ORIGINAL RESEARCH
published: 04 March 2021
doi: 10.3389/fcomp.2021.605772

A Declarative Model for Web Accessibility Requirements and its Implementation

*Jens Pelzetter**

Faculty 3—Mathematics and Computer Science, University of Bremen, Bremen, Germany

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

OPEN ACCESS

Edited by:
Carlos Duarte,
University of Lisbon, Portugal

Reviewed by:
Fabio Paternò,
Italian National Research Council, Italy
J. Eduardo Pérez,
University of the Basque Country, Spain

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

Specialty section:
This article was submitted to Human-Media Interaction, a section of the journal Frontiers in Computer Science

Received: *13 September 2020*
Accepted: *06 January 2021*
Published: *04 March 2021*

Citation:
Pelzetter J (2021) A Declarative Model for Web Accessibility Requirements and its Implementation. Front. Comput. Sci. 3:605772. doi: 10.3389/fcomp.2021.605772

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 website is a time-consuming task that requires good expertize 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 article presents a declarative model for describing accessibility requirements for web pages according to the Web Content Accessibility Guidelines by the 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 article 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 article 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 ontology and a prototype of a tool using the declarative model are presented. **Section 5** discusses the results and future work.

This article is an extended version of a previously published article (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 the WCAG* also describe 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 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 the 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 the 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 the 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, and 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

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

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

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

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:

- 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 an 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 cannot 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 26–35% false positives and missed 26–35% 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, Brajnik et al., 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 article 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 article does not list all available conditions.

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 the 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 the WCAG (Campbell et al., 2019) does not contain information on when a technique is applicable. This information is provided in an additional document, *How to meet the 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 to 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 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 the 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]

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]https://www.totalvalidator.com.

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

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

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 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 to 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 unpredictable 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 or 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 (ARIA) (Diggs et al., 2017) recommendation 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, and 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, and 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 as follows:

