Semi-Automatic Wrapper Generation for Commercial Web Sources

Alberto Pan[1], Juan Raposo[2], Manuel Álvarez[2], Justo Hidalgo[1] and Ángel Viña[2]
[1]Denodo Technologies. [2]Communications and Information Technology Department-University of A Coruña

Abstract: Semi-automatic wrapper generation tools aim to ease the task of building structured views over semi-structured web sources. But the wrapper generation techniques presented up to date are unable to properly deal with sources requiring complex navigational sequences for accessing data. In this paper, we present Wargo, a semi-automatic wrapper generation tool, which has been used by non-programmer staff to successfully wrap more than 700 commercial web sources in several industrial applications. We describe our approach for wrapper generation and show the difficulties found with other systems for wrapping this kind of sources.

Key words: wrapper generation, mediator systems

1. INTRODUCTION

WWW data is available only in a human-readable form In order to let software programs gain full benefit from semi-structured web sources, wrapper programs must be built to provide a structured view over them. This task is often known in industry as “screen-scraping”.

Web scraping typically entails two main tasks:

– Access to pages containing the required data, and
– Obtain a structured view over the information contained in the retrieved HTML pages

Most of the research works in wrapper generation have been focused only on the second problem, and have assumed the first one to be trivially solved by using simple HTTP clients.

Nevertheless, in the modern commercial web environment this is far from truth. Today's commercial web sources use Javascript, dynamic HTML, advanced session maintenance mechanisms based on session identifiers, etc. Citing a recent industrial paper [9], "commercial screen-scraping is not merely intelligent parsing. It also includes the intricacies of navigating Javascript pages, dealing with cookies and passwords, and interfacing with HTTPS-protected sites"

In turn, the problem of parsing the required information has received much more attention from the research community. The approaches presented until now can be roughly classified into three groups:

– Wrapper programming languages. They provide specialized languages for building parsers for the desired web pages.
– Inductive learning techniques. They learn about web pages structure from labelled examples.
– Supervised interactive tools. This approach relies on interactive graphical wizards to guide the user in the wrapper generation process.

In this paper we present WARGO, a system which lets non-technical users generate complete wrappers for web sources, even today's complex commercial ones.

To specify how to access the pages containing the required data, users can generate complex web flows by simply navigating with a web browser, without needing to worry about Javascript, Dynamic HTML or session maintenance mechanisms. For parsing the required data, we provide users with a supervised interactive tool (integrated into their web browsers) in order to enable them to generate complex extraction patterns by simply highlighting relevant data from very few example pages, and answering some simple questions. The system relies on two wrapper programming languages: Navigation SEQuence Language (NSEQL) for specifying navigation sequences and Data EXTraction Language (DEXTL) to specify extraction patterns. Nevertheless, they are hidden from the users, who only need to use simple graphical wizards.

The paper is structured as follows. In the next section, we describe the main modules of WARGO. Section 3 and 4 describe, respectively, the languages used for specifying navigation sequences (NSEQL) and extraction patterns

(DEXTL). The basics of the interactive process needed to generate a wrapper using the Wargo graphical tool are shown in Section 5. Section 6 provides some experimental data collected during the industrial use of the system. Finally, Section 7 discusses related work.

2. SYSTEM OVERVIEW

2.1 Architecture at Runtime

Figure 1 shows WARGO Architecture at runtime. An explanation of its main modules follows.

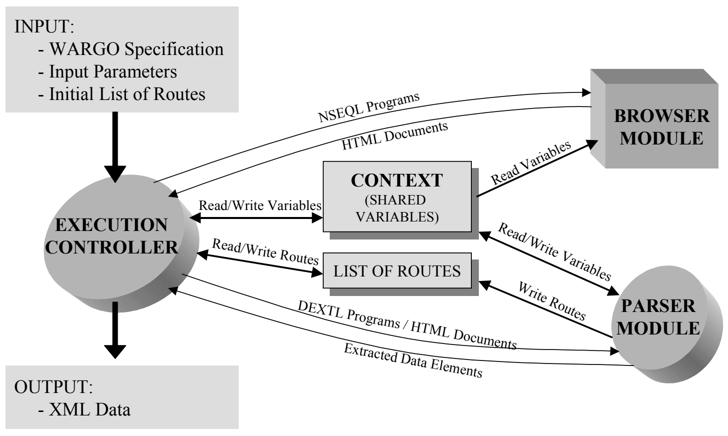


Figure 1. Wargo Architecture at runtime

Wargo receives as inputs a wrapper specification, a list of input parameters and a list of routes to documents. As its output it will return a XML document containing the required data from the set of documents.

