

Towards Sophisticated Wrapping of Web-based Information Repositories

Boris Chidlovskii, Uwe M. Borghoff and Pierre-Yves Chevalier

Rank Xerox Research Centre, Grenoble Laboratory
6, chemin de Maupertuis. F-38240 Meylan, France

E-mail: {chidlovskii, borghoff, chevalier}@grenoble.rxc.xerox.com Tel.+33 4 76615050

Abstract

Access to on-line information via the Web is exploding. Index and retrieval engines already start to integrate a huge variety of heterogeneous repositories. However, the heterogeneity issue remains, both in terms of the search formats and the formats of the result pages.

In this paper we focus on html-based search and result presentations. We discuss our experience in the design, the development and the maintenance of wrappers (in the context of the *Knowledge Broker* project). We outline different ways to write wrappers, illustrate some of the lessons learned, and conclude by describing a semi-automatic approach for an efficient wrapping of Web-based information repositories. Throughout the paper, we give illustrating examples for hands-on readers.

KeyWords

World Wide Web; heterogeneous repositories; wrapping; information extraction; rule-based parsing.

1 Introduction

The continuous growth of documents available on the Internet, especially via the World Wide Web, creates the need for new technologies and mechanisms that query the different information stored in heterogeneous repositories [BoSc96]. These query facilities must provide homogeneous access (to the underlying heterogeneity) and allow for homogeneous representation of the information found.

The existing indexing and retrieval engines on the Web already support rudimentary ways for querying the huge amount of information stored in Web-repositories [ODL93, SEK⁺92]. However, there still remains the problem of dealing with the diversity of the indexing schemes and the formats the results appear in on the Web. In addition, the information retrieved from the different repositories must be merged and unified. Apart from the communication with the retrieval engines, the problem consists in sophisticated analyses of result pages returned by the retrieval engines, and in putting the parsed answers into a unified format. In the literature these activities are covered by the term *wrapping*. The time-saving and

highly declarative level approaches to the wrapping were a cornerstone of several recent projects on heterogeneous information retrieval [KnAm96, QRS⁺95].

In reality, the main problem we encounter when working with the Web-repositories is *un*-wrapping and not wrapping. The extraction of relevant information from the (raw) result pages is where most difficulties arise, while storing the extracted information within a unified format, i.e., wrapping it, is in most cases a minor problem. However, in the remainder of this paper we shall use the term *wrapping* both for the unwrapping and wrapping of information.

This paper presents our work and experience on high-level wrapping of heterogeneous information retrieved from the Web-repositories. This work is part of the *Knowledge Broker* project [ABP⁺95, ABP96]. Our approach to the integration of distributed information repositories contrasts with the one used in most federated database projects in that we do not rely on meta-schema and meta-indexes but rather keep our system as independent as possible from the external sources. This provides greater flexibility for integrating remote sources but introduces additional problems with respect to the maintenance of wrappers. We report here on how different approaches to wrapping were exploited, starting from a naive hand-made coding, to an advanced approach exploiting rule-based analysis. Throughout the paper, we assume a http-based protocol for both the querying of the Web-repositories and the retrieving of the result pages.

The paper is organized as follows. Section 2 describes the problem and gives pointers to related work. In Section 3, we classify aspects of result pages provided in html-format. Here we distinguish between format uniqueness and completeness, and between different html-quality levels. Section 4 introduces the reliability problem for the result page analysis. The wrapper design, approached in three different ways, is illustrated in Section 5. In Section 6, we discuss the particular implementation issues. Section 7 concludes the paper.

2 Problem Description and Related Work

The key problems in any wrapper subsystem consist of the *backend analysis*, the *wrapper coding*, and the *wrapper maintenance*.

- The *backend analysis* is an intrinsic requirement for any high-quality solution to the problem of a clean wrapper design. The problem aspects vary from detecting the initial URLs and the needed cgi-structures of the retrieval engines of the backends in question, to analyzing the html-pages returned by these engines.
- Three main approaches to the *wrapper coding* are analyzed in the paper. The first is a naive hand-made coding approach. The second is based on a sgml-parser which spots html tags and extracts raw data between the tags. It works very much like a state-automata where html tags are used to define transitions. The use of such a parser makes the wrapper structure more transparent, but usually longer. In our experience, the length of the parser for a particular wrapper may be up to 200-300 lines of code. The third approach uses lexical and syntactical grammar rules to automatically generate the parser. The experience shows that a full parser is defined with 10 to 15 rules.
- Among the key problems listed above, the most difficult issue is sometimes the *wrapper maintenance*. Information on the Web is continuously updated. Any change in any of the numerous backend aspects may deeply impacts the corresponding wrapper. What makes the problem worse is that in many cases an interactive user does not even detect a chang-

