the element. The coordinates and the size of the bounding box have been obtained in the first phase of the evaluation. The questions used in these dialogs are simple yes/no questions. The answer corresponds with one of the outcomes `passed` and `failed` (the web application currently uses a switch control for selecting the result). Figure 4 shows an example of the dialog. When all manual evaluation questions are done, the results of the evaluation are presented to the user. An example is shown in figure 5.

The results display shows the results for all rules. Details of the results can be viewed by clicking on one of the rules. The details view shows the results for all tested elements.

5 DISCUSSION

The declarative model presented in this article was developed with the goal of providing a foundation for different types of accessibility tools and to minimize the required maintenance for such tools. Moreover, this model is easy to extend and adapt to changes in the requirements.

The current version of the model is based on the ACT Rules Format (Fiers et al., 2019) and the rules developed by the ACT Rules Community Group. During the development of the prototype implementation, the *web-a11y-auditor*, the rules published by the ACT Rules Community Group have been updated several times. These changes included the addition of new rules, the removal of some rules, and changes to the applicability definitions and expectations of some rules. For some rules, requirements were added to the applicability definitions and expectations or removed from them. As expected, no changes in the code of

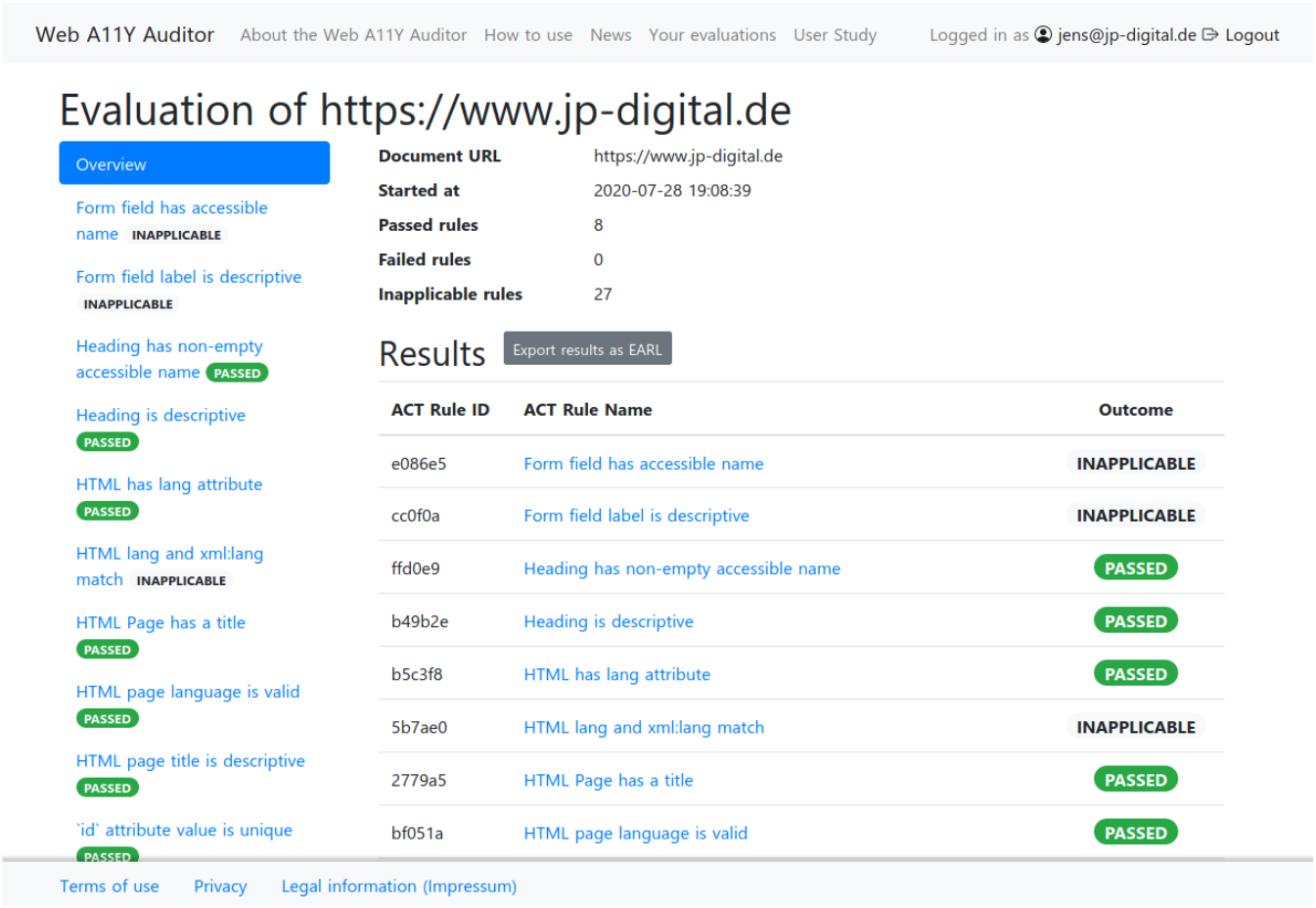

Figure 5. Results of an evaluation in the web-a11y-auditor

the prototype were necessary to integrate the updated rules into the prototype. Only the ontology was edited to match the updated rules. After replacing the ontology, the *web-a11y-auditor* used the updated rules.

At first glance, the approach presented in this paper looks very similar to the approach used by the MAUVE project (Schiavone and Paternò, 2015). However, the approaches differ in many ways. In LGWD, the definitions of the conditions and checks are much more oriented towards a DOM API (at least in the examples shown in the paper). This makes it difficult, if not impossible, to add checks that cannot be done using the DOM API, for example, checking for keyboard traps. Another difference is the model itself. LGWD is a custom XML language. The model presented in this paper uses an ontology allowing the combination of the knowledge represented in the ontology with other ontologies.