The input parameters are a list of name/value pairs. For instance, a wrapper for retrieving new messages from an on-line webmail service, could accept two input parameters login and password, thus allowing the invoking application to decide at execution time which user account should messages be retrieved from.

The list of routes contains a set of navigation sequences pointing to documents which must be explored by the wrapper to extract the required data.

The execution controller module rules the execution process. It interprets the Wargo specification, it uses the browser module to perform NSEQL navigation sequences, and it uses the parser module to execute DEXTL programs, thus extracting the required data from the retrieved documents.

In our current implementation, the browser module is based on the Microsoft Internet Explorer WebBrowser Control component. The parser module uses JFlex-generated scanners for lexical analysis, and a Wargo-proprietary non deterministic parser for syntactical analysis.

A shared space called context is used to store variables whose values can be accessed at runtime by NSEQL and DEXTL programs. Initially the context is loaded with a variable for each input parameter provided by the invoking application. DEXTL programs can also load new variables as the data extraction process proceeds.

2.2 Architecture at Wrapper-Generation Time

Figure 2 shows Wargo architecture at wrapper-generation time.

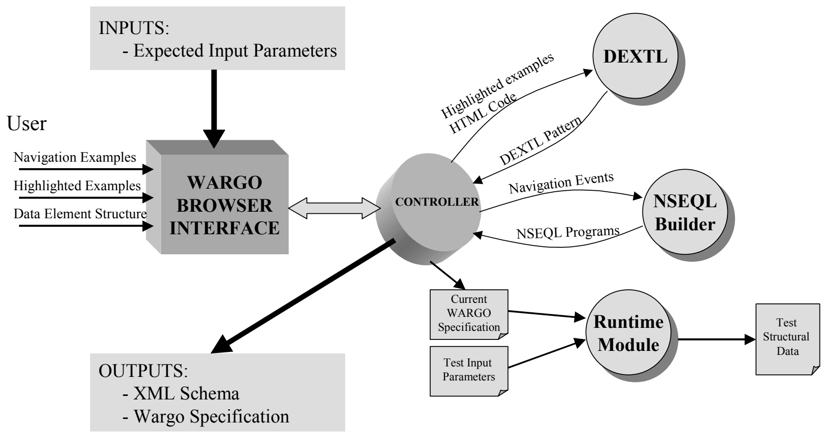


Figure 2. Wargo Architecture at wrapper-generation time

The input for the process is the name of the expected input parameters for the wrapper at runtime. As outputs, the process obtains a complete wrapper specification and the XML schema from the data to be extracted by the wrapper.

Users interact with the system through the Wargo browser. The tool guides them through the whole specification generation process.

As it was already said, specifications combine NSEQL navigation sequences and DEXTL extraction programs. The NSEQL Builder module is responsible of managing the process of NSEQL programs generation from user-provided navigation-examples.

The DEXTL builder has the analogous responsibility in DEXTL programs generation. In this case, the wrapper-generator provides input by highlighting in the Wargo browser instances of the required information.

Finally, the Runtime Module (which architecture was shown in Figure 1) is used at wrapper-generation time for specification-testing purposes.

3. NSEQL BASICS

NSEQL is a language to declaratively define sequences of events on the interface provided by a web browser.

Working at browser layer instead of at HTTP layer, lets us forget about problems such as:

– Many sources use non-standard session maintenance mechanisms. These mechanisms rely on generating session identifiers which are passed between pages through hidden HTML form fields. In these situations, to perform a navigation sequence by using HTTP requests, forces wrapper generator to write ad-hoc code for each source to extract the session identifier from the HTML code and include it in the subsequent HTTP requests.

– Many sources use Javascript for a variety of purposes, including dynamically building HTTP requests, managing dynamic HTML layers, etc. Using a Javascript interpreter does not solve the problem, since Javascript programs often assume a set of browser-provided objects to be available. So, very often, ad-hoc code must be written to replicate Javascript functionality.

These problems are accentuated by issues such as frames, dynamic HTML or HTTPS. Nevertheless, all these troubles are transparent to the wrapper generator when working at browser-interface layer.

We have defined NSEQL, a language for defining navigation "macros" using MSIE WebBrowser Control. Consider the following example to get a first flavor of it.

