Accessibility for Simple to Moderate-Complexity DHTML Web Sites

Cynthia C. Shelly
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
cyns@microsoft.com

George Young
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052 USA
gcyoung@microsoft.com

ABSTRACT
In this paper, we describe specific design and coding techniques for the creation of simple to medium complexity Dynamic HTML and AJAX applications, which are accessible to people with disabilities using mainstream user agents and assistive technology available at the time of this writing.

Categories and Subject Descriptors
I.7.2 **[Document and Text Processing]**:
Document Preparation – Markup languages, Scripting languages, Standards

General Terms
Design, Human Factors

Keywords
Dynamic HTML, HTML, JavaScript, Accessibility

1. INTRODUCTION
A great deal has changed since the World Wide Web Consortium's Web Accessibility Initiative released the Web Content Accessibility Guidelines 1.0 in 1999. At the time of WCAG's release, most web sites were static HTML, or created from server script, with only small amounts of client script used for usability enhancements and form validation. Today, most web sites contain some level of interactivity built in client script and some are entire applications within the browser window.

We submit that there is a continuum of complexity in web sites, with the static sites of the 1990s on one end, and highly complex applications on the other. The bulk of Dynamic HTML web products in the market today fall between these two extremes. For these products in the middle, a good level of functional accessibility can be achieved right now, with current browsers and assistive technology, by following a few straight-forward coding standards.

2. COMPLEXITY OF WEB SITES
2.1 Static HTML
These are traditional web pages. They are essentially documents, not applications. They may include simple forms which post back to the server for processing. The semantics of HTML map quite nicely to the structure of these documents. Making them accessible requires the use of appropriate header markup <h1> - <h6>), table headers, explicitly associated labels for form elements, and alternative text for images. WCAG 1.0 addresses the accessibility of these pages, and this paper will not cover the details of making these types of pages accessible.

2.2 Simple Dynamic HTML
These pages are still basically documents, and map easily to HTML semantics. They may contain client-side form interaction and validation using in-built user interaction mechanisms like alerts, pop-ups, and server-trip page navigation. The DOM is not typically modified client-side, and there is no asynchronous content updating (AJAX). These pages are well supported by current user agents and assistive technology.

2.3 Moderate-complexity DHTML
These pages use common UI widgets like toolbars, menus, dialogs, and sliders. They may involve unusual variations on these controls. These applications frequently modify the Document Object Model (DOM) at runtime, to add content, change layout, display error messages, etc. Areas of the page may be updated asynchronously, and full-page navigation is kept to a minimum.

This paper deals primarily with this type of application. Most of the common UI widgets can be mapped to HTML constructs with close-enough semantics. Careful modeling and coding of markup and script, along with tested workarounds to shortcomings in current browsers, can be used to create highly useful and innovative user interfaces, which are accessible in current browsers.

2.4 Complex DHTML
A very small number of applications go beyond what can reasonably be modeled in HTML. For example, while it is possible to create a spreadsheet out of a table of text input boxes, it quickly becomes cumbersome both to code and to use. For these applications, a new approach is needed. The W3C Roadmap for Accessible Rich Internet Applications (WAI-ARIA) is targeted at this type of application. A discussion of these applications is beyond the scope of this paper.

3. WEB ACCESSIBILITY ARCHITECTURE
3.1 HTML Semantics
Correct HTML Semantics are the key to working with a variety of User Agents and a variety of Assistive Technology. Each element in HTML was created to add a particular structural meaning, or semantic, to the document. All user agents use these semantics to determine how to present the document to their users. Some Assistive Technology (AT) works directly with the

W4A2007 - Technical Paper, May 07–08, 2007, Banff, Canada. Co-Located with the 16th International World Wide Web Conference.

HTML in the Document Object Model, and relies on these semantics to build an alternative user interface for its users.

In static HTML pages, the user agent creates the Document Object Model from the elements in the HTML sent from the server. In Dynamic HTML pages, the DOM is created or modified after the page is loaded, using script. Regardless of how the DOM is created, the elements within it must be semantically correct in order for the page to be accessible.

