WebViews: Accessing Personalized Web Content and Services

Juliana Freire
juliana@bell-labs.com

Bharat Kumar
bharat@bell-labs.com

Daniel Lieuwen
lieuwen@bell-labs.com

ABSTRACT

The ability to take information, entertainment and e-commerce on the go has great promise. However, the existing Web infrastructure and content were designed for desktop computers and are not well-suited for other types of accesses, *e.g.,* devices that have less processing power and memory, small screens, and limited input facilities, or through wireless data networks with low bandwidth and high latency. Thus, there is a growing need for techniques that provide alternative means to access Web content and services, be it the ability to browse the Web through a wireless PDA or smart phone, or hands-free access through voice interfaces.

In this paper, we discuss issues involved in making existing Web content and services available for diverse environments, and describe WebViews, a system that allows casual Web users to easily create customized views of Web sites that are well-suited for different types of terminals. In particular, we describe our approach to provide voice access to these Web views and experiences in building the system.

Keywords

content transcoding, dynamic content, electronic commerce, information delivery, personalization, smart bookmarks, voice interfaces, Web clipping, wrappers

1. INTRODUCTION

The explosion in the use and availability of wireless devices and the ability they give people to access information anytime and anywhere has great promise. In the U.S. alone, Dataquest expects that the number of wireless data subscribers will explode from 3 million in 1999 to 36 million in 2003. Thus, very soon, millions of people will be able to access the Web, and order services and goods from wireless Internet devices. However, the existing Web infrastructure and content were designed for desktop computers and are not well-suited for devices that have less processing power and memory, small screens, limited input capabilities, and that are connected through networks that have high latencies and low bandwidth.

Consider for example accessing the Web from a PDA such as the Palm Pilot, using a wireless data service such as Omnisky [20]. Omnisky runs over CDPD[1] and has effective throughput rates that vary from 5-6 kbps up to 12-13 kbps. Combining that with a screen size of 160x160 pixels on a 6x6cm surface, it can be very hard to browse through large pages with rich graphics. Given that these devices have significantly less memory and processing power than desktops, the available browsers only have a small subset of the features of the widely used browsers (*e.g.,* they do not support Java, Javascript, and are not able to display GIF or JPEG images). In addition, input facilities are limited—even with Palm's Grafitti text input system, entering text can be very time consuming. Similar difficulties arise while trying to access the Web using WAP phones (Internet-ready mobile phones). Newer additions to the PDA family, such as the Compaq iPAQ, have more memory, powerful processors, and could eventually have more complete browser support; however, the screen size, limited input capabilities, and high latency for page accesses still apply.

Voice interfaces have received much attention recently as an effective means of user interaction which both simplifies the input process, and provides more convenient (hands-free) access. The advances in voice recognition and text-to-speech (TTS), combined with the steady increase in computing power has made these technologies viable for end-users. Standards such as Voice eXtensible Markup Language (VoiceXML) have been proposed for making Web content and information accessible via voice and phone. But even though there are some VoiceXML-based services available, most content on the Web consists of HTML pages and cannot be easily accessed through voice interfaces.

The reality is that the Web is not really accessible anytime or anywhere. Different attempts have been made to address these shortcomings:

- *Re-engineering existing Web sites and services:* content providers create different versions of their Web sites that provide content formatted for specific devices. For example: The New York Times has a *palm-friendly* section [18]; Amazon provides a specialized interface for Web-enabled phones, as well as for the Palm VII [3]; and various other Web sites now have mobile phone-friendly versions (see [26] for a list such sites).
- *Creating specialized wrappers that export a different view of a Web page or service:* services such as `everypath.com`

[1]CDPD [7] is a wireless IP network that overlays on the existing AMPS (analog) cellular infrastructure.

and oraclemobile.com provide tools and services to create wrappers which can export wireless-friendly clippings of a set of Web pages and services, such as stock quotes, traffic, and weather information. Similar voice-enabled services are provided by tellme.com and heyanita.com.

- *Using proxies that filter and reformat Web content:* proxies can be programmed to transform content according to client's display size and capabilities. For example, ProxiWeb [23] transforms HTML pages and embedded figures into a format that can be displayed on a Palm Pilot. PhoneBrowser [22] similarly transforms HTML pages into a form that can be read using TTS over the phone.

All these approaches have drawbacks. From a content provider's perspective, creating and maintaining multiple versions of a Web site to support different devices is labor intensive and can be very expensive. Even though wrappers require no modifications to the underlying Web sites, they can be costly to create and need updating whenever the corresponding Web site or service changes. From a user's point of view, both solutions are restrictive, as neither do all Web sites support all kinds of devices, nor do existing wrapper-based solutions offer clippings for all content and services a user may need.

Proxy transcoders, on the other hand, perform on-the-fly content translation and, in theory, they are a good general solution for allowing users to browse any Web site. But since Web pages must be presented as faithfully as possible, these general purpose proxies do not perform any personalization. This is clearly not the ideal solution for somebody accessing the Web through a cellular phone with a 3-line display, or through a voice interface. Besides, some features are hard or even impossible to translate. It is not unusual that proxies fail to properly transcode *complex* pages, or even simple, but badly designed pages (Section 4.1 discusses this issue).

Example 1.1. *[A Web view: Pricing airfare]* Consider the following scenario. Juliana plans to attend the WWW10 conference and she is looking for flights from Newark Airport to Hong Kong that leave from Newark on April 29th and return from Hong Kong on May 6th. She goes to www.travelocity.com and after navigating through 6 pages, and having 650 Kb of data transferred (300 Kb with images turned off), a page with the list of nine flights (as well as ads and additional navigational information) is displayed.

From a desktop browser, repeating this series of steps can be rather tedious if one performs this type of query often in an attempt to find a cheap flight. The problem is compounded if one tries to access the flight list from a wireless device such as the Palm Pilot with a wireless modem. In fact, we were not able to access the flight lists from Travelocity using either ProxiWeb [23] browser version 3.5, or AvantGo [6] version 3.3—neither was able to transcode the first page properly.

The ideal in either scenario would be to create a shortcut to the flight list. Since the final page also contains additional information, it would also be useful to extract from that page *just* the list of flights. Note that in the Palm Pilot scenario, the shortcut could be executed at a server (with a better connection to the Internet), and only the final results (the clipped flight list) delivered to the device. Ignoring latencies, downloading the 650kb required to access the flight list from Travelocity takes anywhere between 60 and 180 secs over CDPD, whereas from desktop computer connected to the Internet through a cable modem the transfer would take less than 5 seconds.

In general, it would be useful if one could *easily* create not only simple shortcuts, but different *views* of Web sites that are better suited to be accessed from different terminals. For wireless devices and voice interfaces, it would be useful to reduce the number of required interactions, and the amount of data input and transferred. For example, one could create a clipping template for searching for flights from Travelocity that would automatically login and navigate to the page where Juliana enters the details of her itinerary (with Newark automatically filled in as the departure airport by the system). □

Our Contributions: We propose the WebViews architecture as a solution for creating customized views of Web content and services. The main idea is to let end-users easily create and maintain simplified views of Web content and services, from CNN health headlines to bank balances. By allowing users to create their own Web views, a service can be offered that is *personalized* and not restricted to a set of supported Web sites, in contrast with services such as tellme.com that only allows access to a pre-specified menu of popular choices. Users can easily customize such Web views for specific devices. In addition, since these Web views provide a simpler view of sites and services, they are considerably simpler to transcode into other languages (or formats).