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 to check whether an accessible name is available for an element:

```
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 as follows:

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 as follows:

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

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

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

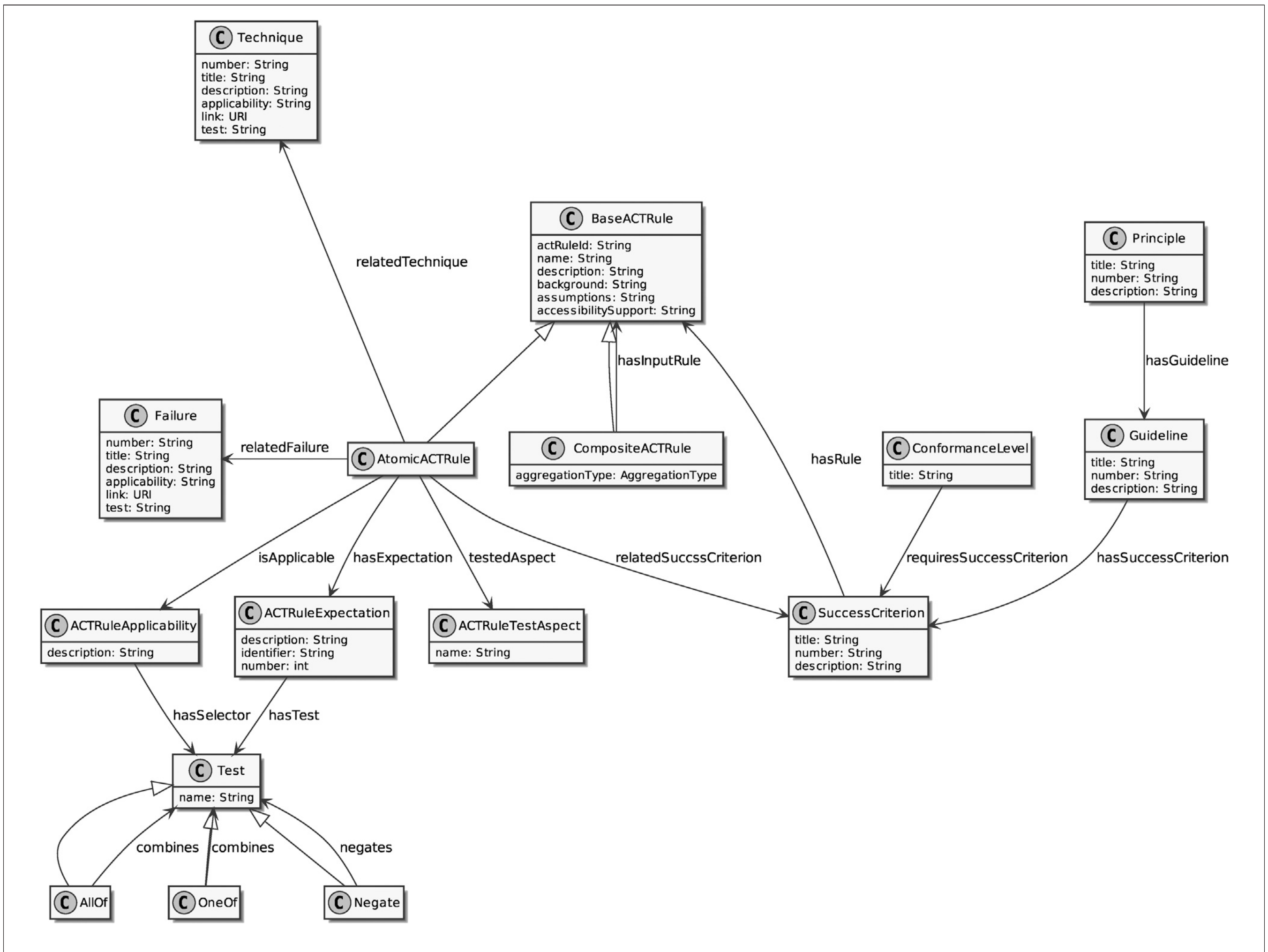


FIGURE 1 | UML diagram showing the primary classes of the *web-a11y-auditor* ontology.

These conditions can be expressed as
oneOf (hasAccessibleName()
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 article. 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

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

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.

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 an 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. 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] **Figure 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 *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

[10]https://nuxtjs.org/.
[11]https://selenium.dev/.
[12]https://activemq.apache.org/components/artemis/.

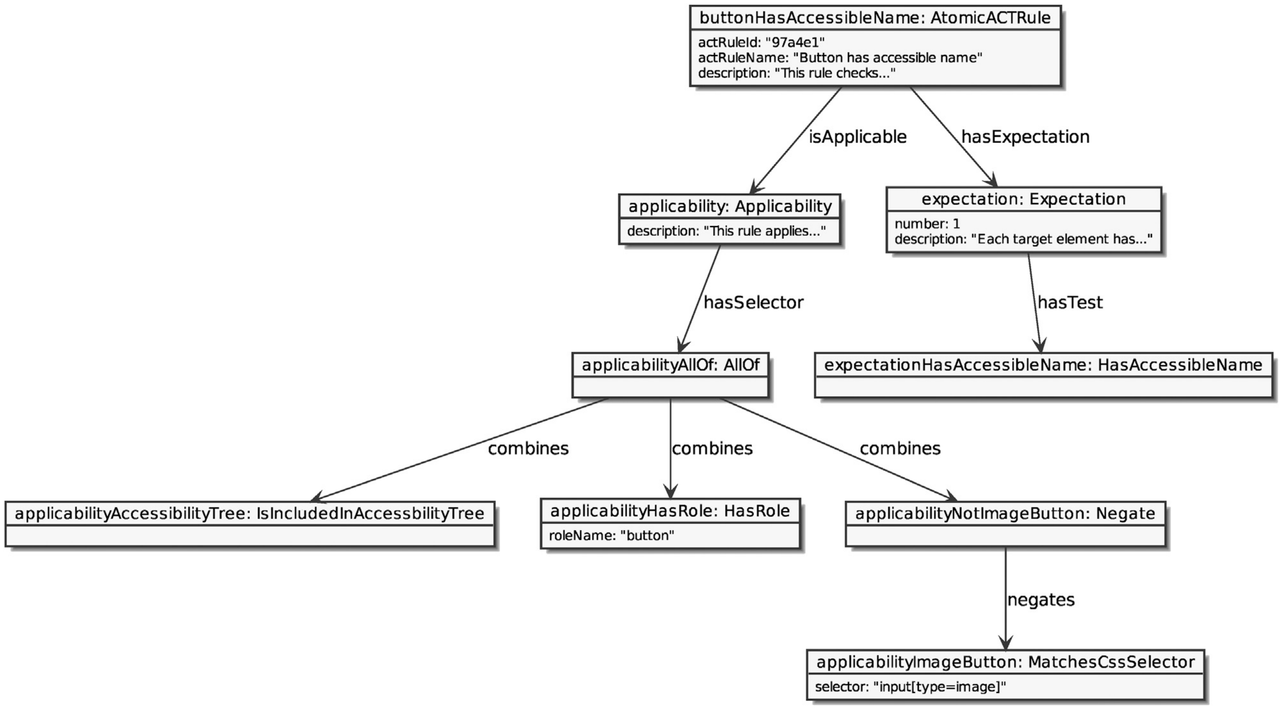

buttonHasAccessibleName: AtomicACTRule
actRuleId: "97a4e1"
actRuleName: "Button has accessible name"
description: "This rule checks..."
isApplicable
hasExpectation
applicability: Applicability
description: "This rule applies..."
expectation: Expectation
number: 1
description: "Each target element has..."
hasSelector
hasTest
applicabilityAllOf: AllOf
expectationHasAccessibleName: HasAccessibleName
combines
combines
combines
applicabilityAccessibilityTree: IsIncludedInAccessbilityTree
applicabilityHasRole: HasRole
roleName: "button"