Originally it was intended to put much more logic into the ontology, for example, inferring whether a document passed a rule. The first experiments showed several problems with that approach. One problem already emerged during the development of the first version of the ontology itself. OWL uses an Open World Assumption. Expressing that there are no more instances of a class than those specified in the ontology required complex additional modeling. The second problem was performance. With only a few (less than 40 elements), the reasoning worked as expected. However, even small web pages often contain several hundred HTML elements. For each element, the applicable rules have to be determined. For each rule and element, an additional individual for the result has to be added. For a web page with 500 elements - which is not an unusual number - and the 35 rules currently supported by the web-a11y-auditor, this would

produce 17500 results for the rules. Most applicability definitions also contain more than one test. Therefore the number of test results is even larger. Each of these results would be an individual in the ontology, with several properties. For each of these properties, another axiom is added to the ontology. All these axioms have to be processed by the reasoner. With that number of axioms, reasoning on OWL ontologies with the available reasoners like Openllet[15] becomes extremely slow and requires several Gigabytes of memory. Sometimes this even causes out of memory errors. Therefore, it was decided to use the ontology only to model the ACT Rules and to use the ontology as a knowledge base only, and not as a rule engine. Based on the experience gained during the development of the web-a11y-auditor, it would also require a significant amount of code to put the data gathered from the website into the ontology.

It was planned to compare the effectiveness of the *web-a11y-auditor* with other tools and manual evaluations. Unfortunately, due to the COVID-19 pandemic, it was not possible to recruit enough people for a study. Nevertheless, the *web-a11y-auditor* was tested by some users during the development. The most valuable insights for the development of the *web-a11y-auditor* come not from the results of the evaluations, but from the responses from the users. The users who tested the *web-a11y-auditor* found the tool easy to use and also found the instructions for the manual tests very helpful.

6 CONCLUSIONS

In this paper, a declarative model for accessibility requirements of web pages was presented. The foundations of this model are so-called atomic tests, which are small, easy to implement tests. Each of these tests only checks a specific aspect. These tests are combined to formulate rules for testing the accessibility of web pages. Based on the rules developed by the ACT Rules Community Group, several atomic tests can be developed. This approach could be a possible option for creating a machine-readable model for rules in the ACT Rules Format or other similar rules. One possible serialization of this model using the *Web Ontology Language* (OWL) was also presented, together with an example of a tool - the *web-a11y-auditor* - that uses the declarative model.