A number of requirements exist for a system that creates Web views. First, since it is targeted to end-users, it must be easy to use and require no programming expertise. Another important requirement is that Web views should be robust. Since Web sites may change often, Web views should degrade gracefully with these changes, *i.e.,* if changes are minor, the Web views should still return the desired content. In the event of radical changes, they should be easy to update.

In this paper we describe how WebViews fulfills these requirements, and discuss how it can be used for information delivery to diverse devices. We also describe the implementation of (and our experiences with) VoiceViews, a system that provides voice-enabled access to WebViews. Our goal is to cover the main issues involved in building such a system, but to keep within space limitation, we are forced to make some restrictions in scope. For example, we do not discuss issues such as security and scalability.

The structure of the paper is as follows. Section 2 describes the architecture of WebViews, and how the system fits into an information delivery platform. Section 3 provides a brief overview of voice-enabled interfaces and presents the techniques we used for transcoding Web views into VoiceXML, including how we produce robust clippings and how we generate markup to allow navigation within the voice dialogues. We discuss our experiences in building the system in Section 4 and related work in Section 5. We conclude in Section 6, where we also outline directions for future work.

2. SYSTEM OVERVIEW

There are two main steps involved in creating Web views: retrieving a Web page that contains the desired information, and extracting relevant content from the retrieved page. Given the growing trend of creating interactive Web sites that publish data on demand, retrieving information from the Web is becoming increasingly complicated. Many sites, from online classified ads to banks, require users to fill a sequence of forms and/or follow a sequence of links to access a page they need, and often, these hard-to-reach pages cannot be bookmarked using the bookmark facilities implemented in popular browsers. To create Web views of these pages, the process to access them must be automated. Also, as described in Example 1.1, once the desired page is retrieved, a user may want to specify individual elements of the page she is interested in, so that irrelevant information is filtered out. A Web view thus must encapsulate the actions required to retrieve a particular page, along with the specification of which elements should be extracted from the retrieved page.

It is possible to automate the retrieval of pages by writing wrappers in Java or in specialized languages such as WebL [14]. One can also write Perl scripts to extract individual fragments of Web pages. However, this approach is not feasible for casual Web users that are not programmers. In addition, given the dynamic nature of the Web, maintaining these programs and scripts can be very costly, as they might require modifications every time Web sites change. The WebViews architecture addresses these problems by providing a VCR-style interface similar to the WebVCR [4] to transparently record browsing steps; and a point-and-click interface to let users select page fragments. From a desktop, using a browser and the WebViews Recorder applet, a user can create a Web view by simply browsing to the desired page and selecting on that page the components of interest—no programming is required. Furthermore, the system uses techniques that enhance the robustness of Web views, so that they work even if certain changes occur in the underlying Web sites.

After a Web view is created, it can be accessed through a WebViews server, which is a Web-hosted service located at an ISP, ASP, or a company intranet. As Figure 1 shows, the WebViews server accepts requests from HTTP clients and returns XHTML responses. A request to the WebViews server contains an identifier for a particular Web view (and additional parameters—see Section 2.5) which when executed, accesses a particular Web page, clips it, and returns the resulting content to the requesting client. Note that requesting clients can be proxy transcoders that translate the clipped content into various formats (*e.g.,* HDML, WML, VoiceXML, etc.).

Example 2.1. *[Usage Scenario: pricing airfare]* If Juliana wishes to create a Web view for the Travelocity example presented in Section 1, she starts a WebViews Recorder applet at her desktop. She then goes to the main Travelocity page (`www.travelocity.com`), hits the *Record* button on the applet, and browses to the itinerary page. As soon as the *Record* button is clicked, the applet transparently records all her navigation actions. When the desired page is reached, she hits the *Stop* button, and specifies the content to be extracted from the final page (*e.g.,* only the itinerary details). At this point, the Recorder applet has all the information required for the Web view, which can be saved. After it is uploaded to the WebViews server, the Web view is then accessible to any HTTP client. When Juliana wants to access this view from her Palm Pilot, she accesses the WebViews server, which after authenticating her request, automatically navigates to the itinerary page, extracts the specified content from the page, and returns the extracted XHTML content (which ProxyWeb transcodes before it reaches her Palm Pilot). □

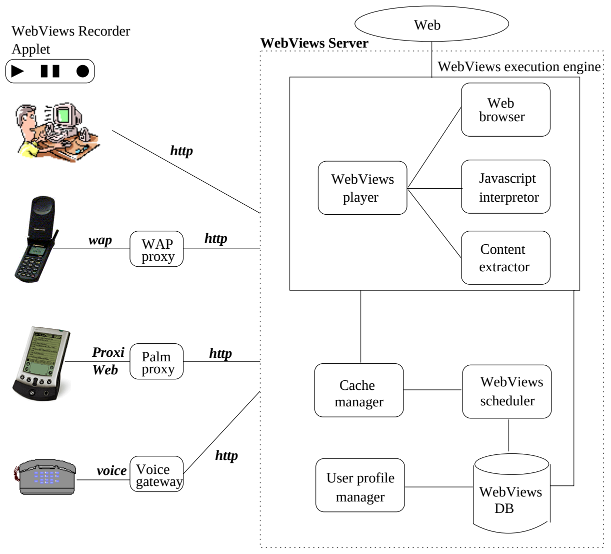


Figure 1: WebViews server architecture

The architecture of the WebViews server is shown in Figure 1. The WebViews server consists of the following modules: 1) the WebViews DB, which stores Web view specifications; 2) the user profile manager, that performs user authentication for sensitive Web views stored on the server (*e.g.,* a Web view that retrieves a user's 401(k) balance), as well as manages other aspects of the user account; 3) the WebViews scheduler, that periodically executes Web views (if so specified by the Web view requestor); 4) the cache manager, that stores cached Web views; and 5) the WebViews execution engine, that interacts with the WebViews player, which together with the Web browser and the Javascript interpretor, retrieves Web documents and parses HTML pages; and the content extractor (which clips *interesting* components of retrieved pages). We now give a more detailed description of Web view creation and execution.

2.1 Creating Smart Bookmarks

To create a Web view, a user must first specify the page to be clipped. If the page requires multiple steps to be retrieved and does not have a well-defined URL, the user can use the recording component of the applet to create the script to access the page. It has a VCR-style interface to transparently record browsing actions—users can simply navigate their way to the final page while their actions (links traversed, forms filled along with the user inputs, and any other interactions with active content) are transparently recorded and saved in a *smart bookmark* (SMB).

