Browser-Oriented Data Extraction

*I-Chen Wu, *Jui-Yuan Su, and †Loon-Been Chen
* *Department of Computer Science and Information Engineering*
National Chiao Tung University, Hsinchu, Taiwan
† Department of Computer Science and Information Engineering, Tunghai University, Taichung, Taiwan
{icwu,rysu,lbchen}@csie.nctu.edu.tw

Abstract: *Traditionally, most researchers used the URL-oriented data extraction model for data extraction. In this model, the systems extract URLs from pages and then use the extracted URLs to access next pages. However, more and more pages currently use script functions to access next pages. Since it is hard to extract URLs from script programs, it is inappropriate to use this model for such pages.*

For solving this problem, this paper proposed a new data extraction model, named the browser-oriented data extraction model. In this model, the system built on top of browsers accesses pages by simulating users' operations on browsers, which can also trigger script functions.

Besides, this paper defines a scripting language, named the BODED (Browser-Oriented Data Extraction Description) Language, which instructs the system to do data extraction.

Keywords: data extraction, Internet, BODED.

1. Introduction

With the rapid development of World Wide Web (WWW) recently, more and more information is published in WWW. Hence, it becomes significant for many users to collect useful information over Internet. Usually, these users simply want to extract some needed segments from web pages, instead of retrieving the whole web pages. For example, customers want to extract products' data (e.g., names and prices) from different web sites for price comparison; managers need to retrieve business news regularly for business analysis; and researchers want to extract references of academic articles from some archive sites. Since most of the data to be extracted are usually regularly located inside or among web pages, it becomes possible and helpful to automate the process of data extraction from these pages.

```
<TABLE>
  <TR>
    <TD><A href="db.html">Databases</A></TD>
    <TD><A href="al.html">Algorithms</A></TD>
    . . .
  </TR>
</TABLE>
```

Figure 1. The main category page.

```
<TABLE>
  <TR>
    <TD><A href="de.html">Data Extraction</A>
    </TD>
    <TD><A href="dm.html">Data Mining</A>
    </TD>
    . . .
  </TR>
</TABLE>
```

Figure 2. The databases subcategory page located at `db.html`.

```
<TABLE border=1 width="100%">
  <TR>
    <TD>BODED: A Data Extraction Service
        Description Language</TD>
    <TD>I-C. Wu, J.-Y. Su, L.-B. Chen </TD>
    <TD>Submitted to ICS 2004.</TD>
  </TR>
  <TR>
    <TD> Managing Web-based data - Databases
        models and transformations </TD>
    <TD> Atzeni, P., Mecca, G., Merialdo,
        P.</TD>
    <TD> IEEE Internet Computing 6(4):
        33-37 2002</TD>
  </TR>
  . . .
</TABLE>
<A href=nextpage.html>next</A>
```

Figure 3. A paper list page at `de.html`.

Consider an example of data extraction: a simplified bibliography archive site including two-level category pages. Figure 1 (below) shows the HTML file of the main category page that links to subcategory pages, one of which is shown in Figure 2. Subcategory pages link to a list of paper list pages, one of which is shown in Figure 3. A paper list page lists author names, titles, and publishers of article references. At the end of the paper list page, a URL links to the next paper list page for more references in the same subcategory.

Data extraction for the above example is to extract all article references in the whole bibliography archive. In order to extract all article references, the data extraction system needs to traverse all the paper list pages in the archive and then extracts the article references from each paper list page.

Traditionally, for traversing the web pages, the approach of most researchers [1][2][3][4][5][6] is to extract URLs from web pages and then use these extracted URLs to retrieve next pages via the HTTP protocol. For example, after extracting the URLs, say `db.html` from the main category page in Figure 1, the system uses these URLs to read the next subcategory pages, as shown in Figure 2, for more data extraction. Such a data extraction model is called the *URL-oriented data extraction model*, since all pages are accessed via given URLs.

However, more and more current web pages include scripting languages, such as JavaScript or VBScript, to make the presentation of web pages more flexible and friendly. If scripts are used, it becomes much harder to do data extraction by using the URL-oriented data extraction model.

```
<SCRIPT language=Javascript>
    function DirectToNext(name){
       window.open(name+".html")
    }
</SCRIPT>
<TABLE border=1 width="100%">
  . . . <!-- The same as those in Figure 3 -->
</TABLE>
<A href="DirectToNext('nextpage')">next</A>
```

Figure 4. A paper list page with a Javascript function.

