Noname manuscript No.
(will be inserted by the editor)

Widget Identification: A High-Level Approach to Accessibility

Alex Q. Chen · Simon Harper · Darren Lunn · Andrew Brown

Received: date / Accepted: date

Abstract The Web 2.0 sees once static pages evolve into hybrid applications, and content that was previously simple, now becoming increasingly complicated due to the many updating components located throughout the page. While beneficial for some users, these components (widgets) are often complex and will lead to confusion and frustration for others, notably those for whom accessibility is already an issue. While users and developers often perceive widgets as complete components (a Slideshow, or an Auto Suggest List), they are in-fact heterogeneous collections of code, and are therefore hard to computationally identify. Identification is critical if we wish to reverse engineer inaccessible widgets or 'inject' missing 'WAI-ARIA' into 'RIAs'. In this case, we introduce a technique that analyses the code associated with a Web page to identify widgets using combinations of code constructs which enable uniquely identification. We go on to technically evaluate our approach with the most difficult widgets to distinguish between – Slideshows and Carousels – and then describe two prototype applications for visually impaired and older users by means of example.

Keywords Web 2.0 · Widget Identification · Widget Classification · Disabilities · Ageing

Alex Qiang Chen
School of Computer Science, Information Management Group, The University of Manchester, Kilburn Building, Oxford Road, Manchester M13 9P, UK
E-mail: chenqa@cs.manchester.ac.uk

Simon Harper
E-mail: simon.harper@manchester.ac.uk

Darren Lunn
E-mail: darren.lunn@cs.manchester.ac.uk

Andrew Brown
E-mail: andrew.brown@cs.manchester.ac.uk

1 Introduction

The World Wide Web (Web) is undergoing a profound change from version 1.0 to version 2.0. There is no precise definition of what this new Web is, but it has an emphasis on interaction, community, and the active contribution of users; while at a technical level it is generally accepted that Web 2.0 technologies are based around Asynchronous JavaScript and XML (AJAX) and now HTML5 technologies. Components, called widgets, built from these technologies enable small sections of the page to be updated as opposed to requiring a full page refresh, are extensively used in Web 2.0 sites. While this can provide a richer Web experience for a majority of users, for some the addition of dynamic content can be problematic. For users who are not familiar with Web concepts, or for those who require assistive technologies to access Web content, the increased complexity can prove to be a hindrance that detracts from the benefits that the Web can provide.

The shift in the way the Web works come with a corresponding increase in the cognitive load required to understand and interact with it [26]. This additional load can produce lower performance and higher levels of frustration which negatively effect both work and social activity, especially for those users that may not be experienced with using the Web or may require assistive technologies to access Web content, as discussed in §2. Older users, for example, experience a heightened cautiousness and a hesitancy about making responses that may be incorrect [22], while users with visually impairments often receive insufficient or inappropriate feedback from screen readers [3].

One approach to support such users is to provide tools that will assist them as they interact with Web 2.0 content. However, before such tools can provide meaningful functionality, the type of content delivery method used by the developer must first be identified from the content in the page. In this paper, we present a technique that analyses the combinations of tell-signs[1] to identify the types of widgets that are present in the page. §3 describes how these widgets can be formally classified using an ontology that matches tell-signs to characterise the high-level widgets that users interact with.

Next we ascertain the most popular widgets and their extent (see §4) arriving at nine distinct instances. We continue §5 by describing the identification process for just the 'Carousel' and 'Slideshow' as these posses similar properties and are therefore the hardest to reliably identify. A technical evaluation of the method is provided in §6. By using such identification techniques, prototype tools (see §7) are being developed that assist both visually impaired and older Web users as they interact with Web 2.0 content.

[1] Tell-signs are the attributes or properties of a widget, or its component parts, that indicate the presence of that widget on a page. The term originates from hunting, where a tell-sign (e.g., tracks) indicates the presence of the quarry.

2 Related Work

Studies have demonstrated that dynamic content attracts a user's attention more than static content [5]. However, Web 2.0 pages often have multiple areas of dynamic content. When users are faced with differing types of content, they have difficulty in dividing their attention between the elements to complete tasks effectively [24]. This additional load is a major problem for an ageing population of knowledge workers expected to work longer into old age. The general effects of ageing include changes in attention, cognition and behaviour, all of which affect how people use the Web [16]. Studies have shown that elderly Web users experience a heightened cautiousness and a hesitancy about making responses that may be incorrect [20][22]. In addition, elderly users show difficulty in maintaining attention, focus, and concentration on tasks where there is a lot of distracting information [18].

For visually impaired users, advancements in content delivery raise many new issues previously not seen [15]. These people often use assistive technologies, such as screen readers, to access Web content [1], but a review of how users of different assistive technologies deal with dynamically updating pages [2] showed that older browser/screen reader combinations did not respond to some updates, in particular those which occurred automatically. Even newer technologies often did not notify the user that the page had changed.

One approach to the problem of accessibility is to semantically annotate widgets so that tools can identify what content is present in the page and the role of that content. To achieve this annotation, the World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) has developed Accessible Rich Internet Applications (WAI-ARIA) [10]. WAI-ARIA is a mechanism that allows developers to enhance the accessibility of rich content contained within their Website by defining the roles, states, and properties of interactive elements contained within the page [25]. The annotations allow the roles of components to be made more explicit, and enable controls to be related to the content they change. While this approach has merit, from the perspective of people using the Web today, it suffers from two flaws. The first is the limited use of markup on new and existing sites; the second is that WAI-ARIA is generally concerned with making the components accessible, and appears to be less well suited for helping users understand widgets as coherent objects. Our approach aims to provide a solution that may work in tandem with WAI-ARIA to address both of these limitations by using code analysis to automatically identify widgets, thereby enabling them to be presented holistically.