ing backend structure (be it in terms of the search form, or in terms of the layout of the result page). Wrapper maintenance is difficult due to changes in the html-page of the retrieval engine of the backend, the file locations, the html-format of the result page, and, most importantly, due to changes of the search and retrieval functionality. Although many public-access repositories on the Web are stable for quite a while (e. g., *Library of Congress*, *AltaVista*, *WebCrawler*), some other backends are in the phase of interrupted existence, continuous improvements and heavy modifications (e. g., the on-line catalogue for a database of “Databases and Logic Programming” maintained at <http://www.informatik.uni-trier.de/~ley/db/index.html>). This becomes even worse when one has no control over external sources, and when no global meta-data such as a catalog or an index is available. For any system of interest to a wide range of users, we need a high-level tool which allows us to create and maintain dozens and even hundreds of wrappers in a simple and transparent mode.

The wrapping problem was addressed in several research projects on the heterogeneous information retrieval. The *TSIMMIS* project [CGH⁺94, GHI⁺95, QRS⁺95] is a joint project of Stanford University’s Database Group and IBM. The project focus is on the integration of structured and unstructured data sources, techniques for the rapid prototyping of wrappers and techniques for implementing mediators. The system put a lot of emphasis on the conversion of high-level SQL-like queries into source-specific queries and on the translation of returned results into its data meta-model. It follows an approach that is naturally biased towards database issues and which might lack the full flexibility needed to accommodate to the diversity and rapid evolution of the World Wide Web.

The group around K. Geihs [PMG⁺95, PuBu96] at Univ. Frankfurt in Germany exploits *conceptual graphs* for the matching problem of functionally equivalent attributes despite different data schemata. The main focus of their work lies in building some mediator functionality for the trading and interworking in open distributed environments. AI techniques are used to implement matching and learning algorithms.

SIMS is an information mediator that provides access to heterogeneous data and knowledge bases [KnAm96]. It is structured as a network of cooperating information agents where each agent provides access to a subset of resources available and can server as an information source to other agents. The focus of *SIMS* is more on the representation of knowledge and the management of semantic issues than on the extraction of information per se. One of the strengths of this system is its ability to cope with (semantic) changes at the information sources in a transparent manner. However, the actual connection to an external source is very specific and in some respect hard-wired. *SIMS* lacks the flexibility required to extract raw information from external sources and to cope with evolution of those sources.

Other projects related to the wrapping problem include *COIN* (MIT), *Harvest* (Univ. Colorado), *Garlic* (IBM Almaden), *HERMES* (Univ. Maryland), *Information Manifold* (AT&T), *InfoSleuth* (MCC), and *KRAFT* (Univ. Aberdeen, Univ. Wales, Univ. Liverpool, BT).

3 Classification Aspects of Result Pages in HTML

In this section we look briefly at some classification aspects concerning the result pages (in html-format) our wrappers work with.

3.1 Format uniqueness + Completeness (homogeneity)

This parameter pair provides the most obvious classification of html-pages returned by the different backends. We hereby define completeness to be homogeneity, that is, a search request always returns the same set of elements within the resulting html-page. The following classification can be given:

- *Rigorous structure*, i.e. unique format + complete information: for example, *AltaVista* always returns a “name” and a “body” element (even though with rather unstructured, raw information).
- *Semi-rigorous structure*, i.e. unique format + incomplete information: for example, *Library of Congress* always provides a list of attributes for each returned item but not all of the attributes are common for all the items.
- *Semi-relaxed structure*, i.e. no unique format + complete information: semi-relaxed structures result from data sets put together from different legacy collections.
- *Relaxed structure*, i.e. no unique format + incomplete information: most home pages on the Web have relaxed structures.

Experience shows that many useful sources of information on the Web publish data which are stored in an internal structured format (e.g., BibTeX or any database formats). These data are wrapped into a html-format by a corresponding generator (see Figure 1). This wrapped format is what a user gets when he queries the search engine of the backend. The original structure of the data is normally lost but the generated html keeps some sort of structure. Some http-gateways (together with html-forms for searching) lose some of the structures of the formerly full-fledged, attribute-rich Z39.50 protocol. To name just two, OPAC-gateways were written for the Library of Congress and the Staatsbibliothek München, Munich, Germany.