For example, let the paper list page in Figure 3 be rewritten with a JavaScript function, as shown in Figure 4. The system based on the URL-oriented data extraction model can easily access the next page for the page in Figure 3, by extracting the URL `nextpage.html` from the attribute `href` of the element `A`. However, for the page in Figure 4, the URL of the next page is hidden in the JavaScript program. Since script programs are usually less regular or harder to predict when compared with HTML/XML structures, data inside programs are harder to be extracted than those in the structures of HTML/XML. Thus, traditional systems in the URL-oriented data extraction model can rarely process the pages with script programs.

In order to solve the above problem, this paper presents a new data extraction model, called the *browser-oriented data extraction model*. In this model, the system accesses web pages by simply simulating human operations, such as a click operation on the browser. Obviously, it is easy for the case to work in this new model, but hard or almost impossible in the traditional URL-oriented data extraction model.

In this paper, Section 2 presents the browser-oriented data extraction model. Based on the model, Section 3 defines the scripting language, named the *BODED* (*Browser-Oriented Data Extraction Description*) Language, which instructs the data extraction system to do data extraction. Section 4 discusses some related issues and gives a conclusion.

2. Browser-Oriented Data Extraction Model

In a browser-oriented data extraction model, the system uses a set of browsers for data extraction. First, Subsection 2.1 briefly reviews commonly used browsers, such as Internet Explorers and Netscape. Then, we describe the browser-oriented data extraction model in the rest of this section. Subsection 2.2 describes the extraction model inside browsers, while Subsection 2.3 describes the interaction model between browsers. The former is called the *intra-browser model*, and the latter the *inter-browser model*. Finally, Subsection 2.4 introduces services that control the operations of browsers.

2.1. Browsers

Given a URL, a browser uses the HTTP protocol to retrieve the page file in HTML located at URL, and then display the page after retrieval. In an HTML page, each tag is called an *element* and each element has several *attributes*, each of which may or may not have a value. In Figure 1, the anchor element `A` is a *hyperlink* and its attribute `href` has a URL value, indicating the location of the linked page. When the user clicks on the element, the browser uses the HTTP protocol to access the linked page located at the URL.

In order to make the presentation of pages more flexible and friendly, most commonly used browsers support JavaScript or VBScript in an event-driven model. When *events* are issued, the script functions specified in the events will be called. Events include mouse click (on some element), mouse down, mouse over, content change or filling (of an element), loading (of a page), timer, etc.

Events can be classified into *external events* and *internal events*, in the sense of issuing sources. External events are those issued by external users, such as mouse click or typing texts into some text areas. If users directly type URLs to trigger the system to access the corresponding page, this is called a *URL event*, also an external event. Internal events are those issued or caused by other script programs internally. For example, the timer events are usually issued by other script functions, and the corresponding script functions are called after

designated times; and the content change events are issued when the content of some element or some attribute is changed (normally caused by other script functions).

For simplicity, the browser in our model, in principle, follows the above, but with some slightly modification for data extraction as described in the rest of this section. In the rest of this section, unless explicitly specified, the term *browser* implicitly indicates the browser in our model, not the commonly used browsers, such as Internet Explorers and Netscape.

2.2. Intra-Browser Model

Initially, browsers display no pages and are called *empty*, or in the *empty state*. The current displayed pages of browsers, if not empty, are called the *active pages* of the browsers.

Inside a browser, the data extraction system in our model uses a script or expression to locate and extract elements inside the active pages. The well-known examples are the Document Object Model (DOM), and the XPath language. In this paper, we choose the XPath language, since XPath has advantages over the DOM as mentioned in.

Using an XPath expression, we can locate a set of elements or extract the content of these elements. In our model, it is very important to locate elements, since this allows the system to issue events, such as mouse click, on the located elements.

A problem with locating elements is: if some elements were already located but are being changed (e.g., relocated or simply gone) due to running some script functions, then it will become unexpected to issue events on these elements. In this case, these elements are called *volatile* elements in this paper. A data extraction system without volatile elements is said to be *consistent*.

In our model, we will prevent elements from being volatile, so that the data extraction system in our model is consistent. In order to avoid the problem, the model needs to include the following two restrictions. One is to locate elements or issue events on located elements when no script programs are running. We will describe it in the rest of this subsection. The other is to prevent from issuing events on the browser including some located elements that will be used later, as described in next subsection.

In our model, after an external event is issued, the corresponding script functions are invoked, and the browser is called to enter the *volatile state*. The external event may issue more internal events that in turns may also invoke more internal events repeatedly. The volatile state ends when the external event as well as all these internal events has been completed. Note that when one timer event was just issued but not executed yet, the browser is still in the volatile state. The period from entering to ending a volatile state is called the *lifetime* of the external event.