There have been limited previous attempts at automatic widget detection. Miyashita et al. [23] developed a method of making media players accessible by identifying interface components such as play and stop buttons, and the volume slider. While this approach was sufficient for Media Players, we assert that it is difficult to apply as a general solution to widget accessibility. Many widgets share common interface parts, such as a next button, that have similar functionality but when viewed in the context of the content area, have different meanings to the users. Approaches to identify dynamic content using Pixel-

based methods [11] offers an alternative approach, but without interpreting the widget's processes this may result in detection ambiguity as reported in [6], and cause false expectation to their user.

Indeed, it is at a high-level content area that many developers and users perceive widgets: as a coherent region of controls and content grouped by visual association. When developing a widget, design patterns help developers to formulate the widget by using the correct combination of component parts [14]. Studies have demonstrated that design patterns can be detected from the source code using code comprehension techniques to automatically identify the nature of the component [13][12]. However when attempting to locate widgets within a Web page, these techniques can fail due to the diverse range of development technologies on the Web [6].

While previous accessibility solutions have so far concentrated on the interface components [23], we aim to detect and present widgets at a higher level. Our approach, therefore, is to automatically detect widgets using code comprehension techniques to search for the particular combinations of components and their properties that characterise widgets, to identify what dynamic content is present on the page. By combining both the high-level nature and the components of widgets, we assert that richer information can be relayed to the user while still maintaining the ability to have fine-grained control over the component.

3 Widget Classification

As a first step towards supporting users when they interact with Web 2.0 content, a method to automatically identify widgets that are present on the page was developed. This technique allows subsequent tools, designed for a variety of user groups, to be able to target assistance and provide support for those widgets currently displayed to the user.

A widget is a coherent unit of content contained within a Web page that users can interact with. Figure 1 shows an example of a Slideshow widget. This widget organises a list of content to display a limited number of items at a time in the *Display Window*[2]. The Slideshow widget has a set of controls that allows the user to browse through the list at their own pace. In this example, the user can use the *Previous* and *Next* buttons to skip through the content, or use the *Pause/Play* button to enable the widget to display the content automatically, with a small delay between each set of displaying items.

While users view widgets as a high-level object, they are derived from a set of components. Figure 2 shows how these components are combined to create a high-level Carousel widget. What users consider to be a Carousel is composed of three components – a *Previous* Button, a *Next* Button, and a *Display Window* – each of which allows the user to perceive and interact with the content on the page. In a similar fashion, the components that make up

[2] The *Display Window* is an area in the Web page that is part of the widget where content is presented to the user.

Fig. 1 Illustration of the components a Slideshow widget possesses that users can interact with using the given set of controls.

widgets have a series of properties that provide the functionality that users interact with. In Figure 2 for example, a *Next* Button has an Incremental Property which is used by the underlying control mechanism to display the next set of content in the list to the user.

Tell-signs are the properties of the components that, when combined, provide an indication that a widget is present in the Web page. By identifying tell-signs from the underlying code, it is then possible to see what combinations of components co-exist in order to determine the widgets that are present in the page.

Problems can arise when widgets share common features. Consider the Slideshow and the Carousel widgets shown in figures 1 and 2 respectively. Both widgets deliver content in a sequential fashion with only a limited number of items displayed to the user at a time. The user can browse through the list of content by using a set of controls, such as the *Previous* and *Next* buttons. However, a Carousel allows the user to loop around the list of content endlessly in bi-direction, while a Slideshow does not. Often, a Slideshow even provide additional functionality whereby the user can force the widget to display content automatically through the use of the *Pause/Play* button. The subtle differences between widgets means that they can be classified differently based on individual points-of-view, experiences, and how the widget is applied [14]. As we have seen, a Slideshow has common functionality to a Carousel

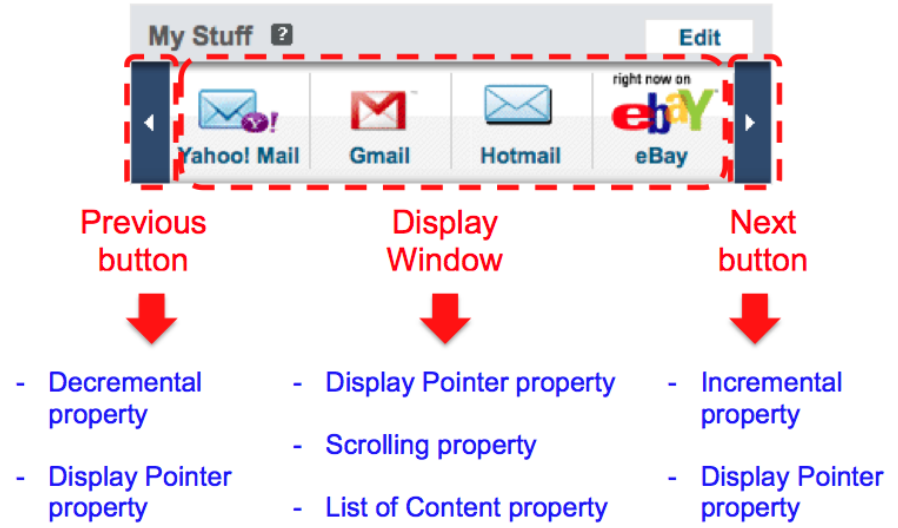

Fig. 2 Example of the components and properties that are combined to create a Carousel widget.

leading to some developers using the two names interchangeably, and users of these types of widget to have different expectations.

In this case, we created an ontology that formally defined the components a widget is composed of, and the properties that a component consisted of. By creating a formal classification system, widgets are identified by the processes it does and the actions that users performed, rather than an informal name used by developers. Such a classification system reduces any definition ambiguity and misinterpretations of widgets that may arise due to different designers having differing views of what a widget should be called. By formalising what a widget is, inconsistencies in operation between accessibility tools are reduced as tools are able to explicitly identify what widgets are on the page, and the components necessary for the users to interact with the widget.

