ChromeVox
A Screen Reader Built Using Web Technology

T. V. Raman, Charles L. Chen, Dominic Mazzoni,
Rachel Shearer, Chaitanya Gharpure,
James DeBoer, David Tseng
Google Inc
1600 Amphitheatre Parkway
Mountain View, CA 94043

April 26, 2012

Abstract

The evolution of the web programming model defined by the HTML DOM, CSS and JavaScript has moved us from a web of documents to a web of applications. We leverage this evolution to build ChromeVox — a screen reader for Chrome OS that is built as a web application. By creating this accessibility solution using the same technical underpinnings as modern web applications, we are able to support the latest in accessibility standards including HTML5 and W3C ARIA to enable complete access to rich web applications. By being an adaptive technology that is built for the web, ChromeVox is able to move beyond the status-quo when it comes to enabling access to full-featured web applications such as document editors and spreadsheets. Thanks to the rapid progress in the web programming model, ChromeVox is helping us bring the same pace of rapid innovation to the field of web access that mainstream users have come to expect. By building ChromeVox on top of the WebKit-based Chrome, we are able to bring ChromeVox to a multiplicity of platforms, including Android.

1 Introduction

The move to Web 2.0 is the move from a web of documents to a web of applications, where applications are in turn built out of web parts — see [15]. Furthermore, web applications are now significantly easier to build because the interaction layer can be built using HTML, CSS and JavaScript — see [10, 2, 5]. Visually rich applications ranging from interactive maps to fully functional word processors can now be built entirely using web-centric technologies — see [6].

The web programming model is also evolving to support rich user interaction that matches that found on traditional desktop clients. This in turn leads to the question

> Can we build adaptive technologies such as screen readers for the blind using this same web programming model?

ChromeVox answers this question in the affirmative by implementing a complete screen reader for web applications within Chrome. In the rest of this paper, we describe the architecture of ChromeVox, after first outlining the differences in architecture from screen readers built for traditional operating systems such as Windows and Mac OS.

1.1 Architecture: Screen Readers

First, we briefly outline the architecture of adaptive technologies such as screen readers found on traditional operating systems. Here are the design requirements and constraints under which these were designed:

- Applications such as word processors, spreadsheets and mail readers implement platform-specific APIs that provide accessible information and control of their user interfaces.
- Adaptive technologies run as separate processes and communicate with applications using these APIs.
- These adaptive technologies expose commands and capabilities to users, and provide alternative feedback via text-to-speech or braille.
- When generic platform APIs fall short, many applications expose custom interfaces that adaptive technology can leverage to create a customized experience for that particular application. In other cases, some adaptive technology may resort to hacks including "screen scraping" to provide a more accessible experience for users.

Microsoft Windows accessibility APIs include Microsoft Active Accessibility (MSAA), which was the original accessibility API for Windows; IAccessible2, an extension to MSAA developed by the Linux Foundation; and UI Automation, a newer API from Microsoft. All of these Windows APIs are Component Object Model (COM) interfaces, which are supported in almost every major programming language. Mac OS X and iOS support the NSAccessibility protocol, which is available to any application built using the Objective-C Cocoa APIs, and a bridge between NSAccessibility and Java's accessibility API is built-in. Most Linux distributions support the GNOME Accessibility Toolkit (ATK), which was developed jointly with the IAccessible2 API for Windows and shares a compatible semantic programming model. Adaptive technologies such as screen readers that leverage these APIs to provide accessibility may be provided by the operating system or by third parties.

In this environment, an *accessible* application is one that implements the platform-specific accessibility API within its user interface. Because native operating system widgets and many other user interface toolkits already implement the accessibility APIs, applications built exclusively using these widgets may be accessible without any extra work on the part of the application developer. However, such an application may still be more difficult for an adaptive technology user to work with if the conceptual design of the application made assumptions, *e.g.*, that the user can see the spatial layout of controls. Thus, the application *publishes* information that is of interest to adaptive technology, *e.g.*, the label of the currently focused widget. Screen Readers *subscribe* to this information via the platform accessibility API 1. When working with an *accessible* application, the screen reader receives notifications via the accessibility API and subsequently provides appropriate alternative feedback, *e.g.*, by speaking the label of the widget that is focused. Additionally, adaptive technologies like screen readers can query application state and in certain cases even drive the application, *e.g.*, by setting focus to a given user interface control.