The publication of structured data on the Web is a trend that is likely to increase in the near future. We thus consider sources that exhibit a rigorous or a semi-rigorous structures are our primary targets.

Broker systems, like our *Knowledge Brokers*, try to unify and homogenize result pages of different backends. They also try to extract attribute-based information to increase the precision of the hits through local constraint solving techniques [BCW96]. Such systems have to solve the inverse problem of the Library of Congress which is *unwrapping* semi-structured html into structured data. Two different aspects should be distinguished here.

- The first aspect deals with the work that is fully inverse to the html-generator. Since most html-generators are quite simple, this part is simple too.
- The second part concerns the data themselves. This part is harder because the data were (typically) input manually, with no or little emphasis on accuracy and coherence. During the parsing step of the unwrapping, for example, you can expect symbols or sentences in an unpredictable position.

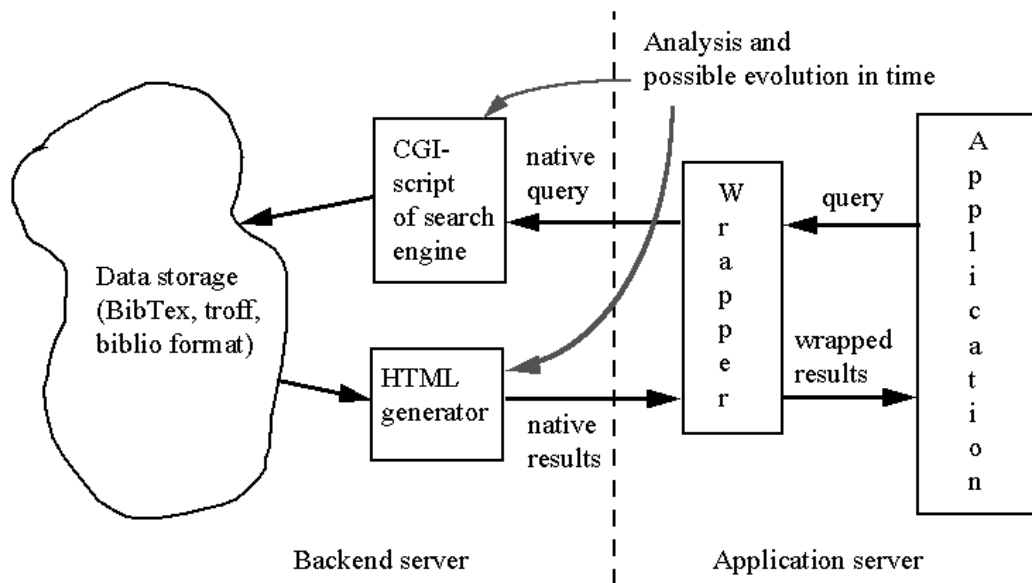


Figure 1. Typical scheme of the application - backend interaction

3.2 Html-Quality Level

This aspect is even more important than the previous one. The classification ranges from *high level* to *low level*. In existing information repositories, almost all variation between these two extreme classifications can be found.

- *High level*: each item in the result page is surrounded by a couple of html-tags, such as `--`. Each of these “tagged” items corresponds to *exactly one* attribute of the original (bibliographic) data (e.g., title, author, reference, body, etc.). The on-line catalogue of the IEEE-server (<http://www.computer.org/cgi-bin/wwwwais/>), for example, returns the following sentence:

```
<dt> <b>1: </b><a href="http://www.computer.org/cspres/catalog/proc3top.htm">
Conference Proceedings - Object-Oriented and Database Computing</a><dd> Score:
<b>1000</b>, Size: <b>1 kbytes</b>, Type: <b>html file</b>
```

- *Low level*: a string between two html-tags corresponds to *more than one* output attributes. In this case, some additional plain-text separators like “.”, “;”, “:” are used for separating the different attributes of the original (bibliographic) data. Some html-generators provide the “author” and the “title” attributes within a single “author-title” string. The separation of both attributes is context-dependent. Here, the analysis of the html-structure of the result pages is not enough, and we are faced with plain text analysis which is an unsatisfactory task!

