# The Caméléon Web Wrapper Engine

**Aykut Firat, Stuart Madnick, Michael Siegel**

Contact author: Stuart Madnick
MIT Sloan School of Management, Cambridge, MA 02139, USA
Email: {aykut, smadnick, msiegel}@mit.edu; phone: 617/253-2656

**Abstract**

The web is rapidly becoming the universal repository of information. A major challenge is the ability to support the effective flow of information among the sources and services on the web and their interconnection with legacy systems that were designed to operate with traditional relational databases. This paper describes a technology and infrastructure to address these needs, based on the design of a web wrapper engine called Caméléon. Caméléon extracts data from web pages using declarative specification files that define extraction rules. Caméléon is based on the relational model and designed to work as a relational front-end to web sources. ODBC drivers can be used to send SQL queries to Caméléon. Query results by Caméléon are presented in either XML or HTML table formats. Users can also easily call Caméléon from other applications (e.g. Microsoft Excel by using Caméléon web query file (Caméléon.iqy)). Unlike its predecessor, Grenouille, Caméléon lets users segment web pages and define independent extraction patterns for each attribute. The HTTPClient package used in Caméléon supports both GET and POST methods and is able to deal with authentication, redirection, and cookie issues when connecting to web pages.

## Introduction

More and more information sources and services are becoming available on the Internet, and the benefits of achieving interoperability among these sources and services are compounding. The emergence of XML in the recent years promises to facilitate interoperability by delivering structured content over the web, yet it does not solve the interoperability problem completely. At least there are and will be a huge number of legacy HTML applications, which will need wrapping. Web wrappers are thus essential in interoperability efforts and used in selectively extracting structured content from semi-structured web sources. A review of projects (see Appendix I) in this area reveal that several web wrappers have been developed by various commercial and research institutions to deliver structured content over the web. These wrappers either support rich query languages such as SQL or OQL ([Roth & Schwarz 97], see [Abiteboul 97] for more examples) to query the web sources, or emphasize conversions from HTML to XML (e.g. XWRAP[Liu et.al 99], W4F[Sahuguent & Azavant 99]). In this paper we introduce a powerful general-purpose wrapper engine, called Caméléon after its predecessor Grenouille [Bressan & Bonnet 97]. Both Caméléon and Grenouille wrapper engines are based on the relational model and designed to work as relational front-ends to web sources. They treat the web as a giant relational database and support SQL for querying it. The engines differ in the structure of their specification files (spec-files), which include schema information, and extraction rules.

Caméléon makes spec-file creation easier by providing more flexibility in the spec file structure: users can narrow the scope of pattern matching, and define independent patterns for each attribute in the schema. This flexibility is expected to be essential in on-going efforts of automating the spec-file generation.

When connecting to web pages, Caméléon uses the HTTPClient [Tschalär 99] package, which deals with connection (get & post), authentication, redirection, and cookie issues. The package was also made capable of extracting information from sites that use SSL(Secure Socket Layer). These features allow us to connect virtually to any site on the net and extract the desired information.

Among the advanced features of Caméléon are its ability to traverse multiple pages to locate a desired page and to supply input and extracted data to the subsequent patterns.

Currently Caméléon runs as a Java Servlet and its output can be presented in either XML or HTML table formats. Users can send SQL queries to Caméléon through http or ODBC drivers. They can also call Caméléon directly from other applications (e.g. Microsoft Excel by obtaining the Caméléon web query file (Caméléon.iqy)).

In the next section we first briefly overview the past efforts of web wrapping in our group, which focuses on achieving interoperability among heterogeneous data sources through context mediation. Then we analyze where Caméléon stands within the framework of wrapper engines. Next we explain the Caméléon Architecture and talk about how the new engine works. Then we describe the new spec-file structure. After mentioning some of our example applications, we finally present our future plans.

## Overview of Wrapper Efforts in the COIN Group

The wrapper efforts in our COntext INterchange (COIN) group date back to 1995 and earlier. [Qu 96] developed the first wrapper engine in Perl, called Generic Screen Scraper, using regular expressions and finite state automata. In this system each web page was regarded as a state, and transitions determined which page to go next and the input required to go that page. The specification file was essentially a directed acyclic graph. An SQL query was interpreted as traversing a path beginning at the start state of a data source. Her system used regular expressions to extract intermediate values to be supplied to the following states as well as the requested data in the final state. Regular expressions, in Qu's system, were defined independently for each attribute.

Later [Jakóbisiak 96] pursued the same approach and implemented a wrapper generator with minor changes. She also constructed a multidatabase browser to provide a single query interface to heterogeneous sources.

Then came Grenouille [Bressan & Bonnet 97] with a slightly different approach. In order to be consistent with the relational model, Bressan defined regular expressions that would extract a tuple at a time, not a single attribute value. This solved the problem of tuple identification, but made regular expression generation harder for some cases. His system allowed multiple page transitions and input replacement in patterns. Later [Ambrose 98] converted Grenouille from Perl to Java keeping the same design.

Finally we have developed Caméléon in Java based on a new design, which is explained in detail in this paper.

## Structure

The analysis of literature shows that there are two fundamentally different approaches taken in the design of web wrapper engines. In the first approach wrappers utilize the HTML tag-based hierarchy and web pages are completely parsed into a tree. Then, depending on the granularity of the text to be extracted, regular expressions may be applied. W4F [Sahuguent & Azavant 99] and OnDisplay [OnDisplay 97] for instance belong to this category. The second approach completely ignores the HTML tag-based hierarchy and simply applies regular expressions with any level of granularity. Caméléon belongs to this second class of web wrappers. Both approaches have strong and weak points, some of which are shown in Table 1 and Table 2 below.

