Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 157 (2006) 87–100
www.elsevier.com/locate/entcs

Web Accessibility Evaluation Tools: A Survey and Some Improvements

Vicente Luque Centeno[1] Carlos Delgado Kloos[2]
Jesús Arias Fisteus[3] Luis Álvarez Álvarez

Department of Telematic Engineering
Carlos III University of Madrid

Abstract

Web Content Accessibility Guidelines (WCAG) from W3C consist of a set of 65 checkpoints or specifications that Web pages should accomplish in order to be accessible to people with disabilities or using alternative browsers. Many of these 65 checkpoints can only be checked by a human operator, thus implying a very high cost for full evaluation. However, some checkpoints can be automatically evaluated, thus spotting accessibility barriers in a very effective manner. Well known tools like Bobby, Tawdis or WebXACT evaluate Web accessibility by using a mixture of automated, manual and semiautomated evaluations. However, the automation degree of these Web evaluations is not the same for each of these tools. Since WCAG are not written in a formalized manner, these evaluators may have different "interpretations" of what these rules mean. As a result, different evaluation results might be obtained for a single page depending on which evaluation tool is being used.
Taking into account that a fully automated Web accessibility evaluation is not feasible because of the nature of WCAG, this paper evaluates the WCAG automation coverage of some well known Web accessibility evaluation tools spotting their weaknesses and differences. We also provide a formalized specification for those checkpoints where these differences have been detected, thus challenging nowaday's tools for more automated coverage.

Keywords: Automated evaluation, WAI, WCAG, Web Accessibility, Web site evaluators

[1] Email: vlc@it.uc3m.es
[2] Email: cdk@it.uc3m.es
[3] Email: jaf@it.uc3m.es

doi:10.1016/j.entcs.2005.12.048

1. Introduction

Web Content Accessibility Guidelines (WCAG) 1.0 [1] were defined by W3C as a set of 14 guidelines and 65 checkpoints that Web pages should accomplish in order to be accessible to people with disabilities, people using alternative browsers, or even Web agents, as shown at [15]. However, they were defined using fuzzy, ambiguous and subjective terms not being so much focused on the underlying HTML format. As a result, many of those checkpoints require human judgement and they don't provide an objective criteria to be followed. This implies that it is not possible to build a fully automated accessibility evaluator only based on cheap-to-be-evaluated automatically conditions.

In order to fully determine Web accessibility, some semiautomated and manual checkpoints should also be evaluated. Thus, we can classify checkpoints into the four following groups.

(i) **Objectively automated** rules are clearly defined and clearly specify a condition that nobody might reject or misunderstand. Checking whether a set of well defined mandatory elements and attributes are present within the HTML markup is, for example, a typical checkpoint within this category.

(ii) **Subjectively automated** rules specify fuzzy conditions that can be automated, but they require some extra particular *non-fuzzy* interpretation might be accepted or rejected by different groups of people. For example, checking whether a text is *too long* is a subjective condition since it mentions a condition which is fuzzy to evaluate. Even though some people might agree that an alternative text longer than 150 characters might be classified as *too long*, some other people might not think so. It would be desirable that these extra particular *non-fuzzy* interpretation required by subjective rules could be customizable in evaluation tools, but they can't be easily customized because they are usually implemented as objectively automated rules. For these kind of subjective automatable conditions, W3C has defined a set of heuristics [2] that provide some help.

(iii) **Semi-automated** rules, which can not be evaluated automatically, but tool's assistance can focus user's interest on relevant markup. Thus, they require no user's focus if no element can trigger the condition's evaluation.

(iv) **Manual** rules, which require human judgement and can't be explicitly associated to any specific markup because they refer the document as a whole.

Both semi-automated and manual rules are very expensive to evaluate and

should be kept to a minimum. Automatable rules, even though subjectiveness, imply very cheap evaluations.