Example 1: The following NSEQL program is able of executing the login process at YahooMail and navigating to the list of messages from the Inbox folder.

```
Navigate("http://mail.yahoo.com");
FindFormByName("login_form",0);
SetInputValue("login",0,@LOGIN);
SetInputValue("passwd",0,@PASSWORD);
ClickOnElement(.save,Input,0);
ClickOnAnchorByText ("Go to
Inbox",0,false);
```

Figure 3. NSEQL Program

The Navigate command makes the browser navigate to the given URL. Its effects are equivalent to that of a human user typing the URL on his/her browser address bar.

The 'FindFormByName(name, position)' command looks for the position-th HTML form in the page with the given name attribute value. Then, the 'SetInputValue(fieldName, position, value)' commands are used to assign values to the form fields. The value parameter might be a string (closed by "") or a variable included in the context (see Figure 1) prefixed by the '@'character. As it was formerly said, variables are replaced by its value at execution time.

The 'clickOnElement(name,type,position)' command, clicks on the position-th element of the given type and name from the current selected form. In this case, it is used to submit the form and load the result page. The 'ClickOnAnchorByText (text, position)' command looks for the position-th anchor which encloses the given text and generates a browser click event on it. This will cause the browser to navigate to the page pointed by the anchor.

NSEQL also includes commands to deal with frames, pop-up windows, etc. The main NSEQL commands are shown in Figure 4.

Command	Description
Navigate(url,headers,target)	Navigates to an "url" adding the specified "headers" (optional) to the request and loads it on the "target" (optional) frame.
PostData(url,data,headers,target)	Posts "data" to "url" adding the specified "headers" (optional) to the request and loads the result on the "target" (optional) frame.
FindFrameByName (name,position,equals)	Searches for the "n"th frame with the specified "name" (equals=true) or which name contains the specified "name" (equals=false).
GetSourceCode()	Obtains the source code of the current selected frame
ClickOnAnchorByPosition(n)	Clicks on the "n"th anchor in the document.
ClickOnAnchorByText(text,n,equals)	Clicks on the "n"th anchor with the specified "text" (equals=true) or containing the specified "text" (equals=false).
FindFormByName(name,n)	Searches for the "n"th form with the specified name
SetInputValue(name,n,value)	Establishes the "value" of the "n"th input field with the specified "name" on last selected form
SelectIndexByText(name,n,text,n2,equals)	Selects the "n2"th opction with the specified "text" (equals=true) or containig it (equals=false) in the "n"th select field with the specified "name" in the last selected form
clickOnElement(name,type,n)	Clicks on the "n"th element of the selected form, with the "name" (optional) and the "type" specified.
FireEvent(eventName,type,name,n)	Fires the event "eventName" over the "n"th field with the specified "type" and "name" (optional) in the last selected form
SubmitForm()	Submits the last selected form.
FindElementByText(type,text,n,equals)	Searchs for the "n"th element with the specified "type" and "text" (equals=true) or containing the specified "text" (equals=false)
FindElementByAttribute(type,name,value,n,equals)	Searchs for the "n"th element of the specified "type" containing an attribute with the specified "name" and "value" (equals=true) or containing the "value" (equals=false)
ClickOnSelectedElement()	Clicks on last selected element
FireEventOnSelectedElement(eventName)	Fires the "eventName" event over the last selected element
OpenNewWindow(url,name,features)	Opens a child window wtih the specified "name" and "features" and loads the specified "url" on it.

Figure 4. Main NSEQL Commands

4. DEXTL BASICS

A DEXTL program is composed of hierarchically structured elements, much in the way in which XML elements are structured. The elements in a DEXTL program correspond with the information items we wish to extract from a given document. For instance, if we are building a program for parsing a list of e-mail messages from a Webmail source, we will probably have DEXTL elements for information items such as messages, subjects, senders, etc.

Each element can be either atomic (those without sub-elements) or non-atomic (those with sub-elements).

Each sub-element from a non-atomic element has associated one of the following cardinalities: 1 (a sub-element for each parent element), 0-1 (zero or one sub-element for each parent element), + (one or many sub-elements for each parent element) and * (zero or many sub-elements for each element).

Atomic elements can be typed. In DEXTL, a data-type is represented through a function which receives an string as input parameter, and returns true if the string is a valid instance of the data type, and false otherwise. By default atomic elements are considered as strings (a function which always returns true). Other available data-types are integer, float, date, money, etc. New data-types can be added if required.