*HTML Tag-Based Approach*

| ADVANTAGES | DISADVANTAGES |
|---|---|
| **Easy rule construction:** Allows easier construction of extraction rules by utilizing the HTML tag hierarchy.<br>**Concise rules:** Allows powerful yet concise extraction rules by the using the DOM model.<br>**Preservation of Hierarchy:** Associations among hierarchy elements, i.e. column name and column elements is possible. | **Irregularity:** Less than 20% of the pages are conforming to the HTML standards, therefore HTML parsers need error recovery mechanisms. [Sahuguent & Azavant 99].<br>**Currency:** Engine has to be updated with changing HTML versions.<br>**Performance:** Parsing HTML into a tree is expensive. |

Table 1. Advantages and Disadvantages of HTML Tag Based Approach

*Regular Expression Approach*

| ADVANTAGES | DISADVANTAGES |
|---|---|
| **Generality:** Independent of HTML, thus not effected by the web languages. It would work just fine with XML or anything else.<br>**Granularity:** Matching with any level of granularity is possible.<br>**Performance:** Regular Expression pattern matching is orders of magnitude faster than parsing. | **Complexity:** Regular Expressions are harder to form and understand by regular users compared to HTML tag-based approach.<br>**No-nesting:** In order to be efficient regular-expression pattern matching does not backtrack.<br>**Limited Association:** Hard to associate hierarchical relations, i.e. inferring column elements from the column name or number. |

Table 2. Advantages and Disadvantages of Regular Expression Based Approach

Despite its complexity we preferred the regular expression based approach, in order to avoid becoming obsolete in the fast changing world of web. Because our approach is not specific to HTML, it can also be applied to other domains without any extra difficulty. We can wrap and treat XML or text sources as relational databases with our approach, and execute SQL queries against them. By choosing a regular expression based approach, however, we are forgoing the opportunity to utilize the information hidden in the tag hierarchy. Yet this does not constitute a significant problem for us given our relational front end which has query processing capabilities and the speed of regular expression pattern matching.

Our system also produces XML output after a post-processing of the extracted data. Thus it can be used just for the purposes of converting HTML pages into XML.

## Caméléon Architecture

There are two major components that make Caméléon look like a relational database: the relational front-end, and the core engine. Relational front-end consists of a planner, optimizer, and an executioner. Because the front-end has been developed as a separate process, we will not go into any details of it in this paper, but suffice it to say that it takes an SQL query, creates a plan, optimizes it and then runs it concurrently using the core Caméléon.

The core of Caméléon is composed of Query Handling, Extraction, and Retrieval modules as shown in Figure 1. Based on input queries and interaction with the registry the query handler determines which spec file needs to be retrieved and sends a request to the spec-file parser. Spec file parser, implemented using JavaCC (parser-generator written in Java) and JJTree (an add-on which allows the generated parsers to produce syntax trees), fills in relevant internal data structures with pattern and schema information. Query handler then calls the extractor. Extractor module first calls the retrieval module, which in turn uses HTTPClient to retrieve the requested web page. As mentioned before, HTTPClient handles authentication, redirection, cookie and SSL issues. When the web page is retrieved and returned back, the extractor calls the regular expression engine and supplies the relevant attribute patterns to obtain the desired data items. After the patterns are matched and data items are obtained they are returned back to the query handler in a plain format. The query handler then adjusts the data format and returns the answer to the user.

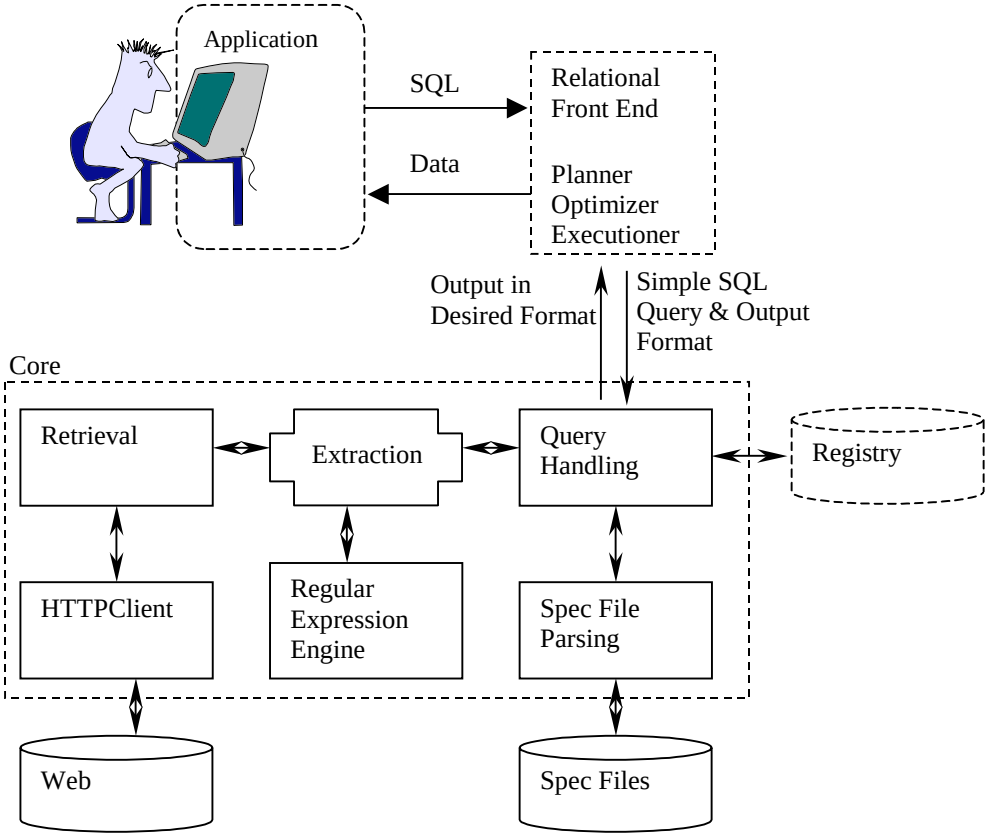


Figure 1. Caméléon Architecture

Our modular approach enables us to remove one component of the system and replace it with something functionally similar. It was in fact quite easy for us to create a server version of Caméléon, by simply adding a Java servlet class which retrieved requests and redirected them to the query handler.

