Presenting the SASWAT interfaces through WAI-ARIA

WEL— Accessibility Catch-Up Technical Report 1, November 2011

School of Computer Science

Information Management Group

Presenting the SASWAT interfaces through WAI-ARIA

Andy Brown
Web Ergonimics Lab
School of Computer Science
University of Manchester
UK

The SASWAT project used eye-tracking studies to identify effective ways of presenting dynamic Web content to screen reader users. These techniques were iteratively tested, then evaluated, using an implementation based on the FireVox screen reader extension to the Firefox Web browser. This project aims to extend and widen that work, allowing improved interaction and presentation techniques to be available sooner to users. Initial work, reported here, has re-engineered the SASWAT implementation so that its core functionality is available through code injected into a Web page. This implementation is described, and the plans for further work in the project discussed.

WEL

Web Ergonomics Lab

Accessibility Catch-Up

This Google-funded project is a follow on from the SASWAT project (`http://hcw.cs.manchester.ac.uk/research/saswat/`). The aim is to make the findings of that project immediately beneficial to screen reader users, and to expore ways of speeding up the dissemination of accessibility research findings.

Accessibility Catch-Up Reports

This report is in the series of WEL Accessibility Catch-Up technical reports. Other reports in this series may be found in our data repository, at `http://hcw-eprints.cs.man.ac.uk/view/subjects/saswat.html`. Reports from other Web Ergonomics Lab projects are also available at `http://wel-eprints.cs.manchester.ac.uk/`.

Acknowledgements

This report forms the first technical report for the "Accessibility Catch-Up" project (although [2] describes related work undertaken as part of this project), which is funded by a Google research award, and is a continuation of the SASWAT project.

License

Contents

Web Ergonimics Lab
School of Computer Science
University of Manchester
Kilburn Building
Oxford Road
Manchester
M13 9PL
UK

tel: +44 161 275 7821
http://wel.cs.manchester.ac.uk/

Corresponding author:
Andy Brown
tel: +44 (161) 275 7821
andrew.brown-3@manchester.ac.uk
http://www.cs.man.ac.uk/~browna

1 Introduction

The SASWAT project (see section 1.1, below) identified effective ways for dynamic content to be presented to people interacting with Web pages via a screen reader. The techniques used proved more effective and more popular than the approach used by a typical screen reader/browser. Unfortunately, users often benefit from research like this after a considerable lag (if, indeed, they ever benefit). There can be several reasons for this:

- Dissemination of knowledge: where research is undertaken outside the technology organisations, there is often a considerable time between completion of experiments and publication. There can then be further delay before those able to implement changes become aware of the research.
- Implementation priorities: even when developers become aware of improved techniques, time is constrained and implementing them is not a priority — bug fixes and planned development will both have higher priority.
- Poor understanding. Research can be poorly communicated, and developers may spend insufficient time understanding it. In this situation, the nature of the benefits or the scale of the problem they addressed might be underappreciated, so implementation gets a low priority.
- Development takes time. Research is testing ideas, but implementing these in a fully robust manner, suitable for a widely-used commercial (or free) product can be time-consuming.

The aim of this project is to make it quicker and easier for accessibility research findings to benefit users. We aim to develop a framework that enables improvements in user-interface design (particularly audio user-interfaces for screen reader users) to be passed on to users without such long delays. The project proposal enumerated the following research questions:

1. Can JavaScript be injected into a Web page to monitor and classify updates? Monitoring pages involves identifying and grouping changes to a page, discovering both what has been removed and what has been inserted, and which changes are related to one another. Classification in the SASWAT system relies on analysis of the content of the changes and monitoring user activity — can this be translated into a system that runs like AxsJAX?
2. Can WAI-ARIA markup injected into the page after an update modify how a screen reader (that is ARIA aware) behaves?
3. Can this be done in an efficient way? The prototype Fire Vox-based system does not have adequate performance on large pages, and is not sufficiently robust: investigation is required to use techniques that do not suffer these deficiencies.

A further aim was:

> In addition to the specific questions relating to dynamic updates, the theme of reducing the accessibility lag will run through the project, asking how the systems can be designed to be most widely usable, and for what other types of accessibility advances could it be used.