3.2 Accessibility APIs

In addition to accessing the semantics of the HTML directly through the DOM, many assistive technologies can also take advantage of an Accessibility API such as Microsoft Active Accessibility (MSAA) on the Windows operating system. An Accessibility API defines a contract between applications on the one hand, and assistive technology on the other. This approach can greatly simplify the development of both sides. Applications need only write to the API, and Assistive Technologies need only read from it. Without this, Assistive Technologies must understand the details of each application they wish to support, and applications must understand the quirks of many different assistive technologies.

3.2.1 User Agent Mapping from HTML Semantics to API

For a Web application to be able to take advantage of an Accessibility API, the user agent must support it, and pass information from the markup and script in that page through the API to the various Assistive Technologies that use that API.

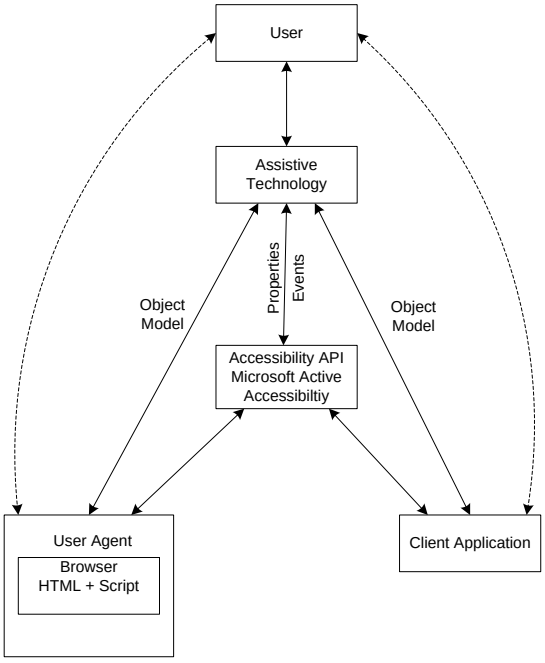

Figure 1 shows the interaction between the user, assistive technology, Accessibility API, browser and web code.

This is exactly what happens in Microsoft Windows Internet Explorer and FireFox on the Windows operating system. Each of these user agents maps a subset of the semantics encoded in the DOM to the Microsoft Active Accessibility Interface. These mappings are read-only, and web developers have no direct access to the API and must rely on the semantics exposed by the user agent.

4. TECHNIQUES FOR CREATING ACCESSIBLE DHTML USER INTERFACES

This section provides specific information and examples that you can use when developing accessible Web applications. This discussion can be seen as a cascading or layered approach, with the more fundamental requirements discussed first. The guidelines fall into the following general categories (see Table 1):

- HTML issues – Use semantically correct HTML. Focus on correct use of roles and names, and always supply alternative text for visual elements.
- Dynamic HTML (DHTML) issues – Maintain source order and use elements with built-in action semantics.
- Asynchronous JavaScript And XML (AJAX) issues – AJAX is DHTML with asynchronous updates. When you use AJAX, you need to notify AT that the page has been updated.

HTML	**DHTML**	**AJAX**
HTML Web pages. These can be hard-coded, server- generated, or generated from client-side scripts.	HTML plus user interaction through client-side scripting.	DHTML plus asynchronous updating.
• HTML semantics • DOM order	• Links and buttons • Click event • DOM order	Callback notification to AT

Table 1: Web Technology and Accessibility Categories

The following sections describe these categories and the corresponding accessibility guidelines in greater detail.

4.1 HTML: User Interface Semantics

Again, the key to creating an accessible application for today's browsers is to use the elements built into HTML, and to use them the way they were designed to be used.

In many ways, HTML is poorly suited to the creation of user interfaces. There are many elements that describe document structures like paragraphs, tables, lists and headings. There are links. There are a few, fairly primitive form elements. However, there are no in-built structures for menus, toolbars, sliders, and many other common user interface elements that one might want to use to create an application. However, the simple semantics of HTML can be combined to approximate many of the missing structures in a way that provides functional accessibility.

The first and most important step in creating an accessible web application is to understand the structure of your user interface, and to model that structure, as best you can, in the semantics of HTML.

4.1.1 Actionable Elements: Using the click event of links and buttons

On Windows, Internet Explorer and Firefox map the link and button elements to their corresponding values in the Microsoft Active Accessibility API. JAWS and WindowEyes will add "link" when reading these. The default action is mapped to the onclick event. The default action occurs when the user clicks the element with a mouse, but it also occurs when the user focuses the element and hits enter or space, and when the element is triggered via the accessibility API.