To develop the widget ontology, design pattern libraries (such as Yahoo! Design Pattern Library[3] and Welie[4]) were utilised to identify common widget development practices that use combinations of components. Once the tell-signs and their properties had been derived for a widget, the ontology could be used as a basis for determining the presence or absence of a widget in a Web page.

A cursory examination of our Widget Identification Ontology[5] (using say, the Protégé tool) we can see how two widgets with very similar character-

[3] Yahoo! Developer Network: Design Pattern Library – `http://developer.yahoo.com/ypatterns/`

[4] Welie.com: Patterns in Interaction Design – `http://welie.com/patterns/`

[5] `http://wel-eprints.cs.manchester.ac.uk/136/`

istics can be differentiated based on their user interaction components and processes characteristics. Not only can the ontology allow these differences to be explicitly stated, but it is flexible enough to allow more tell-signs to be included that define more widgets. Existing tell-signs can also be modified to suit the evolution of widget's design. The ontology provides developers with a medium to communicate their concepts and definition of a widget, minimising misinterpretations of a widget, and reducing the need to reinvent similar types of widget. These definitions can then be used to identify instances of widgets that are present on a given Web page.

4 Widget Properties and Extent

To ascertain the most popular widgets and their extent we conducted a manual census of the Alexa top 50 English language website. We then identified different widgets using our ontology and countered the instances of each widget. This census gave us nine distinct widgets which we assert are good indicators of the kinds of widgets used over the entire AJAX coded Web - rationalising that modern AJAX technology will be most systematically used in the highest visibility websites.

1) Auto Suggest List widget Provides the user with a list suggested phases identified from the characters entered in a text field. Found in 52% of sites. Tell-signs: Monitors a text field for a change; Updates the Display window of the widget; Polling for whether the up or down key was pressed; Auto Complete attribute of the text element is set to off.
2) Popup Content widget Provides a floating area for content to be displayed above the rest of the page. Found in 50% of sites. Tell-signs: Set Display Window's z-index style value to > or <than 0; Triggered by an user or window event; Make Display Window appear or disappear.
3) Tabs widget Generally used to breakup content into multiple sections stacked together, so that the desired section can be displayed when requested by the user without refreshing the page. Found in 46% of sites. Tell-signs: Triggered by an user event; Make selected content appear and the rest in the list to disappear.
4) Carousel widget Presents user with a list of content, so that they can scroll through the different content in the list. This widget loops around the list of content in bi-direction. Found in 30% of sites. Tell-signs: Must have a next and a previous button; Make sure that the content will loop around to the first item when the last item in the list is reached, and vice-versa.
5) Collapsible Panels widget Allow users to hide irrelevant content and show the content of interest. Found in 20% of sites. Tell-signs: Triggered by an user event; Make Display Window appear or disappear.
6) Slideshow widget Presents the user with a list of content, so that they can scroll through the different content in the list, but do not allow the user to loop around the list. Found in 14% of sites. Tell-signs: Must have a next and a previous button; Make sure that the content do not loop around to

the first item when the last item in the list is reached and vice-versa; Must have a display pointer; Updates the Display window of the widget.

7) Ticker widget Presents user with a list of contents that will automatically scroll through the different content in the list at a fix interval. This widget loops around the content in a single direction. Found in 12% of sites. Tell-signs: Must have a display pointer; Updates the Display window of the widget; Uses a delay before the next item in the list is presented; Make sure that the content will loop around to the first item when the last item in the list is reached.
8) Popup Window widget A link or a form button that creates a pop up window. Found in 12% of sites. Tell-signs: Triggered by a user event on a element; Executes JavaScript's new window method – `window.open(...)`.
9) Customisable Content widget This widget allows the user to customise content on a Web page. Unlike Collapsible Panels, this widget only allows the user to remove irrelevant content from the page. Found in 10% of sites. Tell-signs: Triggered by an user event; Make Display Window disappear.

We will continue by describing the identification process for just the 'Carousel' and 'Slideshow' as these posses similar properties and are therefore the hardest to reliably identify.

5 Widgets Identification

As discussed previously in §3, combinations of components help to define a widget. Similarly components themselves are combinations of different properties. A fusion of bottom-up and top-down approaches was used to seek instances of the widget's properties (tell-signs) and determine if the widgets existed on the Web page. Figure 3 illustrates the process used to gather tell-signs and identify the widgets that were available to the Website [7].

The initial stage, indicated as ①, processes the Document Object Model (DOM) of the Web page, in order to identify the locations of any HTML, JavaScript, and Cascading Style Sheets (CSS) that may exist within the document. Once the individual source code files have been identified they are sent to the Code Synthesiser, indicated as ②. This stage assembled all the separate code resources of the page into one large source file to enable the code to be profiled. As Websites can call external files to enable code reuse, ② also involved accessing the Web to pull in any external resources and add them to the single source file. The profiler, indicated as ③, creates a file which can be thought of as being executed, in that instances of factory method's concepts and run-time code constructs are logged. The final stage of the process, indicated as ④ involves systematically processing the combined HTML, profiled JavaScript, and CSS code of the Web page to identify widgets that may exist within it. This stage relied upon the Widget definitions that were formulated in the Widget Identification Ontology. For each widget, the code was parsed