Each non-atomic element has the following parts: Name (which will be used to generate the XML output), From clause, Until clause, and Extraction clause, where Name and Extraction clause are mandatory.

The FROM clause identifies the beginning of the document region where element instances will be searched for and the UNTIL clause identifies the end of it. This region is called the element search-region.

If no FROM clause is specified for a given element, the FROM clause from its parent is assumed. The same applies with UNTIL sections.

If an element has no parent and no FROM clause, the beginning of the document is considered as search-region start delimiter. In the same way, if an element has no parent and no UNTIL clause, the end of the document is assumed as search-region end delimiter.

The EXTRACTION clause describes the element sub-elements and how they are laid in the search region, thus allowing to identify and to extract the element data instances in the document.

The basic structure used to specify FROM-UNTIL and EXTRACTION clauses is called a DEXTL pattern. These patterns are the “core” of DEXTL programs and will be described in detail in the next sub-section. By now, we will only say that a pattern defines a sequence of data elements and separators among them.

The EXTRACTION clause will contain one DEXTL pattern for each possible visual layout of the element inside the search region. At runtime, a new data instance for the element will be created for each occurrence inside the search-region of a pattern from the element EXTRACTION clause.

DEXTL patterns are also used to construct FROM and UNTIL clauses. When the first occurrence of a pattern included inside the FROM clause is found, Wargo will start to search for EXTRACTION clause patterns occurrences (which correspond with the actual data to extract), till the first occurrence of a pattern included inside the UNTIL clause is found.

4.1 DEXTL Patterns

A DEXTL pattern is comprised of a list of text tokens which are laid consecutively in the document and which are delimited by separators.

Text tokens represent text in the browser-displayed page. They are enclosed between '[' and ']' and they can be divided into portions by applying Perl5-like regular expressions.

A name (prefixed by the '$' character) can be assigned to the parts of the regular expression enclosed between '(' and ')'. The name may correspond either with an atomic data element or with a special value called IRRELEVANT, which is used to represent non-constant strings appearing in the pattern which we do not wish to extract.

A datatype can be added to a portion to indicate that it must verify the additional restrictions imposed by the function associated to the specified datatype. This is indicated by adding a ':' character after the name, followed by the desired datatype.

It is also possible to have a portion divided into substrings using character offset positions. For instance, the expression '(0-5:$NAME1 6-11:$NAME2)' will cause the text portion to be divided in three parts, having the first six characters assigned to NAME1 and the following six characters assigned to NAME2.

Separators represent a regular expression concerning HTML document tags. For instance, we could define a separator called EOL (EndOfLine) as follows:

```
EOL =("<br>"| "</p>"| "</tr>" | ”</td>” ([ \n\r\t])* </tr>”)
```

Though they can be defined to suit, DEXTL includes built-in separators which are enough for most situations (in fact, we still have not needed to create new separators, though having successfully wrapping more than 700 sources). They are typically used to traduce from HTML tags to more general layout primitives that can be used in HTML documents to visually separate data elements. Each DEXTL pattern has a set of associated separators, which are indicated through the SEPARATORS construction. Figure 5 shows some of the main built-in separators in DEXTL.

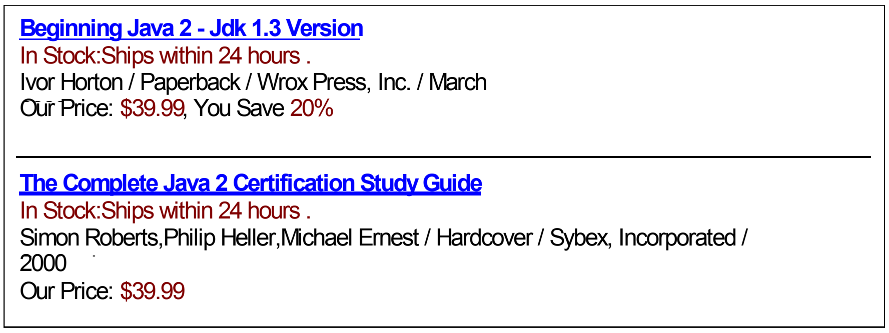
Separator	Definition	Description
EOL	"</TD>" [\n\r\t]* "</TR>" | "<BR>" | "<LI>" | "<P"~">" | "</TR>" | "</OPTION>" | "<DD>" | "<DT>" | "<DL>" | "</UL>" | "</H1>" | "</H2>" | "</H3>" | "</H4>" | "</H5>" | "</H6>" | "</TH>"	End of line
TAB	"</TD>"	Tabulator
ANCHOR	"<A" [^>]* ">"	HTML anchor
ENDANCHOR	</A>	End of HTML anchor
RADIO	"<INPUT" [^>]* TYPE=" [\"]? "RADIO" [^>]* ">"	Radio input
OPTION	"<OPTION [^>]* ">"	Select option
ENDOPTION	</OPTION>	End of select option
CHECKBOX	"<INPUT" [^>]* "TYPE=" [\"]? "CHECKBOX" [^>]* ">"	Checkbox