During recording, if the user is required to fill out forms, she can optionally specify which field values are to be stored in the Web view specification itself, and which ones are to be requested from the user every time the Web view is executed. This allows the user to create parameterized Web views. For example, a Web view to retrieve a restaurant list from the Yellow Pages at `mapsonus.com` can have a zip code parameter, so the user does not need to create a separate Web view for each city. Also, for security reasons, a user may choose not to save certain kinds of information such passwords inside a Web view (parameterization issues are discussed in Section 2.5), or to save it encrypted. An excerpt of the SMB to retrieve the itinerary page from `travelocity.com` (discussed in Example 1.1) is shown in Figure 2.

```
<BOOKMARK id="juliana_travel">
  <URL> http://travelocity.com </URL>
  <LINK>
    <loc> document.links[8] </loc>
    <href> <![CDATA[http://dps1.travelocity.com/
lognlogin.ctl?tr_module=AIRG&SEQ=1]]> </href>
    <text> null </text> <target> null
</target>
  </LINK>
  <FORM>
    <!-- Login form --> ...
  </FORM>
  <LINK>
    <!-- 9 Best Itineraries link --> ...
  </LINK>
  <FORM>
    <loc> document.forms[0] </loc>
    <action>
      <![CDATA[https://dps1.travelocity.com:443/
lognmain.ctl?SEQ=1]]>
    </action>
    <method> POST </method> <name> null
</name>
    <target> null </target>
    <ATTRS>
      <ATTR> <name> trip_option </name>
             <loc> 5 </loc>
             <type> radio </type>
             <prop> stored </prop>
             <val> roundtrp </val> </ATTR>
      <ATTR> <name> depart_airport </name>
             <loc> 10 </loc>
             <type> text </type>
             <prop> stored </prop>
             <val> EWR </val> </ATTR>
      <ATTR> <name> depart_month </name>
             <loc> 11 </loc>
             <type> select-one </type>
             <prop> stored </prop>
             <selected_index> 3 </selected_index>
             <text> Apr </text> </ATTR>
      <ATTR> <name> depart_day </name>
             <loc> 12 </loc>
             <type> select-one </type>
             <prop> stored </prop>
             <selected_index> 28 </selected_index>
             <text> 29 </text> </ATTR>
      ...
    </ATTRS>
  </FORM>
</BOOKMARK>
```

Figure 2: Smart Bookmark to retrieve itineraries from Travelocity

2.2 Creating Web Views

Once the desired page is retrieved, the clipping component of the WebViews applet can be used to specify the fragments of the page that should be extracted. An interesting problem is how to identify these fragments. In general, any extraction specification needs to provide the ability to 1) address individual or groups of arbitrary elements in a page, and 2) specify rules (that use the above addressing scheme) to extract the relevant content from the page. We wanted a solution that was standard, powerful, portable and efficient, and most importantly, which could be used to easily create robust extraction expressions (*i.e.,* that would not break under minor changes to page structure).

We chose XPath [32] as the mechanism for specifying extraction expressions. XPath views an XML document as a tree and provides a flexible mechanism for addressing any node in this tree. One drawback of using XPath is its requirement that pages be well-formed. Since browsers are very forgiving in this respect, many Web sites generate pages that are ill-formed (*e.g.,* have overlapping tags, missing end tags, etc.). Consequently, the WebViews system must first *clean up* HTML pages (*e.g.,* using tools such as HTML Tidy [27]) before XPath can be applied. Another alternative we considered for specifying extraction expressions was the XML DOM API. However, XPath allows a more *flexible* and *easier* way to create robust clipping expressions that are immune to minor changes in page structure. DOM addresses (without storing extra information, or using other heuristics to compensate for page changes) can be very *brittle* even to minor layout changes.

Continuing with Example 2.1, suppose Juliana is only interested in the first three itineraries from Travelocity (where each itinerary is represented by two HTML tables—one with pricing information, the other with route information). After recording the navigation steps, she specifies an XPath expression that will extract only the desired content from the final page. For example, she could use either of the following expressions (note that (1) is very similar to using DOM addresses):

```
(1) //html/body/center[2]/div/table[2]/tr/td/
        table[position()>=3 and position()<=8]

(2) //table/tr/td[(contains(string(),'Price:')
      or contains(string(),'Option')) and
      not(descendant::table)]/parent::tr/
      parent::table[position() >= 1 and
                    position() <= 6]
```

These expressions can be rather complicated, and writing them can be an involved task. In addition, as seen above, there are multiple ways to specify a particular page element, and some may be preferable to other in terms of robustness (as explained in Section 2.4). Since our system is directed towards naive users, we cannot expect them to know about, far less be able to specify, XPath expressions. To address these problems, we are currently designing a point-and-click interface that lets users select portions of Web pages (as they see them in the Web browser), and it automatically generates extraction expressions. The point-and-click interface will provide users with different levels of abstraction corresponding to a breadth-first search in the portion of the document tree that is visible in the browser. For example, if a user is interested in particular cells of a table, she must first select the table and then zoom into the table to select the desired cells. Section 2.6 gives more details on how the GUI produces XPaths and other clipping information.

Figure 3 illustrates a Web view specification for Example 2.1, simplified for exposition purposes. The first part of a Web view specification points to the SMB that retrieves the desired page (see Figure 2). The `<EXTRACT>` elements contain the extraction specifications. Note that multiple fragments can be specified, and users may choose to specify these fragments according to the terminal where they will be displayed. For example, if the Web view is to be displayed in a Palm Pilot, the user could choose to extract the first 3 itineraries (the extraction tag with `fragment_name = "first_3_itineraries"`), whereas if the Web view is to be displayed in a Web-enabled cellular phone with a 3-line display, a single itinerary may be preferable (*e.g.,* the extraction tag with `fragment_name = "first_itinerary"`).

2.3 Executing Web Views

After a Web view is specified, it can be saved and uploaded to a WebViews server. Users may then access Web views

```
<WEB-VIEW id="juliana_clippings">
  <BOOKMARK idref="juliana_travel" />
  <REFRESH-INTERVAL> 24 Hours </REFRESH-INTERVAL>
  <EXTRACT fragment_name = "first_3_itineraries">
    <![CDATA[
(//table/tr/td[(contains(string(),'Price:')
or contains(string(), 'Option')) and
not(descendant::table)]/parent::tr/
parent::table)
[position() >= 1 and position() <= 6]
    ]]>
  </EXTRACT>
  <EXTRACT fragment_name = "first_itinerary">
    <![CDATA[
(//table/tr/td[(contains(string(),'Price:')
or contains(string(), 'Option')) and
not(descendant::table)]/parent::tr/
parent::table)
[position() >= 1 and position() <= 2]
    ]]>
  </EXTRACT>
</WEB-VIEW>
```

Figure 3: Web View for airfares from Travelocity

via URLs that uniquely identify them. Users may further specify additional parameters such as input values for a Web view (*e.g.,* the password to access a bank account); the mode of operation (pull or push); whether the Web view should be cached and how often it should be refreshed. Given the Web's unpredictable behavior (network delays, unreachable sites, etc.), caching plays an important role in a WebViews server. Users can specify for each Web view, if and how often it should be executed and cached (*e.g.,* execute the Web view in Figure 3 every 24 hours, and cache the itineraries).

In addition, users may also specify how they want the Web view to be delivered. In *pull mode*, the URL invokes a CGI script at the server, which in turn executes the Web view specification and immediately returns the clipped content to the requesting client. In *push mode*, the execution and delivery of the Web view are asynchronous, *i.e.,* the Web view can be returned to the client later, possibly through protocols other than HTTP (*e.g.,* Web views could be emailed). Push mode is preferable when back-end Web sites are slow or temporarily unreachable, or when the end user cannot or does not want to keep a session open for too long. (Some wireless data services, such as Sprint PCS, charge for usage time.)