applicabilityNotImageButton: Negate
negates
applicabilityImageButton: MatchesCssSelector
selector: "input[type=image]"

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

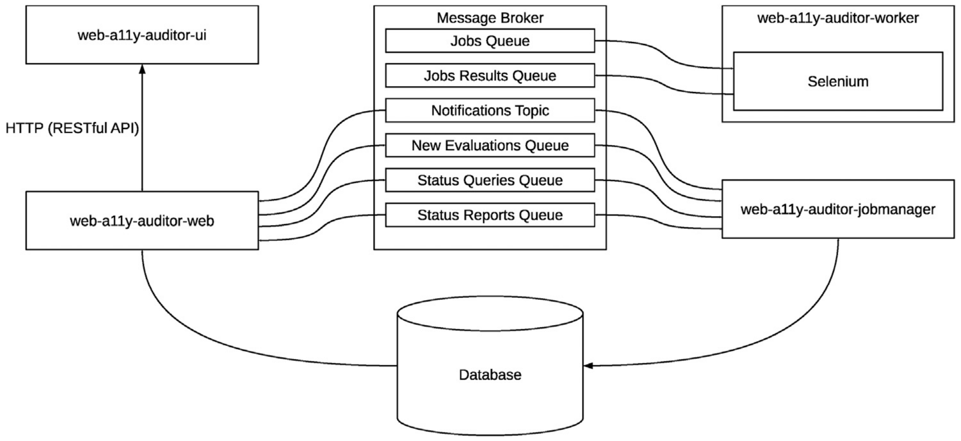


FIGURE 3 | Diagram showing the architecture of the *web-a11y-auditor*.

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 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 cannot 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?

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

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

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 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.

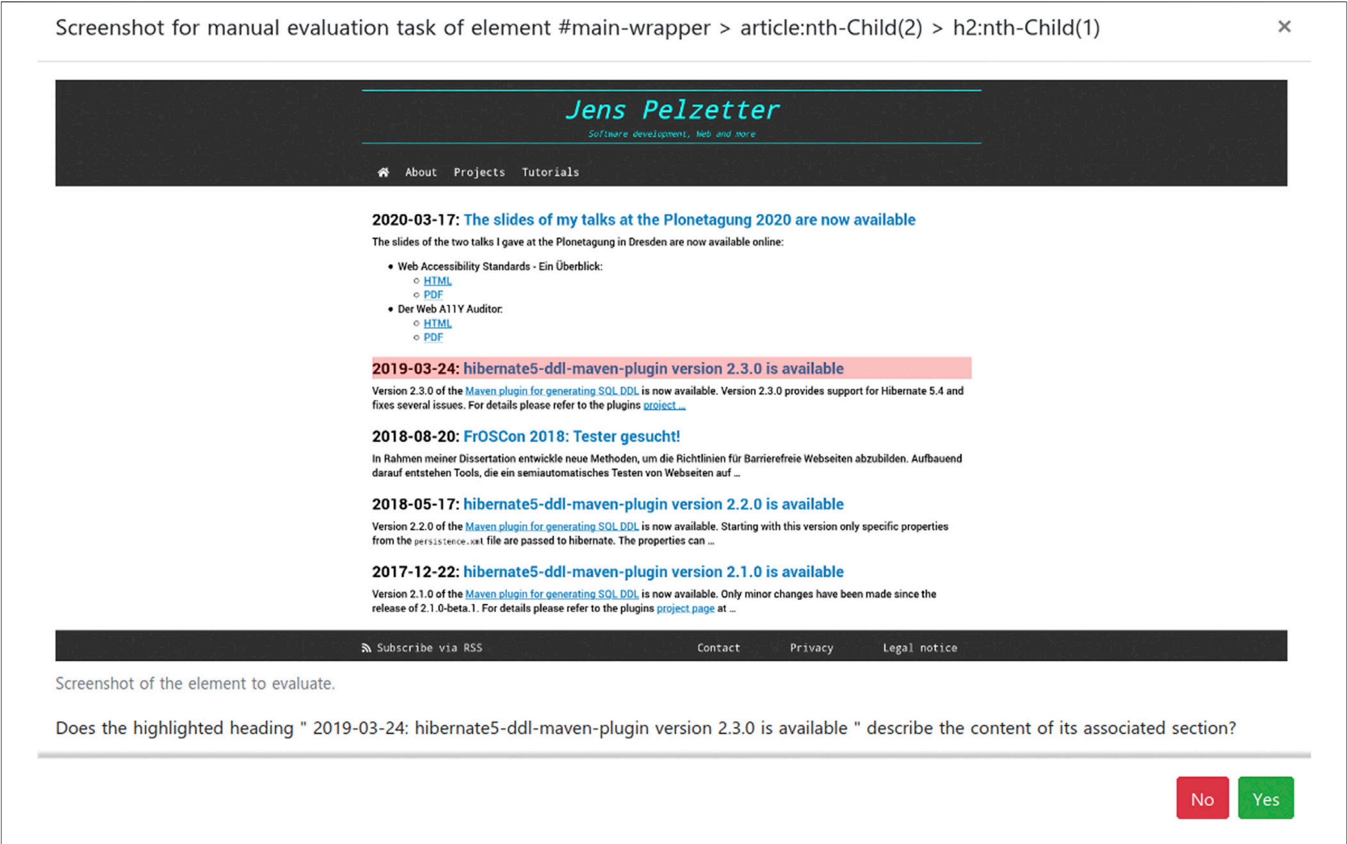
5 DISCUSSION

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 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 article 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 toward a DOM API (at least in the examples shown in the article). 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 article 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


Screenshot for manual evaluation task of element #main-wrapper > article:nth-Child(2) > h2:nth-Child(1)

×

Jens Pelzetter

Software development, Web and more

About Projects Tutorials

2020-03-17: The slides of my talks at the Plonetagung 2020 are now available

The slides of the two talks I gave at the Plonetagung in Dresden are now available online:

- Web Accessibility Standards - Ein Überblick:
  - HTML
  - PDF
- Der Web A11Y Auditor:
  - HTML
  - PDF

2019-03-24: hibernate5-ddl-maven-plugin version 2.3.0 is available

Version 2.3.0 of the Maven plugin for generating SQL DDL is now available. Version 2.3.0 provides support for Hibernate 5.4 and fixes several issues. For details please refer to the plugins project ...