The click event of these elements sends an event to the operating system that can be trapped by AT, making it device independent for these elements. This is not true for other events and other elements. This makes perfect sense in terms of Semantic HTML. The only elements that are intended to trigger actions are links and buttons, and these elements have a single, default action. These are the basic units of action in HTML, and all other interaction must be built on from them.

The <a> element is only mapped to the link role when it includes the href attribute. In this case, the link target is "#", but it could be anything. The "return false;" at the end of the doStuff() event handling function tells the browser not to navigate to the URL. Without it, the page would refresh after the script ran.

```
<script>
        function doStuff()
        {
                //do stuff
                return false;
        }
</script>

<a href="#" onclick="return doStuff();">do stuff</a>
```

This technique will not work with other elements, such as divs or spans, which are made to look like a link with CSS. Even if tabindex and onkeypress handlers are added, the role will be wrong, and the events will not be sent to the operating system.

The same technique can be used with the buttons that are implemented using one of the following elements:

```
<button>display text</button>
<input type="button">display text</input>
<input type="submit">display text</input>
<input type="reset">display text</input>
<input type="image" title="alternative text"/>
```

These are true buttons, mapped to the push-button role. JAWS and WindowEyes will add "button" when reading these controls, which can be activated with the enter and space keys. It is fairly common to see constructs that look like buttons but are created with tables or styled divs. These do not have the correct semantics, and will not expose the right role or send the right events. Both <button> and <input type="image"> are very flexible, and can be use for almost any desired visual treatment.

Below are some examples that build on functionality built into link semantics, and use a combination of script and CSS to achieve a particular user experience.

4.1.1.1 Creating larger hit targets using link semantics

User Interface designers often want to allow users to click on a large area to trigger an action. Larger hit targets are also easier for users of low-resolution pointing devices, and can improve usability for all users.

In order to be device independent, the click must occur on a link or a button. However, the HTML <a href> element, being an inline element, cannot be wrapped around block elements such as or <table> without creating invalid HTML.

This technique has two pieces. The first is a link with an onclick handler, as discussed above. The second is a larger UI element which contains the link, and which passes any clicks it receives to the link by calling the click() method on the <a href>.

This example is a list of messages, where each message has a sender, a subject and a receipt time. The user can click anywhere in the message to open it. Similar UI is often implemented by adding mouse handling to divs or spans or other elements that don't have native support for actions, which doesn't interoperate with many types of assistive technology.

Because this example is a list of mails, the HTML for this example is a <ul>,which is the appropriate semantic mapping for a list. Each list item has three div elements containing message information. This approach can work with any container, including and <td>.

```
<ul class="msgList">
   <li class="msg" onclick="OnMsgClick(event,this);"
      onmouseover="OnMsgOver(event,this)"
onmouseout="OnMsgOut(event,this);">
      <div class="date">Wed Jan 11 2006</div>
      <div class="from">Sarah Smith</div>
      <div class="subject"><a href="#"
onclick="OnMsgLinkClick(event);">Ready for the
presentation?</a></div>
   </li>
   ...

</ul>
```

The script for the example is simple. If we find a link child in the container, we call click() on it.

```
function OnMsgClick(evt,el)
{
   HarmonizeEvent(evt);
   var link = el.getElementsByTagName("a")[0];
   if (link)
   {
      link.click();
   }
}
```

CSS is used to style the elements to look like a message list.

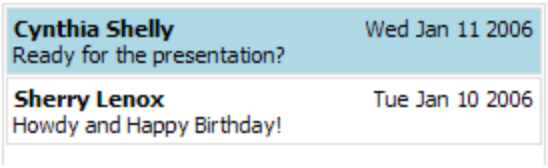

Figure 2 Shows the clickable area, which is highlighted onmouseover

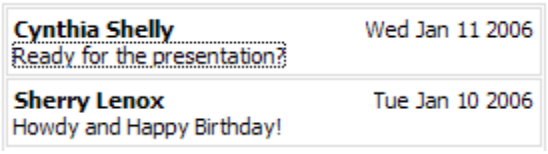

Figure 3 Shows the underlying link, with keyboard focus

These are ordinary HTML links, and will appear in AT just like any other link. For example, they are available in the link list in JAWS and VoiceOver, and in Window Eyes link Navigation. Additionally, because they use the device-independent onclick event, their script will be triggered when those tools activate the links.