## Spec File Structure

Before describing the new spec file structure, we will first explain the shortcomings of the Grenouille spec file structure to provide the motivation behind our new design.

### Grenouille Spec Files

In Grenouille, a spec file can contain multiple pages. But each page can only contain a single content element that describes the extraction rules for a set of attributes. An example is shown in Figure 2.

```
#HEADER
#RELATION = quicken
#HREF=POST http://quicken.elogic.com
#EXPORT= quicken.Ticker quicken.CompTicker   quicken.Competitors
quicken.Marketcap
#ENDHEADER
#BODY
 #PAGE
  #HREF=GET
http://www.quicken.com/investments/comparison/?symbol=##quicken.Ticker##
 #CONTENT=
<A\s*HREF="/investments/quotes/\?symbol=.*?">##quicken.CompTicker:(.*?)##</A></
FONT></TD>\s*<TD ALIGN=LEFT><FONT SIZE=2
FACE="Arial">##quicken.Competitors##</FONT></TD>\s* <TD ALIGN=RIGHT>
<FONT\s*SIZE=2\s*FACE="Arial">.##quicken.Marketcap##.nbsp;.nbsp;</FONT>
#ENDPAGE
#ENDBODY
```

Figure 2. Example Grenouille Spec File

Grenouille has a page based approach and applies the regular expression pattern described within the Content tag to the whole text of the page. This may cause two problems:

- Inefficiency because of applying (usually complex) patterns to the whole page.
- Inability to limit patterns to particular segments of the web pages, thus the difficulty of creating a globally unique expression.

In fact the second problem created significant difficulties for our Instant Web Wrapper Generator (IWrap), which is a tool designed to automatically create spec files based on minimal user input. When IWrap found patterns that would extract the requested information, the patterns were usually only locally unique. Thus the result included spurious tuples and the specification file creation failed. Consider for instance the following simplified html from CNN:

*World:*
*<ul>*
*<li>Mexico City rocked by quake*
*<li>Guenter Grass wins Nobel Prize in literature*
*</ul>*
*U.S.:*
*<ul>*
*<li>Standardized tests under fire*
*<li>CIA sets up venture capital firm in Silicon Valley*
</ul>

Without using local scopes it is extremely difficult (perhaps impossible) to extract the World and U.S. news items in relational form using a single pattern.

The following pattern*

*World:.*?<li>(.*)$.*?U.S.:.*?<li>(.*)$*

for instance would only match the first items, “Mexico City rocked by quake” and “Standardized tests under fire” for the reason that regular expressions do not backtrack when the match is successful. In fact [Friedl 97] recognizes this difficulty and suggests a two step approach.

The simple solution proposed is as follows:

1. *Match up to a landmark expression*
2. *Then apply the pattern to match items after that point*

## Caméléon Spec File Structure

Caméléon attempts to solve the above-defined problem by limiting the scope of the patterns. Patterns are now defined within start and end points. (If the user wants to have a global pattern, it is enough to specify the begin and end points with the <html>, </html> tags respectively.) In addition patterns are defined separately for each attribute. For instance we would wrap the CNN example as follows:

*#Attribute = WorldNews#String*
*#Begin = World:*
*#Pattern = <li>(.*)$*
*#End = </ul>*

*#Attribute = USNews#String*
*#Begin = US:*
*#Pattern = <li>(.*)$*
*#End = </ul>*

This approach has two basic advantages:

- It makes the task of automatically creating a pattern for a selected text easier by limiting the scope of patterns.
- Complex regular expressions are applied only to a small portion of the page, which increases the efficiency of the extraction.

In the attribute definitions shown above the attribute tag describes the name and the type of the attribute. All other elements in the attribute block are regular expressions. As explained above regular expressions are used in begin and end blocks to create regions in the page. Pattern element denotes the data to be extracted in enclosed parentheses. Three additional features introduced by our new approach to spec file creation are listed below:

**1. Disjunction by allowing multiple patterns.**

If the item to be extracted has multiple patterns, this feature may be quite useful. For instance consider the following extract from the http://www.quicken.com domain, and assume the user is only interested in the bolded text.

```
<B>Last Trade</B></FONT>
</TD>
<TD WIDTH=130>
<FONT SIZE=2 FACE="Arial"> 
<B>120 <FONT SIZE=1>1/16</FONT>
</B></FONT></TD>
```

Notice thatthe fractional part of the last trade value is put into special tags. Sometimes, however, last trade values do not have fractions and the html may look like:

```
<B>Last Trade</B></FONT>
```

* Please refer to Appendix II for a quick reference to regular expressions used in our examples.

```
</TD>
<TD WIDTH=130>
<FONT SIZE=2 FACE="Arial"> 
<B>81
</B></FONT></TD>
```

In these kinds of cases it may be quite difficult, if not impossible, to find a single pattern to match data items.