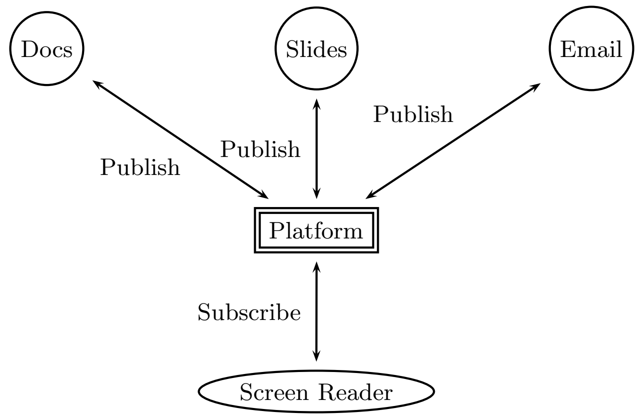

Figure 1: Applications communicate bidirectionally to a screen reader via platform accessibility APIs.

1.2 Architecture: ChromeVox

Chrome OS is an operating system created by Google. A key insight behind its design is that it is now possible to do most (if not all) of one's work within a web browser. Building on this insight, Chrome OS machines boot up within seconds of turning on and immediately launch Chrome; from that point onwards all user interaction happens within the browser and all applications are implemented as web applications.

We created ChromeVox to be a screen reader built into Chrome OS which led to the following design requirements:

- All applications such as word-processors, mail readers and spreadsheets are implemented using HTML, CSS, and JavaScript.
- All user interaction happens within the web browser.
- The screen reader needs to respond to user interaction within all browser tabs. Though the user is accessing web content the majority of the time, the screen reader must still provide access to everything else on the system — toolbars, menus, and dialogs that are part of Chrome and Chrome OS and which might not be implemented in the web programming model.
- Alternative feedback such as speech feedback needs to be either implemented using web technology, or provided on the platform and accessed via web APIs.

Since all applications on Chrome OS are *web* applications running in Chrome, and since an increasing proportion of the platform's user interface is itself implemented in the web programming model, we chose to do away with the traditional metaphor of applications communicating via an accessibility API intermediated by the operating system. Instead, ChromeVox uses the HTML Document Object Model (DOM) as the means of communicating with web applications running within Chrome 2.

For the small portion of the user interface that is not implemented in HTML, we expose these interaction elements to JavaScript via a light-weight custom interface. User interface elements in the "chrome of Chrome", *e.g.*, toolbar buttons, menus, and the address bar, raise events in response to user interaction that in turn produce appropriate alternative feedback through ChromeVox. But as HTML evolves to support ever-richer user interaction, we are seeing some of these native user interface components migrating to a web interaction model, possibly obviating these additional APIs over time.

2 Overview: Chrome Extensions

ChromeVox is built as a Chrome extension — see [9]. A detailed explanation of Chrome extensions is beyond the scope of this paper; however, we briefly outline the design of Chrome extensions in the context of the ChromeVox implementation.

A Chrome *extension* is a zipped bundle of files— HTML, CSS, JavaScript, images, and anything else needed by the extension — that adds functionality to the Google Chrome browser. Chrome extensions were inspired by Mozilla Firefox add-ons — in particular Greasemonkey, a Firefox add-on that made it easy to inject custom JavaScript into specific webpages — see [12]. A Chrome extension consists of primarily two parts: injected content and background content. The injected content of an extension consists of style sheets and JavaScript code that can be added to every running page. Content scripts run in a separate JavaScript context called an *isolated world* that does not share any global