4.1.1.2 Secondary Links for Hover-Based User Interface

Hover pop-ups -- bits of content which appear on page in response to the user mousing over an element -- are becoming more and more prevalent. These can often enhance usability for mouse users, but rely on the device dependant onmouseover and onmouseout events.

This technique adds a hidden link immediately following hoverable element. Mouse users never know this is there. Because it is an actual HTML link (<a href>), a keyboard user will encounter it when tabbing through the page, right after the hoverable link. It is also exposed as a link to Assistive Technology.

The hidden link may contain either text or an image, indicating that the link is for displaying a hover popup. Hidden by default, the link expands to its full size when it receives focus. The onclick event of the link calls the same script as the hover effect it is augmenting, ensuring that all users can access the User Interface triggered by the hover.

This example simply launches a JavaScript alert window. The same idea can be applied to call any script, to show any UI. The HTML shows two links. The first is the standard link with the onmouseover() event-handler. The second link is contains a hidden image that displays onfocus() and whose onclick() handler calls the same function as the onmouseover() handler on the normal link.

```
This a some sample text with a
<a href="#" class="hoverPopup" onmouseover="alert('hello');"
   >hover popup</a>
<a class="clickPopup" href="#"
   onfocus="TogglePopupImage(event,true)"
onblur="TogglePopupImage(event,false);"
   onclick="alert('hello');" ><img
   height="10" width="10" src="smile.bmp"  alt="Toggle popup
for 'Sample Link 1'"/></a>
```

in the paragraph.
The TogglePopupImage() function handles the showing and hiding of the hidden image when its link receives or loses focus.

```
function TogglePopupImage(evt,show)
{
   HarmonizeEvent(evt);
   var src = evt.target;
   var img = src.getElementsByTagName("img")[0];
   if (img)
   {
      img.style.width = (show ? "10" : "0");
      img.style.marginLeft = (show ? "0.3em" : "0");
   }
}
```

CSS is used to set the initial width of image to 0

Like the previous example, these are ordinary links, in this case containing 0-width graphics. They will appear in link navigation in screen readers and other AT.

4.1.2 Lists of Links: Menus, toolbars, trees and navigation bars

Links are one of the basic units of action in HTML, perhaps even more basic than buttons, and many user interface constructs can be modeled as groupings of links. HTML has a structure ideal for grouping like objects: the unordered list.

AT can gain a great deal of information about the relationship of items in a list, and about the relationships between nested lists. This approach also offers a benefit to keyboard users without AT, as most browsers will follow the structure and nesting of the lists to determine tab order.

Because these are links, like the earlier examples, the functionality will be available alongside any other links in specialized AT user interface.

In all of the examples below, sub-lists are hidden by default and shown only after the user activates a link. Many techniques for exposing navigation menus to users with disability rely on making the sub-lists visible at all times. While that approach does make the lists available, it creates a great deal of noise for AT users, often in repeated navigational elements they would prefer to ignore.

Many common user interface constructs can be modeled as lists of links or nested lists of links. Below, nested lists of links are used for a navigation bar, a toolbar with dropdown menus, and a tree. The underlying structure is the same: a hierarchy of actionable items.

4.1.2.1 Menu Bar

A menu is a list of actions related to a parent category on a toolbar. For example, “Save” and “Open” are actions related to the “File” category. So, the best semantic representation for a menu is a list nested under the toolbar, which is also a list

The actionable elements of the toolbar and its child menus are created with links, and the script to hide and show the menu is triggered from the onclick event of the toolbar links. CSS is used to make the underlying structure of the nested lists appear as a toolbar and menus in visual user agents.