Figure 5. Main buil-in HTML separators

A DEXTL pattern can also include actions. Actions are arbitrary functions receiving a list of input parameters (which can be constant strings, context variables or even functions) performing an action which can modify the current context and/or the current list of routes. Each action associated with a pattern will be executed every time an instance of such pattern is found. As we will see in section 5.2.2, actions can be used to combine navigation and data extraction.

Now, let us consider the following example to show DEXTL patterns in action.

Example 2: Figure 6 shows two search results from B&N electronic shop. Note that there are some differences between them. More precisely, the first result includes discount information while the other does not.



Beginning Java 2 - Jdk 1.3 Version
In Stock:Ships within 24 hours .
Ivor Horton / Paperback / Wrox Press, Inc. / March
Our Price: $39.99, You Save 20%

The Complete Java 2 Certification Study Guide
In Stock:Ships within 24 hours .
Simon Roberts,Philip Heller,Michael Ernest / Hardcover / Sybex, Incorporated / 2000
Our Price: $39.99

Figure 6. Two search results from Barnes & Noble

Figure 7 shows two DEXTL patterns to extract occurrences of an element P, with atomic sub-elements TITLE, AUTHORS, FORMAT, DISCOUNT and PRICE.

```
PATTERN
{
  SEPARATORS {EOL}
  [($TITLE)] EOL
  [($IRRELEVANT)] EOL
  [($AUTHORS) "/" ($FORMAT) "/"($IRRELEVANT)]
EOL
  ["Our Price:" ($PRICE:Money) ",You save:"
($DISCOUNT)] EOL
}

PATTERN
{
  SEPARATORS {EOL}
  [($TITLE)] EOL
  [($IRRELEVANT)] EOL
  [($AUTHORS) "/" ($FORMAT) "/"($IRRELEVANT)]
EOL
  ["Our Price:" ($PRICE:Money)] EOL
}
```

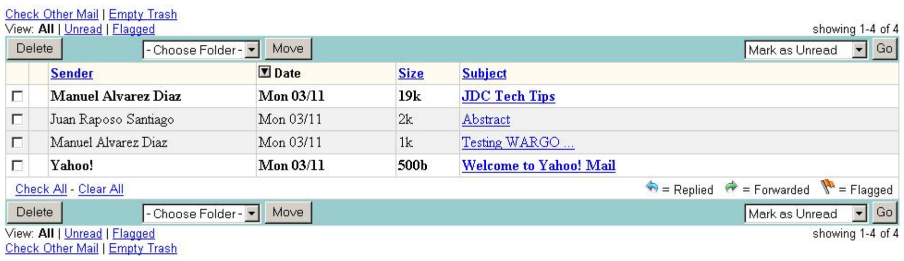
Figure 7. B&N DEXTL Patterns

The only separator used in both patterns is EOL. This is the only one actually needed in this case to separate the required data elements. PRICE is specified to be of money datatype.

Now, we have shown how DEXTL patterns are built, we are ready to show a complete DEXTL example in the net section.

4.2 A Complete DEXTL Example

Figure 9 shows a complete DEXTL program to extract a list of message headers from a YahooMail folder (see Figure 8).