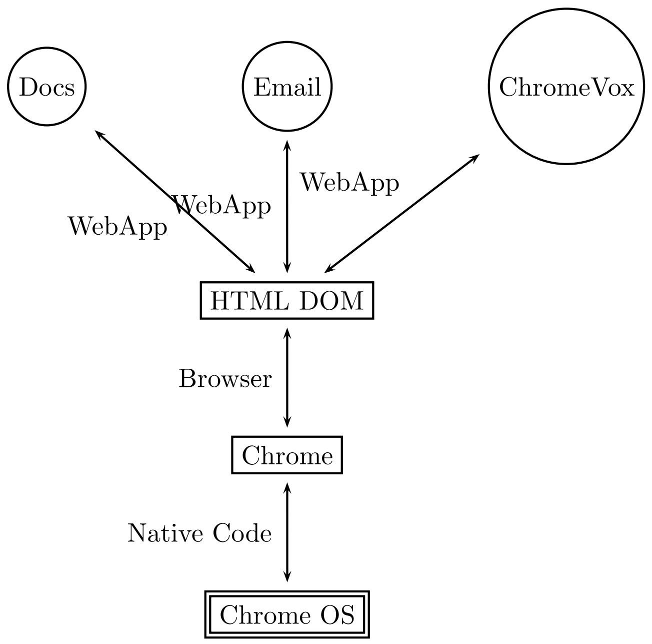

Figure 2: Web Applications including ChromeVox communicate with Chrome via the HTML DOM.

state with the scripts running in the page, while still having access to the DOM. Note that a content script functions *not* as part of the extension that it was packaged with, but as part of the loaded page into which it is *injected*. Background content consists of custom UI that the extension wants to show (built like any web app, with HTML, CSS, and JavaScript) and purely background scripts that are used to coordinate between different pages or keep track of global state. JavaScript running in the background portion of an extension can access some APIs that are not normally available to web pages, including information about bookmarks and tabs, speech APIs, and the ability to make web requests without the same-origin restriction. Content scripts and background scripts communicate using message passing.

A Chrome extension can provide any or all of the following types of functionality:

- Web mashups that present an optimized interface to oft-performed tasks.
- Additional user interface affordances such as browser action icons for performing common actions.
- New behaviors for web pages and applications in the browser.

3 APIs used by ChromeVox

ChromeVox uses a number of relatively new APIs that are part of the HTML5 web platform. These APIs are available to any web author targeting a modern browser, but some of them are particularly useful for accessibility.

ChromeVox also takes advantage of a number of extension APIs that give it capabilities not normally possible in a webpage.

3.1 HTML5 Audio

In addition to spoken feedback, ChromeVox augments the user interface with auditory icons —[14, 1]. ChromeVox uses auditory icons to identify specific HTML constructs such as list items and hyperlinks, or to indicate when events have occurred such as when a page is fully loaded and ready for user interaction. We use HTML5 Audio to play earcons from prerecorded sound files; this is an example of using new HTML5 APIs from within a Chrome extension.

3.2 Mutation Observers

Mutation Observers are a way for a JavaScript program to register for and receive callbacks every time a change is made to the DOM. They replace and deprecate the DOM mutation events that were partially implemented in some web browsers but proved to be impossible to implement completely without serious performance implications.

Mutation Observers are important to ChromeVox because of W3C ARIA *live regions* — see [16, 13]: when a web author has marked part of a page as a live region, changes to that part of the page should be announced to the user right away, even if those changes were not directly the result of a user interaction. For example, a live region might display new headlines or chat messages. With Mutation Observers, ChromeVox can observe all changes to the page as they occur and decide which ones need to be announced.

3.3 Text-to-speech extension API

Text-to-speech (TTS) is an essential service for implementing a screen reader. Chrome includes a TTS extension API that allows an extension that requests this permission to generate synthesized speech. On Chrome OS, a free speech engine is included with the platform as a baseline, but other Chrome extensions can build on this functionality by providing additional voices using a companion TTS engine API. As of this writing, there are several additional voices implemented using Native Client (see [7]) that can be installed directly from the Chrome Web Store.

There are plans to make TTS part of the web platform. We are closely following the *html-speech* activity at the W3C and in the future we expect that web applications will be able to generate synthesized speech using open, cross-browser standards, without relying on extension APIs.

Note that some screen readers support braille output in addition to, or instead of, speech output. The current version of ChromeVox does not yet support braille, so the discussion in this paper is therefore speech-centric.