In this example, the menus start out in the DOM as nested lists, and are hidden and shown with script that modifies the display CSS property. The menus could also be constructed in script, as long as the resulting DOM matched the one below, with the menu as a nested list.

```
<ul class="menubar">
  <li class="first"><a href="Cat1.htm">Category 1</a></li>
  <li><a href="Cat2.htm"
onclick="ToggleMenu(event);">Category 2</a>
    <ul>
      <li><a href="SubCat21.htm">Sub-Category 2.1</a></li>
      <li><a href="SubCat22.htm">Sub-Category 2.2</a></li>
      <li><a href="SubCat23.htm">Sub-Category 2.3</a></li>
    </ul>
  </li>
  <li><a href="Cat3.htm">Category 3</a></li>
</ul>
```

The ToggleMenu() function introduces the script that is used with little variation in the related techniques. After harmonizing the Firefox and Internet Explorer events, the child list element is located, and its display is toggled between "block" and "none". If displayed, the menu is positioned using a helper function.

```
function ToggleMenu(evt)
{
  HarmonizeEvent(evt);
  evt.preventDefault();

  var src = evt.target;
  var menu = src.parentNode.getElementsByTagName("UL")[0];
  if (menu)
  {
    if ("block" == menu.style.display)
    {
      menu.style.display = "none";
    }
    else
    {
      menu.style.display = "block";
      PositionElement(menu,src,false,true);
    }
  }
}
```

The CSS for the menu bar lays the toolbar list out horizontally, and sets the position of the menu to absolute, its display to none, and its float to none.

```
/* menubar and menus */
ul.menubar, ul.menubar ul { margin:0; padding:0; list-style-
type:none; }
ul.menubar li { float:left; border:1px solid gray; border-left:none;
  background-color:#eeeeee; }
ul.menubar li.first { border-left:1px solid gray; }
ul.menubar li a { color:black; text-decoration:none; width:100%;
height:100%;
  padding:0.2em 2em 0.2em 0.5em; width:8em; display:block; }
ul.menubar a:hover { text-decoration:underline; }

/* menu specific */
ul.menubar ul { position:absolute; display:none; border:1px solid
gray; }
ul.menubar ul li { float:none; border:none; }
ul.menubar ul li a { width:10em; }
```

4.1.2.2 Click Menu

This technique is almost identical to the related Menu Bar technique, except that the visual display of the top-level list is vertical. A vertical navigation bar is a group of hierarchically equal items, and the HTML list elements (<ul> and <ol>) are ideal for expressing these semantics. The list element provides the grouping mechanism, and a list item represents each node in the group. A menu is really a list of actions related to the category on the navigation bar, so the best semantic representation for a menu is a list nested under the navigation bar list item. The actionable elements of the navigation bar and its child menus are created with links, which ensures that they are available in a device-independent manner. Script to hide and show the menu is triggered from the device-independent onclick event of the navigation bar links. CSS is used to make the underlying structure of the nested lists appear as a navigation bar and menus in visual user agents.

```
<ul id="navMenu">
  <li><a onclick="ToggleMenu(event);"
href="Cat1.htm">Category 1</a>
    <ul>
      <li><a href="SubCat11.htm">Sub-Category 1.1</a></li>
      <li><a href="SubCat12.htm">Sub-Category 1.2</a></li>
      <li><a href="SubCat13.htm">Sub-Category 1.3</a></li>
    </ul>
  </li>
  ...
</ul>
```

As is the case in the other tree samples, ToggleMenu() looks for a child list element and then toggles its display property, also tracking an activeMenu property, so that at most one menu is displayed at any given time.

```
function ToggleMenu(evt)
{
  HarmonizeEvent(evt);
  evt.preventDefault();

  var src = evt.target;
  var menu = src.parentNode.getElementsByTagName("UL")[0];
  if (menu)
  {
    var active = window.activeMenu;
    if (active && active != menu)
    {
      active.style.display = "none";
      window.activeMenu = null
    }
    if ("block" == menu.style.display)
    {
      menu.style.display = "none";
      window.activeMenu = null
    }
    else
    {
      menu.style.display = "block";
      PositionElement(menu,src);
      window.activeMenu = menu;
```

```
    }
  }
}
```

4.1.2.3 Tree Control

A tree control is a list of items which can be expanded to show more items under them, and collapsed to hide them. The tree control, much like menus, is semantically a set of nested lists of links. CSS and script are used to make the list look and behave like a tree control.