Check Other Mail | Empty Trash
View: All | Unread | Flagged showing 1-4 of 4

	Sender	Date	Size	Subject
☐	Manuel Alvarez Diaz	Mon 03/11	19k	JDC Tech Tips
☐	Juan Raposo Santiago	Mon 03/11	2k	Abstract
☐	Manuel Alvarez Diaz	Mon 03/11	1k	Testing WARGO ...
☐	Yahoo!	Mon 03/11	500b	Welcome to Yahoo! Mail

Check All - Clear All = Replied = Forwarded = Flagged
View: All | Unread | Flagged showing 1-4 of 4
Check Other Mail | Empty Trash

Figure 8. YahooMail folder

```
MESSAGES{
  FROM {
    PATTERN {
      SEPARATORS {TAB,EOL}
      ["Sender"] TAB ["Date"] TAB ["Size"]
TAB ["Subject"] EOL
    }
  }

  EXTRACTION {
    *:MESSAGE {
      EXTRACTION {
        PATTERN {
          SEPARATORS {TAB, EOL}
          [($SENDER)] TAB
[($DATE:Timestamp)] TAB [($SIZE)]
TAB [($SUBJECT)] EOL
        }
      }
    }
  }
}
```

Figure 9. A Complete DEXTL Program

The specification defines an element named Messages. The FROM clause of the region contains a single pattern. No UNTIL clause is specified, so the end of the document will be assumed.

The Extraction region identifies one non-atomic sub-element whose name is Message, and which cardinality is '*'. Each Message element is composed from three atomic sub-elements: Sender, Subject and Date. Sender and Subject data-type is string (which is the default data-type and, thus, need not be specified). Date element has associated the Timestamp data-type, which accepts several formats of date and time information.

Though we have used here a more concise representation, DEXTL programs can also be written using XML syntax. In the extended version of this paper[11], the complete DTD for DEXTL is provided.

5. INTERACTIVE WRAPPER GENERATION

Though NSEQL and DEXTL can be used directly by wrapper creators, we have created tools for interactively generating specifications, thus improving ease of use. In fact, the wrapper creators using WARGO need not to know anything about the languages.

We begin by showing how DEXTL programs are generated by means of examples. Then, we will show the equivalent for NSEQL programs.

5.1 Generating DEXTL Programs

The first step to generate DEXTL programs is identifying elements schema, which involves specifying element names, datatypes, cardinalities and nesting structure. This is made in Wargo by using a hierarchical Windows-Explorer like graphical interface. This way, a tree data structure which can be directly mapped to an XML schema is built.

Once the basic elements structure is created, the DEXTL patterns for the FROM, UNTIL and EXTRACTION clauses of the non-atomic elements must be created.

Wargo requires the FROM and UNTIL clauses of an element to be created before the EXTRACTION clause. Wargo also requires parent elements clauses to be specified before its sub-elements ones. This way, when an DEXTL pattern is created, Wargo can know what is the region where pattern occurrences must be searched.

Having this in mind, an DEXTL pattern can be created by the user as follows:

1. User selects in the tree the element and clause the pattern will apply to.
2. User chooses the menu option 'Patterns-New Pattern'.
3. User selects the separators which should be used in this pattern. The tool shows each separator along with its HTML graphical representation.
4. User highlights an occurrence of the pattern directly in the browser page.
5. Wargo automatically obtains the HTML code associated to the highlighted region and divides it into tokens, thus obtaining an DEXTL pattern composed by separators and text tokens.

6. Then Wargo launches a new pop-up window showing the matched text tokens in the highlighted pattern. For each text token, a 'Split' button and an 'Assign' button are shown.
7. The 'Split' button let users access a new window where text token can be divided into several ones by iteratively applying two simple functions: one divides the text using a string as delimiter and the other uses a position offset for that purpose. When the user clicks the 'Accept' button, Wargo comes back to the previous window showing the new list of text tokens after the splitting process.
8. The 'Assign' button lets user assign a text token to an atomic sub-element. It the selected text is not a valid instance of the sub-element data-type, a warning window is launched.
9. Once the user has split the tokens and has made the correspondences between text portions and atomic sub-elements, Wargo searches for other occurrences of the pattern inside the relevant region (see section 4), and automatically highlights them.
10. If any of the required element instances have not been highlighted by Wargo, the user can add a new pattern by highlighting one of the missing instances, thus making Wargo match the element instances following that format. The process ends when all the required instances are matched.

The last step involved in generating DEXTL programs is adding actions to patterns. The Wargo tool interface includes several commonly used built-in actions, though new ones can be added in a modular fashion. The most important ones are $GOROUTE and $ADDROUTE, which can be used to dynamically add new routes at runtime (see section 5.2.1). Actions can be inserted by selecting a non-atomic element from the tree, clicking mouse right button, and choosing the 'New Action' menu option. The user will then choose the action to be added. Each action has an associated wizard to let user provide values for its parameters.