The execution of a Web view is as follows. The SMB in the Web view specification is replayed, and after the final page has been retrieved (and tidied), the extraction expressions are evaluated to extract the desired content by an XSLT [33] processor such as XT [34] or Xalan [31]. The extracted content is then returned to the client.

To allow access to diverse devices, we assume the presence of appropriate gateways that perform protocol conversion to and from HTTP, as well as the necessary transcoding of content retrieved from the WebViews server, *e.g.,* a WAP proxy to allow access to WAP-enabled devices, a Voice gateway to enable voice access to Web content, and even specialized gateways such as a Palm proxy. In Section 3.3, we discuss how annotations to the extraction specification can lead to better transcoding for voice access.

Note that all processing (retrieval and extraction) is done at the WebViews server. Only select portions of Web pages are returned to the requesting client, effectively giving users one-click access to desired content, and considerably reducing communication between the client and the WebViews server. This feature is especially useful in wireless environments where users must access the Web through high-latency, low-bandwidth connections and where some navigation steps (*e.g.,* those involving Javascript) are impossible. Further, if the desired content is to be sent to a handheld device or via a voice interface, the task of transcoding the content becomes much easier—only a portion of the final page needs to be transcoded, and none of the intermediate pages.

2.4 Robustness Issues

Since Web pages may change between the time of creation and execution of a Web view, the system uses techniques to ensure that replaying a sequence of recorded actions will lead to the intended page, and that the correct fragments are extracted—even when the underlying pages are modified. Usually, changes to Web pages do not pose problems to a user browsing the Web, but they do present a challenge to a system that performs automatic navigation. In a sequence of recorded browsing actions, some links may contain embedded session ids, and forms may contain hidden elements that change from one interaction to the next. Thus, for each user action during replay, the WebViews system must locate the correct object (link, form or button) to be operated on, and this can be challenging in the presence of changes to Web pages (*e.g.,* addition/removal of banner ads). Moreover, any algorithm used to determine the new position of the object on the changed page must be reasonably fast, since it needs to be executed for every recorded user action. Hence, we cannot rely on algorithms that require expensive parsing or pattern matching (*e.g.,* [21, 9]). As discussed in [4], during replay, if an exact match for a navigation action cannot be found in a page, heuristics (and optionally, users' hints) are an effective means to find the *closest match* for the action.

Extraction expressions also need to be made robust to changes to Web pages. For example, in the XPath expression (1) above, if the position of the `center` tag containing the desired tables changes (*e.g.,* a new preceding sibling `center` tag appears in the document), the expression will no longer retrieve the correct tables. Instead of absolute positions of nodes, the specification needs to include other information that helps the system uniquely identify elements to be extracted, even if the node positions happen to change. For instance, the XPath expression (2) specifies tables that contain the "Price" or "Option" string—this expression would still retrieve the correct itineraries even if new `center` tags are added (in Section 2.6 we discuss how these expressions can be automatically generated).

Even though the heuristics we have developed are robust to minor changes in the page structure, they can still break if the page structure changes radically. In such cases there needs to be a mechanism to detect and report errors to the client. During replay, if the WebViews player is not able to locate an object involved in a recorded action, it suspends the replay and notifies the user. The user may then re-record the SMB (to correct the problematic step) before the corresponding Web view is used again. Similarly, if the result of applying the XPath extraction expression to the final page returns nothing, the system reports "not found". Depending on what is sought, this may be an error. It may also mean that what is sought—*e.g.,* a column by a particular columnist—may not be available at present, but might well be tomorrow. It is also possible that the XPath expression returns an undesired object (if the page structure changes radically). In such cases, the user may need to correct the extraction expression.

2.5 Parameterizing Web Views

A nice feature of smart bookmarks (SMBs) is the ability it gives users to change the input parameters used during navigation at each replay. For example, in Example 1.1, if Juliana wants to check airfares on different travel dates, she need not record a new SMB, instead she can easily *parameterize* her SMB. Parameterization is possible due to the robustness features of SMBs, and their ability to navigate through dynamic pages. The original WebVCR prototype allowed users to specify whether form elements should be automatically filled, or whether they should be provided by the user during replay. In the latter case, the replay would stop at each page where attributes needed to be input. This approach works well if replay takes place at a desktop, but it may not be feasible in a thin-client environment (as these pages would have to be shipped to the client). Instead, the WebViews engine generates a page where a user can specify the values of all parameters required for navigation steps. The user-specified values are then fed into the SMB at the appropriate times during replay.

However, two issues need to be resolved to support reliable parameterization of Web views: internal attribute names that are undescriptive and invalid selections. Consider for example the *Book a flight* page at Travelocity, where users specify the itinerary details. The internal name for the departure month attribute is `dep_dt_mn_1`, which can be hard to identify. Also month values must have a specific format, *e.g.,* April is represented by "Apr", and if users input "April", the submission will fail. The WebVCR component of WebViews was extended to allow users to edit input attributes directly in the Web view. At recording time, users can specify mappings from internal names to more descriptive tags of their choice. In addition, extra information is saved for elements (*e.g.,* all values in a selection list are saved) so that inputs can be checked for validity (as we discuss in Section 3.3, this additional information can be very useful for transcoding Web views). Note that checking for validity of values is only possible for elements such as selection lists and radio buttons, where a domain is well-defined—it is not possible for text fields. Thus, even though we can reduce the likelihood of failures, they cannot be entirely avoided.

It is worth pointing out that parameterization only works reliably for deterministic sites—if there are different navigation paths for different values (or combination of values) of parameters, it will invariably fail when an alternate path is taken. For example, in DBLP[2] one can search for authors of papers in databases and logic programming by name. If the name is unambiguous, the result of the search will be a list of all that author's papers. But if multiple authors share that name, a list with these authors is displayed. To create a parameterized Web view for DBLP, WebViews would need the ability to express conditional statements.

2.6 GUI Design

As discussed above, writing robust XPath expressions can be a rather involved task—not a task for naive users. We are currently building a GUI that can produce clippings automatically. The GUI will allow users to choose a logical section of a page (*e.g.,* a table, a paragraph) to extract (much like `ezlogin's JumpPage service` [10] does) and produce the appropriate XPath.

If a user chooses a non-tabular entity (*e.g.,* a paragraph), the GUI asks for predecessor and successor text. The system then determines whether the predecessor (successor) text is within the selected page section or not. Using this information, an XPath expression can be automatically generated.

If the user chooses a table, the GUI asks her to specify the first row of the table that she is interested in, whether that row contains column labels, and if so, which column labels are of interest. Identifying the first row of interest within the table allows clipping to eliminate elements within the table that are of no interest (*e.g.,* links included in the table simply for layout purposes). For instance, for Figure 4, the user might specify the row containing `DATE` as the first row of interest, that that row contains column labels, and that the columns `MAKE/MODEL`, `YEAR`, `PRICE`, and `MILES` are of interest. The GUI also allows the user to optionally specify a phrase immediately prior to the chosen row—*e.g.,* the word "Showing" in Figure 4. The user may similarly specify a phrase in the text immediately past the last row she is interested in clipping. (If none is specified, the system assumes that all rows are of interest.) The GUI also asks her to identify the row labels that are of interest (if any) and whether the table is laid out row-wise (*e.g.,* Yahoo! stock quotes) or column-wise (*e.g.,* Quicken stock quotes). This information is very valuable for transcoding data for output to a small screen device or a telephone.