2. Comparison of Web Accessibility Evaluation Tools

One main problem of Web accessibility evaluation tools is that they provide a poor coverage of WCAG. Their behaviour consists on addressing only a few accessibility barriers on Web pages, thus they can not guarantee that accessibility is achieved, but they can guarantee that accessibility is faulty whenever a barrier is found. This weakness of nowadays' Web accessibility evaluation tools is mostly based on the nature of how WCAG were specified.

A major problem, however, is that we can easily find different evaluation results for a page depending on which tool is being used. The problem is that, while a checkpoint can be automated in one tool, it probably may be semi-automated or ignored in other tools. Even though they are considered in both tools, since each tool has its own *interpretation*, a condition triggered in a tool may easily not be triggered in other tools.

Several Web accessibility tools have been evaluated in this article in order to determine their automation coverage of WCAG 1.0, and some formalized rules are given as a specification (but also as implementation) for these tools.

2.1. Tools being evaluated

Web accessibility evaluation tools selected for this comparison are Bobby [12], Tawdis [13] (supported by the Spanish Ministry of Social Affairs) and WebXACT [14] (from the same company of Bobby). We have chosen these because they are well-known and can be used online for free. We know of no other similar evaluation tools (except Torquemada [18], which, sadly, is no longer available) that provide such final results without human intervention.

2.2. Checkpoints Vs Checkpoints' conditions

WCAG are very heterogeneous. Many of them imply a condition that only human judgement may evaluate. Some of them imply a set of several lower level conditions. In order to properly evaluate, we defined the WCAG checkpoints' conditions, a set of 103 conditions that perform some more detailed evaluation that the original 65 WCAG conditions. A WCAG condition may imply 1 or several WCAG checkpoint's conditions. For example, checkpoint WCAG 4.1 (Clearly identify changes in the natural language) might imply checkpoint's conditions WCAG 4.1a (Use `xml:lang` attribute for pieces of text in a different language within the same document) and WCAG 4.1b (Use `hreflang`

attribute on links pointing to documents in a different language).

2.3. Automated coverage of checkpoints evaluation

Table 1 indicates how the 65 WCAG checkpoints can be classified into the categories previously mentioned according to each tool. Both semi-automated and manual rules imply human evaluation cost, so our interest is focused on automated rules mainly. From a total of 14 possible objectively automated checkpoints, WebXACT detects 5 of them, Bobby only detects (completely) 4, and only a single one is detected by Taw. At least 14 - 5 = 9 objectively automatable checkpoints, some of them provided in the following sections, are not completely implemented by any analyzed tool. Subjectively automated rules are used in these tools as semi-automated, because the user has no ability to specify preferences for those subjective conditions, thus requiring extra human intervention.

	Theoretical	**Bobby**	**Taw**	**WebXACT**
Objectively automated	14	4	1	5
Subjectively automated	2	0	0	0
Semi-automated	31	33	20	33
Purely manual	18	28	44	27
Total	**65**	**65**	**65**	**65**

Table 1
Comparison of automation of WCAG checkpoints

Table 2 indicates the same as table 1 , but looking at our 103 WCAG checkpoints' conditions. The number of objectively automated conditions reveals the *hidden* value of tools like WebXACT and Bobby, having 25 and 24 objectively automated conditions respectively. It becomes rather difficult to pass all those tests for a Web page whose design was taken without accessibility in mind, so this number is enough to provide the feeling that accessibility is difficult to achieve by a random page. But this number is still far from the 43 possible conditions that can have, in fact, objectively automatable evaluations. This implies that at least 43 - 25 = 18 objectively automated conditions can not be fully evaluated by any of our tools. The rest is only usable for partial evaluation. Taw has a poor result of 10 rules from a set of 43 (thus validating a significant number of Web pages).