5.2 Generating NSEQL Programs

A Wargo user can create new navigation sequences by choosing the 'Navigation-New Navigation Sequence' menu option.

Then a new browser window is launched where user-produced browser events are recorded. The main events Wargo monitors are:

– Navigation events. Occur when the user introduces an URL in the browser bar. It translates to a navigate NSEQL command.
– Click events over anchor elements. An event of this kind translates to a clickOnAnchorByText command (note that if several anchors share the same text, the position parameter is used to disambiguate).

– Submit events over form elements. These events translate to a sequence of the form:
  1. A findFormByName command identifying the submitted form.
  2. A command for each not-hidden filled field of the submitted form, setting the field to the value used in the form submission. The exact command used depends on the field type. For input elements, setInputValue commands will be used. selectByIndex commands for select elements, etc.
  3. A clickOnInputElement command for clicking in the form submit button.
– If the anchor clicked or the form submitted by the user is inside a frame, a findFrame command is included before the corresponding sequence for each case.

Besides of recording events, the Wargo tool launches a pop-up window whenever a form is submitted. The window lets user assign variable values from the context to the form text-fields. This is needed because form fields often need to be filled with data obtained at runtime. For instance, in the YahooMail example, the exact login/password pair is not known until execution-time (when they are provided through input parameters).

5.2.1 Dynamically added routes

It often occurs that routes are not known until run-time. For instance, if we wished to retrieve the first 100 bytes from each message in a YahooMail account, we would need to cross the anchor on the message subject. Nevertheless, there is no way for the wrapper creator to know in advance neither how many messages the account will contain, nor what will be the exact NSEQL commands required.

Other example arises when web sources shows a list of items divided into several pages connected through 'Next' and 'Previous' links: since the number of items is not known a priori, there is no way to know how many 'Next' links will have to be followed to extract all the items.

Wargo addresses this kind of problems combining DEXTL actions and context variables.

There are two pre-defined DEXTL actions used to dynamically manage the list of routes:

– ADDROUTE(Sequence,ElementName). It causes the execution of the new route at the end of processing the current document.

– GOROUTE(Sequence,ElementName). It causes the immediate execution of the received route. When the route is finished, the process will be continued at the same point where it was interrupted.

These routes are inserted in an specification by selecting an Extraction clause pattern P from a non-atomic element, clicking mouse right button, choosing the 'New Action' menu option, and selecting either the $GOROUTE or the $ADDROUTE actions.

Then, a wizard let user performs an example of the desired navigation sequence in a similar way to that shown in section 6.2. The main difference is that if the sequence includes generating events on HTML elements included in an instance of the pattern, then Wargo will consider that event as an example which, at runtime, must be repeated on the analogous element from each matched pattern instance.

See the extended version of this paper [11] for a more detailed explanation of this process.

6. EXPERIENCE USING WARGO

WARGO has been used to build wrappers for more than 700 commercial web sources, including e-shops, auction sites, online banks, employment sites, webmail providers, online newspapers, online travel agencies, Internet portals, government bulletins, etc.

These wrappers are being used in industrial applications which are currently online in several major Internet portals and financial sites. The maintenance work of these sources is entirely made by non-programmer staff.

Figure 10 shows number of wrapped sources, average wrapper creation times, average number of navigation sequences and average number of DEXTL patterns required for the wrappers created in the following application domains: financial aggregation, comparison shopping, e-mail aggregation and news search. Average times for NSEQL programs and for DEXTL programs are also shown.

Domain	Wrapped sources	Creation time average (minutes)	Navigation sequences average	DEXTL elements average
Financial Aggregation	37	211.2	9	8.1
Comparison Shopping	318	14.4	2.8	2.4
E-mail Aggregation	12	26.3	3.1	2.6
News Search	33	20.1	2	1.8

Figure 10. Wrapped Sources

7. RELATED WORK

The wrapper generation techniques presented so far, have been mainly focused in the parsing problem.

Nevertheless, as we have already discussed, the problem of creating navigation sequences to access the documents is at least as important.

Many systems, such as [8][7] did not consider the navigation problem in any way. Several others, such as [5][2], provided simple http-level based functions to build navigation sequences and dynamically access new documents during the extraction process. Since all these approaches relied upon a http-level of abstraction, a number of very common complexities in the commercial websites of today were not supported or they required from complex custom code for each source:

– Non-standard session maintenance mechanisms.
– Javascript code.
– HTTPS
– Dynamic HTML

Wargo works at a higher abstraction level, thus getting independence from these low-level complexities. NSEQL allows Wargo wrapper-generators to specify navigation sequences at a browser interface-level.