The solution with our new approach is straightforward. We define two patterns and both of them are applied within the specified interval.

```
#Attribute=LastTrade#String
#Begin=Last\s*Trade
#Pattern=<B>\s*(.*?)\s*<FONT\s*SIZE=1>(.*?)</FONT>
#Pattern=<B>\s*(\d+)\s*</B>
#End=</TR>
```

Note, however, that these disjunctive patterns are not mutually exclusive and occasionally special care must be taken to construct patterns whose intersections are empty. Otherwise the same item will be matched multiple times and repeated in the output.

**2.Concatenation within a pattern element.**

As seen in the quicken example above, the last trade value 120 1/16 is separated by unwanted text. In Grénouille, it was impossible to extract a single value divided by unwanted text without secondary processing. The last trade values in the quicken example would be extracted as:

```
120 <FONT SIZE=1>1/16</FONT>
```

and it would look just right when displayed in HTML. Nevertheless, the extracted values were not really precise.

In Caméléon all matched elements within a pattern element are concatenated. In the quicken example the first pattern has two enclosing parentheses. The first one matches the whole part, the second one the fractional part. The matched elements are then concatenated to form a single value. This feature is very useful as it is usually the case that unwanted tags separate the desired information.

**3.Object Groupings**

As mentioned before, patterns in Caméléon spec files are defined for each attribute. Each attribute is also allowed to have multiple values. This may create a problem in identifying units of data. Consider for instance the following simplified html extract:

```
Research Abstracts:
<i>IBM</i>:<b>Abstract1 Abstract2 </b>
<i>Microsoft</i>:<b>Abstract3 Abstract4 Abstract5 </b>
</BODY>
```

In this example suppose that we would like to have the tuples shown in Table 3:

| CNAME | ABSTRACTS |
|---|---|
| IBM | Abstract1 |
| IBM | Abstract2 |
| Microsoft | Abstract3 |
| Microsoft | Abstract4 |
| Microsoft | Abstract5 |

Table 3. Desired tuples

To obtain the tuples shown in Table 3, we need to associate attribute values with each other to form the correct tuples. To accomplish that we need to know what constitutes a data unit. We use the tags #ObjectBegin and

#ObjectEnd to indicate the boundaries of a data unit. For instance in the above example we use the following constructs:

```
#Attribute = CName
#Begin = Research Abstracts
#ObjectBegin =<i>
#Pattern = (.*?)
#ObjectEnd = </i>
#End = </BODY>

#Attribute = Abstracts
#Begin = Research Abstracts #ObjectBegin = <b>
#Pattern = (\S+\s+)
#ObjectEnd = </b>
#End = </BODY>
```

The extraction engine first extracts the text between “Research Abstracts” and “</BODY>”. Within this text it then extracts anything between the  and  tags. Any attribute values that will be matched within this text after applying the pattern will be considered as one unit. In this case we match IBM. Then we move to the second begin-end group and extract anything between “Research Abstracts” and “</BODY>”. This time a unit of data is between  and , thus we apply the pattern to the text within this interval. We match two values: Abstract1 and Abstract2. Cartesian product of IBM and Abstract1, Abstract2 gives us the right tuples. Next we go back to the first attribute and repeat the steps again until there is no match. This approach greatly simplifies the tuple identification problem.

## Advanced Spec File Creation

**1. Multi-page transitions**

When wrapping web pages, we sometimes need to traverse multiple pages to locate the page we want to extract information from. This situation occurs when the URLs are created dynamically (e.g. a session ID is assigned for each access to the page), or a cookie needs to be established before you can go to the desired page, or there is no simple way of deducing the desired URL without visiting a particular page, or the data is spread through multiple pages. To handle these kinds of cases Caméléon has a feature that lets us wrap multiple pages for a relation. Consider for instance the following example:

```
#Source=http://www.edgar-
online.com/bin/esearch/default.asp?sort=F&sym=#Ticker#&date=1994

#Attribute=Link#String
#Begin=10-K
#Pattern=<A\s*HREF="([^"]*)">FDS
#End=People

#Source=http://www.edgar-online.com/#Link#

#Attribute=INVENTORY#String
#Begin=<TABLE\s*BORDER=1\s*CELLSPACING=0\s*CELLPADDING=1>
#Pattern=INVENTORY[\0-\377]*?<TD[^>]*><[^>]*><[^>]*>([^<]*)<
#End=</TABLE>
```

In the above example we first extract the subsequent page link from the first web page. We need to perform this step in this case because there is no simple way of deducing in advance what the link is supposed to be. Then the link is supplied to the next source element, which takes us to the page where we want to extract the values of financial attributes.

This approach is quite useful if we know in advance how many pages we are going to traverse. In other cases, where the number of intermediate pages is not deterministically known a finite state machine approach (FSM)

would be useful. Associating every source definition above with states and introducing transition rules, would enable us to solve the problem of non-deterministic multiple page traversals. Unlike [Qu 96]'s state transitions, however, our state transitions will sometimes be cyclic. Consider for instance search engines that only display ten results at a time and you need to click on "next" to see other results. The number of "next" clicks, however, is not static, thus there is a need to cycle in our state transition diagrams. We are in the process of implementing this feature in Caméléon.