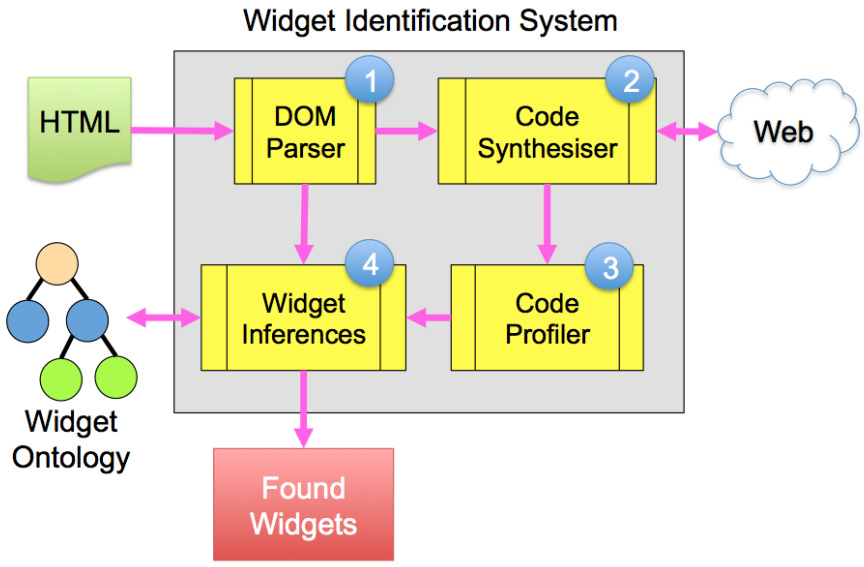

Fig. 3 Widget identification system architecture

to identify the tell-signs that were used to compose that widget and determine the presence or absence of the widget on the page.

As an example of the page analysis method, consider the Carousel widget, shown previously in Figure 2 and defined in the ontology. When our system receives the page from the Web browser, the DOM is parsed and any external HTML, JavaScript, and CSS files are extracted. These are passed to ② where the code is concatenated together. It should be noted that concatenation occurs only for the same code type. Therefore all CSS code is concatenated into a single file and all JavaScript code is concatenated into a separate, single file.

Before analysing the code to identify widgets, documenting the concepts and structures of the code will provide a directory of the code to assist widget detection. A customized profiling module is used to build such a directory for the JavaScript code to be analysed. This module provides a platform to assist the code comprehension process when identifying widgets in ③ of Figure 3.

It is during ④ that the Carousel widget is determined by searching through the information provided by ③. The Widget Inferences module attempts to match for components that are defined in the Widget Ontology as being composite parts of a Carousel Widget. In this case, instances of the *next* and *previous* button are the key tell-signs. As these components are buttons, the analyser looks for button-like elements. Buttons can be implemented in a number of styles, the analyser looks for all possible forms, which include:

Form elements

```
<input type="button"...>
<input type="submit"...>
```

Anchor links with text

```
<a href="..." ...>...</a>
```

Anchor links with images

```
<a href="..." ...><img src="..." ...></a>
```

As *Next* and *Previous* buttons are interactive, they can trigger JavaScript code through an event initiated by the user. In this case, the event will trigger either an increment or decrement of the display pointer to update the currently displayed content. The triggering events from a mouse include `onclick`, `onmousedown`, `onmouseover`, and `onmouseup`, and from a keyboard include `onkeypress` and `onkeydown`. By following the handling function triggered by these events, the fragment of JavaScript code that changes the *Display Pointer* value can be identified. To differentiate between the *Next* and *Previous* buttons, key code constructs are looked for. For example, variables with `++`, or variables that are assigned with a value from an addition, are chosen for further analysis for an incremental process for the *Next* button, while variables with `--`, or variables that are assigned with a value from a subtraction, will be chosen for further analysis for a decremental process for the *Previous* button.

The *Next* and *Previous* buttons' tell-signs can only be confirmed after the Display Window tell-sign is determined. This is because there will be a large number of candidates *Next* and *Previous* buttons, and a Carousel requires all three coherent components. It is only by identifying all the tell-signs and establishing the relationships between them that we can confirm the existence of all components, and hence the widget. This interdependence means that our process is iterative as we identify potential tell-signs for a given widget and discard them as new evidence is discovered.

To identify *Display Window*, we search for a display pointer, as instances of this functionality cause an update to the *Display Window*. A *Display Pointer* will consist of two "tell-tale" signs: firstly, a *Display Pointer* will constantly check to ensure that it refers to a location within the list of available content. When searching for instances that these conditions are checked in the suspected section of code, the following regular expression is used to search for the different patterns of this type of checks.

```
"/if\s*\(\s*[\w\W]*\s*(==|<=|>=|>|<)?\s*" + possible[i]
          + "\s*(==|<=|>=|>|<)?\s*[\w\W]*\s*\)/i"
```

The concatenated array variable within the regular expression, `possible[i]` stores the candidate *Display Pointers* variable identifier identified during the analysis and filters out the less likely *Display Pointer* candidates. To ensure that the pattern is not case sensitive, the modifier '`i`' is included after '`/`' at the end of the regular expression.

Secondly, evidence to prove that the variable is used to point to a location within a list is required. To cater for different variations of programming styles and approaches, the following regular expression is employed to analyse for these instances:

```
"/([-_0-9a-z]+)\[" + possible[i] + "\]/i"
```

As with before, the concatenated array variable `possible[i]` stores the remaining candidate *Display Pointer* variable identifier, while this search further segregate the likely *Display Pointer* candidates. However, from this search process, candidates for the list of content variable can be found too. This combination of findings will lead to the discovery of a Display Window tell-sign.

To confirm that a *Display Window* has been identified, a final analysis occurs to establish if the *Display Pointer* updates the DOM in order to display content to the user. Based on anecdotal evidence, there are two popular ways of updating the content in the Web page. The first method updates the content that is currently stored in the DOM by changing the `innerHTML` property of an element. The second method changes the styling properties of the content so that it is either displayed (`display="block"`) or hidden (`display="none"`). Other DOM mutation methods that facilitates updating the content, such as different techniques to change the structure of the DOM tree, are not incorporated at this stage, because we are only evaluating our concepts and proving its feasibility. It is beyond the scope of this paper to provide the regular expressions that can identify these updates, however the process used is similar to those that was demonstrated previously.

Identifying widgets not only relies on identifying the presence of tell-signs, but also the absence of tell-signs. As highlighted in §3, widgets can share similar features, with the difference between a Carousel and a Slideshow being the presence of absence of a looping feature. Using a similar process as above, the code is analysed using a combination of DOM analysis and JavaScript analysis to search for indications that a looping feature exists. For example a common tell-sign for this feature is assigning the display pointer to either the start or end of the list.