	Theoretical	Bobby	Taw	WebXACT
Objectively automated	43	24	10	25
Subjectively automated	11	0	0	0
Semi-automated	21	30	18	30
Purely manual	28	49	75	48
Total	**103**	**103**	**103**	**103**

Table 2
Comparison of automation of WCAG checkpoints' conditions

3. Checkpoints having similar behaviours

From tables 1 and 2 , we determine that, even though our tools might have similar behaviours for some checkpoints, some other checkpoints clearly have different behaviours. Table 3 shows some checkpoint conditions having the same automated behaviour on all evaluated tools. As expected, the most well known rule 1.1a (provide `alt` attribute for images), is within this set. The second column of table 3 provides a formalized XPath 1.0 [3] and XPath 2.0 [4] expression that can be used to address all the HTML elements breaking the rule within that page. The condition to check if the rule is broken consists on comparing the result of these expressions against the empty set. If equal, then no barrier is found. For example, rule for 1.1a addresses all images that have no `alt` attribute. If the result if this expression is the empty set, rule 1.1a is fulfilled. Rule for 1.1b addresses all image buttons without an alternative text, rule for 1.1e addresses all framesets not providing a `noframes` section, ...

WCAG #	Formalized rule	Bobby,WebXACT,Taw
1.1a	//img[not(@alt)]	Auto Obj.
1.1b	//input[@type="image"][not(@alt)]	Auto Obj.
1.1e	//frameset[not(noframes)]	Auto Obj.
1.5	//area[not(@alt)]	Auto Obj.
7.2a	//blink	Auto Obj.
7.3a	//marquee	Auto Obj.
13.2b	//head[not(title)]	Auto Obj.

Table 3
Some rules with same behaviour in automated evaluators

Rule 3.2: Validate document against a public grammar

According to [16], Web accessibility evaluators are not XHTML [7] validators. Thus, none of our selected tools provide support for validation against a well known DTD or XML Schema, because there are specific tools (like [11]) for this task and because accessibility is more than just XHTML validation (and because their authors wanted to make accessibility still applyable for non validated documents). However, we must say that rule 3.2 from WCAG probably guarantees more accessibility than any other rule by itself, because several WCAG rules are partially based on statements that are automatically achieved whenever a document is validated against a DTD or XML Schema. In fact, we have found that, depending on the *flavour* of XHTML being used, accessibility might be easier (or more difficult) to be achieved. We have found that XHTML 2.0 [10] provides more accessibility features than XHTML Basic 1.0 [8], which itself provides more accessibility features than XHTML 1.1 [9].

Examples of WCAG checkpoints achieved when a proper DTD or XML Schema is chosen and validated against, are:

- Rule 3.3: Use style sheets to control layout and presentation
- Rule 3.6: Mark up lists and list items properly
- Rule 11.2: Avoid deprecated features of W3C technologies

4. Checkpoints having different results

Tables 4 and 5 provide a summary of the most important differences on WCAG's coverage in our evaluated tools. Each row corresponds to an (objectively or subjectively) fully automatable rule. However, WebXACT and Bobby only provide a fully automated evaluation on rules 4.3, 9.2a-c, 10.4a-d, 10.5, 12.4b and 13.1a, leaving the others as semi-automated or manual, despite they could be fully automated. Rules marked up as "Pseudo" refer to rules that detected a problem only under particular conditions, but could not evaluate with the same accuracy as the formalized rule.

Table 4 includes W3C-standards-based rules for specifying when to use the `longdesc` attribute (rule 1.1c), when heading elements are used properly (rules 3.5), whether document's idiom is specified (rule 4.3), and the detection of some broken links (rule 6.3b), among others. Table 5 indicates that frames should have a description (`title` attribute). However, though this is easily automatable, neither Bobby nor Taw require it so. This is the only improvement we have found on WebXACT if compared to Bobby.