The approaches presented until now to deal with the parsing problem can be roughly classified into three groups:

– Wrapper programming languages.
– Inductive learning techniques.
– Supervised interactive tools.

The first approach consists in providing some sort of special language which eases the construction of parsers for the desired web pages.

Some of the proposed systems following this approach are the Tsimmis language for web wrapper generation [2], WebL [5] Araneus [1] or Jedi [4]. All of them are expressive but complex, and thus, its use is reserved to experts, even when dealing with simple sources.

Inductive learning techniques do not require technically-skilled staff to generate wrappers, since the only task needed is to label example pages. Some remarkable systems are WIEN [7], SoftMealy [3], and Ariadne [6].

The main drawback from inductive-learning techniques is its limited expressive power. For instance, many of these systems can deal only with pages following the so-called HLRT structure (head-left-right-tail), where there is only one extraction area in the document prefixed by a head, suffixed by a tail, and where each item has a left and a right separator.

Inductive learning techniques also require of a big number of examples even when wrapping simple sources, thus making the process lengthy and tedious for industrial users. Nevertheless, inductive learning techniques are the most promising ones for wrapper auto-repairing techniques in presence of source changes. That is why we plan to introduce some learning capabilities into Wargo as part of our future work.

Interactive supervised graphic tools often rely also on some kind of wrapper programming language, but they try to hide its complexities under easy-to-use graphical wizards and interactive processes.

Some systems following this approach are Xwrap[8] and Lixto [10]. Xwrap makes an intensive use of heuristics which are not satisfied by many sources. It limits its expressiveness. Besides, the Xwrap html-tree based interface is less intuitive than the Wargo approach, which uses the document displayed "as is" in the browser instead of the HTML tree.

Lixto also uses the idea of highlighting pattern occurrences in the browser page in order to generate extraction programs (they use a Datalog-like language called Elog). Nevertheless, pattern creation is more tedious than in Wargo, since user must create patterns even for the atomic elements. Another advantage of Wargo is its already-mentioned integration with browser-interface level navigation sequences.

(see [11] for further related work discussion).

8. REFERENCES

[1] P. Atzeni, G. Mecca. "Cut and Paste". Proceedings of the PODS Conference. 1997

[2] J. Hammer, H. García-Molina, J. Cho, R. Aranha, A. Crespo. "Extracting semi-structured data from the web". Proceedings of Workshop on Management of Semi-structured data. 1997

[3] C-H. Hsu, M-T Dung. "Generating Finite-state Transducers for semi-structured Data Extraction from the Web". Proceedings of the fifteenth National Conference on Artificial Intelligence (AAAI). 1998

[4] G. Huck, P. Fankhauser, K. Aberer and E.J. Neuhold. "JEDI: Extracting and Synthesizing information from the web". Proceedings of the CoopIS Conference. 1998

[5] T. Kistlera, H. Marais. "WebL: A Programming Language for the Web" http://www.research.digital.com/SRC/WebL/index.html. 1998

[6] C.A. Knoblock, K. Lerman, S. Minton and I. Muslea. "Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach". Bulletin of the IEEE Computer Society Technical Committee on Data Enginnering. 1999

[7] N. Kushmerick, D.S. Weld, R. Doorembos. "Wrapper induction for information extraction". Proccedings of the fifteenth International Joint Conference on Artificial Intelligence (IJCAI). 1997

[8] Ling Liu, Carlton Pu and Wei Han. "XWRAP: An XML-enabled wrapper construction system for web information sources". Proceedings of the 16th International Conference on Data Engineering. 2000

[9] M. Stonebraker, J. Hellerstein. "Content Integration for E-Business". Proceedings of the ACM SIGMOD Conference. 2001

[10] R.Baumgartner, S.Flesca and G.Gottlob. "Visual Web Information Extraction with Lixto". Proceedings of the VLDB Conference. 2001

[11] A.Pan, J.Raposo, M.Álvarez, J.Hidalgo, A. Viña. "The Wargo System: Semi automatic wrapper generation in presence of complex data access modes (extended version)". http://www.tic.udc.es/~gris/publications/ExtendedEISIC.pdf