Since similar techniques are used to generate a robust XPath for both column-wise and row-wise layout, we will only describe XPath generation for the latter. For both, the GUI can generate robust clippings in XPath—clippings that survive significant changes to the HTML structure that leave the table of interest's form alone. For instance, the table can move from being top-level to being nested several levels deep in another table or vice versa and the XPath will continue to work properly. The following form of XPath is used for row-wise layout with a user-specified first row.

```
(//table/tr/td[contains(string(),
  '<USER-SPECIFIED-LABEL>')
  and not(descendant::table)]/parent::tr)[1]
```

The expression looks for a table containing a row with "*<USER-SPECIFIED-LABEL>*" (*e.g.,* "YEAR") as a data item. The "`not(descendant::table)`" part makes sure that the clipping gets the most deeply nested table for which such a row exists. Without it, when the table of interest is embedded inside another table, the expression would retrieve the containing table—XPath finds the outermost entity matching an expression when going down the tree (and the most deeply nested element matching an expression when going up). The clipping grabs that header row (retrieved using `parent::tr`) and succeeding rows. Starting at the header row skips extraneous information preceding the proper table content. An example of how to extend such an expression to handle predecessor text is given in Section 4.2. Note that column and row clipping (if any) is a post-processing step on the XHTML returned by XPath application.

The system allows users to specify a lot of information. However, it should be noted that all the information requested other than choosing the unit to clip is optional. The more information given, the more robust, the clipping can be. Also, the information about row-wise vs. column-wise

[2]http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/index.html

layout is very valuable information that can be passed to the transcoder. Without it, the transcoder must make an informed guess as to how the table is structured using a technique like that in [13]. Having the user specify this, increases the likelihood that their custom content will be transcoded in a meaningful form (see Section 3.3).

3. VOICE-ENABLING WEBVIEWS

3.1 Voice Interfaces

Interactive voice response (IVR) systems have traditionally been expensive to develop and run, being restricted to proprietary application development environments and to running on specialized hardware. As a result, for a long time, voice access was limited to a few services, such as accessing financial institutions, or consulting yellow pages. The steady increase in processor speeds, combined with developments in TTS and voice recognition technologies has made it possible to run IVR systems on commodity hardware with specialized boards. This opens up exciting possibilities for providing multi-modal input to computers (and soon to PDAs) as well as making it possible for small companies or even individuals to provide services over the phone.

VoiceXML [28] has recently been proposed as a standard XML-based markup language for IVR systems. VoiceXML replaces the familiar HTML interpreter (Web browser) with a VoiceXML interpreter and the mouse and keyboard with the human voice. As noted in [15], "the Web revolution largely bypassed the huge market for information and services represented by the worldwide installed base of telephones, for which voice input and audio output provide the sole means of interaction". VoiceXML has the potential to remedy this oversight.

It should be noted that some Web content is available by phone already. Also, some voice browsers and HTML to VoiceXML transcoders have been built (*e.g.,* pwWebSpeak [24], PhoneBrowser [22], and Vox Portal [12]). However, as Goose et al. [12] point out, the effectiveness of such systems is compromised in the presence of improperly structured documents [29]. Furthermore, much of the information on a Web page is not related to the primary purpose of the person browsing to the page (*e.g.,* ads, links to other parts of the site). Thus, it is very painful to hear such pages transcoded into voice, substantially reducing the utility of such browsers. By simplifying the content retrieval process, and filtering out uninteresting components of Web pages, the WebViews architecture can render the desired information in voice in a terser, far more user-friendly manner than more general voice browsers can.

It is worth pointing out that transcoding even simplified versions of HTML pages into VoiceXML presents interesting challenges. The serial nature of voice interfaces is in many ways incompatible with visual interfaces. For example, voice interaction becomes intractable if users are presented with too many choices (*e.g.,* a list with all American states). In contrast, *displaying* many choices may be inconvenient, but it is still manageable. In what follows, we describe the VoiceViews prototype, and our approach to provide voice access to Web views.

3.2 Transcoding Web Views into VXML

Section 2 assumed the presence of transcoding proxies to enable access to various devices. Note, however, that the

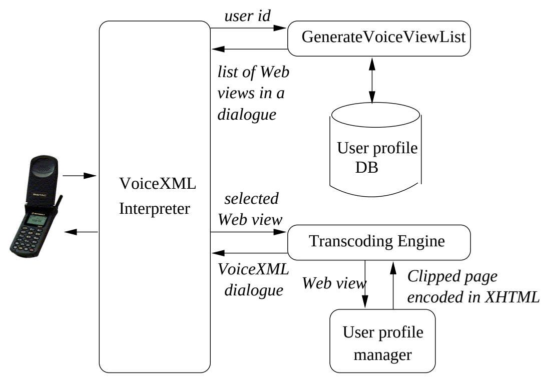


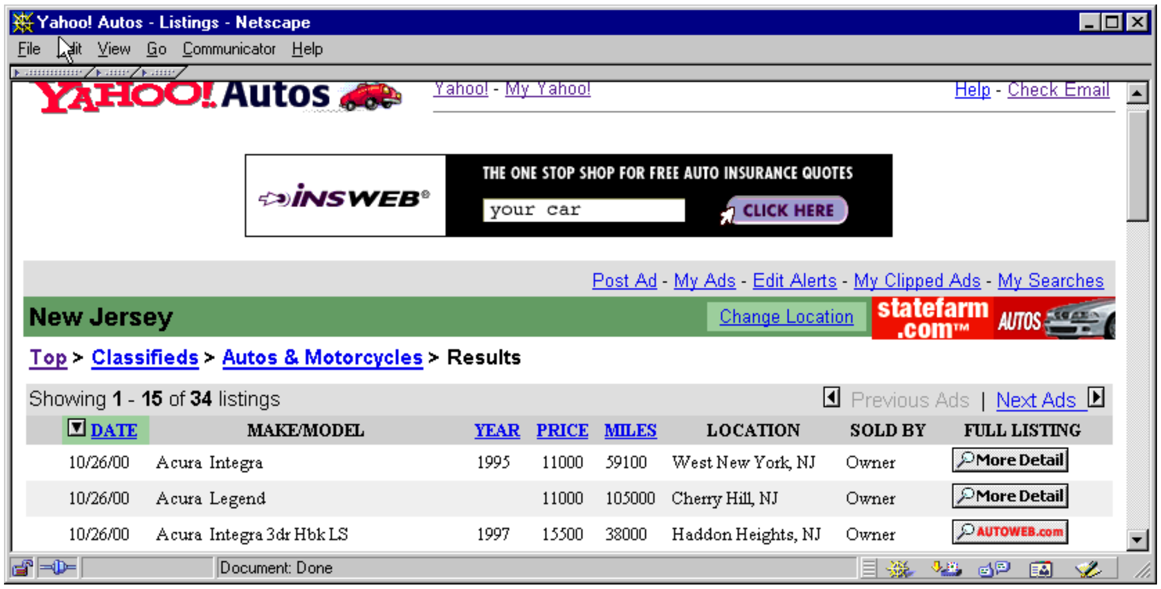
Figure 5: Low-level architecture of the VoiceViews prototype

transcoding functionality can also be incorporated into the WebViews server, with the added advantage that the user can now *annotate* the Web view and supply extra information that can be used in the transcoding process to produce better quality content, and a more user-friendly experience. The main drawback of this tight-coupling it that the transcoding engine and server must agree on a set of annotations. If they are developed by separate parties, this will sometimes discourage such an architecture, unless, of course, a standard for such annotations exists.