4 Reliability Problem for a Result Page Analysis

Before we discuss different techniques of the Web wrapper implementation, we introduce here one important aspect common for all of them, namely the *wrapper reliability*. We explain the problem for the example of the rule-based parsing. It concerns the fundamental difference between the standard use of parsers [AhU172] and our use for html-parsing. In the standard use, a grammar-based parser is invoked to scan input files of a highly rigorous

structure with well-defined syntax and as free of inconsistencies as any program is. If such a program contains a syntax error, an error message is generated and no code is produced. Clearly, none of the above properties are usually verified for the data stored on the Web, and moreover no error messages or states are allowed for a Web information parser. *All* information (even with inconsistencies!) should be parsed and accepted. Therefore, these parsers have to be quite smart.

Another aspect of the reliability problem is that the analysis of the backend results is essentially *partial*. In practice, the process of the Web parser creation proceeds as follows. A set of *a priori* defined queries are generated with the goal to obtain the corresponding result pages and generate the parser which works for *these* pages. The set may be big or small, but cannot cover all data in the backend repository which may contain a huge amount of information. During the wrapper testing, if a parsing error happens with a new return page, the parser is adapted to properly handle that particular error. The wrapper testing goes on until one becomes confident that the wrapper works.

Therefore, the problem is to validate how reliable the parser is when new queries (not covered by the testing queries) are submitted to the backend. As obtaining an absolutely reliable parser by scanning the entire Web repository is unacceptable, there always exists a probability to have a parsing fault due to some inconsistencies not met in the previous result pages. Another reason why the 100% level of the parser reliability is never reached in practice is because information in most backends is continuously updated. Thus, the state of having all information correctly parsed becomes unreachable.

Since the probability of a parsing fault remains independent of how many queries have been successfully processed, the parser reliability should itself be expressed as a probability which gives to what extent the parser is reliable.

To estimate the parser reliability, we apply some basic statistical methods. During the testing phase, we provide the parser with a set of queries that are generated randomly. Given random queries for a new version of parser, we now assume that the results pages obtained for these queries have n items in total and all the pages are parsed accurately. Note that the less rigorously a page is structured, the more work is required to reach this level. Basically, the high reliability requires that the probability p that the parser will fail on a new result page should be very low. For the statistical evaluation, we apply the central limit theorem [MRT70] which states that the random variable for an input error

$$z = \text{sqrt}(p n / (1-p))$$

is approximately distributed according to the standard normal distribution. Therefore, we can calculate, for example, that it is enough to have about $n=350$ items in the successfully parsed result pages to be 95% confident that p is less than 0.03. In practice, the value n should be estimated from the level of confidence and the fault probability demanded by the user.

We can now sum up the *main requirements* of the wrapper design procedure:

- A wrapper should permit the analysis of the html-structure, and should, as explained before, look inside these structures.
- It should be open to possible backend modifications. As a consequence and in order to ease its maintenance, it should be as short and transparent as possible.
- It should be fast and highly reliable.

Within our project, we have looked at and implemented three main approaches to the backends result analysis: *manual coding*, *parsing by tag-marks*, and *specification by rules*. Let's now discuss these approaches and the lessons learned so far in more detail.

5 Wrapper Design

We start from the previous approaches used for the wrapper design and then move to the semiautomatic design based on the specification by rules.

```
<HR>
Bass, Leonard J.,
  A generalized user interface for applications programs (II),
<I>Commun. ACM</I> <B>28</B>, 6 (June 1985), 617-627.
<A HREF=/pubs/toc/Abstracts/cacm/3816.html>[Index Terms and Review]</A>
<P>
Berzins, Valdis, Gray, Michael and Naumann, David,
  Abstraction-based software development,
<I>Commun. ACM</I> <B>29</B>, 5 (May 1986), 402-415.
<A HREF=/pubs/toc/Abstracts/cacm/5691.html>[Index Terms and Review]</A>
<P>
Rushinek, Avi and Rushinek, Sara F.,
  What makes users happy?,
<I>Commun. ACM</I> <B>29</B>, 7 (July 1986), 594-598.
<A HREF=/pubs/toc/Abstracts/cacm/6140.html>[Index Terms and Review]</A>
<P>
Van Dyke, Carolynn,
  Taking "computer literacy" literally,
<I>Commun. ACM</I> <B>30</B>, 5 (May 1987), 366-374.
<A HREF=/pubs/toc/Abstracts/cacm/22901.html>[Index Terms and Review]</A>
<P>
Bilson, A. J.,
  Get into the groove : designing for participation,
<I>interactions</I> <B>2</B>, 2 (April 1995), 17-22.
<A HREF=/pubs/toc/Abstracts/interactions/205353.html>[Index terms]</A>
<HR>
```