A browser is called to enter the *steady state* when the volatile state ends. In the steady state, there are no script programs to be called. This implies that the browser changes no more page content or structure in this state (unless more internal events are issued to enter volatile states again). So, in our model, operations such as extracting data, locating elements and issuing external events must be done in the steady state, not in the volatile state.

In our model, the lifetime of an external event must be finite, that is, the script programs triggered by the event ends eventually. Unfortunately, there are several ways to let the programs not end eventually. For example, the timer function is issued every one second, or an infinite loop is hidden in a script program. Since it is proved that there are no ways to detect whether programs will end eventually, our data extraction model works only for those external events with finite lifetimes.

2.3. Inter-Browser Model

In our model, it is assumed that the data extraction system includes an infinite number of browsers. All the browsers are mutually independent, that is, any two different browsers do not share the same data, objects, or status. Initially, all the browsers are *empty* (i.e., have no active pages) and available for initiation in the following two ways. First, for an empty browser, the system can issue a URL event on the browser to load the web page located at the URL.

Second, for an empty browser, the system can replicate from another designated browser to this empty browser, when the original browser is in the steady state. After replication, the active page and all the associated data (including variables in a JavaScript or VBScript) and status of the replicated browser are the same as those of the original. Thus, the behavior of issuing an external event to some element node of the replicated would be the same as that to the corresponding element node of the original.

Browser replication is an important technique to prevent the located elements from being volatile. Consider the following example. For the main category page in Figure 1, we first use an XPath expression to locate those links to the subcategory pages and then issue mouse click events on these located links to access the subcategory pages,

respectively. Accessing multiple pages from a browser is called *multi-way navigation* in this paper.

Now, suppose to use one browser only for the above multi-way navigation. Then, we will use the following steps, as also shown in Figure 5 (below).

1. Locate the elements linking to subcategory pages in the browser.
2. Issue a mouse click event on the first element node (linking to the page at `db.html`).
3. Load the page, "`db.html`", into the browser.
4. Process the page.
5. Go back to the main category page.

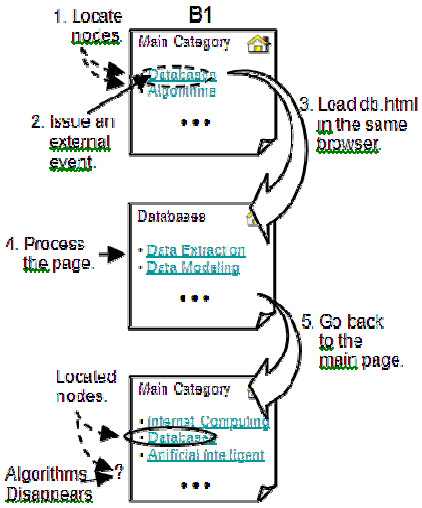

Figure 5. Data extraction on two pages with one browser only.

If the mouse click event in Step 2 triggers a JavaScript function which may change the content of the main category page as shown in Figure 5, then issuing an event in Step 2 makes the located elements volatile, e.g., the node linking to the next subcategory page "`al.html`" disappears and the node linking to "`db.html`" appears in a different place. Thus, data extraction for other subcategory pages becomes hard to be predicted. In this case, the system becomes inconsistent.

In order to make the system consistent for multi-way navigation, our model forces browser replication before each external event is issued. In our model, the following steps, also shown in Figure 6 (below), are used to prevent the located links from being volatile.

1. Locate the elements linking to subcategory pages in the current browser, B1.
2. Replicate the current browser to a new browser, B2.
3. Issue a mouse click event on the element of B2, corresponding to the first located element of B1.
4. Load a new page into B2.
5. Process the page of B2.

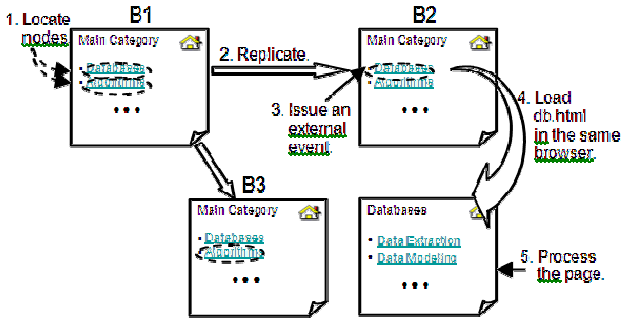

Figure 6. Data extraction on two pages with browser replication.

For browser replication as above, after Step 5, the elements located in Step 1 are still in the original browser B1 without any change. Thus, the system is still consistent.

In order to make the system consistent, our model simply forces browser replication before each external event is issued. Thus, we obtain the following property.