This report describes the first stage of this process — a re-implementation of the SASWAT system so that the techniques found in that project to be effective for handling and presenting updates can be used by any users with ARIA-aware browser/screen reader, not just using our experimental Firefox extension. Here, a system is described that enables code to be injected into any Web page so that dynamic updates can be analysed and classified (this report, therefore, does not fully answer the first research question; to do this an evaluation is required). This report also describes the first attempts to use WAI-ARIA markup to present updates to users.

1.1 The SASWAT system

The SASWAT system [1] was an adaptation of the FireVox screen reader extension[1] for the Firefox Web browser. It implemented rules for presenting dynamic updates, which had been derived from eye-tracking studies [6] that examined how sighted users allocated their attention when interacting with dynamic content. The effectiveness of this approach was validated by two studies [7, 5].

The way in which the SASWAT implementation operated can be sketched as follows. On loading a Web page, a map of the Document Object Model (DOM)[2] was created, and event listeners were added that reacted to DOMMutationEvents, which are fired whenever the DOM tree is modified. When events were detected, a new map of the DOM was created and compared to the old map. This comparison allowed regions that had changed to be detected and, crucially, grouped into meaningful groups (DOMMutationEvents may be fired for each individual change so that, for example, removal of a section of page might result in separate events for each paragraph and each line of whitespace removed). Once the changes had been resolved into distinct modifications, update objects were created and added to a queue. (Unfortunately, this process, while relatively effective at grouping, was relatively slow and demanding on the processor, particularly for larger pages.)

Concurrent to detecting and analysing changes to the page was a process of monitoring user activity. This involved:

- Tracking the focus of the user: since the screen reader and user-activity monitoring system were closely coupled, the monitoring system was able to use the information from the screen reader to exactly identify what the user was reading at any time.

- Tracking the actions of the user: all user input was via the keyboard, so event listeners were implemented that detected keystrokes and identified the

[1] http://www.firevox.clcworld.net/

[2] For an overview of the DOM, events and listeners, see `http://www.w3.org/DOM/` and `http://www.w3.org/TR/DOM-Level-2-Events/events.html`.

commands issued. Event listeners were also implemented to identify 'click' events generated, for example, when users follow a link.

Combining these two pieces of information: the focus of the user and their last input, it was possible to obtain a reasonably accurate idea of their last activity. This, in combination with their location can be used to infer whether or not an update was triggered by the user or happened automatically. For example a user might have navigated into an input box (which may trigger a pop-up calendar to appear), or typed into an input field (which may trigger an auto-suggest list), or clicked a JavaScript link (which may trigger updates). Alternatively, if the user has simply been reading the page content, an update can be classified as automatic.

Presentation of updates was the key contribution of SASWAT, for which all the update analysis was performed. Users were notified of *all* updates by brief non-speech sounds. Two sounds were used to differentiate between automatic and manual updates, and users had the option to disable notification for any dynamic region. For user-initiated updates, in addition to the non-speech notification, the user's focus was moved to the new content, and the first item read (the original location was bookmarked and could be returned to with a simple command). More sophisticated user-interaction was implemented for two common sub-classes of user-initiated updates: pop-up calendars and auto-suggest lists.

1.2 Disseminating SASWAT

While the main aim of the SASWAT project was to increase understanding of how to deal with dynamic content in a non-visual environment (by developing a model of how sighted users interact with such content and translating the benefits they receive into an audio interface), application of this understanding (e.g., by screen readers improving their default handling of updates, or by Web developers using ARIA to create interfaces that have been *demonstrated* to be effective) is likely to take some time. Sadly, this is the case with much research. This project is looking at how the benefits of improved understanding of user interfaces can be passed on more quickly. The aim is to develop a framework that will allow improvements in Web interface design more generally to be disseminated rapidly.

One approach to making novel interaction systems available is to inject them directly into the Web page. There are several routes for doing this: using a browser extension or bookmarklet, browsing via a proxy server, or using a system such as Greasemonkey[3]. The AxsJAX[4] framework [4] behaved in this way, providing a mechanism for injecting code to change the behaviour of Web pages, e.g., adding hot-keys to jump between sections, or defining speech output. The plan for this project is to develop a similar framework to AxsJAX, but with the following enhancements:

- Improve generalisability, so the behaviour of a widget can be defined and applied to all such widgets, rather than on a page-by-page basis.