We have built an engine that transcodes clipped HTML content into VoiceXML. In what follows, we describe its architecture, and discuss how it can be combined with the WebViews server to create VoiceViews. Note that we *still* assume the presence of a Voice gateway (running a VoiceXML interpretor) that connects the PSTN network to the IP network. The user calls a phone number, and the VoiceXML interpretor requests the WebViews server for any VoiceXML dialogs to be played to the user.

The current transcoding architecture is diagramed in Figure 5. The usage scenario is as follows. When the user calls a special phone number, a fixed *caller identification* VoiceXML dialog is started. The dialogue attempts to identify the user using his caller ID (which it uses as `userid`); if that is not available, the system interrogates the user for the `userid`. Once the `userid` is obtained, the list of Web views associated with that user is looked up (via GenerateVoiceViewList) from the User Profile DB, and a VoiceXML dialog is generated that prompts the user to select between one of his recorded Web views. The user can then make his selection via touch-tone or spoken input. Once the user makes a choice, the VoiceXML interpreter passes the Web view information to the Transcoding Engine, which in turn queries the WebViews execution engine to execute the Web view. The Transcoding Engine converts the clipped content into VoiceXML, utilizing the extra annotations supplied with the Web view.

Figure 6 shows the VoiceXML dialogue generated by our transcoder for the clipping (appropriately tidied) of the Yahoo! car page shown in Figure 4. In the dialogue, each car listing is transcoded as a `field` of a VoiceXML `form`. The `form` contains all the data transcoded from a single top-level table. The transcoded output could also have been output as a single `block` so that the entire contents would have been read as a unit using TTS, but that would have given the user



Yahoo! Autos - Listings - Netscape

File Edit View Go Communicator Help

YAHOO! Autos

Yahoo! - My Yahoo!

Help - Check Email

INSWEB THE ONE STOP SHOP FOR FREE AUTO INSURANCE QUOTES your car CLICK HERE

Post Ad - My Ads - Edit Alerts - My Clipped Ads - My Searches

New Jersey Change Location statefarm.com™ AUTOS

Top > Classifieds > Autos & Motorcycles > Results

Showing **1** - **15** of **34** listings Previous Ads | Next Ads

DATE	MAKE/MODEL	YEAR	PRICE	MILES	LOCATION	SOLD BY	FULL LISTING
10/26/00	Acura Integra	1995	11000	59100	West New York, NJ	Owner	More Detail
10/26/00	Acura Legend		11000	105000	Cherry Hill, NJ	Owner	More Detail
10/26/00	Acura Integra 3dr Hbk LS	1997	15500	38000	Haddon Heights, NJ	Owner	AUTOWEB.com

Document: Done

Figure 4: Yahoo! Car page retrieved by the WebVCR for Transcoding

```
<?xml version="1.0" ?>

<vxml version="1.0" caching="safe">

<form>
  <field name="row1">
    <grammar type="application/x-jsgf">
        next {next} | skip {skip} </grammar>
    <prompt bargein="true">
        make/model: Acura Integra
        <break size="medium"/> year: 1995
        <break size="medium"/> price: 11000
        <break size="medium"/> miles: 59100
    </prompt>
    <catch event = "nomatch noinput">
        <assign name="row1" expr="'next'" />
        <goto nextitem="row2" />
    </catch>
    <filled>
        <if cond="row1 == 'next'">
            <goto nextitem="row2" />
        <else />
            <goto next="#end_table" /> </if>
    </filled>
  </field>
  ...
<form id="end_table">
  <block> <goto next=
"http://www.voiceviews.com/GenerateVoiceViewList.
cgi?userid=9081234567" />
  </block>
</form>
</vxml>
```

Figure 6: Transcoding of Yahoo! Auto data into VoiceXML

no option but to listen to the whole table being read or to hang up. As it is, the script listens for special keywords, namely "next" and "skip", allowing the user to quit hearing details of a single row or of the rest of the table. If the user says nothing (`noinput`) or something incomprehensible (`nomatch`), the script goes to the next row.

3.3 Using Clipping Information

In order to intelligently transcode a table, Web view annotations are crucial. A table may be organized row-wise, column-wise, or neither (*e.g.*, being used simply for layout)—and each requires substantially different transcoding. For instance, if the table of interest in Figure 4 is treated as being there simply for layout, the transcoded voice would be partially incomprehensible—something like: "Showing 1 -15 of 34 listings Previous Ads Next Ads DATE MAKE MODEL YEAR PRICE ... FULL LISTING 10/26/00 Acura Integra 1995 11000 ...". The knowledge that MAKE/MODEL, YEAR, PRICE, and MILES are the columns to transcode and that the table is laid out row-wise allows the transcoder to pair headers and values together (see Figure 6), and to eliminate uninteresting data. Similar techniques are useful for small screens (*e.g.*, WAP phones) even where tables are supported, because they produce something much more readable than tables with rows that wrap lines.

Also useful is the information stored for the parameterization of WebViews for entities like radio buttons and pull-down lists to enable users to specify the corresponding values. For these entities, parameterized SMBs store the list of acceptable choices that the user may enter. (Note that this information is not required in un-parameterized bookmarks since the choice is fixed for them). Not only does this information allow the system to prevent bad user-input, it also makes it possible to transcode the parameterized data in a more convenient form. Rather than reading out each possible value with an associated number and having the user enter that number by voice or touchtone, the system can generate a grammar that accepts the legitimate choices. The lists are generally small enough that voice recognition will work reasonably well. For example, when giving a user the choice to enter the name of a state, allowing the user to say "WV" immediately is far more convenient than asking her to wait to hear and respond to "47 WV". It is worth pointing out that due to the limitations of voice-independent recognition systems, this technique does not work well for text fields where no information about the domain is available, or when domains are large.

4. EXPERIENCES

In order to verify that our WebViews creation, extraction and transcoding framework works in practice, we created various Web views of popular as well as not so popular Web sites. In particular, we wished to determine how widely applicable (in terms of coverage) our tools are, verify the effectiveness of our robustness techniques, as well as determine the applicability of our transcoding strategies. We now discuss some of the issues we faced, as well as some general techniques we developed.

4.1 Cleaning HTML Content

Even though we used HTML Tidy [27] to transform pages into XHTML (for easier extraction and transcoding), we found that problems remained for a number of Web sites. We had to perform further post-processing on Tidy's output before the HTML content was usable (*e.g.,* for input to Xalan for evaluating extraction expressions). For example, we found that we had to explicitly remove Javascript from pages after tidying them, since the "<"s that appear in some scripts caused the tidied document to remain malformed XML. We give details on a subset of the problem resolutions below (note that these problems deal only with the final page from which relevant content is to be extracted, and not to the intermediate pages obtained when executing a Web view):

Non-standard tags: Tidy is able to clean up most bad HTML. However, it does require that the document tags be legitimate HTML tags. Some sites generate proprietary tags which are "illegal", but ignored by browsers. To handle illegal tags that showed up in sites we cared about, we installed the corresponding tags in Tidy's default tag table. This was done without modifying the Tidy source code.