The approach for modeling accessibility rules presented in this paper can be used to provide an easily extendable model for accessibility rules and similar structured rules. A prototype of a tool that uses the model is also presented to show that the model can be used as base for creating evaluation tools.

7 FUTURE WORK

The declarative model presented in this paper works as intended but can be optimized. For example, several tests, such as checking if an element is included in the accessibility tree, are used by multiple rules. In the current version of the model and the *web-a11y-auditor*, these tests are repeated for each rule that uses the tests. In an optimized version of the model, these tests should not be repeated. Instead, all rules should refer to the same instance of the test.

The *web-a11y-auditor* is currently only able to check static web pages. Dynamic web pages where new elements are added to the DOM tree cannot be validated completely. A possible approach for allowing a validation tool to check the different states of a dynamic web page is to create a model of these states and the possible transitions between the states. Ideally, this model should be created automatically.

To validate the effectiveness of the model and the prototype implementation, a study that compares the results of the *web-a11y-auditor* with other tools, such as WAVE or MAUVE++, and manual evaluation should be conducted as soon as possible.

[15] `https://github.com/Galigator/openllet`

REFERENCES

Abascal, J., Arrue, M., and Valencia, X. (2019). Tools for web accessibility evaluation. In *Web Accessibility: A Foundation for Research*, eds. Y. Yesilada and S. Harper (London: Springer London). 479–503. doi:10.1007/978-1-4471-7440-0_26

Abou-Zahra, S. and Squillace, M. (2017). *Evaluation and Report Language (EARL) 1.0 Schema*. Tech. rep.

Brajnik, G., Yesilada, Y., and Harper, S. (2010). Testability and validity of WCAG 2.0: the expertise effect. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility* (ACM), 43–50

Brajnik, G., Yesilada, Y., and Harper, S. (2012). Is accessibility conformance an elusive property? A study of validity and reliability of WCAG 2.0. *ACM Transactions on Accessible Computing (TACCESS)* 4, 8

Broccia, G., Manca, M., Paternò, F., and Pulina, F. (2020). Flexible Automatic Support for Web Accessibility Validation. *Proc. ACM Hum.-Comput. Interact.* 4. doi:10.1145/3397871

Caldwell, B., Cooper, M., Reid, L. G., and Vanderheiden, G. (2008). *Web Content Accessibility Guidelines (WCAG) 2.0*. Tech. rep.

[Dataset] Campbell, A., Cooper, M., and Kirkpatrick, A. (2019). Techniques for wcag 2.1

Diggs, J., Craig, J., McCarron, S., and Cooper, M. (2017). *Accessible Rich Internet Applications (WAI-ARIA) 1.1*. Tech. rep., W3C

Diggs, J., Garaventa, B., and Cooper, M. (2018). *Accessible Name and Description Computation 1.1*. Tech. rep., W3C

Eggert, E. and Abou-Zahra, S. (2019). *How to Meet WCAG (Quick Reference)*. Tech. rep., W3C

Faulkner, S. and O'Hara, S. (2020). *ARIA in HTML W3C Working Draft 06 August 2020*. Tech. rep., W3C

Fiers, W., Kraft, M., Mueller, M. J., and Abou-Zahra, S. (2019). *Acessibility Conformance Testing (ACT) Rules Format 1.0*. Tech. rep., W3C

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., and Rudolph, S. (2012). *OWL 2 Web Ontology Language Primer*. Tech. rep.

ISO (2012). *ISO/IEC 40500:2012 Information technology — W3C Web Content Accessibility Guidelines (WCAG) 2.0*. Tech. rep.

Kirkpatrick, A., Connor, J. O., Campbell, A., and Cooper, M. (2018). *Web Content Accessibility Guidelines (WCAG) 2.1*. Tech. rep.