[3]Greasemonkey is a Firefox browser extension that enables user scripts to be run on Web pages in the browser. `http://www.greasespot.net/`

[4]See `http://code.google.com/p/google-axsjax/`

- Simplify UI description. While easy to apply, AxsJAX required considerable coding skills to implement. This project aims to simplify the process, describing the interface at a higher level of abstraction.

The aim of the project is to develop this framework with a focus on the outcomes of the SASWAT project. Can the framework be used to describe and apply interfaces for update handling by injecting code into a Web page?

2 Injected Classification

In the SASWAT system, updates were handled according to their classification. This was done over two axes: how the update affected the page, and how it was initiated. Before implementing any description of how to handle these updates, it is necessary to inject code that can classify them. The classification process involves:

1. Detecting updates.
2. Filtering updates to determine which are relevant.
3. Grouping updates into discrete changes to the page.
4. Monitoring user activity.

The process is summarised in Figure 1, and shown with more detail in Figure 2.

2.1 Event Filtering

Changes to the Web page are detected with DOMMutationEvent listeners. These are added to the document and listen for events fired by changes to the DOM. Events are detected in the 'capture' phase[5]. However, many DOMMutation events are generated that we do not wish to present to the user. For example, insertion or deletion of comment nodes, or nodes containing only whitespace. We may also wish to inject content into the page (e.g., into an ARIA-enabled live region so that the user is notified; see section 3). For these cases it is necessary to filter each event to determine whether or not it needs handling.

The filtering process is as follows:

1. The first pass tests if the update contains user-visible content. Updates are ignored if the target node (i.e., the node generating the event) fulfils any of the following criteria:
   - Is a comment node `(nodeType == COMMENT_NODE)`.
   - Is a text node `(nodeType == TEXT_NODE)` that contains only whitespace.
   - Has a `<script>` or `<style>` tag.

[5]When a change to an element occurs, events fired by the change have two phases. In the 'capture' phase, event handlers on parent elements get the event first, while in the 'bubble' phase, event handlers on the changed element get the event before the parent elements. See `http://www.w3.org/TR/DOM-Level-2-Events/events.html`, section 1.2.

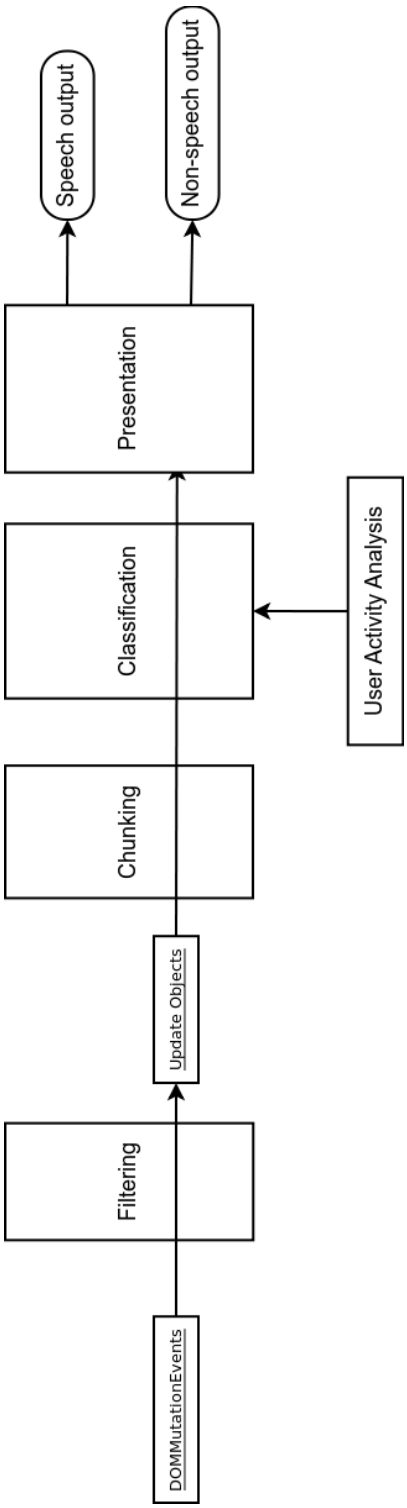

Figure 1: An overview of the processes performed by the injected code.