Figure 2. html-page returned by the ACM publication server. The text in bold font highlights cases that requires specific attention in the rule-based approach due to the presence of special characters such as quotes, columns and question marks

In the remainder of this paper, we use a simplified example from the ACM publication server (<http://www.acm.org/pubs/toc/Search.html>). Figure 2 shows an html-fragment returned by the server. It represents a list of items, with each item corresponding to an article published in one of the ACM journals. Article information is given in four lines, where the first line denotes the author list and the second line denotes the title of the article. The third line shows the journal title, volume, issue, date and pages, whereas the last line stores the html-link to

additional information about the article. Using the classification given in Section 3, this html fragment has a rigorous structure of low html-quality.

5.1 Manual Coding Approach

This naive approach was used in our first prototype for unwrapping/wrapping data for the very initial backends. The code was written in Python and represented a simple step-by-step analysis of the result pages. The main drawbacks were that the code was quite long and that assumptions concerning the structure of the page to be parsed were spread all over the code. The maintenance of a wrapper got very complicated as it was almost impossible to determine what those assumptions were. On the other hand, even staff untrained in parsing could do the work.

```
// transition in state automata depending on state and tag name
//
void handleStartTag(Tag tag) throws IOException {
    String tagName = tag.getElement().getName();
    if (tagName.equals("p") && (_state == SKIP_SEP)) {
        _state = PARSE_HITS;
    } else if (tagName.equals("p")
        && (_state == PARSE_HITS || _state == HIT_DONE)) {
        _state = PARSE_TITLE;
    } else if (tagName.equals("i") && (_state == PARSE_TITLE)) {
        _state = PARSE_JOURNAL;
    }
}

void handleEndTag(Tag tag) throws IOException {
    ...
}

// called when (raw) text is encountered
//
void handleText(byte text[]) throws IOException {

    if (_state == PARSE_TITLE) {
        _title.append(new String(text, 0));
    } else if (_state == PARSE_JOURNAL) {
        _journal.append(new String(text, 0));
    } else if (_state == PARSE_VOL) {
        _volume.append(new String(text, 0));
    }
}
}
```

Figure 3: Subset of the ACM tag-marks wrapper

5.2 Parsing by Tag-marks

Parsing by tag-marks goes one step further than manual coding. It relies on a sgml-parser (implemented both in Python and Java) that spots html-tags within a page. The wrapper keeps track of the transitions from one tag to another and collects raw information between the tags.

Figure 3 shows a subset of a Java wrapper for the ACM publication server. It shows the main methods used to keep track of transitions and to extract information relative to an article. The extraction of the fields requires the definition of seven states, plus some additional post-processing to separate the authors from the title and to build the reference of the article.

This approach mixes a high-level description of the parser, i.e., the definition of transitions based on html-tags, with some low-level processing to extract information fields. It is difficult to follow the flow of the automata and some assumptions about the structure of the html-page are hard-wired in the code. People trained in parsing were needed to design wrappers and the expected simplicity was not reached. The maintenance remains difficult due to the excessive complexity and rigidity of code.

On the other hand, it is possible for such parsers to tolerate inconsistencies and exceptions in the html-page. Errors can be simply ignored; the parser just moves forward to the next tag.

5.3 Specification by Rules

The most advanced approach to the analysis of backend results and wrapper design is through the use of high-level text-processing tools on the base of grammar rules, like UNIX's lex and yacc. Lex generates programs to be used in simple lexical text analyzers for yacc. The source file for lex contains regular expressions to search for, and actions written in a language like C or C++ to be executed when such expressions are found. Yacc converts a context-free grammar into a set of tables for a simple automaton that executes an LALR parsing algorithm.

A first yacc-like parser constructor has been proposed for the Java language in [CUP96]. The corresponding manual describes the basic operation and the basic use of the Java Based Constructor of Useful Parsers (Java-CUP for short). Like yacc, Java-CUP is a system for generating LALR parsers from simple specifications. It actually offers most of the features of yacc. However, unlike the Unix yacc, Java-CUP (written in Java), uses specifications including embedded Java-code. Also, it produces parsers which are implemented in Java.