Nuñez, A., Moquillaza, A., and Paz, F. (2019). Web accessibility evaluation methods: A systematic review. In *Design, User Experience, and Usability. Practice and Case Studies*, eds. A. Marcus and W. Wang (Cham: Springer International Publishing), 226–237

Pelzetter, J. (2020). A Declarative Model for Accessibility Requirements. In *Proceedings of the 17th International Web for All Conference* (New York, NY, USA: Association for Computing Machinery), W4A '20. doi:10.1145/3371300.3383339

Schiavone, A. G. and Paternò, F. (2015). An extensible environment for guideline-based accessibility evaluation of dynamic web applications. *Universal Access in the Information Society* 14, 111–132

Stewart, S. and Burns, D. (2018). *WebDriver W3C Recommendation 05 June 2018*. Tech. rep., W3C

Stewart, S. and Burns, D. (2019). *WebDriver Level 2 W3C Working Draft 24 November 2019*. Tech. rep., W3C

The European Parliament and the Council of the European Union (2016). *Directive (EU) 2016/2102 of the European Parliament and of the Council of 26 October 2016 on the accessibility of the websites and mobile applications of public sector bodies*. Tech. rep.

637 Velasco, C. A., Abou-Zahra, S., and Koch, J. (2017). *Developer Guide for Evaluation and Report Language*
638 *(EARL) 1.0*. Tech. rep.
639 Verordnung (2011). Verordnung zur Schaffung barrierefreier Informationstechnik nach dem
640 Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung - BITV 2.0)
641 WHATWG (2020). *DOM Living Standard*. Tech. rep.

C. Bibliography

[1] *Directive (EU) 2016/2102 of the European Parliament and of the Council of 26 October 2016'on the accessibility of the websites and mobile applications of public sector bodies*, 2016. https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016L2102.

[2] Abascal, Julio, Myriam Arrue, and Xabier Valencia: *Tools for Web Accessibility Evaluation.* In Yesilada, Yeliz and Simon Harper (editors): *Web Accessibility: A Foundation for Research*, pages 479–503. Springer London, London, 2019, ISBN 978-1-4471-7440-0. https://doi.org/10.1007/978-1-4471-7440-0_26.

[3] Abou-Zahra, Shadi and Michael Squillace: *Evaluation and Report Language (EARL) 1.0 Schema.* Technical report, February 2017. https://www.w3.org/TR/EARL10-Schema/.

[4] Adams, Chuck, Alastair Campbell, Rachael Montgomery, Michael Cooper, and Andrew Kirkpatrick: *Web Content Accessibility Guidelines (WCAG) 2.2 - W3C Working Draft 11 August 2020*, August 2020. https://www.w3.org/TR/WCAG22/.

[5] Allan, James, Greg Lowney, Kim Patch, and Jeanne Spellman: *User Agent Accessibility Guidelines (UAAG) 2.0*, 2015. https://www.w3.org/TR/UAAG20/.

[6] Bos, Bert, Tantek Çelik, Ian Hickson, and Håkon Wium Lie: *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* Technical report, W3C, 2016. https://www.w3.org/TR/CSS2/.

[7] Brajnik, Giorgio, Markel Vigo, Yeliz Yesilada, and Simon Harper: *Group vs Individual Web Accessibility Evaluations: Effects with Novice Evaluators.* Interacting with Computers, 28(6):843–861, October 2016, ISSN 0953-5438. https://doi.org/10.1093/iwc/iww006.

[8] Brajnik, Giorgio, Yeliz Yesilada, and Simon Harper: *Is Accessibility Conformance an Elusive Property? A Study of Validity and Reliability of WCAG 2.0.* ACM Trans. Access. Comput., 4(2), March 2012, ISSN 1936-7228. https://doi.org/10.1145/2141943.2141946.

[9] Caldwell, Ben, Michael Cooper, Loretta Guarino Reid, and Gregg Vanderheiden: *Web Content Accessibility Guidelines (WCAG) 2.0*, Dec. 2008. http://www.w3.org/TR/WCAG20/.