- Has no text content.
- Is within the `<head>` element.

This filter is applied as soon as the update event is detected; the events are immediately and permanently ignored. In addition to the criteria above, tests were also performed where nodes that were styled to be invisible were filtered out. However, it was found that, in many cases, a update caused a hidden element to be changed, then made visible; applying this test in the filter caused these updates to be lost.

2. The second pass compares position of the node in the DOM to a list of regions to be ignored. If the node forms all or part of any of these regions it is ignored. This list allows us to maintain a list of areas that the user does not want notification about, and to inject our own dynamic region into the page that can be updated independently from the update handling/presentation system. This filtering occurs after the update event has been added to the queue, before chunking takes place (for simplicity, the figures represent the two phases of filtering as a single process).

2.2 Buffering and chunking

Section 2.1 described how it is necessary to filter out the many events that are generated for updates which are of no interest to the user, including insertion of whitespace. It is also the case that multiple events are generated for what the user would perceive as a single update (section 1.1 described how it was necessary for the SASWAT implementation to group events into meaningful groups). As with SASWAT, updates must be grouped (or 'chunked', to borrow a phrase from Miller [8]) into units that would have been perceived by the user as coherent, discrete changes. For example, consider a page where we have almost simultaneous insertion of a section and updating of a ticker. In this case, the insertion of a section is a single change, but separate DOMMutationEvents are generated for each insertion of heading, paragraphs and whitespace, while one (or more) events are generated by the replacement of ticker text elsewhere on the page. The filtering process has removed events relating to whitespace, but to classify and present these updates effectively, it is still necessary to group the remainder into two.

In this implementation, events that pass the filtering process are used to generate `Update` objects, which are then placed in a buffer (or queue). Adding an `Update` to the buffer triggers a 500ms pause before processing the updates. This allows events from any remaining updates to be captured, filtered and added to the buffer before chunking.

Chunking is done by analysing the `target` elements of the events. Updates from neighbouring elements are grouped and the smallest element of the page containing all components of an update group is identified. This is followed by a pairing process, which matches removals and insertions from the same region and creates an new replacement update. Finally, another filtering process removes unnecessary updates (e.g., duplicates), completing classification on the first axis of the taxonomy: update type.

2.3 User-action monitoring

A key part of the system used by SASWAT-FireVox was the monitoring of user activity. Using a screen-reader integrated into the browser, and processing updates within this enabled effective tracking of the user's actions. This information allowed the system to determine relatively accurately whether the update was likely to have been initiated by some action by the user or automatically. This distinction is critical, as the model of sighted user behaviour showed that this was the key predictor for whether or not an update would be fixated [6]. Replicating this through code injected directly into the page introduces some limitations, but the system outlined below uses heuristics that appear to give a reasonably reliable indication of whether the most recent user action is one that could have triggered an update.

As with the SASWAT implementation, our knowledge of the users actions is limited to detecting key commands. Unlike the SASWAT implementation, however, we do not have access to the commands issued to the screen reader, and we do not have as detailed information about the focus of the user. These limitations make it more difficult to determine the user's activity.

Event listeners are added to the document for two types of event: `click` and `keydown`. Both listen for events in the 'capture' phase, so they are detected before any page scripts might destroy them.

2.3.1 Detecting Clicks

The `click` events are fired whenever a 'click' action occurs on a page, so will detect whenever the user follows a link, whether it is activated by a mouse click, or the keyboard. Detecting these events is critical, as it is the primary means of triggering user-initiated updates.

Once a `click` event has been detected, the event is analysed to determine the `target` of the click. The target allows us to differentiate between links to other pages, links to anchors within the current page, and links that trigger JavaScript. Other clicks, such as those where the target is an `input` element, suggest the user has used the mouse to move focus within the page or may be moving the cursor within an input field; the former may trigger an update (such as a pop-up calendar), the latter is unlikely to. Once analysed, the user's state is set to one of the following:

External link click: The user has clicked a link leading to a new page.