The top-level list items are links which include both an image that shows open/closed state, and a text description of the top-level category. The images are of a + and -, with those symbols in text as the alt attribute. The onclick event of this link triggers the script that hides and shows the sub-lists using CSS, swaps the + and – images, and updates the alt attribute. All actual navigation is in the leaf nodes of the nested lists. The other links just trigger the show/hide scripts. This type of tree can be nested as deeply as the author wishes.

The HTML for the tree is a set of nested list elements. Note that the ToggleTree() event-handler is bound to the <a href> element. Also note the "+" alt attribute on the image.

```
<ul id="tree">
  <li><a onclick="ToggleTree(event,this);"
href="Cat1.htm"><img alt="+"
    height="9" width="9" src="img/p.png" />Category 1</a>
    <ul>
      <li><a href="SubCat11.htm">Sub-Category 1.1</a></li>
      <li><a href="SubCat12.htm">Sub-Category 1.2</a></li>
      <li><a href="SubCat13.htm">Sub-Category 1.3</a></li>
    </ul>
  </li>
  ...
</ul>
```

The ToggleTree() function cancels the default action for the <a href> and then goes to work finding the list item's child list element. Once that's found, the display of that child list element is toggled, and the list item's image and alt are swapped.
This example uses hide/show of lists that already exist in the HTML, but you could also inject the sub-lists from script, as long as you end up with the DOM shown above after the injection. Obviously, the script for this would be somewhat different than that shown below.

```
function ToggleTree(evt,src)
{
  HarmonizeEvent(evt);
  evt.preventDefault();

  var node = src.parentNode.getElementsByTagName("UL")[0];
  if (node)
  {
    var img = src.getElementsByTagName("img")[0];
    if ("block" == node.style.display)
    {
      node.style.display = "none";
      if (img)
      {
        img.src = "p.png";
        img.alt = "+";
      }
    }
    else
    {
      node.style.display = "block";
      if (img)
      {
        img.src = "m.png";
        img.alt = "-";
      }
    }
  }
}
```

The CSS for the tree is fairly simple. Margins, padding and list glyphs are turned off, and child list elements are hidden

```
ul#tree, ul#tree ul { list-style-type:none; margin:0; padding:0; }
ul#tree a { color:black; text-decoration:none; height:1.5em;
padding:0 0.2em;
        display:block; display:inline-block; }
ul#tree a img { border:0; vertical-align:middle; margin-
right:0.2em; }
ul#tree ul { display:none; margin-left:2em; }
```

4.1.3 Form Semantics: Inserting inline error messages

The label element in HTML is used to semantically tie text to a form element. In this example, we modify it at runtime to contain an error message related to the input in the field, explicitly associating the error with the element in a way that is available to AT.

When a user is filling out a form, it is often a good user experience to give immediate feedback on errors visually near the element where the error occurred. There are two aspects to making this compatible with AT. The first is to ensure that the order of the DOM matches the visual order after the insertion. The second is to expose the error message to assistive technology in a way that associates it with the form element where the error occurred.

```
<script type="text/javascript">
function validate()
{
        //do validation
        //return error
        phoneLabel.innerText = "Telephone number must be 9
digits";
}
</script>

<label id="phoneLabel" for="phone">Telephone Number</label>
<input id="phone" type="text" />
```

After the script runs, the DOM will look like this. The value of the input is what the user typed.

```
<label id="Label1" for="phone">Telephone number must be 9
digits</label>
<input id="Text1" type="text" value="999" />
```

The user will encounter the new label the next time he interacts with that form element. In JAWS, for example, the next time the element is focused, the new label will be read “Telephone number must be 9 digits edit.”

4.1.4 Functional Accessibility: DHTML Slider Control

This example shows a slider control implemented by using core HTML elements in a semantically correct way. There is no native HTML slider; therefore, it is not possible to expose a role of "slider." To create an accessible slider, you must understand the functionality of a slider and model that in semantically correct HTML.

A slider must allow the user to increase or decrease the value by using accessibility interfaces and expose the current value to accessibility interfaces. To increase usability for all users, it should also allow the user to move the slider control (or nib) with the keyboard arrow keys or with the mouse.