3.4 Tabs extension API

ChromeVox uses the Tabs extension API in order to get information about all of the open tabs and windows. It uses this API, for example, to announce when the user has switched between tabs, or to broadcast state information from a background script to each of the content scripts.

3.5 Accessibility extension API

Chrome also provides an Accessibility extension API that allows an extension to listen to events generated outside of the webpage, in what we call the "chrome of Chrome", *e.g.*, toolbar buttons, menus, and the address bar. This API is completely implemented on Chrome OS, allowing ChromeVox to describe every UI interaction. On other operating systems, this API does not fully support every Chrome feature; instead, ChromeVox is meant to be used alongside a desktop screen reader that will describe the user interface outside of the webpage, while ChromeVox describes what's in the webpage.

4 ChromeVox

This section details the ChromeVox lifecycle. When it is first installed or enabled, initially only the ChromeVox background script is loaded. The background script loads persistent preferences, registers event listeners from extension APIs, and then injects the ChromeVox content script into every running tab. When the user navigates to a new page or opens a new tab, the ChromeVox content script is run automatically from then on.

The content script first receives persistent preferences from the background script, registers event listeners, and then describes an overview of the page to the user. In order to speak, the content script sends a message to the background script with the utterance and various speech parameters, then the background script uses the TTS extension API to speak.

Once initialization is done, ChromeVox has no running code; it is entirely event-driven, and all of its functionality from here on is based on listening to events generated by the user, or events generated by the page.

ChromeVox listens for the following types of events:

Navigation commands Key-down events matching ChromeVox shortcut keys are intercepted and used to control ChromeVox to navigate the DOM using a variety of *navigation* strategies 4.1, or interact with the page in other ways. Many of these shortcuts directly manipulate the page — for example by moving focus, following a link, or changing a control value — while others are used for passively reading, searching, or otherwise describing the content of the page.

User events Other event listeners respond to user-initiated actions such as moving focus, or typing in editable text controls. In these cases the goal of ChromeVox is to make sure that all of the native interactions provided automatically by the browser are faithfully described to the user. ChromeVox does not override any browser behavior unless absolutely necessary; this ensures that ChromeVox users have the same experience as other users when interacting with web sites and web apps.

Page events ChromeVox also listens for events and notifications from the page that were not user-initiated. The primary use of this is for W3C ARIA live regions, as discussed earlier.

API messages ChromeVox exports its own API that web pages or other extensions can use to communicate with ChromeVox and trigger custom feedback or navigational commands.

4.1 ChromeVox Navigation

When a sighted user reads a web page passively, the extent of their interaction with the browser is typically scrolling the page. When a screen reader user reads a page, listening to the contents of the entire page from top to bottom using synthesized speech may be prohibitively expensive. Instead, the user needs to skim content rapidly and jump through the page.

ChromeVox enables flexible navigation of web pages by giving the user the ability to pick a specific browsing *granularity* at which level they wish to move through the document (see the ChromeVox tutorial [8] for details). As the user traverses the document at the specified granularity, ChromeVox speaks the virtual selection that becomes current, augmented with auditory icons as appropriate. At present, ChromeVox provides the following granularity levels:

Group Content grouped using a heuristic that tries to keep related content together. For example, an HTML paragraph gets treated as a single group.

Object One HTML `object` at a time. Here, HTML elements are considered to be `objects`. Thus, when traversing the following using *object* granularity,

$\boxed{\text{This is}}$ $\boxed{\texttt{<span>a</span>}}$ $\boxed{\text{test.}}$

the content would get split into three pieces as shown above.

Sentence One sentence at a time.

Word One word at a time.

Character One character at a time.

In addition, ChromeVox provides specialized commands for table navigation. When using table navigation, the user can traverse table cells with navigational arrows using a two-dimensional model based on rows and columns. Navigation by table cells speaks the content of the current table cell; ChromeVox also provides the needed functionality to query for row and column headers. Table navigation in ChromeVox incorporates heuristics for distinguishing between *data* and *layout* tables — see [11]. When ChromeVox detects a data table, the user is automatically placed into table navigation mode. Thus, content appearing within tables used purely for layout purposes is typically traversed as text content.