Internal link click: The user has clicked a link that navigates to an anchor on the current page.

JavaScript link: The user has clicked on a link that activates a JavaScript function.

2.3.2 Detecting keystrokes

Keystroke analysis is more complex, and relies on detecting not only which key was pressed, but also where the focus was when it was pressed. The current implementation is based on some simple heuristics. First, it is determined whether the user

was in a `<form>` or (more specifically) `<input>` element, then the key is identified. This information is combined to give a prediction of user activity. The activity is currently assigned to one of the following:

Input typing: The user is typing into an input field.

Navigation: The user has typed a command that moves focus around the page.

Escape: The user has pressed the escape button (this can be used to dismiss certain widgets, such as calendars and auto-suggest lists).

Widget Use: The user is interacting with a widget. This is considered to be the case when arrow keys have been pressed inside a form.

Form submit: The user has pressed the enter key while in a form.

Pressed hotkey The user has pressed a 'normal' (alphanumeric character) key while not in a form.

Finally, if there has been no key activity for a period of time, the user is considered to be 'idle'. This is currently set at 5 seconds, although this value has been set by trial-and-error, and may need to be modified on the basis of user testing. This gives a final type of activity:

Idle: The user has not pressed a key or generated a click event in the last 5 seconds.

2.4 Classification

Alone, the information about the user's last action and the effect of the update (insertion, replacement or removal) is not quite sufficient to classify the update. To get a more accurate estimate of the cause of the update, it is necessary not only to look at what the last action of the user was (and where the focus was when they did that action), it is also necessary to look at where the focus is at the time of the update. For example, the tab command suggests the user's last action was tabbing around the page, or navigating, which may, or may not, trigger an update: navigating to a header element is unlikely to cause the page to change; navigating to an input area might (e.g., to reveal information to help the user complete the form). It is therefore necessary to take the user's last action in combination with information about the focus at the time of update, and use this combined information to classify the update.

3 Injected Presentation

Update classification is simply the means by which we determine the most effective way to present the update to the user. Implementation of a presentation system has begun, but is at an early stage. Testing (using Firefox and NVDA[6]) showed

[6]NonVisual Desktop Access, an open-source screen reader for Windows. See `http://www.nvda-project.org/`