**2. Parameter Replacement**

The second advanced feature we would like to mention is the use of input or extracted attribute values within the subsequent patterns. In the multiple page traversal case, we have seen one example of this. The value of attribute "Link" was used in the next source element. It is also possible to supply any extracted or input attribute value within the attribute definitions. Consider for instance the following SQL query to the CIA Fact Book web site:

*Select capital From cia Where country= "France"*

When this query is executed, the input attribute value, France, replaces #Country# in the pattern of attribute Link during spec file parsing. (Relevant portions of the spec file is shown below.)

```
#Relation=cia

#Source=http://www.odci.gov/cia/publications/factbook/country.html

#Attribute=Link#String
#Begin=Top\s*of\s*Page
#Pattern=<LI><FONT SIZE=-1><a href="([^"]*)">#Country#</a></font>
#End=</[Bb][oO][dD][yY]>

#Source=http://www.odci.gov/cia/publications/factbook/#Link#

#Attribute=capital#String
#Begin=Capital:
#Pattern=</b>\s*([\0-\377]*?)\s*<
#End=Administrative\s*divisions:
```

Replacement of input or extracted attribute values is not limited to only the pattern element, but can be performed at any place in the spec file. #Begin, #EndBegin, #ObjectBegin and #ObjectEnd also replace any attribute specified in between # signs with its value.(Normal # signs need to be escaped with backslash.)

**3. Post Example**

The default method Caméléon uses in communicating with the HTTP protocol is "GET", therefore it needs not be specified explicitly. If the user wants to use "POST" method the parameters have to be specified after the #Param tag. A POST example is shown below involving a login procedure to cdnow.com, a password protected web site. Notice that password attribute needs to be inputted by the user in the SQL query as the first reference to it is enclosed with # signs in the spec file.

```
#Relation=MYCDNOW

#Source=http://www.cdnow.com/cgi
-bin/mserver/pagename=/RP/CDN/ACCT/loginform.html/loginform=

#Method=POST
#Param="lname"="malchik" "fname"="alexander" "password"="#password#"
"autologinflag"="" "passedpagename"="/RP/CDN/ACCT/account_summary.html"
```

**4. Authentication through pop-up windows**

Some web pages perform authentication through pop-up password windows. We handle these kinds of cases with the following scheme. The username and password values have to be inputted within the SQL query, since they are coded as references in this spec file.

```
#Relation=Etrade

#Source=http://game.etrade.com/cgi-bin/cgitrade/TransHistory
#Realm=E*Trade Player (game)
#Username=#username#
#Password=#password#
```

## Extraction Algorithm

The algorithm we have employed in extracting data from web pages is quite simple and depicted in Figure 3.

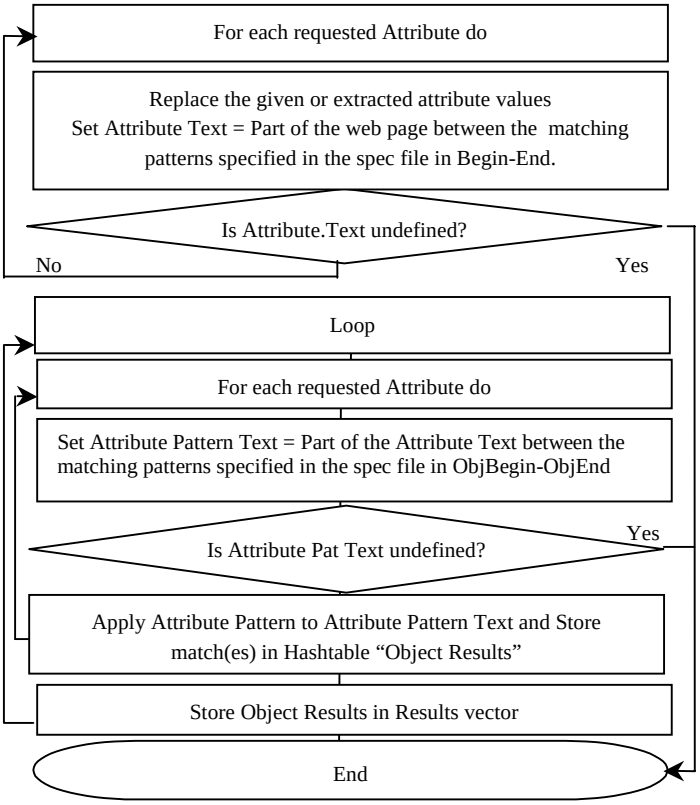


Figure 3. Extraction Algorithm

In the algorithm, we first apply begin-end patterns to extract the relevant text for each attribute from the source html. For instance in the CNN example mentioned before, Attribute Text for WorldNews atribute becomes :

*<ul>*
*<li>Mexico City rocked by quake*
*<li>Guenter Grass wins Nobel Prize in literature*

If any of the Attribute Text values is undefined (i.e. begin-end points are not found) we do not proceed anymore, and conclude that the spec file has become outdated or there is something wrong, thus report an error. Otherwise we do the same thing for all attributes. Next we apply objectbegin-objectend patterns to extract the Attribute Pattern Text for each attribute from the Attribute Text. For instance in the IBM, Microsoft example, the first Attribute Pattern Text would be:

Then we apply the pattern element(s) by using the regular expression engine and obtain the matching elements. In the above example this would result in two values. These values are then stored in a hashtable based on attribute names.

This procedure is then repeated for all attributes until the first attribute has an undefined Attribute Text Pattern. After each round, the hashtables are stored in a vector of objects. Finally the results are returned to the query handler for special formatting.

## Sample Applications

We wrapped several web sites using Caméléon and made some of them available online for demonstration. The samples can be reached from http://context.mit.edu/~coin/demos/ under the heading Caméléon : Web Wrapper.