WCAG #	Formalized rule	Bobby,WebXACT	Taw
1.1c	//img[*toolong_alt*(@alt)][not(@longdesc)]	Semi	Manual
3.5a	//h2[not(preceding::h1)]	Pseudo	Manual
3.5b	See fig 1	Pseudo	Manual
3.5c	See fig 2	Pseudo	Manual
3.5d	See fig 3	Pseudo	Manual
3.5e	See fig 4	Pseudo	Manual
4.3	//html[not(@xml:lang)]	Auto Obj.	Manual
6.4a	//*[@onmouseover != @onfocus]	Semi	Manual
6.4b	//*[@onmouseout != @onblur]	Semi	Manual
7.4, 7.5	//meta[@http-equiv="refresh"]	Semi	Manual
9.2a	//*[@onmousedown != @onkeydown]	Auto Obj.	Manual
9.2b	//*[@onmouseup != @onkeyup]	Auto Obj.	Manual
9.2c	//*[@onclick != @onkeypress]	Auto Obj.	Manual
10.4a	See fig 9	Auto Obj.	Manual
10.4b	//textarea[normalize-space(text())=""]	Auto Obj.	Manual
10.4c	See fig 10	Auto Obj.	Manual
10.4d	See fig 11	Auto Obj.	Manual
10.5	See fig 12	Auto Obj.	Manual
12.4b	See fig 16	Auto Obj.	Manual
13.1a	See fig 17	Auto Obj.	Manual

Table 4
Rules with different behaviour in automated evaluators: WebXACT and Bobby Vs Taw

WCAG #	Formalized rule	WebXACT	Bobby,Taw
12.1	//frame[not(@title)]	Auto Obj.	Manual

Table 5
Rules with different behaviour in automated evaluators: WebXACT Vs Bobby and Taw

On the other hand, table 6 shows a sadly common poor behaviour of all analyzed tools with respect to fully automatable rules, that, however, are improperly treated as manual or semi-automated. It is quite surprising that very simple checkings like Javascript being found on improper attributes (rule 6.3b) or improper `tabindex` or `accesskey` attributes are not detected by any analyzed tool.

WCAG #	Formalized rule	Theory	Tools
3.5f	See fig 5	Auto Subj.	Manual
4.2a	See fig 6	Auto Subj.	Manual
6.3b	//a[starts-with(@href,"javascript:")]	Auto Obj.	Manual
6.3b	//area[starts-with(@href,"javascript:")]	Auto Obj.	Manual
9.4	See fig 7	Auto Obj.	Manual
9.5	See fig 8	Auto Obj.	Manual
10.1a	//*[@target="_blank"]	Auto Obj.	Semi
10.1a	//*[@target="_new"]	Auto Obj.	Semi
12.2	//frame[*toolong*(@title)][not(@longdesc)]	Auto Subj.	Semi
12.3a	See fig 13	Auto Subj.	Semi
12.3b	See fig 14	Auto Subj.	Semi
12.3c	//p[*toolong_p*(text())]	Auto Subj.	Manual
12.4a	See fig 15	Auto Obj.	Manual

Table 6
Rules with common poor behaviour in automated evaluators

Table 6 includes rules for specifying when keyboard and mouse dependant handlers are not paired (rule 9.2), restricting pop-ups and other emerging windows appearances (rule 10.1), when frames provide no textual description (rule 12.1) and some others that are detailed in the following figures (referred from tables 4 , 5 and 6).

Rule 3.5: Use headings properly

Figure 1 specifies an XQuery 1.0 [5] addressing expression for all non properly used `h3` elements. It is necessary that a `h1` and `h2` properly precede any `h3` element. We have sadly found that no evaluated tool requires such

proper heading usage, though W3C recommendation invites us to use headings properly and consistently and recommends using consecutive headings (not jumping from a `h1` to a `h3` without passing by a `h2`).

```
//h3[let $h3:=self::h3 return
let $h2:=$h3/preceding::h2[last()] return
let $h1:=$h3/preceding::h1[last()] return
$h1=() or $h2=() or $h1>>$h2]
```