This final analysis of the code adds to the body of evidence that, when combined, identifies widgets. Evidence of the existence of *Next* and *Previous* buttons, and a *Display Window*, combined with the presence of a looping feature allows us to conclude that a Carousel widget exists within the page. This information can then be relayed to assistive technology in order to allow users to interact with the widget. Examples of how this can be achieved are provided in §7.

6 Evaluation

To establish the feasibility of our method, a technical evaluation was conducted that sought to identify how accurate the approach was at identifying widgets. Two types of widgets were chosen for the evaluation – Auto Suggest Lists (ASLs) as they occur on more sites than any other widget and Carousels as these are the most difficult to identify as they are often misidentified as Slideshows. We are testing for how generalizable our widget detection algorithm is. Carousel and Slide Shows are widgets with similar tell-signs; we want widgets that are a distance from each other to avoid our algorithm to be too specific.

The evaluation dataset consisted of twenty Websites selected from Alexa's global top 500 sites and can be found in [8]. This is a separate set of data used to test the accuracy of our algorithm to find the types of widgets used. The previous dataset discussed in §4 is used to check for the popularity of occurrence of the widgets. Both lists of Websites selected were taken at a different time, so the first list of Websites is not the same as the evaluation list. When choosing the data set a range of popular Websites were chosen so that it gave a good representation of the Web based on the following requisite:

1. No repetition of a Website's domain was allowed. For instance, `google.com.br` was not used as it is a sub-domain of `google.com`.
2. If a repetition did exist, it was ignored and the next domain in the list was examined.
3. Repeat step 1 and 2 until twenty Websites were selected.

The default page for each of the twenty websites was analysed automatically, while the analyser returned either true or false to represent the presence or absence of a particular widget. During the first iteration, the tool looked for ASLs and for the second iteration the tool searched for Carousels. After the automatic analysis was conducted, each page was inspected manually to determine if the widgets had been accurately identified. For the ASL analysis, the analyser identified all the ASL widgets that were present within the pages, but also returned a false positive rate of 16%. Depending on size of a Web page, the execution time of the script will vary; on average it was recorded 0.1707 seconds to detect an ASL. For Carousel analysis, the analyser identified all the Carousels Widgets that were present within the pages, but also returned a false positive rate of 10%. It should be noted that no false negatives were returned during the automatic analysis, only false positives. On average, the execution time for detecting a Carousel is 0.4145 seconds. Further investigations surrounding the false positive results highlighted that instances of these widgets do exist within the page component's library, however, they were not used by the Web page. One can therefore consider these results to be accurate on a Website level, but not on a page level.

Currently, we are investigating a method to address this issue by calculating the distance between the discovered components in the page's DOM. Through this investigation, we intend to locate where the widgets lie within the page and return the XPath of the widgets location to DOM monitoring tools. With this information, DOM monitoring tools can be aware of the live regions where content may be updated. Our current approach identifies components regardless their location in the DOM. By identifying components in close proximity to each other we can assume that they are part of the same widget and therefore false detection. Code optimisation for the individual widget detection scripts are also investigated to improve their execution timings.

While more accurate definitions of widgets are being investigated to ensure a higher successful detection rate, the current results show that the approach is robust enough to use as a platform for developing assistive technologies.

7 Widget Identification Use Case

The techniques described in §3–5 have been used to develop two prototype tools that assist users as they interact with dynamic content. These tools, discussed below, provide older users and visually impaired users with improved access to Web 2.0 content.

7.1 Supporting Older Users of Dynamic Web Content

As discussed in §2, during GSR studies to identify stress levels of older users, Lunn and Harper [21] observed that unlike younger participants, there was a large variance within the results for the older user groups and that there were signs from some users of hesitancy and uncertainty when completing the tasks. Based on these results, a prototype tool was developed to assist older users as they interact with dynamic content. The tool is implemented as a Web Browser extension to allow for rapid prototype development and also to allow users to feel more comfortable. As [17] note, *"users tend to prefer a standard browser with the accessibility transformations added rather than a specialised browser offering only a limited set of features (which would also tend to mark them as being disabled)."*

As the page loads, the browser executes code, based upon the work described in §5, to establish if the page contains any dynamic content. If widgets are present on the page, then an information icon is displayed on the browser to allow users to receive help if they require assistance, indicated as ① in Figure 4. If users are comfortable with interacting with dynamic content, then they can ignore the information icon.

The area indicated as ② shows the panel that is displayed when the user clicks on the assistance icon. A list of the widgets that have been detected is displayed. In this case, three widgets were identified – Auto Suggest List, Carousel, and Tab Box. The user clicks on those buttons to receive help if they do not understand what content is present. In the example shown, the user has clicked on the "Tab Box" button. This results in a short paragraph appearing explaining what a tab box does along with a demonstration video. As the video is being played, the widget on the page is highlighted in a pink circle, indicated as ③. This is to ensure that users are aware of what part of the page is being discussed in the demonstration video. While widgets have similar functionality, they may have slightly difference appearance. The video is designed to be as generic as possible and the pink highlight drawers the user's attention to the widget on the page that is being talked about. Participant feedback and evaluation have been positive.