[10] Campbell, Alastair, Michael Cooper, and Andrew Kirkpatrick: *Techniques for WCAG 2.1*, July 2019. https://www.w3.org/WAI/WCAG21/Techniques/.

[11] Campbell, Alastair, Michael Cooper, and Andrew Kirkpatrick: *Understanding WCAG 2.1*, July 2019. https://www.w3.org/WAI/WCAG21/Understanding/.

[12] Chrisholm, Wendy, Greg Venderheiden, and Ian Jacobs: *Web Content Accessibility Guidelines 1.0 W3C Recommendation 5-May-1999*, May 1999. https://www.w3.org/TR/WAI-WEBCONTENT/.

[13] Cooper, Michael: *Requirements for WCAG 2.0 Checklists and Techniques. W3C Working Draft 25 January 2005*. Technical report, W3C, 2005. https://www.w3.org/WAI/GL/WCAG20/WD-wcag2-tech-req-20050125.html.

[14] Cooper, Michael, Andrew Kirkpatrick, and Joshue O Connor: *Techniques for WCAG 2.0*, 2016. https://www.w3.org/TR/2016/NOTE-WCAG20-TECHS-20161007/.

[15] Diggs, Joanmarie, James Craig, Shane McCarron, and Michael Cooper: *Accessible Rich Internet Applications (WAI-ARIA) 1.1*. Technical report, W3C, December 2017. https://www.w3.org/TR/wai-aria-1.1/.

[16] Diggs, Joanmarie, Bryan Garaventa, and Michael Cooper: *Accessible Name and Description Computation 1.1*. Technical report, W3C, December 2018. https://www.w3.org/TR/accname-1.1/.

[17] Eggert, Eric and Shadi Abou-Zahra: *How to Meet WCAG (Quick Reference)*. Technical report, W3C, October 2019. https://www.w3.org/WAI/WCAG21/quickref/.

[18] Faulkner, Steve and Soctt O'Hara: *ARIA in HTML W3C Working Draft 06 August 2020*. Technical report, W3C, August 2020. https://www.w3.org/TR/2020/WD-html-aria-20200806/.

[19] Fiers, Wilco, Maureen Kraft, Mary Jo Mueller, and Shadi Abou-Zahra: *Acessibility Conformance Testing (ACT) Rules Format 1.0*. Technical report, W3C, December 2019. https://www.w3.org/TR/act-rules-format/.

[20] Frazão, Tânia and Carlos Duarte: *Comparing Accessibility Evaluation Plug-Ins*. In *Proceedings of the 17th International Web for All Conference*, W4A '20, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370561. https://doi.org/10.1145/3371300.3383346.

[21] Harband, Jordan and Kevin Smith: *ECMAScript 2020 Language Specification*. Technical report, ECMA, 2020. http://ecma-international.org/ecma-262/11.0/index.html.

[22] Harrocks, Ian, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*, 2004. https://www.w3.org/Submission/SWRL/.

[23] Hickson, Ian: *HTML5. A vocabulary and associated APIs for HTML and XHTML*, May 2011. http://www.w3.org/TR/html5/.

[24] Hitzler, Pascal, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph: *OWL 2 Web Ontology Language Primer*, Dec. 2012. https://www.w3.org/TR/owl2-primer/.

[25] *ISO/IEC 40500:2012 Information technology — W3C Web Content Accessibility Guidelines (WCAG) 2.0*, 2012. https://www.iso.org/standard/58625.html.

[26] Jr., Tab Atkins, Elika J. Etemad, and Florian Rivoal: *CSS Snapshot 2018W3C Working Group Note, 22 January 2019.* Technical report, W3C, 2019. https://www.w3.org/TR/CSS/.

[27] Kirkpatrick, Andrew, Joshue O. Connor, Alastair Campbell, and Michael Cooper: *Web Content Accessibility Guidelines (WCAG) 2.1*, June 2018. https://www.w3.org/TR/WCAG21/.