Fig. 1. XQuery 1.0 expression for h3 elements breaking WCAG **3.5b**

Figures 2 , 3 and 4 provide addressing expressions for improper `h4`, `h5` and `h6` elements respectively.

```
//h4[let $h4:=self::h4 return
let $h3:=$h4/preceding::h3[last()] return
let $h2:=$h4/preceding::h2[last()] return
let $h1:=$h4/preceding::h1[last()] return
$h1=() or $h2=() or $h3=() or
$h1>>$h2 or $h2>>$h3 or $h3>>$h4]
```

Fig. 2. XQuery 1.0 expression for h4 elements breaking WCAG **3.5c**

```
//h5[let $h5:=self::h5 return
let $h4:=$h5/preceding::h4[last()] return
let $h3:=$h5/preceding::h3[last()] return
let $h2:=$h5/preceding::h2[last()] return
let $h1:=$h5/preceding::h1[last()] return
$h1=() or $h2=() or $h3=() or
$h4=() or $h1>>$h2 or $h2>>$h3 or
$h3>>$h4 or $h4>>$h5]
```

Fig. 3. XQuery 1.0 expression for h6 elements breaking WCAG **3.5d**

Header elements should not be too long, as defined in figure 5 .

Rule 4.2: Use proper abbreviations and acronyms

Abbreviations and acronyms should not be used inconsistently. They should have *semantic* and unique definitions. Though *semantic* value can not be easily addressed automatically, *uniqueness* surely can be. Figure 6 detects all abbreviations and acronyms that provide more different definitions for a single given text. Rule 4.2a, however, is not enough to guarantee rule 4.2, since all possible abbreviations and acronyms should be properly marked up.

```
//h6[let $h6:=self::h6 return
let $h5:=$h6/preceding::h5[last()] return
let $h4:=$h6/preceding::h4[last()] return
let $h3:=$h6/preceding::h3[last()] return
let $h2:=$h6/preceding::h2[last()] return
let $h1:=$h6/preceding::h1[last()] return
$h1=() or $h2=() or $h3=() or
$h4=() or $h5=() or $h1>>$h2 or
$h2>>$h3 or $h3>>$h4 or
$h4>>$h5 or $h5>>$h6]
```

Fig. 4. XQuery 1.0 expression for h6 elements breaking WCAG **3.5e**

```
(//h1|//h2|//h3|//h4|//h5|//h6)[toolong_h(text())]
```

Fig. 5. XQuery 1.0 expression for header elements breaking WCAG **3.5f**

```
(//abbr | //acronym)[let $a:=self::node() return
count((//abbr | //acronym)[text() = $a/text()]) != 1]
```

Fig. 6. XQuery expression for abbr and acronym elements breaking WCAG **4.2a**

Rule 9.4: Specify a proper tab order

Whenever a tabulation order being different from the default is being specified, `tabindex` attributes should be used consistently. Figure 7 spots all elements that have a `tabindex` attribute which is not a proper number or which is shared by several elements (it should be used uniquely).

```
//*[@tabindex][let $n:=self::node()/@tabindex return
not(isnumber($n)) or count(//*[@tabindex=$n]) != 1 or
number($n)<1 or number($n)>count(//*[@tabindex])]
```

Fig. 7. XQuery expression for elements breaking WCAG **9.4**

Rule 9.5: Provide proper keyboard shortcuts

Whenever keyboard shortcuts are being specified, `accesskey` attributes should be used consistently. Figure 8 spots all elements that have an `accesskey` attribute which is not a proper character or which is shared by several elements (it should be used uniquely).

```
//*[@accesskey][let $c:=self::node()/@accesskey return
not(ischar($c)) or count(//*[@accesskey=$c]) != 1]
```