2018-08-20: FrOSCon 2018: Tester gesucht!

In Rahmen meiner Dissertation entwickle neue Methoden, um die Richtlinien für Barrierefreie Webseiten abzubilden. Aufbauend darauf entstehen Tools, die ein semiautomatisches Testen von Webseiten auf ...

2018-05-17: hibernate5-ddl-maven-plugin version 2.2.0 is available

Version 2.2.0 of the Maven plugin for generating SQL DDL is now available. Starting with this version only specific properties from the `persistence.xml` file are passed to hibernate. The properties can ...

2017-12-22: hibernate5-ddl-maven-plugin version 2.1.0 is available

Version 2.1.0 of the Maven plugin for generating SQL DDL is now available. Only minor changes have been made since the release of 2.1.0-beta.1. For details please refer to the plugins project page at ...

Subscribe via RSS Contact Privacy Legal notice

Screenshot of the element to evaluate.

Does the highlighted heading " 2019-03-24: hibernate5-ddl-maven-plugin version 2.3.0 is available " describe the content of its associated section?

No Yes

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

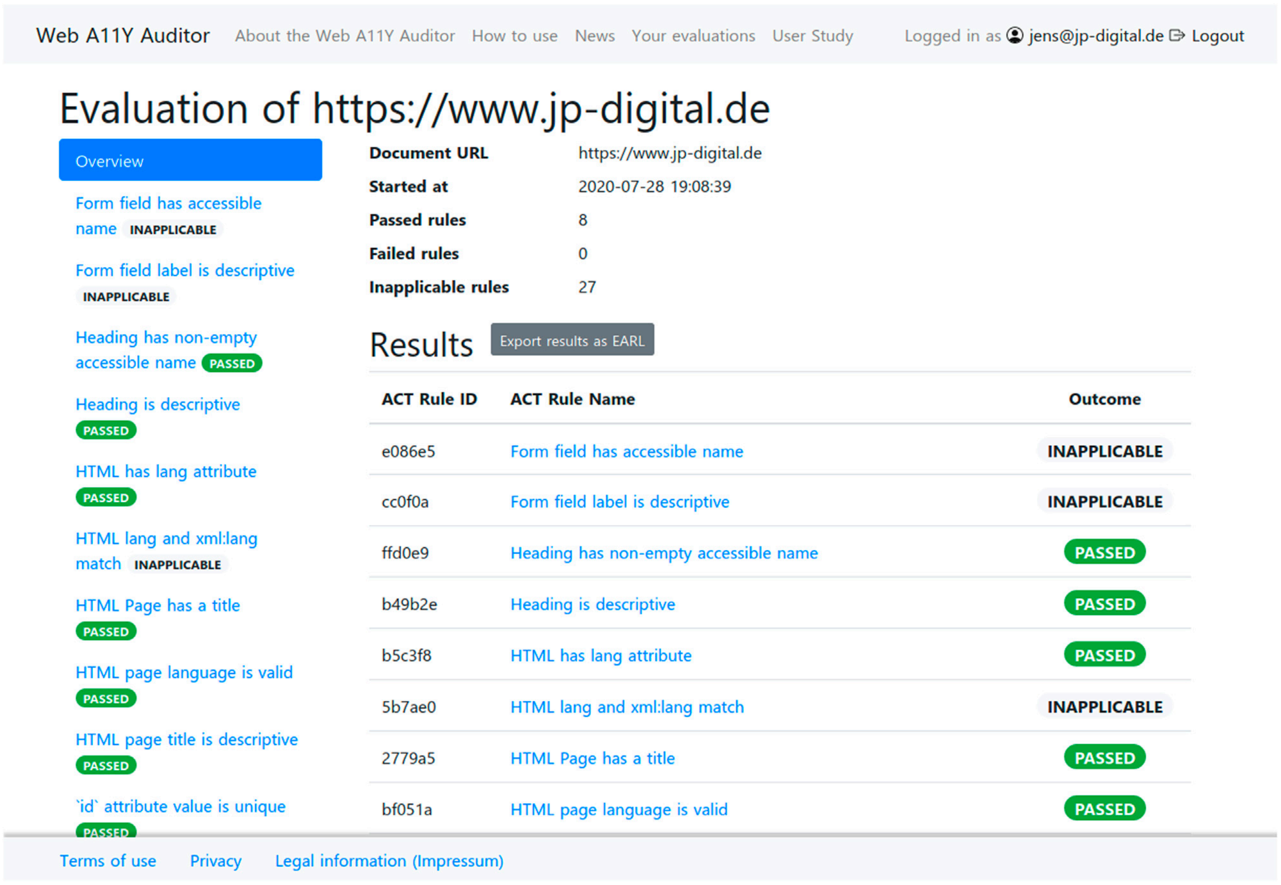
additional modeling. The second problem was performance. With only a few (less than 40 elements), the reasoning worked as expected. However, even small web pages often contain several hundred HTML elements. For each element, the applicable rules have to be determined. For each rule and element, an additional 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 17,500 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 CONCLUSION

In this article, a declarative model for accessibility requirements of web pages was presented. The foundations of this model are the 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.

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