4.2 Earcons

In addition to spoken feedback, ChromeVox augments the user interface with auditory icons —[14, 1]. ChromeVox uses auditory icons to identify specific HTML constructs such as list items and hyperlinks, or to indicate when events have occurred such as when a page is fully loaded and ready for user interaction.

4.3 Visuals

While the primary target audience is blind users, ChromeVox provides rich visuals that highlight its interactions with the page. These are important for developers and testers, for side-by-side use of the same computer by blind and sighted companions, and for low-vision users who need a hybrid interface.

ChromeVox adds visuals by adding new elements to the web page DOM that are rendered on top of page content by giving them a large z-index. The current visuals include:

Active Indicator A glowing outline around the current object or text rage that ChromeVox is focused on; this is synchronized with browser focus but can also land on static content that's not normally focusable.

Lens A visual transcript of the text that ChromeVox spoke in a large font. This enables sighted testers and developers to quickly check accessible behavior without listening to speech (which may be cumbersome for users who aren't used to it), while also providing a hybrid interface for low-vision users.

4.4 User Interface Customization

ChromeVox user preferences are persisted using HTML5 local storage and are initialized from previously saved preferences at start-up. Persisted user preferences include:

TTS Voice The user's preferred voice.

Speech Rate Rate at which text is spoken.

Pitch Default pitch of the speaking voice.

Key bindings All ChromeVox keyboard shortcuts are user configurable.

Verbosity The amount of detail ChromeVox speaks on average.

Visuals Options controlling which visuals are dispayed and where.

ChromeVox user settings are surfaced to the user via an *options* page. This user interface is rendered in HTML, and being a web page can be navigated with ChromeVox providing spoken output. In addition, oft-used settings such as speech-rate and pitch can be quickly adjusted using keyboard short-cuts.

4.5 ChromeVox User Experience

ChromeVox is built into Chrome OS, and is available as a one-click install from the Chrome Web Store for other platforms. On Chrome OS, users can turn on accessibility at the login screen by pressing `Ctrl-Alt-Z` — pressing this key again turns off accessibility. Once turned on, accessibility is *sticky i.e.*, it persists across reboots.

On Chrome OS, virtually the entire user experience, including the initial setup and login screen, are built using HTML. As a result, once ChromeVox is enabled, it provides accessibility to those screens as well as any web page.

For a detailed overview of the ChromeVox user experience, please see the interactive on-line tutorial [8].

5 Accessing Rich Internet Applications

The move to Asynchronous JavaScript and XML (AJAX-style) web pages led to screen readers losing ground with respect to making interactive web content accessible — with JavaScript code now defining the behavior of elements on the page, screen readers could no longer infer the interaction semantics of user interface controls on the page just from the HTML. Access to Rich Internet Applications (W3C ARIA [13, 16]) helps adaptive technologies regain much of this lost ground by adding the following authoring features to HTML content:

Role Used to annotate HTML elements with interaction semantics, *e.g.*, *menu*. This avoids adaptive technologies having to discover these semantics by examining the DOM (which is difficult and error-prone) or by examining JavaScript code (which is impossible, as JavaScript is Turing-complete).

Properties Additional properties, *e.g.*, the range of admissible values for a *slider* along with its current value.

State The current interaction state of controls. These can be dynamically updated by JavaScript code in response to changes *e.g.*, to indicate that a toggle button is *pressed*.

Live Regions Attributes to indicate that a piece of content on a web page may be updated dynamically; this enables conformant browsers to raise an event informing adaptive technologies of changes to portions of a web page. A screen reader can then respond by speaking updated content on the page.

W3C ARIA is now a Candidate Recommendation (CR) and is supported to varying degrees by today's modern browser and screen reader combinations. ChromeVox implements a large percentage of the ARIA specification, and it is our goal to implement all aspects of the final W3C Recommendation. Below, we illustrate the differences that arise when implementing W3C ARIA within a screen reader built as a web application versus a traditional screen reader. For a high-level overview, see figures fig. 3 and fig. 4

ARIA Support In Traditional Screen Readers

ARIA support in a traditional screen reader requires the following actors to collaborate:

Web App Making a web app accessible with ARIA requires the collaboration of the web application — in this case, that translates to web authors adhering to good authoring practice.