We have also developed demonstration applications that aggregate data from multiple web pages and present them in a unified interface. One such example, called Personal Investor Wizard (PIW), aggregates data from ten different web sources including cnn, fox, yahoo, quicken, fortune and edgar and is available online for demonstration [http://context.mit.edu/~coin/demos/]. PIW scrolls daily headlines (extracted from FoxNews and CNNfn), lets users view companies in a selected industry (obtained from yahoo), and display the competitors of selected companies (taken from quicken). A snapshot of this screen is shown in figure 4. Then when the user wants to compare a set of companies, PIW displays in a second screen the profile info for each company (extracted from yahoo), the analyst recommendations (extracted from quicken), financial figures (extracted from edgar-online), and recent news(extracted from fortune). A snapshot of this screen is shown in figure 5.

We built PIW using Java Server Pages(jsp) and simply embedded SQL queries in the jsp page. It is important to note that once the spec file for a web site is set up, it can be used by many applications and the developer of any of these applications merely views the web site as a traditional relational database. Development time of PIW, therefore, was quite short.

Because Caméléon accepts SQL queries and has a Java Servlet version, it is also very easy to call it from other applications. The Java version of PIW, an example of this flexibility, is also available in our web site for download.

## Conclusions & Future Work

Caméléon is able to provide a robust infrastructure for web automation as [Allen 97] defines it. Specifically:

- it has full interaction with HTML forms i.e. it supports both get and post methods;
- it handles both HTTP Authentication and Cookies;
- both on-demand and scheduled extraction of targeted web data is possible as demonstrated by PIW Java version.
- it facilitates aggregation of data from a number of web sources as demonstrated by PIW.
- it can extract data across multiple web sites through chaining.
- it is very easy to integrate it with traditional application development languages and environments as it provides a SQL interface to web pages.
- our declarative way of specifying extraction rules provides a clean framework for managing change in both the locations and structures of Web documents.

We are currently working on automating the spec file creation, which is going to facilitate the task of managing change in both the locations and structures of Web documents for large scale projects. Further improvements to handle non-deterministic multiple page traversals are also under consideration.

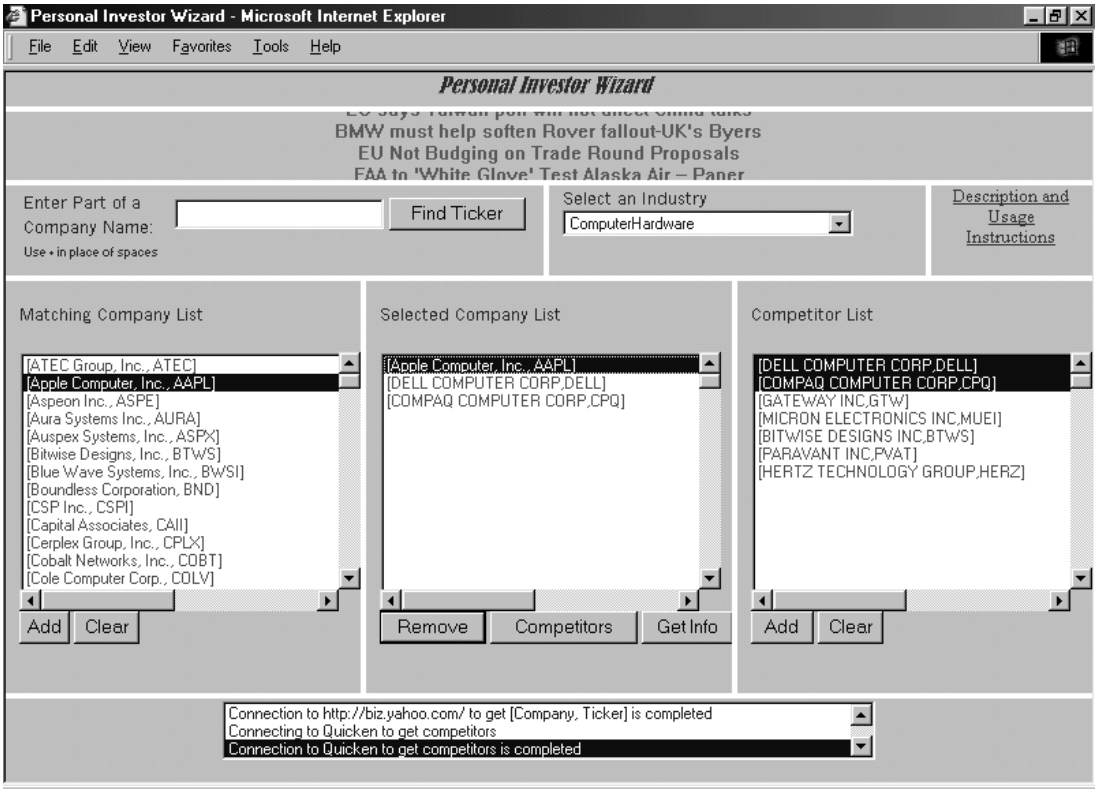


Figure 4. Personal Investor Wizard - Main Screen (Profile, Financial Info & News Display)