Property 1. *The browser-oriented data extraction system described above is consistent.*

Since the system always does browser replication upon issuing an external event, all the browsers have at most three periods. The first period is in the empty state, the second in the volatile state (due to an external event), and the third in the steady state.

2.4. Services for Browsers

In our model, *services* are to process the operations of browsers, e.g., extract data from the active pages, locate the elements, initiate new browsers, and issue external events on elements of the page. When a browser is initiated, one and only one service must be assigned to the browser. Similarly, when the system initially initiates an empty browser via a URL event, an initial service also needs to be assigned to the browser.

If service S1 assigned to browser B1 initiates a browser B2 and assigns service S2 to B2, we call S1 the *parent* of S2 and B1 the *parent* of B2. Since a service may initiate several browsers each associated with one service, all the services form a *service tree*. Similarly, all initiated browsers form a *navigation tree*. The initial browser is the root of the navigation tree and the initial service is the root of the service tree.

3. BODED

This section defines a script language, named Browser-Oriented Data Extraction Description (BODED) Language, an XML-based language. The BODED language describes the operations in a browser-oriented data extraction system, described in the previous section.

A BODED script is enclosed by the element `BODED`. This element contains two types of elements, `INIT` and `SERVICE`. Each `SERVICE` element specifies the operations of a class of services, named in the attribute `name`. As defined in Section 2, when a browser is initiated, a service is instantiated and assigned to the browser to process the active page of the browser, e.g., extract data from the associated pages, process these data, and initiate other services.

The element `INIT` designates the URL of the initial web page in the attribute `url` and the name of the initial service in the attribute `service`. Based on the description in Section 2, the data extraction system initiates a browser by issuing a URL event with the given URL and assigns to the browser the service, named in the attribute `service`. In our system, we only consider *one* initial web page, so that there is only one navigation tree and the initial browser becomes the root of the navigation tree.

```
<BODED>
  <INIT service="PaperList"
   url=http://boded.csie.nctu.edu.tw/de.html />
  <SERVICE name="PaperList">
    <VAR name="Title"
         xpath="//TABLE[1]/TR/TD[1]" />
    <VAR name="Author"
         xpath="//TABLE[1]/TR/TD[2]" />
    <VAR name="Publication"
         xpath="//TABLE[1]/TR/TD[3]" />
    <BODEDLET code=SavePaperList.dll />
  </SERVICE>
</BODED>
```

Figure 7. A BODED script to extract data from the HTML file in Figure 3.

For the paper list page, in Figure 3 (in Section 1), a BODED script in Figure 7 is used to extract papers' titles, authors, and publications of the page. In this script, the element `INIT` indicates that the BODED system initially loads a web page at the URL `http://boded.csie.nctu.edu.tw/de.html` and then uses the service, named `PaperList` (specified in the attribute `service`), to process the page. The service `PaperList` is specified in the `SERVICE` element named `PaperList`.

Intra-Browser

This subsection describes intra-page data extraction of BODED. In the BODED language, XPath is used as the format of query expressions to extract data or locate elements inside the associated page, since XPath is a standard intra-page data extraction language defined by W3C and supported as packages in many systems. Inside the element `SERVICE`, the attribute `xpath` of the elements `VAR` specifies an XPath expression that locates the data.

For example, the service `PaperList` in Figure 7 is used to extract papers' titles, authors, and publications of the paper list page in Figure 3. The `VAR` element named `Title` extracts papers' titles, according to the XPath expression `"//TABLE[1]/TR/TD[1]"` specified in the attribute `xpath`, and then sets the extracted data into a variable named `Title`. More specifically, this XPath query expression follows the XPath standard to extract data in the page as follows: locate the first `TABLE` element and then extract the first `TD` element of all the `TR` elements. The variable `Title` is an array of two `TD` elements locating the two papers' titles in Figure 3. Similarly, the other two `VAR` elements extract both papers' author names and publications into two variables named `Author` and `Publication`, respectively.

Inter-Browser

This subsection describes inter-browser data extraction of BODED. When accessing a new page via a given URL, a service will initiate a browser, as described in Section 2. Hence, the service needs to designate the URL and the corresponding service to serve the new browser.

```
<SERVICE name=AccessNext>
  <EVENT service=Next xpath="//A[1]/@href"
         type=URL />
</SERVICE>
```

Figure 8. The service AccessNext.

Consider the service `AccessNext` in Figure 8 (above), which serves the browser with the page in Figure 3. This service contains an element `EVENT`, specifying an external event. In the `EVENT` element, the attribute `xpath` includes an XPath expression by which the system extracts the `href` attribute of the first `A` element, the link to the next bibliography page. The attribute `type` with the value `URL` indicates to issue a URL event with a URL specified in `xpath`. From above, the element `EVENT` finds the URL link to the next page and then issues a URL event on the newly replicated browser.