Browser The browser needs to consume ARIA markup and expose platform-specific accessibility objects to native screen readers — HTML elements augmented with ARIA roles get mapped to their platform-equivalent widgets. Events on these generic HTML objects need to get translated into platform-specific events on the accessible objects. For features that have no corresponding platform equivalent, browsers need to expose custom details.

Platform The platform needs to propagate these objects and events to the adaptive technology (AT).

AT Adaptive technology needs to generate the appropriate feedback for these objects and events. In many cases, widgets built using ARIA may not resemble native platform widgets in detailed form and function, even though they user the same roles. As a result, describing them may require a lot of custom handling.

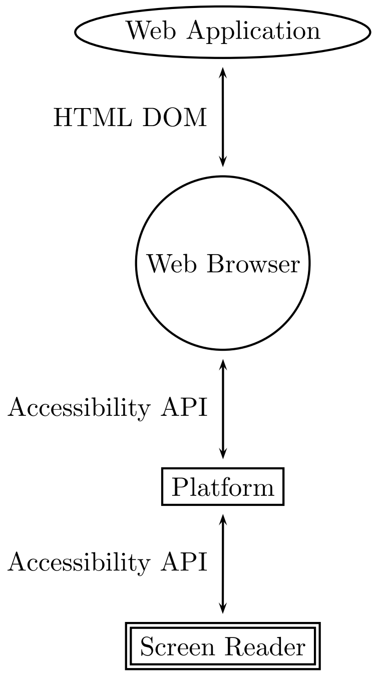

Figure 3: W3C ARIA in a traditional screen reader.

ARIA In ChromeVox

The work on ARIA started sometime in 2003, with early working drafts published in 2006. ARIA was first shown to be a viable solution when the desired behavior was implemented in Fire Vox[3, 17], a Firefox extension that turned Firefox into a self-voicing browser. In this instance, a complete implementation was possible without the multiple actors enumerated in the previous section needing to collaborate. Because FireVox was a Firefox extension that handled all aspects of the auditory user interface, it could directly respond to the ARIA markup present in the HTML DOM. The ARIA implementation in ChromeVox is inspired by the Fire Vox implementation.

The ChromeVox implementation of ARIA is achieved in the following steps. To deliver effective access to rich web applications, we only need the collaboration of the web application author.

Mapping Map controls identified via the various ARIA *role* values to their corresponding widget types.

Labels W3C ARIA defines a number of new authoring idioms that enable web developers to more flexibly label controls — this sets up the right associations so that the user hears the relevant label text spoken when navigating the page.

Configure Process attributes in the HTML DOM that define *properties* of user interface controls such as sliders.

Listeners Register *listeners* that process updates to the dynamic attributes that reflect the *state* of user interface controls.

Live Regions Attach JavaScript handlers that process updates to *live regions* to produce appropriate spoken feedback in response to dynamic updates to page content.

Notice that the implementation of ARIA in ChromeVox is significantly simpler than that observed with traditional screenreaders. The simplicity can be attributed to the following factors:

- Coordinating among fewer actors.
- No impedance mismatch in the programming model used by the web application being accessed and the screenreader that is used to make the application accessible — they're both web applications.

As shall be seen in the next section, this lack of impedance mismatch enables us to push the envelope with respect to implementing accessible web applications, and in the process discover new programming idioms that once proven could accelerate the coming of fully accessible web applications. One reason for the present lack of effective accessibility to web applications is the mismatch in the speed with which the web platform is itself evolving as against the pace of evolution seen in traditional desktop screen readers. The road to ARIA has been long and complex; we hope that in the future, the field of adaptive technology will come to see the same rapid pace of innovation that mainstream users on the web have experienced to date.