Evaluation of https://www.jp-digital.de

- Form field has accessible name INAPPLICABLE
- Form field label is descriptive INAPPLICABLE
- Heading has non-empty accessible name PASSED
- Heading is descriptive PASSED
- HTML has lang attribute PASSED
- HTML lang and xml:lang match INAPPLICABLE
- HTML Page has a title PASSED
- HTML page language is valid PASSED
- HTML page title is descriptive PASSED
- `id` attribute value is unique PASSED

Document URL	https://www.jp-digital.de
Started at	2020-07-28 19:08:39
Passed rules	8
Failed rules	0
Inapplicable rules	27

Results

ACT Rule ID	ACT Rule Name	Outcome
e086e5	Form field has accessible name	INAPPLICABLE
cc0f0a	Form field label is descriptive	INAPPLICABLE
ffd0e9	Heading has non-empty accessible name	PASSED
b49b2e	Heading is descriptive	PASSED
b5c3f8	HTML has lang attribute	PASSED
5b7ae0	HTML lang and xml:lang match	INAPPLICABLE
2779a5	HTML Page has a title	PASSED
bf051a	HTML page language is valid	PASSED

FIGURE 5 | Results of an evaluation in the *web-a11y-auditor*.

The approach for modeling accessibility rules presented in this article 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 article 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.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. These data can be found here: https://ontologies.web-a11y-auditor.net.

AUTHOR CONTRIBUTIONS

The author confirms being the sole contributor of this work and has approved it for publication.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fcomp.2021.605772/full#supplementary-material.

REFERENCES

Abascal, J., Arrue, M., and Valencia, X. (2019). “Tools for web accessibility evaluation,” in *Web accessibility: a foundation for research*. Editors Y. Yesilada and S. Harper (London, United Kingdom: 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. World Wide Web Consortium (W3C). Available at: https://www.w3.org/TR/EARL10-Schema/.