7.2 Supporting Visually Impaired Users of Dynamic Web Content

The second use-case for widget identification as an aid to accessibility explores how identifying widgets as coherent high-level units, rather than low-level con-

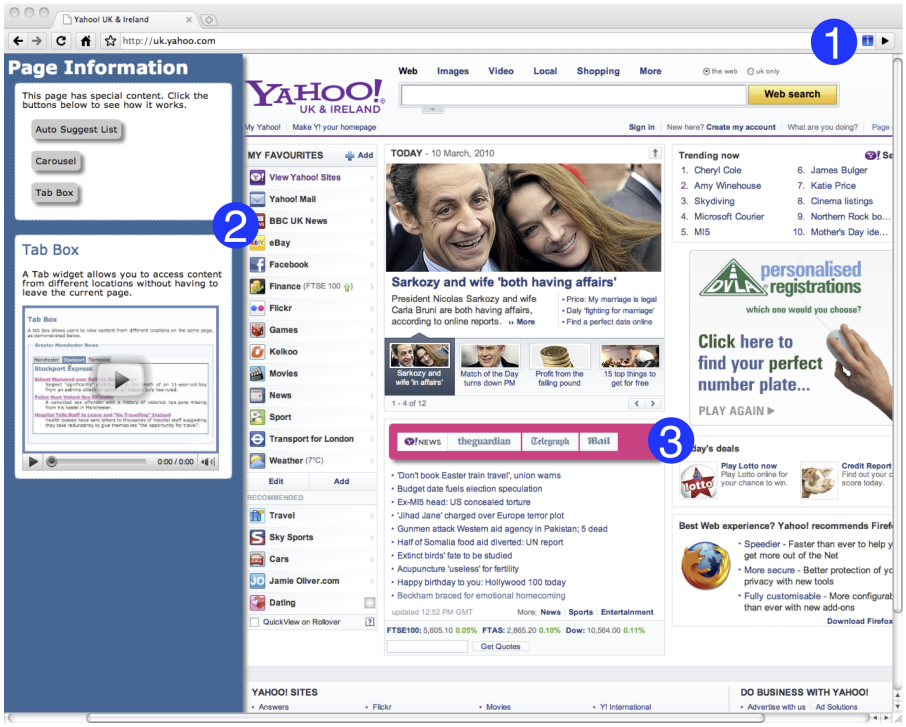

Fig. 4 Interface for the Older User Widget Assistant.

trols, could benefit screen reader users. Eye-tracking studies of sighted users have given insight into how these people interact with, and benefit from, different types of dynamic content [4][3], allowing development of rules for presenting updates according to how they were initiated and their effect on the page [19]. During evaluation of these rules [19] the nature of each of the test pages, and their dynamic content, was explained to the blind and visually impaired participants. This was noted to be beneficial to the users, as it gave them a more accurate mental model of how the interactions were likely to work, and thus enable them to use the widgets more effectively.

Sighted users are able to scan a page, or a section of a page, rapidly, perceiving the content at a glance. Visual clues, such as borders and spacing enable them to identify units within the page, and layout conventions help them to recognise the type of widgets present. For example, the Yahoo! home page contains an area with news content, organised into 4 sections, and presented as tabs; clicking on the tab title changes the content (Figure 5). This section is clearly separated from the surrounding content by spacing and a border, and the tab titles are styled in such a way that most sighted users will recognise the metaphor and understand how to interact with it.

The clues are, however, mostly implicit, so those people who wish to browse these pages using a screen reader miss many or all of them, and are thus unable

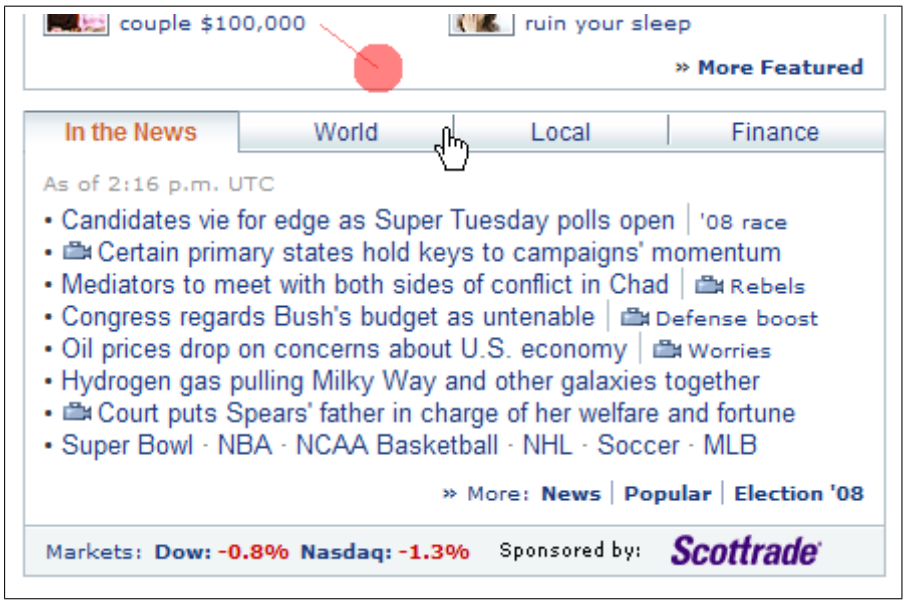

Fig. 5 Tabs on the Yahoo! home page.

to recognise the content. The visual differentiation is not available to them, so it is not clear that they are navigating into a new region (that may be a widget). The ability to glance over an area is also much diminished, so relating components to each other is more difficult, particularly as the 2-dimensional visual layout is translated into a 1-dimensional audio one. For these people, therefore, who encounter a linear sequence of content and controls, un-grouped and undifferentiated from surrounding content, it is very difficult to recognise the form and function of a widget. Understanding its use is thus more difficult, as is forming an expectation of the result of any action. Coupled with the lack of information current screen readers provide when content changes, this adds up to make using these widgets a demanding experience.

A prototype application is thus underway that will utilise the tell-sign approach to widget recognition and apply it to making the presence of widgets explicit to screen-reader users. This is being done in two ways: by inserting text into the DOM so that the user is informed when entering or leaving a widget, and by modifying the widget controls (for example AxsJAX [9]) so that their effect will be more explicit. Although development is underway and user-testing is yet to be performed, it is expected that these techniques will mean that screen reader users will no longer have to interact with unrelated low-level controls, but that they will be encounter widgets at a high-level, thereby understanding better the function and effects of the low-level controls within. This should lead to more confident and efficient use of dynamic content.