Suppose that the reference to the next paper list page is a JavaScript function, instead of a URL. We simply modify the value of the attribute `type` to other external events, such as `ONCLICK`, as shown in Figure 9. In addition to `ONCLICK` and `URL`, there are a set of event types, such as `ONLOAD`, `ONCHANGE`, etc. If the result of the XPath expression is a string, the default value of the attribute `type` is `URL`. Otherwise, the default value is `ONCLICK` for the located element.

```
<SERVICE name="Next">
  <EVENT service="Next" xpath="//A[1]"
         type="ONCLICK" />
</SERVICE>
```

Figure 9. The service Next with an external event.

In order to support multi-way navigation, BODED contains the element FOREACH in services, in which the attribute from is an XPath expression used to locate an array of elements. Let the size of the array be *n*. The FOREACH first replicates the current browser, *B*, to *n* browsers (from the current browser), B_i, where $1 \le i \le n$, as described in Section 2. In B_i, a variable named in the attribute name of the FOREACH is set to the *i*th element in the array. Besides, assign a service S_i to each browser B_i, for all $1 \le i \le n$. Each service S_i runs the script inside the element FOREACH, independently. The original service continues to process the following siblings of the element FOREACH.

Now, consider the service, instantiated from MainCategory, for the browser with the main category page, as shown in Figure 10. The service uses the element FOREACH to replicate browsers and services for each of the links to the subcategory pages, and then issues external events, ONCLICK, on the link of each browser.

```
<SERVICE name="MainCategory">
  <FOREACH name="Link"
           from="//TABLE[1]/TR[1]/TD/A">
    <EVENT service="SubCategory" xpath="/" />
  </FOREACH>
</SERVICE>
```

Figure 10. The MainCategory service.

Now, consider the whole bibliography web site including two-level category pages. The main category page, as shown in Figure 1, includes several subcategory pages, such as databases and algorithms shown in Figure 2, each of which includes a list of articles, as shown in Figure 3. A BODED script of extracting all the article data in the web site is shown in Figure 11.

```
<BODED>
  <INIT url="http://BODED.csie.nctu.edu.tw/bib/"
        service="MainCategory" />
  <SERVICE name="MainCategory">
    . . . <!-- See Figure 10 in detail -->
  </SERVICE>
  <SERVICE name="SubCategory">
    <FOREACH name="Link"
             from="//TABLE[1]/TR[1]/TD/A">
      <EVENT service="PaperList"
             xpath="/" />
    </FOREACH>
  </SERVICE>
  <SERVICE name="PaperList">
    . . . <!-- See Figure 7 in detail -->
  </SERVICE>
</BODED>
```

Figure 11. A BODED script of extracting all the article data.

4. Discussions and Conclusion

Traditionally, most researchers used the URL-oriented data extraction model for data extraction. In this model, their systems extract URLs from pages and then use the extracted URLs to access next pages for data extraction. However, more and more pages currently use script functions to access next pages. Since it is hard to extract URLs from script programs, it is inappropriate to use this model for such pages.

For solving this problem, this paper proposed a new data extraction model, named the browser-oriented data extraction model. In this model, the system built on top of browsers accesses pages by simulating users' operations on browsers, which can also trigger script functions.

Besides, this paper defines a scripting language, named the BODED (Browser-Oriented Data Extraction Description) Language, which instructs the system to do data extraction.

Acknowledgements

The authors would like to thank the National Science Council of the Republic of China for financially supporting this research under contract No. NSC 91-2213-E-009-114 and NSC 92-2213-E-009-116.

References

[1] G. Arocena and A. Mendelzon. "WebOQL: Restructuring Documents, Databases, and the Web", In *Proceedings of ICDE*, Orlando, Florida, 1998.

[2] R. Baumgartner, S. Flesca, G. Gottlob. "Visual Web Information Extraction with Lixto", In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, 2001.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, "XQuery 1.0: An XML Query Language", W3C Working Draft, W3C Consortium, July 2004.

[4] D. Konopnicki and O. Shmueli. "Information gathering in the World-Wide Web: the W3QL query language and the W3QS system", *ACM Transactions on Database Systems (TODS)*, Volume 23 Issue 4, Dec. 1998.

[5] P. Merrick and C. Allen. "Web Interface Definition Language", W3C Note, W3C Consortium, Sep. 1997.

[6] J. Robie, J. Lapp, D. Schach. "XQL: XML Query Language", *Workshop on XML Query Languages*, Dec. 1998.