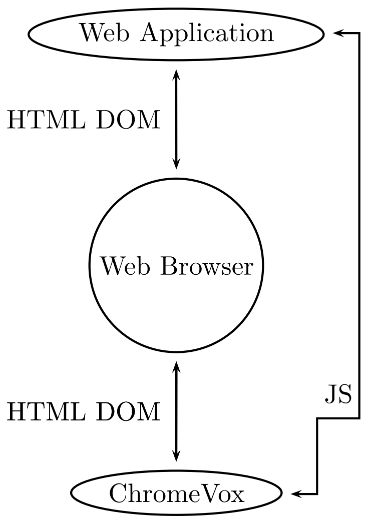

Figure 4: W3C ARIA in ChromeVox.

6 Exposing Accessibility APIs

As covered in sec. 1.2, ChromeVox is just another web application on Chrome OS — see fig. 2. It injects accessibility affordances for blind and low-vision users into all running web applications as part of its default behavior. In addition, ChromeVox exposes a light-weight API that allows web applications to customize the overall accessibility experience for a given application. This API allows us to experiment with new strategies for adding accessibility to web applications that stretch the limits of today's web platform. In this section, we'll illustrate this with a concrete example.

The word processor in Google Docs is designed to provide a desktop-like experience for WYSIWYGS text editing. A key design requirement was to enable collaboration across the web in all supported browser/OS combinations, and to enable reliable round-tripping of the document independent of browser differences in HTML serialization.

Google Docs achieves this by storing an in-memory model of the document content in JavaScript that is updated continuously in response to user input. As this model updates, the results are rendered to the visual display by appropriately updating the DOM, including drawing the visual caret and appropriately styling the presentation to match the user's intent. Notice that in this implementation, the content being edited is not directly serialized in the HTML DOM; instead, the implementation uses an underlying Model, View, Controller (MVC) abstraction where the document model as encapsulated in the internal JavaScript data structures are continuously reflected in the view layer represented by the HTML DOM. The controller, which handles user input is implemented as an off-screen text input field that the user never sees; all aspects of user interaction are managed by the web application itself, rather than relying on the browser.

The MVC design pattern adopted by Google Docs enables users on different browser/OS combinations to collaborate effectively without differences in HTML serialization among browsers affecting the ability to effectively round-trip content. Thus, this word-processor as designed has a number of advantages over more traditional approaches of using HTML's *content-editable* feature. However, it presents an interesting challenge to traditional approaches to access-enabling web applications that rely solely on the document markup containing the needed accessibility affordances. To illustrate, access technologies like screen readers watch for the insertion caret when producing spoken feedback within a word processor. However, in the described implementation, the insertion caret is drawn to the screen by the web application along with the stylized visual presentation of the content being authored; the controller that receives user input is itself never rendered to the screen.

To help provide better accessibility in web applications that extend the browser platform, ChromeVox exposes a light-weight API that enables Web developers affect the following aspects of the overall accessibility experience:

Speak Specify feedback to be spoken using ChromeVox' speech abstraction layer.

Earcons Provide auditory icons that can be played to augment the overall experience.

Synchronize Map the *current* position of the ChromeVox `NavigationManager` to the navigation position maintained by the application.

The Google Docs product uses this API to implement an accessible word processor in conjunction with ChromeVox. The same techniques are used by Google Spreadsheets; this provides an end-user accessibility experience that closely matches and will eventually surpass accessibility that users have come to expect in desktop applications.

6.1 Leveraging ChromeVox API

The previous section outlined how Google Docs uses the ChromeVox API to customize the spoken feedback provided by ChromeVox. In addition to the web application author calling the ChromeVox API, ChromeVox can directly include application-specific scripts that customize spoken feedback on a per-application basis. This type of application-level customization was first demonstrated in AxsJAX (see [4]) and we have applied the lessons learnt from that project within ChromeVox. As an example, ChromeVox leverages this functionality to provide better spoken feedback within Gmail and Google Search — we detail this briefly in the paragraphs below. In the case of Gmail, we use a Gmail JavaScript API to get programmatic access to the current message; in the case of Google Search, we leverage structure present in the page-level HTML markup.