A slider can meet these requirements as follows:

- Allowing the user to use accessibility interfaces to increase or decrease the value: Image links use an onclick event to trigger scripts that increase and decrease the value by one. The onclick event on links and buttons is device independent and exposed to accessibility interfaces. Additionally, triggering such a link sends a navigate event to the accessibility interface.
- The images are + and - icons. These icons can be changed, depending on the functionality of the scroll bar in the application. The image links have descriptive alt text. For example, the text could be “increase by 1” and "decrease by 1". The links trigger the same scripts that are called when the slider nib is moved with the mouse or keyboard.
- Exposing the current value to accessibility interfaces: The slider nib is an image with alt text, and the alt text includes the value and units; for example, “3 widgets”. The increase and decrease scripts change the alt text to the new value, in addition to moving the nib visually.
- Allowing the slider nib to be moved with the keyboard arrow keys: The slider nib image is surrounded by a link (<a href>), which exposes it in the tab order. The slider nib link includes an onkeypress handler. The onkeypress handler calls the increase function for a right arrow key press and the decrease function for a left key press. These are the same scripts used with mouse or link operation.
- Allowing the slider nib to be moved with the mouse: The slider nib link includes ondragstart and ondragstop handlers. The ondragstart handler calls the same increase function when the nib is dragged to the right and the same decrease function when the nib is dragged to the left.

The following example implements the first two items from the list above.

The example HTML for the slider has a link around each of the two slider decrease and increase images. Note that the alt attribute values indicate that clicking these activates the slider. The empty alt on the button image is set in the script to the current value.

For ease of illustration, all of the standard (and inaccessible) mouse handler code used to move the slider image with the mouse has been removed.

```
<div class="slider">
<a href="#" onclick="OnSliderClick(event,-1);"><img alt="Show
fewer lines"
   height="17" width="22" src="less.gif"" /></a>
<img id="btn" height="17" width="10" alt="" src="btn.gif" />
<img id="bar" height="17" width="76" alt="" src="bar.gif" />
<a href="#" onclick="OnSliderClick(event,1);"><img alt="Show
more lines"
   height="17" width="22" src="more.gif"" /></a>
</div>
```

The two functions in the script handle the image link clicks and the positioning of the button. Note that the alt attribute on the button image is updated to reflect the new position.

```
var pos = 0;

function OnSliderClick(evt,dir)
{
   HarmonizeEvent(evt);
   evt.preventDefault();

   var newpos = pos + dir;
   if (0 <= newpos && newpos < range.length)
   {
      pos = newpos;
   }
   PositionButton();
}

function PositionButton()
{
   var btn = document.getElementById("btn");
   var bar = document.getElementById("bar");
   PositionElement(btn,bar);
   btn.style.left = btn.offsetLeft - 4 + ((bar.offsetWidth/3 + 12) *
pos) + "px";
   btn.alt = range[pos] + " lines";
}

window.onload = PositionButton;
```

The experience in a screen reader is of two image links and an image. Activating either of the links changes the alt value of all three images, which may then be read with the normal text interaction mechanisms.

4.2 DHTML: Semantic Script

The examples above use script in a way that fits with the inherent semantics of HTML. The click events of links and buttons, which are exposed to AT, are used as the building blocks of user interactivity. In addition to the semantics of the elements used, the order of those elements is also essential to creating a DOM which can be used by a variety of AT. The reading order in a screen-reader is based on the order of the HTML elements in the DOM, as is the default keyboard navigation, or tab, order.

This technique inserts new content into the DOM immediately following the element that was activated to trigger the script. The triggering element must be a link or a button, and the script must be called from its onclick event, as described above.

The user experience with the keyboard, both for sighted users and for screen-reader users, is smooth and natural. The user tabs to

the link and hits enter. The focus remains on that link. The new content, whether it's a sub menu or a dialog, is the next item in both the reading order and the tab order. When the user tabs out of the new content, he returns to the original tab order of the page.

Note that this technique works for synchronous updates. For asynchronous updates (sometimes called AJAX), an additional technique is needed to inform the assistive technology that the asynchronous content has been inserted.