Duplicate attributes: Another problem we found in many pages was the presence of multiple attribute definitions, for example:

```
<td bgcolor="#000000" align="CENTER" valign="TOP"
  align="CENTER">
```

Tidy does not detect duplicate attribute definitions. However, our XML parser gave up parsing when it found an attribute multiply defined. To solve this problem, we had to go over the Tidy output and eliminate multiply defined attributes.

4.2 Extracting Desired Content

There were a number of problems that we encountered when clipping content from HTML pages, most relating to robustness of extraction expressions, and the expressiveness of XPath. Some of these problems are discussed below.

Changes to page structure: In building robust extraction expressions, keeping around a fair bit of context information is quite valuable as some sites change their HTML layout frequently, while maintaining a very similar appearance. One particularly dynamic site is Yahoo! Autos. In the course of the day, the same SMB can return pages in at least three different forms: with all the cars in a single top-level table, with all the cars in a single table embedded in another table, and with some of the cars in a "Featured Cars" table and the rest in a much bigger table—both tables having the same structure (once one gets past the layout data to the logical data). The first two variants are taken care of by ignoring the path to the Auto table and simply looking at its form—which was fixed. The third was taken care of use predecessor text. The column headers of the main listing always followed a row containing "Showing 1 - n of m listings". The row following "Showing" contained the column header information. The following XPath which uses both predecessor information and the form of the table works for all cases:

```
 //table/tr/td[contains(string(),'Showing') and
not(descendant::table)]/parent::tr/
following-sibling::tr/
td[contains(string(), 'YEAR')]/parent::tr
```

Note that in this case "Showing" was *inside* the table being transcoded. For handling text preceding (following) the table, the system would find a text node containing the desired anchor text and then search the following (preceding) nodes for a table of the desired form.

WYSIWYG – Not!: We found that we had to be very careful in our assumptions about the `string()` that would come back from an XML entry, when looking for key strings that the user had specified to identify which document fragments to clip. Line breaks appear in unexpected places. For example, while "All\nListings" and "All Listings" in the XHTML look the same when rendered on the screen, they do not compare as the same. Which form the text came out in appeared random within a document—sometimes the text was all on a single line, sometimes it was broken across lines. Consequently, we normalize the output of Tidy to replace line breaks with blanks before using XPath to extract data. Note that this has no effect on voice transcoding.

Limitations of XPath: We found one limitation of XPath which was the lack of an axis for location paths combining "self" and "following-sibling". One can simulate this by using "preceding-sibling" to go backwards in the document and then choosing "following-sibling" from that point provided a preceding sibling exists. If no preceding sibling exists, then combining "parent" and "child" can be used. However, no one expression can express this relationship. This is unfortunate, because when clipping a contiguous collection of HTML entities, the ability to specify that an entity containing a particular piece of text and some number of succeeding entities should be grouped together for clipping would be quite useful. However, such an expression cannot be written unless it is known whether or not the entity of interest will have predecessor siblings or not. Such an expression could have been used to write expression (2) (from Section 2.2) in a different way that requires less user input:

```
(//table/tr/td[contains(string(),'Price:') and
not(descendant::table)]/parent::tr/
parent::table/following-sibling-or-self::table)
[position() >= 1 and position() <= 6]
```

4.3 Transcoding Issues

In the general WebViews server architecture presented in Figure 1, we assume that the transcoding for various formats would be performed by an appropriate gateway that communicates with the WebViews server. However, generic transcoders often do not work well with complex HTML, even after the relevant content is extracted from the retrieved page. However, we found that certain extra information could be specified in the Web view, which could be used to greatly improve the quality of the transcoding, and hence result in a better user experience.

Table Layout: A few sites layout their rows as a single row whose embedded data entries, `<td>`s, contain line breaks. To handle such sites, the GUI needs to check for embedded linebreaks within `<td>`s and ask the user whether or not the peculiar layout of rows is being used or the row entry is simply long—if at least two entries have a similar number of linebreaks.

VoiceXML: Unfortunately, standards compliance with the VoiceXML 1.0 spec is spotty. We have experimented with three VoiceXML interpreters, and all of them have missed some useful features of the spec. One of them not only failed to implement one tag according to the spec, but implemented a tag with the same name but different semantics. (We gave up trying to develop VoiceXML documents on that platform because of its frequent disregard for the standard.) Even for the two better behaved platforms, the difference in implemented feature sets makes it impossible at present to develop scripts of any complexity that will run on both.

These lacks also made VoiceXML dialogues much longer than they would have been in a full-featured VoiceXML implementation. For instance, the transcoding of each row in Figure 6 could have been considerably simplified (*i.e.,* the `grammar`, `catch`, and `filled` parts could have been removed) had `form` level grammars (grammars that apply to the whole form and not just to an individual field within the form) been implemented in the VoiceXML platform we are using. Then, those implementation details could have been shared across rows being transcoded. We also would have supported more features like "back". We dropped them to keep the VoiceXML file from becoming even larger. When transcoding large tables, we did not want to make the VoiceXML file so large that XML parsing becomes noticeable. Consequently, we dropped useful but infrequently used navigation options. If the code were shared at the form level, this would not have been an issue.

5. RELATED WORK

There are two main areas of work related to WebViews: wrapper creation, and information delivery to diverse terminals. We review some of the important literature in these fields, and mention a few commercial products.

In the area of information integration, many systems and techniques have been proposed to *wrap* Web sites. Most of the work in this area focuses on extracting structure from semi-structured data (*e.g.,* [5, 2]). The extractor component of the WebViews system is not concerned with *understanding* the structure or discovering the schema of the underlying data, but in providing robust mechanisms to identify high-level HTML/XML syntactic components (*e.g.,* the first table after a specific string). The first version of WebViews uses XPath to address specific components to be extracted from Web pages. Other languages could also be used for this purpose, for example WebL [14] or the scheme used by W4F [25]. These languages provide good mechanisms to extract fragments from documents—in some cases, they are easier to use than XPath. However, XPath is a widely accepted standard and there are freely available tools to process XPath expressions.

Whereas data extraction has been studied extensively, not much attention has been given to data retrieval. WebVCR [4] was to the best of our knowledge the first proposal for automating Web navigation without requiring programming expertise. In order to provide customization of Web views, the WebViews system extends the WebVCR to handle parameterized SMBs (as described in Section 2).

The issue of wrapper robustness has been studied recently in [4, 21, 9]. In our previous work [4] we introduced a set of heuristics to improve the robustness of automated navigation for SMBs. Davulcu et al. [9] propose techniques to generate resilient regular expressions that are able to identify specific objects in many variations of the same document. Phelps and Wilensky [21] proposed the use of redundant specifications in order to maintain *robust locations* within documents so as to support document annotation. Their techniques are closely related to those we use for producing robust clippings. Our focus is slightly different, instead of finding arbitrary locations in text documents, we need to find more well-defined structures like tables for transcoding and so have access to more semantics, and unlike [21], our matching routines take internal structure of elements into account.