8 Discussion and Future Work

While the initial results of the technical evaluation and the prototypes are positive, there are a number of issues that need to be addressed before wide-scale deployment can commence.

8.1 Interdependent and Optional Tell-Signs

As discussed in §3, widgets share common tell-signs that contain similar characteristics and capabilities. Tell-signs can therefore be reused like objects in an Object-Oriented programming language. However, there are certain characteristics of tell-signs that may complicate the identification method. The first is the interdependent tell-signs issue. Typically, tell-signs exist within the code and are executed depending on what actions the user has performed. However, some development techniques dynamically generate tell-signs as the user interacts with the widgets. As an example, consider a form that assists the users when entering data into it by guiding them through the form filling process. One method of creating a form is to have all the form components present in the page code and display only the necessary elements to the user as they enter data. A second technique automatically generate the next section of the form as the user completes each section, this time only the form elements that are presented to the user is loaded in the code, and when the next section is required, it will be loaded using remote scripting techniques. In this method, only a limited number of tell-signs will be available during the initial analysis and the remaining tell-signs are dependent upon certain aspects of code being executed. In order to identify all the tell-signs, a method of virtually executing the code must be used in order to interpret the developers intention of the process, establish what elements are generated and use the virtual execution tell-signs as part of the widget identification.

The second issue is designers may consider tell-signs to be optional for the functioning of high-level widget. In §3, the subtle differences between a Carousel and a Slideshow were highlighted, with the defining factor being a Carousel loops around the list of content. This was identified through the presence of assigning the display pointer to the start or end of the list when either ends of the list is reached respectively. However, some designers may implement a Carousel but allow its users to stop or start the content from automatically displaying. In this case additional tell-sign will be present during the identification phase of the method. Our initial insight to this problem suggest a probability based model to adapt our identification method for the varied styles of developing them, and to include flexibility to cater for their different characteristics. Each tell-sign will be allocated a value depending on their importunateness, and whenever a tell-sign is discovered from our identification process, its value is accumulated to form the identification probability. Then, this value will facilitate the identification process to make a decision

whether the widget exist by comparing it with the threshold value allocated to the widget.

8.2 Website Development Tool

The widget identification method, and subsequent prototype tools, has thus far focused on assisting users as they interact with Web 2.0 pages. However, there is scope for the Widget Identification method to be used in the development phase when Websites are being designed. The first areas that could make use of the method is the general usability of Websites. The ontological definitions are based upon commonalities between widgets that are found on Websites. As part of the development process, the Widget Identification Tool could check the pages for widgets that are present and highlight any potential missing components. For example, the Carousel with the looping feature missing, as discussed previously. Users interact with Websites more effectively when they follow a standard pattern. Therefore informing developers that their widgets do not conform to expected interaction methods would allow them to make suitable changes to their Websites and improve usability.

Secondly, the Widget Identification method could be used to improve accessibility. By identifying widgets that are present on the page, the tool could suggest appropriate mark-up that should be added to the page in order to improve accessibility. For example, an indication of the high-level widget that is present would inform users of assistive technology what type of widget they were interacting with. This, coupled with suitable WAI-ARIA mark-up would then allow users of assistive technology to interact with the widgets on the Website more effectively without the need for our tool running in the back-end.

9 Conclusion

In this paper we have presented a novel method to automatically identify dynamic content that is present within a Web page. This process identifies widgets as complete units of content, as perceived by the user, and the underlying processes rather than as individual component parts as previous techniques have done. The technique is based upon the identification of tell-signs and an analysis of the combinations in which they occur to determine the presence or absence of a particular widget within the page. As widgets share many common components, and as developers can use widget names interchangeably, an ontology was developed to act as a classification system for defining widgets. The classification was based on formal functionality and component parts of the widget, rather than relying on developers' perceptions of widgets based on naming conventions. Such an approach allows for more formal identification of widgets when the Web page is analysed for evidence that tell-signs of widgets existed within the code.

We have demonstrated how the Widget Detection Algorithm can be used as the basis for developing tools that assist both visually impaired and older Web users. We expect them to allow users to interact with dynamic content more easily and provide a more enjoyable Web experience.

Acknowledgements This work was conducted as part of the Senior Citizens On The Web 2.0 (SCWeb2) project, funded by The Leverhulme Trust(F/00 120/BL) and the Single Structured Accessibility Stream for Web 2.0 Access Technologies (SASWAT) project, funded by the UK EPSRC (EP/E062954/1).