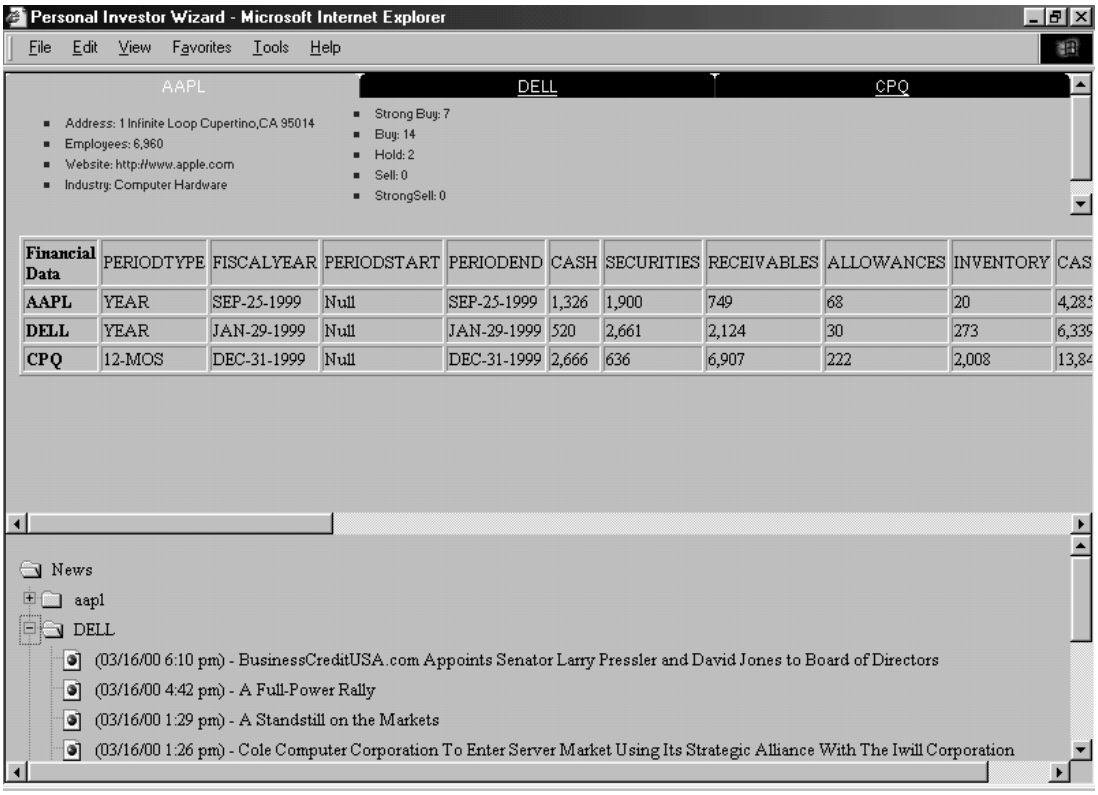


Figure 5. Personal Investor Wizard - Comparison Screen (Invoked by “Get Info” button)

## Appendix I-Related Projects

*Academic Projects*

World Wide Web Wrapper Factory (W4F)

URL: http://db.cis.upenn.edu/W4F

W4F is a toolkit for the generation of web wrappers, developed by F.Azavant and A. Sahuguet at the University of Pennsylvania. It relies mainly on the HTML tag-based approach, but also allows regular expressions to handle finer levels of granularity. The tool has a WYSIWYG user interface that helps one find the rough extraction rule of a highlighted element in their DOM-centric HEL language. Because wrappers are generated as Java classes the users have to go through compilation process each time they create new wrappers or make some modifications in the existing ones. Cookie and javascript issues are not supported. The results are presented in xml based on a user defined mapping.

The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS)

URL: http://www-db.stanford.edu/tsimmis/tsimmis.html

TSIMMIS project uses a wrapper for web sources as described in [Hammer et. al 97]. This wrapper uses a regular expression like declarative language to express the extraction rules. The results are presented in OEM format which is recognized by other TSIMMIS components.

The Ariadne & Theseus Projects

URL: http://www.isi.edu/ariadne/index.html

Ariadne and Theseus are extensions of the SIMS information mediator developed by the Information Sciences Institute of University of South Carolina. Ariadne focuses more on using AI techniques(e.g. Hierarchical Induction in Stalker) in automatically generating wrappers, while the Theseus project is trying to build an efficient plant execution system for information agents [Barish 00].

The Wrapper Generation Project

URL: http://www.umiacs.umd.edu/labs/CLIP/WW.html

This is a project of the Computational Linguistics and Information Processing Laboratory (CLIP) at the university of Maryland Institute for Advanced Computer Studies (UMIACS). The wrapper toolkit developed in this project (including graphical interfaces) helps one define the capabilities of web sources and build wrappers for them through the use of an underlying DOM-centric language. [Gruser 98]

XWRAP

URL: http://www.cse.ogi.edu/DISC/XWRAP/

XWRAP has been developed by the Computer Science department of the Oregon Graduate Institute. The graphical wrapper generation toolkit maps an HTML page to an XML one by utilizing user input and machine learning techniques.

Java Extraction and Dissemination of Information (JEDI)

URL: http://www.darmstadt.gmd.de/oasys/projects/jedi/

JEDI, built completely in Java, is developed by the German National Center for Information Technology. It combines grammar and pattern matching based approaches in creating wrappers [Huck et. al 98].

The Wrapper Induction for Information Extraction Project

URL: http://www.cs.washington.edu/homes/weld/wrappers.html

The essential contribution of this project, undertaken by the computer science department of University of Washington, is to introduce wrapper induction, a technique for automatically constructing wrappers from labeled examples of a resource's content. Its central algorithm uses the labeled examples to automatically output a grammar by which the data can be extracted from subsequent pages. [Kushmerick et. al 97]

The Araneus Project

URL: http://www.difa.unibas.it/Araneus/

Araneus is a set of tools not only for web wrapping and querying but also for the design and implementation of web sites. The tools are written in pure Java. It is a joint project of the database groups of the Universita di Roma Tre and Universita della Basilicata (Italy). [Mecca 99]

Mediation of Information using XML (MIX)