[28] Lim, Zui Young, Jia Min Chua, Kaiting Yang, Wei Shin Tan, and Yinn Chai: *Web Accessibility Testing for Singapore Government E-Services.* In *Proceedings of the 17th International Web for All Conference*, W4A '20, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370561. https://doi.org/10.1145/3371300.3383353.

[29] Nuñez, Almendra, Arturo Moquillaza, and Freddy Paz: *Web Accessibility Evaluation Methods: A Systematic Review.* In Marcus, Aaron and Wentao Wang (editors): *Design, User Experience, and Usability. Practice and Case Studies*, pages 226–237, Cham, 2019. Springer International Publishing, ISBN 978-3-030-23535-2. https://www.doi.org/10.1007/978-3-030-23535-2_17.

[30] Pelzetter, Jens: *A Knowledge-Based Framework for Improving Accessibility of Web Sites.* In Blomqvist, Eva, Diana Maynard, Aldo Gangemi, Rinke Hoekstra, Pascal Hitzler, and Olaf Hartig (editors): *The Semantic Web*, pages 226–235, Cham, 2017. Springer International Publishing, ISBN 978-3-319-58451-5. https://doi.org/10.1007/978-3-319-58451-5_17.

[31] Pelzetter, Jens: *Using Ontologies As a Foundation for Web Accessibility Tools.* In *Proceedings of the Internet of Accessible Things*, W4A '18, pages 26:1–26:2, New York, NY, USA, 2018. ACM, ISBN 978-1-4503-5651-0. http://doi.acm.org/10.1145/3192714.3196316.

[32] Pelzetter, Jens: *A Declarative Model for Accessibility Requirements.* In *Proceedings of the 17th International Web for All Conference*, W4A '20, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370561. https://doi.org/10.1145/3371300.3383339.

[33] Pelzetter, Jens: *A declarative model for web accessibility requirements and its implementation.* Frontiers in Computer Science, 2021, ISSN 2624-9898. https://doi.org/10.3389/fcomp.2021.605772.

[34] Richards, Jan, Jeanne Spellman, and Jutta Treviranus: *Authoring Tool Accessibility Guidelines (ATAG) 2.0*, 2015. https://www.w3.org/TR/ATAG20/.

[35] Sajka, Janina, Michael Cooper, Peter Korn, and Charles Hall: *Challenges with Accessibility Guideline Conformance and Testing and Approaches for Mitigating Them*. Technical report, W3C, June 2020. https://www.w3.org/TR/accessibility-conformance-challenges/, Working Draft 19 June 2020.

[36] Schiavone, Antonio Giovanni and Fabio Paternò: *An extensible environment for guideline-based accessibility evaluation of dynamic Web applications*. Universal Access in the Information Society, 14(1):111–132, March 2015, ISSN 1615-5297. https://doi.org/10.1007/s10209-014-0399-3.

[37] Spellman, Jeanne, Rachael Montgomery, Shawn Lauriat, and Michael Cooper: *W3C Accessibility Guidelines (WCAG) 3.0*. Technical report, W3C, September 2020. https://w3c.github.io/silver/guidelines/, W3C Editor's Draft 16 September 2020.

[38] Stewart, Simon and David Burns: *WebDriver Level 2 W3C Working Draft 24 November 2019*. Technical report, W3C, November 2019. https://www.w3.org/TR/webdriver/.

[39] Velasco, Carlos A, Shadi Abou-Zahra, and Johannes Koch: *Developer Guide for Evaluation and Report Language (EARL) 1.0*, February 2017. https://www.w3.org/TR/EARL10-Guide/.

[40] *Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung - BITV 2.0*, September 2011. https://www.gesetze-im-internet.de/bitv_2_0/BJNR184300011.html.

[41] *How to Meet WCAG 2.0. A customizable quick reference to Web Content Accessibility Guidelines (WCAG) 2.0 requirements (success criteria) and techniques*. https://www.w3.org/WAI/WCAG20/quickref/.

[42] WHATWG: *DOMLiving Standard*. Technical report, June 2020. https://dom.spec.whatwg.org/.