References

1. Asakawa, C., Itoh, T.: User Interface of a Home Page Reader. In: ASSETS '98: Proceedings of the Third International ACM conference on Assistive technologies, pp. 149–156. ACM (1998). DOI http://doi.acm.org/10.1145/274497.274526
2. Brown, A., Jay, C.: A Review of Assistive Technologies: Can Users Access Dynamically Updating Information? SASWAT Technical Report 2, The University of Manchester, School of Computer Science, UK (2008). DOI http://wel-eprints.cs.manchester.ac.uk/70/. Http://wel-eprints.cs.manchester.ac.uk/70/
3. Brown, A., Jay, C., Harper, S.: Audio Presentation of Auto-Suggest Lists. In: W4A '09: Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibility (W4A), pp. 58–61. ACM (2009). DOI http://doi.acm.org/10.1145/1535654.1535667
4. Brown, A., Jay, C., Harper, S.: Audio Access to Calendars. In: W4A 2010: Proceedings of the 2010 International Cross-Disciplinary Conference on Web Accessibility (W4A). ACM (2010)
5. Carmi, R., Itti, L.: Causal Saliency Effects During Natural Vision. In: ETRA '06: Proceedings of the 2006 Symposium on Eye Tracking Research & Applications, pp. 11–18. ACM (2006). DOI http://doi.acm.org/10.1145/1117309.0410. URL `http://portal.acm.org/citation.cfm?id=1117309.0410`
6. Chen, A.Q.: Widget identification and modification for web 2.0 access technologies (WIMWAT). SIGACCESS Access. Comput. pp. 11–18 (2010). DOI http://doi.acm.org/10.1145/1731849.1731851. URL `http://doi.acm.org/10.1145/1731849.1731851`
7. Chen, A.Q.: Profiling The Web Page's Behaviour Layer. WIMWAT Technical Report 5, The University of Manchester, School of Computer Science, UK (2011). Http://wel-eprints.cs.manchester.ac.uk/135/
8. Chen, A.Q., Harper, S.: Identifying Web Widgets. WIMWAT Technical Report 1, The University of Manchester, School of Computer Science, UK (2009). DOI http://wel-eprints.cs.manchester.ac.uk/115/. Http://wel-eprints.cs.manchester.ac.uk/115/
9. Chen, C.L., Raman, T.V.: AxsJAX: a talking translation bot using google im: bringing web-2.0 applications to life. In: Proceedings of the 2008 international cross-disciplinary conference on Web accessibility (W4A), W4A '08, pp. 54–56. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1368044.1368056. URL `http://doi.acm.org/10.1145/1368044.1368056`
10. Craig, J., Cooper, M., Pappas, L., Schwerdtfeger, R., Seeman, L.: Accessible Rich Internet Applications (WAI-ARIA) 1.0. `http://www.w3.org/TR/wai-aria/` (2009)
11. Dixon, M., Leventhal, D., Fogarty, J.: Content and hierarchy in pixel-based methods for reverse engineering interface structure. In: Proceedings of the 2011 annual conference on Human factors in computing systems, CHI '11, pp. 969–978. ACM, New York, NY, USA (2011). DOI http://doi.acm.org/10.1145/1978942.1979086
12. Dong, J., Sun, Y., Zhao, Y.: Design pattern detection by template matching. In: SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, pp. 765–769. ACM (2008)
13. Fukaya, K., Kubo, A., Washizaki, H., Fukazawa, Y.: Design pattern detection using source code of before applying design patterns. In: 1st International Workshop on Software Patterns and Quality (2007)

14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Hailpern, J., Reid, L.G., Boardman, R., Annam, S.: Web 2.0: blind to an accessible new world. In: WWW '09: Proceedings of the 18th international conference on World wide web, pp. 821–830. ACM (2009)
16. Hanson, V.L.: Age and Web Access: The Next Generation. In: W4A '09: Proceedings of the 2009 International Cross-Disciplinary Conference on Web Accessibility (W4A), pp. 7–15. ACM (2009). DOI http://doi.acm.org/10.1145/1535654.1535658
17. Hanson, V.L., Richards, J.T.: Achieving a More Usable World Wide Web. Behaviour & Information Technology **24**(3), 231–246 (2005). DOI 10.1080/01449290412331327465. URL http://www.informaworld.com/10.1080/01449290412331327465
18. Hartley, A.A.: Attention. In: F.I. Craik, T.A. Salthouse (eds.) The Handbook of Aging and Cognition, chap. 1, pp. 3–53. Laurence Erlbaum Associates (1992). ISBN: 0–8058-0713-6
19. Jay, C., Brown, A., Harper, S.: Internal Evaluation of the SASWAT Audio Browser. SASWAT Technical Report 6, The University of Manchester, School of Computer Science, UK (2010). DOI http://wel-eprints.cs.manchester.ac.uk/125/. Http://wel-eprints.cs.manchester.ac.uk/125/
20. Kurniawan, S., King, A., Evans, D., Blenkhorn, P.: Personalising Web Page Presentation for Older People. Interacting with Computers **18**(3), 457–477 (2006). DOI DOI:10.1016/j.intcom.2005.11.006. URL http://www.sciencedirect.com/science/article/B6V0D-4HYMY31-6/2/0c620c176c9eed3ef2acc2d287de3e9a. Human Factors in Personalised Systems and Services
21. Lunn, D., Harper, S.: Using Galvanic Skin Response Measures To Identify Areas of Frustration for Older Web 2.0 Users. In: W4A '10: Proceedings of the 2010 International Cross-Disciplinary Conference on Web Accessibility (W4A). ACM (2010)
22. Memmert, D.: The Effects of Eye Movements, Age, and Expertise on Inattentional Blindness. Consciousness and Cognition **15**(3), 620–627 (2006). DOI DOI:10.1016/j.concog.2006.01.001. URL http://www.sciencedirect.com/science/article/B6WD0-4J91R6C-1/2/764cb264eedae395b41d0a9cc7cb9ad9
23. Miyashita, H., Sato, D., Takagi, H., Asakawa, C.: Aibrowser for multimedia: introducing multimedia content accessibility for visually impaired users. In: Assets '07: Proceedings of the 9th international ACM SIGACCESS conference on Computers and accessibility, pp. 91–98. ACM (2007). DOI http://doi.acm.org/10.1145/1296843.1296860
24. Sàenz, M., Buracas, G.T., Boynton, G.M.: Global Feature-Based Attention for Motion and Color. Vision Research **43**(6), 629 – 637 (2003). DOI DOI:10.1016/S0042-6989(02)00595-3. URL http://www.sciencedirect.com/science/article/B6T0W-47XSSK6-1/2/b540ce8624f967c247b3e30a4e2ad97f
25. Thiessen, P., Chen, C.: Ajax live regions: chat as a case example. In: W4A '07: Proceedings of the 2007 International Cross-disciplinary Conference on Web accessibility (W4A), pp. 7–14. ACM (2007)
26. Varakin, D.A., Levin, D.T.: Change Blindness and Visual Memory: Visual Representations Get Rich and Act Poor. British Journal of Psychology **97**(1), 51–77 (2006). DOI 10.1348/000712605X68906. URL http://www.ingentaconnect.com/content/bpsoc/bjp/2006/00000097/00000001/art00004