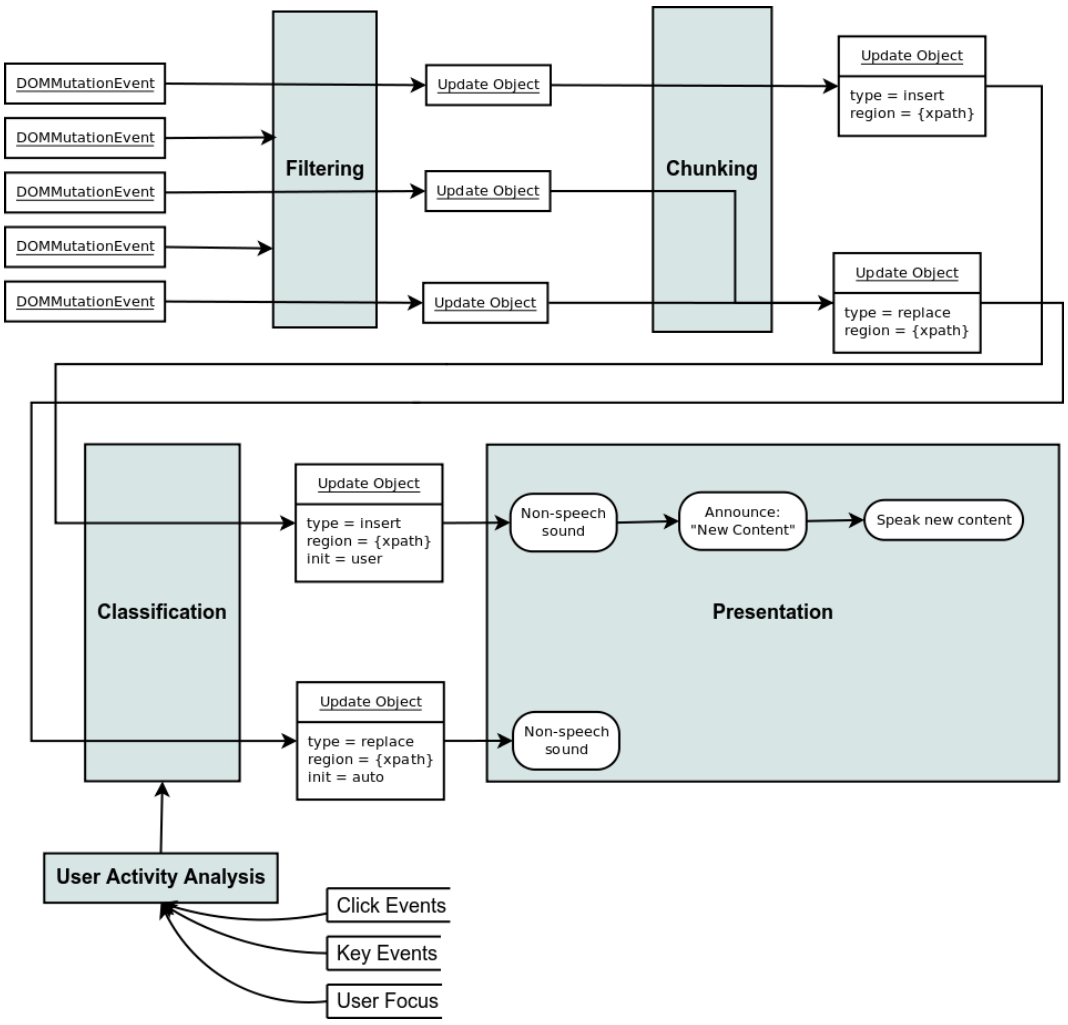

Figure 2: A more detailed schematic of the current implementation.

that ARIA injected after the update had, unsurprisingly, no effect: regions set as `aria:live='assertive'`, for example, were not announced. Nevertheless, it remains important that the information is relayed quickly and effectively to the user.

3.1 Injected ARIA live region

One solution to this is to inject a region into the page when it is loaded, mark this up as an ARIA live region, and use it to make announcements. An implementation of this concept injects the following into the page, appending it to the `<body>` element of the DOM:

```
<div id='saswatAriaNotification' aria-live='assertive'>
</div>
```

JavaScript functions then allow injection of messages and clearing the `<div>`. Limited tests with Firefox and NVDA suggest that this is an effective way of requesting speech output from the screen reader, and can be used to make announcements. Further experimentation is required to determine how this affects the user's focus (and if focus can be manipulated directly) and if it can be used with more interactive updates, such as calendars and auto-suggest lists.

3.2 Injected sound generation

The SASWAT project not only identified the importance of speaking user-initiated new content immediately, but also the effectiveness of announcing automatic updates in an unobtrusive manner as possible. In that implementation, brief non-speech sounds were played when updates occurred, with two sounds differentiating between automatic and manual updates. As with announcements, one solution for implementing this within the page/browser is to inject sound into the page, and play the sounds on demand.

For example, the HTML5 code below can be injected into the page, given the URL for a sound file.

```
<div id='saswatAutoSoundWrapper'>
  <audio src='http://path_to_sound/file.wav'
       autobuffer
       id='saswatSoundAuto'
       aria-hidden='true'>
  </audio>
</div>
```

The sound can be played with the following JavaScript:

```
var soundRegion = document.getElementById('saswatSoundAuto');
soundRegion.play();
```

Injecting two such regions, one for automatic updates and one for manual updates allows the non-speech notification of the SASWAT browser to be replicated in a normal browser/screen reader combination by injecting some simple code into the page.

4 Future Work

This implementation forms the basis for answering the first research question of this project (can updates be classified using injected JavaScript?). In this section, the plans for completing the project objectives (Section 1) are described. In summary, technical evaluations will be performed on the implementation, answering research questions 1 and 3; the use of the ARIA notification area will be completed to answer question 2. The remaining effort will be directed to the issues of generalisability, exploring how the classification system can be built into a framework that will allow rapid prototyping, testing and delivery of new techniques for handling dynamic Web content. The different strands are described below, and summarised in Figure 3, which shows how the future work extends the current implementation.