This example creates a menu when a link is clicked and inserts it after the link. The onclick event of the link is used to call the ShowHide script, passing in an ID for the new menu as a parameter.

```
<p><a href="#" onclick="ShowHide('foo',this)">Toggle</a></p>
```

The ShowHide script creates a div containing the new menu, and inserts a link into it. The last line is the core of the script. It finds the parent of the element that triggered the script, and appends the div it created as a new child to it. This causes the new div to be in the DOM after the link. When the user hits tab, the focus will go to the first focusable item in the menu, the link we created.

```
function ShowHide(id,src)
{
        var el = document.getElementById(id);
        if (!el)
        {
                el = document.createElement("div");
                el.id = id;
                var link = document.createElement("a");
                link.href = "javascript:void(0)";

        link.appendChild(document.createTextNode("Content")
);
                el.appendChild(link);
                src.parentElement.appendChild(el);
        }
        else
        {
                el.style.display = ('none' == el.style.display ?
'block' : 'none');
        }
}
```

The experience in screen readers is as though the content was there all along, it is available to the user when he reaches the updated area of the page.

4.3 AJAX: Notify AT of Asynchronous Updates

Asynchronous content injection (sometimes called JavaScript And Xml [AJAX]) presents one specific challenge for accessibility: that of letting screen readers know when content has been updated on the page. This is an issue because screen readers work by taking a snapshot of the page after it loads and then allowing the user to move around the snapshot. A screen reader must receive a navigate event to update the snapshot. The browser generates a navigate event in response to an onclick event on a link <href>.

AJAX works by updating only those page elements that actually change. Therefore, the snapshot is not updated when an AJAX call subsequently updates a single element on the page, such as when a user adds a stock to a stock tracking application or changes the city in a weather application. The browser does not initiate an event, and the AT device snapshot does not update its snapshot.

Screen readers take a snapshot of Web content and store it in a buffer, and users interact with that snapshot. When a user navigates to a new page, a navigate event is sent to the operating system, and the screen reader catches that event and updates its buffer. This also happens when a script is triggered from an onclick event on a link or button. With asynchronous updates, the content is returned and added to the DOM after the navigate event is processed. Adding an asynchronous event to the DOM does not generate any events that the screen reader can trap, and it is not possible to generate the navigate event directly from script.

However, a navigate event is generated for any navigation, including navigation inside a <frame> or <iframe>. The technique creates a hidden <iframe>in script and navigates to it at the end of the function that retrieves content from the asynchronous call. The <iframe>is navigated to a unique URL, created from a time stamp to avoid cache hits and to ensure that the event is generated, and location.replace is used to avoid cluttering the user agent history. After the returned XML is confirmed, the callback function pastes the new content into the DOM or produces an error message. Then, the code generates a unique URL value and calls location.replace() on the hidden <iframe>.

```
<script>
function Callback(response)
{
   var target = document.getElementById("output");
   var out = null;
   if ("200" == response.status)
   {
      out = response.responseText;
   }
   else
   {
      out = response.status + ":" + response.statusText;
   }
   target.innerHTML = out;

   // create a unique href and navigate the iframe
   var url = 'blank.htm?' + (new Date()).getMilliseconds();
   window.frames['nav-i-frame'].location.replace(url);
}
</script>
```

The following is the HTML for the hidden <iframe>.

```
<iframe tabindex="-1" id="nav-i-frame" name="nav-i-frame"
   src="blank.htm" height="0" width="0"></iframe>
```

Like the previous example, the experience in a screen reader is as though the content was there all along. It is available to the user when he reaches the updated area of the page.

5. CONCLUSION

Most web applications today use a common set of user interface elements, which can be mapped fairly closely to the semantics available in native HTML elements. By understanding the semantics of HTML elements and the events associated with them, developers can model most of the user interface interactions

desired for today's web applications in a way that is functionally accessible with today's user agents and assistive technology.

6. REFERENCES

[1] Schwerdtfeger, R. et al. *Roadmap for Accessible Rich Internet Applications (WAI-ARIA Roadmap).* World Wide Web Consortium Recommendation Working Draft, 2006.

[2] Shelly, C., Young, G. *Writing Accessible Web Applications*. Microsoft Developer Network, 2006.

[3] Vanderheiden, G. et al. *Web Content Accessibility Guidelines 1.0,* World Wide Web Consortium Recommendation, 1999

[4] Vanderheiden, G. et al. *Web Content Accessibility Guidelines 2.0,* World Wide Web Consortium Recommendation Working Draft, 2006.