Fig. 8. XQuery expression for elements breaking WCAG **9.5**

Rule 10.4: Include default, place-holding characters in form fields

Editable (not hidden) form fields different from text areas (i.e, text fields, radio or checkbox buttons) that have null or no default values are detected by the XQuery addressing expression from figure 9 . Empty text areas are detected by the expression of 10.4b from table 6 .

```
//input[@type!="hidden"][not(@value) or @value=""]
```

Fig. 9. XQuery expression for input elements breaking WCAG **10.4a**

Select boxes having selected option by default are detected by expression from figure 10 . Figure 11 addresses sets of radio buttons where no option has been checked by default.

```
//select[not(@multiple)][not(.//option[@selected])]
```

Fig. 10. XQuery expression for select elements breaking WCAG **10.4c**

```
//input[@type="radio"][let $n:=self::input/@name return
not(ancestor::form//input[@name=$n][@checked])]
```

Fig. 11. XQuery expression for radio buttons breaking WCAG **10.4d**

Rule 10.5: Include non-link, printable characters between adjacent links

Figure 12 addresses all consecutive links that have nothing more that white spaces between them. For this rule we use a combination of both XQuery and XPointer [6].

```
//a[position() < last()][let $a:=self::a return
normalize-space((end-point($a)/range-to($a/following::a))/text())=""]
```

Fig. 12. XPointer + XQuery based rule for WCAG **10.5**

Rule 12.3: Divide large blocks of information into more manageable groups

The `fieldset` elements is highly recommended for forms that have several form fields. It can be used to group several of them which are semantically related within a same group, thus providing more manageable groups of form fields. Though it could be a subjectively automated rule (*toomany_inputs* should be defined with some subjective criteria), both Bobby and Taw treat this rule as semi-automated, asking a human evaluator whenever a form addressable by expression from figure 13 is found. The same applies for rule 12.3b from figure 14 (for non grouped options within a select).

```
//form[toomany_inputs(.//input)][not(.//fieldset)]
```

Fig. 13. XQuery expression for forms breaking WCAG **12.3a**

```
//select[toomany_options(option)][not(optgroup)]
```

Fig. 14. XQuery expression for select elements breaking WCAG **12.3b**

Rule 12.4: Associate labels explicitly with their controls

This rule implies two different conditions. Figure 15 addresses all labels which are not properly used, i.e., that point to more than a single control (rule 12.4a). Vice-versa, all form fields should have no more than a single label (rule 12.4b). Otherwise, they can be detected by expression from figure 16 . If both rules 12.4a and 12.4b are followed, there exists a bijective mapping between labels and form fields, as desirable.

```
//label[let $l:=self::label return
count((//select|//input|//textarea)[@id=$l/@for]) != 1]
```

Fig. 15. XQuery expression for labels breaking WCAG **12.4a**

```
(//select|//textarea|//input[@type="text" or @type="password" or
@type="radio" or @type="checkbox"])[let $ff:=self::node() return
count(//label[@for=$ff/@id]) != 1]
```

Fig. 16. XQuery expression for form fields breaking WCAG **12.4b**

Rule 13.1: Clearly identify the target of each link

It is not possible to automatically determine whether the text of a link can be understood by itself when read out of context (which could be rule 13.1b). However, we still can determine if there two links are used in a confusing manner. Figure 17 contains a XQuery expression for finding all links that, having same text and descriptions, point to different targets.

```
(//a | //area)[let $a:=self::node() return
(//a | //area)[@title = $a/@title and
text() = $a/text() and @href != $a/@href] != ()]
```

Fig. 17. XQuery expression for links breaking WCAG **13.1a**

5. Conclusions and future work

As a result of this evaluation, we can determine that it is easier to detect accessibility barriers using Bobby or WebXACT than using Taw. However, Bobby

and WebXACT do not provide as much coverage as we could expect. In fact, both tools are far away from having a good automated coverage of WCAG. It is also easy to explain that all elements from table 3 are all objectively automated. Objectivity leads into avoiding misinterpretations.