Gmail ChromeVox uses the Gmail JS API to get access to the DOM node containing the currently displayed message and to get application level status information about that message. After *synchronizing* the current selection to this node, ChromeVox automatically *speaks* the email message. Having this application level logic is especially important as it is what enables ChromeVox to intelligently describe a message as being *unread, starred* rather than merely speaking the lower level visual representation of *bold, checked.*

Search The ChromeVox script for Google Search causes ChromeVox to speak the complete snippet for the current result when navigating the result list. In addition, pressing `enter` while hearing the snippet automatically opens the result, thereby obviating the need to explicitly navigate among the various links in the snippet.

7 Challenges

As we push the envelope on what is possible using today's HTML DOM, we have also discovered short-comings in today's underlying browser implementations and the specifications they implement. As an example, it's impossible for ChromeVox to present visuals without modifying the DOM. These modifications can be detected by web applications and can lead to compatibility problems when apps don't expect the DOM structure to change.

As another example, ChromeVox leverages the ability to attach event handlers and receive events via standard DOM event propagation. This works well with well-behaved web applications; however as web developers continue to experiment with different programming idioms, it is fairly easy for a web application to stop event propagation and inadvertently render ChromeVox ineffective within a given web application. We expect to overcome these types of challenges by extending the browser platform whereby specific extensions are given additional privileges *e.g.*, given ChromeVox precedence in the order in which events are received would solve this problem.

8 Conclusion

We set out to test the maturity of the web programming model with respect to building a screen reader as a web application. Our experience as detailed in this paper demonstrates the exciting new opportunities for adaptive technologies that are opened up by the web programming model. Above all, we can now bring the same rapid pace of innovation to adaptive technologies that mainstream users on the web have come to expect.

ChromeVox has delivered end-to-end access on Chrome OS; in addition, it has delivered a useful developer tool for testing accessibility on all major platforms. ChromeVox is easily ported to other WebKit-based browsers; as an example, we have ported ChromeVox to Android to deliver web accessibility on Android.

References

[1] M. Blattner, D. Sumikawa, and R. Greenberg. Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*, 4(1):11–44, 1989.

[2] B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2 CSS2 Specification. *W3C Recommendation, http://www.w3.org/TR/REC-CSS2, May*, 1998.

[3] C. Chen. Clc-4-tts and fire vox: enabling the visually impaired to surf the internet. *Undergraduate Research Journal*, page 32, 2006.

[4] C. Chen and T. Raman. AxsJAX: a talking translation bot using google IM: bringing web-2.0 applications to life. *Proceedings of the 2008 international cross-disciplinary workshop on Web accessibility (W4A)*, pages 54–56, 2008.

[5] D. Flanagan and G. Novak. Java-Script: The Definitive Guide. *Computers in Physics*, 12:41, 1998.

[6] J. Garrett. Ajax: A New Approach to Web Applications. *Adaptive Path*, 18, 2005.

[7] Google, Inc. *Chrome Native Client*. http://code.google.com/chrome/nativeclient/.

[8] Google Inc. *ChromeVox Tutorial*. http://www.chromevox.com.

[9] Google Inc. *Overview: Google Chrome Extensions*. http://code.google.com/chrome/extensions/overview.html.

[10] I. Hickson. HTML 5 Working Draft. *W3C Working Draft, http://www.w3.org/TR/html5/, World Wide Web Consortium (W3C)*, 02 2008.

[11] H. Okada and T. Miura. Detection of layout-purpose table tags based on machine learning. *Universal Access in Human-Computer Interaction. Applications and Services*, pages 116–123, 2007.

[12] M. Pilgrim. *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks)*. O'Reilly Media, Inc., 2005.

[13] W. Protocols and F. Group. ARIA— Access To Rich Internet Applications Overview. Technical report, W3C, 2006. See http://www.w3.org/WAI/intro/aria.

[14] T. Raman. *Auditory User Interfaces: Toward the Speaking Computer.* Kluwer Academic Publishers Norwell, MA, USA, 1997.

[15] T. Raman. Toward 2 w, beyond web 2.0. *Communications of the ACM*, 52(2):52–59, 2009.

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

[17] P. Thiessen and C. Chen. Ajax live regions: Reefchat using the fire vox screen reader as a case example. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 136–137. ACM, 2007.