URL: http://www.npaci.edu/DICE/MIX/

MIX project is a collaboration between the University of California, San Diego Database Laboratory and the Data-intensive Computing Environments (DICE) group at San Diego Supercomputer Center and develops wrappers wrappers that convert HTML into XML. [Ludäscher and Gupta 99]

*Commercial Projects*
Information Manifold (Bell Laboratories)
URL: http://portal.research.bell-labs.com/orgs/ssr/people/levy/manifold.html
WHIRL (ATT Research)
URL: http://whirl.research.att.com/
Garlic (IBM Research)
URL: http://www.almaden.ibm.com/cs/garlic/homepage.html
WebL (Compaq Research)
URL: http://research.compaq.com/SRC/WebL/
CenterStage (OnDisplay)
URL: http://www.ondisplay.com
WebMethods
URL: http://www.webmethods.com
AlphaConnect
URL: http://www.alphaconnect.com
Cambio (Data Junction)
URL: http://www.datajunction.com/products/
Sagent Technologies
URL: http://www.sagent.com
Yodlee
URL: http://www.yodlee.com

## Appendix II- Regular Expressions Used in our Examples

- * Match 0 or more times (greedy).
- *? Match 0 or more times (non-greedy).
- + Match 1 or more times (greedy).
- ? Match 0 or 1 time (greedy).

Greedy quantifiers such as * matches as much as possible, whereas non-greedy quantifiers stop at the minimum match. Example:
Consider the following text:
 hello  lovely   world 
(.*)  would match ‘hello  lovely   world’ whereas
(.*?)  would match ‘hello’ and ‘world’

- . matches everything except \n
- [\0-\377] matches everything
- ^ matches the beginning of a string or line
- [^ a character] matches everything except the specified character. For instance [^<] matches anything but <
- $ matches the end of a string or line
- \s matches a whitespace character
- \S matches a non-whitespace character
- \d matches a digit
- Expressions within parentheses are saved.

More information can be found at:
http://www.savarese.org/oro/developers/docs/OROMatcher/Syntax.html#Perl5Expressions

## References


[Abiteboul 97] Serge Abiteboul. Querying semi-structured data. In Proceedings of ICDT, Jan 1997

[Allen 97] Charles Allen WIDL Application Integration with XML, http://www.xml.com/pub/w3j/s3.allen.html

[Ambrose 98] Ricardo S. Ambrose, A lightweight multi-database execution engine, Thesis (S.M.)--Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998.

[Barish et. al. 00] Greg Barish, Dan DiPasquo, Craig A. Knoblock, Steven Minton. A Dataflow Approach to Agent-based Information Management. IC-AI 2000. Las Vegas, NV. June 2000.

[Bressan & Bonnet 97] S. Bressan, Ph. Bonnet, Extraction and Integration of Data from Semi-structured Documents into Business Applications, Conference on the Industrial Applications of Prolog, 1997.

[Friedl 97] Jeffrey E. F. Friedl, Mastering Regular Expressions, Powerful Techniques for Perl and Other Tools, O'Reilly, 1997.

[Gruser et. al. 98] Jean Robert Gruser, Louiqa Raschid, Maria Esther Vidal, Laura Bright. Wrapper Generation for Web Accessible Data Sources In Proceedings of CoopIS'98.

[Hammer et. al 97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. "Extracting Semistructured Information from the Web". In Proceedings of the Workshop on Management of Semistructured Data. Tucson, Arizona, May 1997

[Huck et. al 98] Gerald Huck, Peter Fankhauser, Karl Aberer, Erich J. Neuhold: JEDI: Extracting and Synthesizing Information from the Web; submitted to COOPIS 98, New York, August, 1998; IEEE Computer Society Press.

[Jakóbisiak 96] Marta Jakóbisiak, Programming the Web- Design and Implementation of a Multidatabase Browser, CISL WP#96-04, May 1996, MIT Sloan School of Management.

[Kushmerick et. al. 97] N. Kushmerick and R. Doorenbos and D. Weld Wrapper Induction for Information Extraction. IJCAI-97, August 1997.

[Liu et. al. 99] Ling Liu, Calton Pu, Wei Han, David Buttler, Wei Tang. `` XWrap: An Extensible Wrapper Construction System for Internet Information Sources", Oregon Graduate Institute of Science and Technology.

[Mecca et. al 99] G. Mecca, P. Merialdo, P. Atzeni ARANEUS in the Era of XML - IEEE Data Engineering Bullettin, Special Issue on XML, September, 1999

[Lacroix 99] Zoé Lacroix: "Object Views through Search Views of Web datasources", International Conference on Conceptual Modeling (ER'99), Paris, France, November 1999

[Ludäscher and Gupta 99] B. Ludäscher, A. Gupta, Modeling Interactive Web Sources for Information Mediation, Intl. Workshop on the World-Wide Web and Conceptual Modeling (WWWCM'99), Paris, France, LNCS, Springer, 1999.

[OnDisplay 97], Centerstage, http://www.ondisplay.com.

[Qu 96] Jessica F. Qu, Data Wrapping on the Worl Wide Web, CISL WP#96-05, February 1996, MIT Sloan School of Management.

[Roth & Schwarz 97] Roth, M.T., Schwarz, P.M., Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. VLDB'97, Proc. of 23rd Int. Conference on Very Large Data Bases, August 25-29, 1997.

[Sahuguent & Azavant 99] A. Sahuguet and F. Azavant, W4F: the WysiWyg Web Wrapper Factory. Technical report, University of Pennsylvania, Department of Computer and Information Science, 1998.

[Tschalär 99] Ronald Tschalär, HTTPClient Version 0.3-2, http://www.innovation.ch/java/HTTPClient/