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, Orlando, FL, October 25–27, 2010 (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 Trans. Accessible Comput. (TACCESS)* 4, 8. doi:10.1145/2141943.2141946

Broccia, G., Manca, M., Paternò, F., and Pulina, F. (2020). Flexible automatic support for web accessibility validation. *Proc. ACM Hum. Comput. Interact.* 4, 1. doi:10.1145/3397871

Caldwell, B., Cooper, M., Reid, L. G., and Vanderheiden, G. (2008). Web content accessibility guidelines (WCAG) 2.0. W3C, Tech. rep.

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

Diggs, J., Craig, J., McCarron, S., and Cooper, M. (2017). Accessible Rich Internet Applications (WAI-ARIA) 1.1. W3C, Tech. rep. Available at: https://www.w3.org/TR/wai-aria-1.1/.

Diggs, J., Garaventa, B., and Cooper, M. (2018). Accessible name and description computation 1.1. W3C, Tech. rep.

Eggert, E., and Abou-Zahra, S. (2019). How to meet WCAG (Quick reference). W3C, Tech. rep. Available at: https://www.w3.org/WAI/WCAG21/quickref/.

Faulkner, S., and O’Hara, S. (2020). ARIA in HTML W3C working draft 06 August 2020. W3C, Tech. rep. Available at: https://www.w3.org/TR/2020/WD-html-aria-20200810/.

Fiers, W., Kraft, M., Mueller, M. J., and Abou-Zahra, S. (2019). Acessibility conformance testing (ACT) rules format 1.0. W3C, Tech. rep.

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., and Rudolph, S. (2012). OWL 2 Web Ontology Language Primer (Second Edition), Tech. rep. World Wide Web Consortium (W3C). Available at: https://www.w3.org/TR/owl2-primer/.

ISO (2012). ISO/IEC 40500:2012 information technology — W3C web content accessibility guidelines (WCAG) 2.0. W3C, Tech. rep.