Recently, Sun has released its own tool for the high-level grammar specifications. The new tool is called JavaCC [JavaCC97] and is written in Java, too. It automatically generates parsers by compiling a high-level grammar specification stored in a text file. Like lex, it builds lexical analyzers from regular expressions, and like yacc, it reduces a grammar specification into a table-driven compiler that could produce code when it had successfully parsed productions from that grammar. The grammars JavaCC accepts are LL(k) grammars as opposed to the LALR grammars used by yacc. The power of automatic parser generation is that it allows users to concentrate on the grammar and not worry about the correctness of the implementation. This can be a tremendous time-saver in both simple and complex projects [McMa96].

We have found JavaCC to be a very powerful tool and used it in our project for the parser generation. To avoid going into details of the JavaCC syntactical agreements, we show the grammar for the ACM result pages in the BNF-like form (see Figure 4).

li-list	::=	"<HR>" article ("<P>" article)* "<HR>"
article	::=	authors "\n" title "\n" source "\n" reference
source	::=	"<I>" title-of-jrnl "</I>" "" volume "," issue "(" date ")" year "," pages
reference	::=	"[" moreinfo "]""
authors	::=	(<NAME>)+
title	::=	(<STRING>)+
name-of-jrnl	::=	(<STRING>)+
year	::=	<NUMBER>
volume	::=	<NUMBER>
issue	::=	<NUMBER>
pages	::=	<NUMBER> "-" <NUMBER>
url	::=	("/" <STRING> <NUMBER>)* ".html"
moreinfo	::=	(<STRING>)+

Figure 4: BNF-like definition for the ACM wrapper

The grammar has three types of elements.

1. The simplest elements are constant sub-strings from the source page used by the parser as tags. They may be the usual html-tags like "<HR>" and "<P>", or plain text characters like "," or "(".
2. The second group are tokens (<NUMBER>, <STRING>, and <NAME>) recognized by the lexical analyzer. A token is a sub-string from the source page which is defined by a regular expression as in the UNIX's lex.
3. Productions are given in lower-case and are divided in two subgroups:
 - Terminal productions correspond to output attributes (*authors*, *title* etc.). They have no other grammar productions on the right-hand side.
 - Nonterminal productions (*li-list*, *article*, *source* and *reference*) drive the parsing process and do not correspond to any output attribute.

Lexical analysis problems. The key problem in the design of an appropriate lexical analyzer is that it should be able to recognize any set of characters. Apart from the auxiliary characters like ":", "?", and different kinds of parenthesis, which occur frequently in article titles (see Figure 2), all possible special characters from other languages (German, French, Greek, etc.) may appear in both titles and authors' names, as well as in fragments of mathematical formulas and should be accepted by the analyzer. To handle this requirement, our analyzers use *negation methods* for the regular expressions more frequently than *direct methods*. The direct regular expression is when we indicate all characters *acceptable* in the token. In contrary, the negation method is when we indicate which characters are *not acceptable* with all other characters being acceptable. Negation allows using a much more compact and transparent definition of the grammar at no performance cost compared to the direct method.

We can now sum up the *main advantages* and *disadvantages* of the approach based on the specification by rules:

Main advantages:

1. It allows processing of any regular search result by a high-level grammar.

2. It is time efficient, because it generates Java-code that is then compiled.
3. The code is short.

Main disadvantages:

1. Since JavaCC is not html-oriented, a special lex-like analyzer for html-pages is missing and had to be implemented.
2. Creating rules for a new parser/backend requires some knowledge about the LR parsing and therefore is not easy for a staff untrained in parsers.
3. The current version of the parser is too rigid as compared to the approach based on sgml-parsers. It does not allow inconsistencies; it just stops and is a source of systematic errors. As a result, it is not able to ignore badly-formatted text and continue smoothly. We hope it will be possible to overcome such limitation in the following versions of the parser.

6 Implementation

Over the last year and an half, we have developed wrappers for more than 20 sources. For most sources, we have used the tag-marks approach and for some of them we have both a tag-marks version and a rule-based version. We have put of lot of emphasis on code reuse which is quite natural in such a context as most wrappers perform similar functions. On average the source-specific part of a tag-marks wrapper is less than 200 lines of code. The implementation time of a tag-marks wrapper ranges from two hours to a few days. The more structured and consistent the html-page is, the easier it is to wrap it.