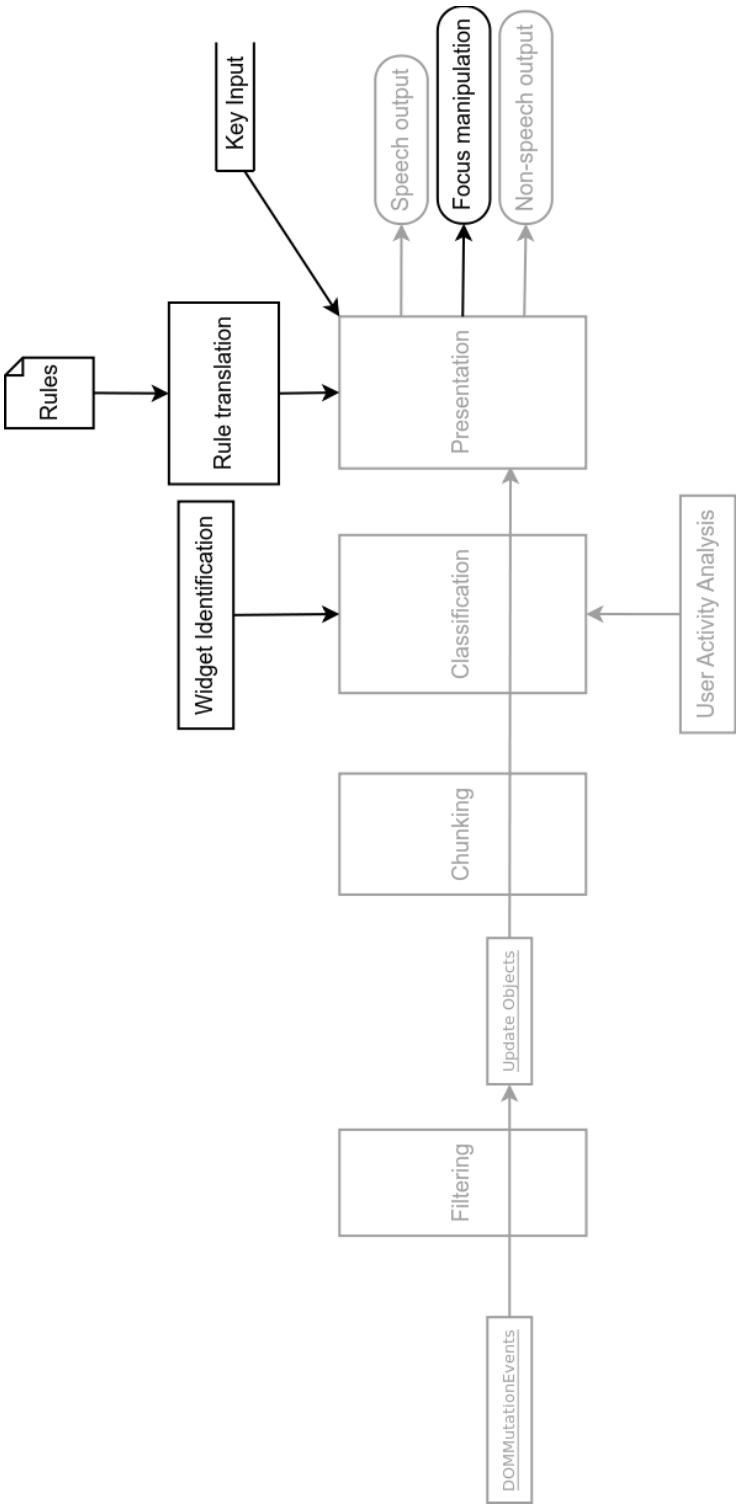

Figure 3: An overview of future development. The current implementation is shown in grey, while black items show planned developments.

4.1 Testing

The implementation described above has only been tested on the pages used for evaluation of the SASWAT system. Further testing on a range of pages is necessary to identify any problems. A technical evaluation is planned in which the code will be injected into a range of pages and the accuracy of update classification determined.

4.2 Providing user control

It will be necessary to provide users with more control over how updates are handled. For example, users should be able to:

- Choose whether or not to inject and play non-speech sounds.
- Change the sounds.
- Silence non-speech notification for an area.
- Turn off all notification or updates for a page or region of a page.