Kirkpatrick, A., Connor, J. O., Campbell, A., and Cooper, M. (2018). Web Content Accessibility Guidelines (WCAG) 2.1. W3C, Tech. rep. Available at: https://www.w3.org/TR/WCAG21/.

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*. Editors A. Marcus and W. Wang (Cham, Switzerland: Springer International Publishing), 226–237.

Pelzetter, J. (2020). “A declarative model for accessibility requirements,” in Proceedings of the 17th international web for all conference, Taipei, Taiwan, April 20–21, 2020 (New York, NY: Association for Computing Machinery). doi:10.1145/3371300.3383339

Schiavone, A. G., and Paternò, F. (2015). An extensible environment for guideline-based accessibility evaluation of dynamic web applications. *Univers. Access Inf. Soc.* 14, 111–132. doi:10.1007/s10209-014-0399-3

Stewart, S., and Burns, D. (2018). WebDriver W3C Recommendation 05 June 2018. W3C, Tech. rep. Available at: https://www.w3.org/TR/webdriver1/.

Stewart, S., and Burns, D. (2019). WebDriver level 2 W3C working draft 24 November 2019. W3C, Tech. rep. Available at: https://www.w3.org/TR/2019/WD-webdriver2-20191124/.

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. Available at: https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016L2102.

Velasco, C. A., Abou-Zahra, S., and Koch, J. (2017). Developer guide for evaluation and report language (EARL) 1.0, Tech. rep. Available at: https://www.w3.org/TR/EARL10-Guide/.

Verordnung (2011). Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behindertengleichstellungsgesetz (Barrierefreie Informationstechnik-Verordnung - BITV 2.0). Federal Republic of Germany. Available at: https://www.gesetze-im-internet.de/bitv_2_0/.

WHATWG (2020). DOM living standard, Tech. rep.

Conflict of Interest: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.