We have also found that conditions not addressable with XQuery expressions rely out of the HTML markup, i.e., CSS files or manual checkpoints. As a result we have decided to implement an accessibility evaluation tool only implemented with:

(i) A XQuery engine

(ii) Our XQuery-based ruleset

(iii) Some customizable preferences (for addressing subjectivity)

(iv) Some little integration code on top of it, inside a servlet

This Web evaluator tool will have a better coverage and we hope it will not be very expensive to build.

Normalization is also an important focus, so that we don't have multiple rulesets for accessibility depending on which country we are. We hope that a formalization attempt like ours may help in that direction. Comparison of WCAG 1.0 and Bobby's implementation of U.S. Section 508 Guidelines [17] has lead us to conclude that both rules have a lot of common rules and minor changes should be added to WCAG 1.0 to include this normative.

There are few tools freely available for accessibility evaluation. Torquemada is no longer operative and WebXACT is offered by Bobby's developers, something which explains why WebXACT only offers a minor enhancement from Bobby. Anyway, different misinterpretations of W3C's WCAG will occur on future implementations as long as there is no common formalized interpretation of these rules. Our proposal is to give some light in this process. That's why we have redefined the set of 65 WCAG into a set of 103 rules which are more focused on low level details than those of WCAG.

6. Acknowledgements

The work reported in this paper has been partially funded by the projects INFOFLEX *TIC2003-07208* and SIEMPRE *TIC2002-03635* of the Spanish Ministry of Science and Research.

References

[1] W3C *Web Content Accessibility Guidelines 1.0*
www.w3.org/TR/WCAG10

[2] W3C *Techniques For Accessibility Evaluation And Repair Tools W3C Working Draft, 26 April 2000*
www.w3.org/TR/AERT

[3] W3C *XML Path Language (XPath) Version 1.0 W3C Recommendation 16 November 1999*
www.w3.org/TR/xpath

[4] W3C *XML Path Language (XPath) 2.0 W3C Working Draft 29 October 2004*
www.w3.org/TR/xpath20

[5] W3C *XQuery 1.0: An XML Query Language W3C Working Draft 29 October 2004*
www.w3.org/TR/xquery

[6] W3C *XML Pointer Language (XPointer), W3C Working Draft 16 August 2002*
www.w3.org/TR/xptr

[7] W3C *XHTML 1.0 TM The Extensible HyperText Markup Language (Second Edition), A Reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 January 2000, revised 1 August 2002*
www.w3.org/TR/xhtml1

[8] W3C *XHTML Basic W3C Recommendation 19 December 2000*
www.w3.org/TR/xhtml-basic

[9] W3C *XHTML TM 1.1 - Module-based XHTML, W3C Recommendation 31 May 2001*
www.w3.org/TR/xhtml11

[10] W3C *XHTML TM 2.0, W3C Working Draft 22 July 2004*
www.w3.org/TR/xhtml2

[11] W3C *Markup Validation Service*
validator.w3.org

[12] Watchfire *Bobby Accessibility tool*
bobby.watchfire.com/bobby/html/en/index.jsp

[13] CEAPAT, Fundación CTIC, Spanish Ministry of Employment and Social Affairs (IMSERSO) *Online Web accessibility test*
www.tawdis.net

[14] Watchfire *WebXACT*
webxact.watchfire.com

[15] Vicente Luque Centeno, Carlos Delgado Kloos, Luis Sánchez Fernández, Norberto Fernández García *Device independence for Web Wrapper Agents*
Workshop on Device Independent Web Engineering (DIWE'04) 26 July 2004, Munich

[16] Peter Blair *A Review of Free, Online Accessibility Tools*
www.webaim.org/techniques/articles/freetools, February 2004

[17] Center for IT Accommodation (CITA) *U.S. Section 508 Guidelines*
www.section508.gov

[18] Fondazione Ugo Bordoni *Torquemada, Web for all*
www.webxtutti.it/testa_en.htm