4.3 Injection of more complex UI components

The current implementation covers the basics: injecting code so that updates can be presented in a similar way to the SASWAT browser. For this to be extended to a framework for rapid roll-out of future research outcomes (or, indeed, for implementing novel UIs for testing), it will be necessary to allow more sophisticated user interaction to be injected. In particular we need:

- Key commands: allowing users to interact with widgets via keys. For example, using the arrow keys to control a slide-show.
- More interactive interfaces. For example, parsing and reading the first items in an auto-suggest list, and allowing users to select them.

In Figure 3, this is represented by 'key input' into the presentation process, and by the 'Focus manipulation' output.

4.4 User-defined UIs: simple descriptions of user interfaces

If we want people to apply novel user interfaces to widgets, it is necessary for them to be able to specify them in a simple, yet unambiguous, manner. We plan to investigate the use of SCXML (State Chart XML). These are represented by the 'Rules' in Figure 3, although in reality, the rules are the output of the UI specification process. Figure 3 also shows another potential requirement for the system: 'Widget Identification'. This represents the possible incorporation of a system that analyses code to detect Web 2.0 widgets[3] into the project. In this case, classification can potentially be enhanced by being able to classify updates in more detail. For example, a slide show might be identified on a page; this knowledge can allow us to inject more tailored user-interface components (e.g., key commands to move back and forward through the slides).

4.5 Translation and injection of user-defined interfaces

Finally, the components of the project can be brought together in a system that translates user interface design, as specified using the component in Section 4.4 so that the necessary interface can be injected into Web pages using the components from Section 4.3. Such a system should allow researchers to both test novel ideas and then to pass on the benefits of the successful experiments using a relatively quick and simple workflow. In Figure 3, this is represented by the 'Rule Translation' process, which feeds into presentation.

5 Summary

This technical report has described how this project aims to extend the SASWAT research. Much of the implementation from that project has been simplified and translated into code that can be injected into a Web page, thereby working in any browser. This has also been done in a way that will allow further extension, so that novel user interaction techniques can be specified and implemented quickly, and their benefits passed on to users more rapidly.

References

[1] Andrew J. Brown and Caroline Jay. Hcw-fire vox user manual. `http://hcw-eprints.cs.man.ac.uk/125/`, October 2009.

[2] Andy Brown and Simon Harper. AJAX time machine. In *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, W4A '11, pages 28:1–28:4, New York, NY, USA, 2011. ACM.

[3] Alex Q. Chen and Simon Harper. Identifying Web widgets. Technical Report, University of Manchester, `http://hcw-eprints.cs.man.ac.uk/141/`, May 2009.

[4] Charles L. Chen and T. V. Raman. 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, pages 54–56, New York, NY, USA, 2008. ACM.

[5] Jenny Craven. Evaluation of the saswat web browser. 2010.

[6] C. Jay and A.J. Brown. User review document: Results of initial sighted and visually disabled user investigations. Technical Report, University of Manchester, `http://hcw-eprints.cs.man.ac.uk/49/`, 2008.

[7] Caroline Jay, Andrew J. Brown, and Simon Harper. Internal evaluation of the saswat audio browser: method, results and experimental materials. Technical Report, University of Manchester, `http://hcw-eprints.cs.man.ac.uk/125/`, 2010.

[8] G.A. Miller. The magical number seven plus or minus two. *Psychological Review*, 63:81–97, 1956.

6 Associated Files

The JavaScript filed described in this report are published on the Web Ergonomics Lab Repository. The url is `http://wel-eprints.cs.man.ac.uk/XX/`. There is a single zip file (code.zip) containing:

- readme.txt: A description of the contents of the zip file.
- user_interaction.js: The code responsible for detecting user activity.
- notification.js: The code repsonsible for injecting announcements into a page.
- rules.js: Some basic code for describing rules that can be applied according to update class.
- update_handler.js: The main code — responsible for detecting, classifying, and applying presentation rules to updates.
- auto.wav: A non-speech sound that may be used for notifying users of updates.
- manual.wav: A non-speech sound that may be used for notifying users of updates.