On the commercial side, there has been a proliferation of personalization systems which offer services that range from notifications about changes to certain Web pages (*e.g.,* Mind-it [17]) to the creation of personal portals (*e.g.,* Yodlee [35]). Most of these services have limited coverage, *i.e.,* they offer *clippings* for a limited number of sites. More recently, services such as Octopus [19] and ezlogin [10], allow creation of clippings from arbitrary Web pages. However, to the best of our knowledge they have some important limitations:[3] they are restricted to pages that have well-defined URLs; and they are very sensitive to changes in these pages. The WebViews system addresses both of these limitations. Since it is able to record navigation actions, it lets users create clippings for virtually any Web site/page. In addition, the Web views it generates are robust.

Another important area of work related to this paper is information delivery to heterogeneous devices. A number of companies (*e.g.,*`tellme`, `heyanita`, `oraclemobile`) provide tools and professional services for Web site creation and hosting. For example, OracleMobile creates and hosts wireless Web sites for online businesses, wireless ISPs, and enterprises. The goal is to ensure that content is instantly available on every mobile device—including Web-enabled phones, PDAs, two-way wireless messaging devices and one-way pagers. In contrast with wireless application service providers (WASPs), the focus of this paper is in providing a service to end-users, not content providers. The requirements and constraints in these scenarios are quite different. Whereas for WebViews no cooperation is required from the sites, WASPs work together with content providers. By gaining access to the internals of the Web sites through the cooperation of content providers, WASPs are able to create reliable views and update them before they break (since it is to content provider's interest to notify the WASP about changes). Nonetheless, the tools and techniques we propose would also be useful in this context—but, of course, some extensions and adaptations would be necessary. For example, a more powerful model for Web sites like navigation maps [8] that support non-determinism and iteration would be necessary to handle more complex navigation, as opposed to linear paths currently supported through SMBs.

Content transcoding has attracted much attention recently. Device-specific transcoders are used that perform on-the-fly content translation in order to provide Web access to a variety of devices. The kind of translation done by these proxies include reduction of image resolution [11, 1], modification of HTML constructs that cannot be effectively viewed

[3]These limitations are also mentioned in [16].

in smaller screens (*e.g.,* ProxiWeb rewrites pages that contain frames so that they display the links corresponding to the frames), and translation from HTML to other languages such as the wireless markup language (WML) [30] and VoiceXML (VXML) [28]. Given the growing complexity of Web sites (*e.g.,* the presence of scripting languages, dynamic content, malformed content), transcoding can be very hard, and in practice, many pages and services are not amenable to transcoding and cannot be accessed through a variety of devices that require non-HTML content such as WAP phones (which require WML) and voice interfaces (*e.g.,* VoiceXML). By allowing users to easily customize services and filter out irrelevant content and complex features, the WebViews system greatly simplifies the transcoding process, increasing the Web coverage for many devices. Note, however, that some of the features provided by the generic transcoding proxies can still be layered on top of the WebViews system, for example, image resolution can be reduced in case the extracted content contains an image.

6. CONCLUDING REMARKS

In this paper we describe the WebViews architecture, how it simplifies the creation of robust customized views of Web content and services, and how these views can be tailored for presentation in different devices. We also describe the implementation of the VoiceViews system, and discuss how Web views can be effectively transcoded into VoiceXML.

Our initial prototype is geared towards letting casual Web users create Web views. Consequently, we cannot assume any cooperation from Web sites, and robustness of Web views is an essential requirement. In addition, since users are not experts, tools have to be very simple to use. One interesting future direction of work is to investigate how the WebViews architecture can be extended to handle the business scenario, where a content provider himself wants to create different views of his own content. For example, more powerful facilities to model Web sites (such as the navigation maps proposed in [8]) that support non-determinism and iteration are likely to be needed.

As a final note, it is worth pointing out that the applications of WebViews go beyond information delivery to diverse environments. For example, Web views can be used as basic building blocks for personal portals such as the ones provided by mynetscape.com.

7. REFERENCES

[1] S. Acharya, H. Korth, and V. Poosala. Systematic multiresolution and its application to the world wide web. In *Proc. of ICDE*, pages 40–49, 1999.

[2] B. Adelberg. NoDoSe - a tool for semi-automatically extracting structured and semi-structured data from text documents. In *Proc. SIGMOD*, pages 283–294, 1998.

[3] http://www.amazon.com/anywhere.

[4] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web navigation with the WebVCR. *WWW9/Computer Networks*, 33(1-6):503–517, 2000.

[5] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[6] http://www.avantgo.com.

[7] CDPD. http://www.wirelessdata.org/develop/cdpdspec.

[8] H. Davulcu, J. Freire, M. Kifer, and I. Ramakrishnan. A layered architecture for querying dynamic web content. In *Proc. SIGMOD*, pages 491–502, 1999.

[9] H. Davulcu, G. Yang, M. Kifer, and I. Ramakrishnan. Computation aspect of resilient data extraction from semistructed sources. In *Proc. of PODS*, 2000.

[10] http://www.ezlogin.com.

[11] A. Fox and E. Brewer. Reducing www latency and bandwidth requirements by real-time distillation. *WWW5/Computer Networks*, 28(7-11):1445–1456, 1996.

[12] S. Goose, M. Newman, C. Schmidt, and L. Hue. Enhancing web accessibility via the Vox Portal and a web-hosted dynamic HTML↔VoxML converter. *WWW9/Computer Networks*, 33(1-6):583–592, 2000.

[13] J. Hu, R. Kashi, D. Lopresti, and G. Wilfong. Medium-independent table detection. In *Proc. of Document Recognition and Retrieval VII (IS&T/SPIE Electronic Imaging)*, volume 3967, pages 291–302, 2000.

[14] T. Kistlera and H. Marais. WebL: A programming language for the Web. *WWW7/Computer Networks*, 30(1-7):259–270, 1998.

[15] B. Lucas. VoiceXML for web-based distributed conversational applications. *Commun. ACM*, 43(9):53–57, 2000.

[16] H. McCracken. Web savvy: Better ways to browse the web. PC World magazine, December 2000.

[17] Mind-it. http://www.netmind.com/.

[18] http://channel.nytimes.com/partners/palm-pilot.

[19] http://www.octopus.com.

[20] http://www.omnisky.com.

[21] T. Phelps and R. Wilensky. Robust intra-document locations. *WWW9/Computer Networks*, 33(1-6):105–118, 2000.

[22] http://phonebrowser.research.bell-labs.com.

[23] ProxiWeb. http://www.proxinet.com.

[24] pwWebSpeak.

[25] A. Sahuguet and F. Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *Proc. of VLDB*, pages 738–741, 1999.

[26] http://www.sprintpcs.com/wireless/wwbrowsing_providers.html.

[27] http://www.w3.org/People/Raggett/tidy.

[28] http://www.voicexml.org.

[29] Web accessibility initiative. http://www.w3.org/WAI.

[30] Wireless Application Protocol Forum. *Wireless Application Protocol: The Complete Standard.* Wiley, 1999.

[31] http://www.luxnet.at/docu/xalan/overview.html.

[32] http://www.w3.org/TR/xpath.

[33] http://www.w3.org/TR/xslt.

[34] http://www.jclark.com/xml/xt.html.

[35] Yodlee2go. http://www.yodlee.com.