Our experience with rule-based wrappers is more recent but shows a significant reduction of the development effort (assuming one masters the concepts of LR parsing). We have found 10 to 15 lines of productions with very few (1 to 3) nonterminal ones to be sufficient to define the main structure of the source page the wrapper expects to parse. We hope the maintenance efforts will be mainly reduced to updating those main rules.

In order to increase the amount of code reuse, we used an approach based on grouping and overlapping wrapper implementations. The main idea is to partition wrappers in groups on the basis of common features their return pages exhibit. Due to this approach, we obtain several so-called *group wrappers*. A group wrapper does not perform a connection to a particular backend. Instead, it is similar to a *parser class* which contains all data and functions common for a given group of backends. The simplest group wrappers are oriented on typical cases seen in most result pages:

1. *One-level one page result*: the search engine of the backend provides one page which contains all items related to the original query (e.g. *Databases and Logic Programming server*).
2. *One-level multi-page result*: the search engine of the backend provides a sequence of pages, that is, portions of the full answer set. A broker's wrapper has to follow "next hits"-links to get the full answer set (e.g. *AltaVista*).
3. *Two-level pages*: for each item in the first-level result page, the wrapper has to follow a "full info"-link in order to navigate to a result page that contains all data related to the query (e.g. *Elsevier Publisher server*, <http://www.elsevier.nl/cgi-bin/inca/search>).

A group wrapper class provides a unified way to scan the html-page returned by the backend. On the first step of the scan, the wrapper extracts the useful data from the return page and splits the data into pieces corresponding to items called *articles*. The articles themselves are processed during a second step. While the first step is the same for any backend in the group, the second step is backend-dependent. In other words, besides a group wrapper, and for each backend in a group, a short description for the processing of an article is prepared. Actually, the specification is designed in order to achieve two main goals:

- recognize the structure of the backend article, and
- associate data in the backend result with output attributes these data correspond to.

With the specifications by rules, the length of the description corresponding to any particular backend is very short (up to 10-15 lines of code).

Eventually, during the second step of the result page scanning, the group wrapper loads into the main memory an article specification corresponding to a given backend, and uses it for extracting and putting article data into corresponding output attributes.

The main benefits of the two-step approach lies in its simplicity and the ease of maintaining a large set of wrappers.

7 Conclusion

Controlled access to structured Web-based information across heterogeneous information repositories is a challenge for any broker (mediator/trading/facilitator) system.

In this paper we described the approach for sophisticated wrapper design as followed in the *Knowledge Broker* project. After a short problem description, we classified the target environment (html-pages) the wrappers work with, discussed some html-quality levels, and explained the reliability problem for the result page analysis. The wrapper design itself was described for three different approaches: the *manual coding* approach, the *parsing by tags*, and, finally, the *specification by rules*. Throughout the paper, we addressed the lessons learned during the course of the project.

Acknowledgments

We wish to thank Natalie Glance for the careful reading of the manuscript. We also wish to thank Jean-Marc Andreoli, Laurent Julliard, Remo Pareschi and Jutta Willamowski for helpful discussions and comments. We thank Michael Ley from University of Trier, Germany, for the help in working with the Databases and Logic Programming server.

References

- [ABP⁺95] J.-M. Andreoli, U. M. Borghoff, R. Pareschi, J. H. Schlichter: Constraint Agents for the Information Age. *J. Universal Computer Science* 1:12, 1995, 762-789. Electronic version available at <http://www.iicm.edu/jucs/>.
- [ABP96] J.-M. Andreoli, U. M. Borghoff, R. Pareschi: The Constraint-Based Knowledge Broker Model: Semantics, Implementation and Analysis. *J. Symbolic Computation*, 1996, in press.

- [AhUI72] A. V. Aho, J. D. Ullman: *The Theory of Parsing, Translation and Compiling*, 2 volumes, Prentice-Hall, 1972.
- [BoSc96] U. M. Borghoff, J. H. Schlichter: On Combining the Knowledge of Heterogeneous Information Repositories. *J. Universal Computer Science* 2:7, 1996, 512-532. Electronic version available at <http://www.iicm.edu/jucs/>.
- [BCW96] U. M. Borghoff, P.-Y. Chevalier, J. Willamowski: Adaptive Refinement of Search Patterns for Distributed Information Gathering. In: Verbraeck, A.: *Proc. Int. Conf. EuroMedia/WEBTEC*, 1996, London, U.K., December, San Diego: The Society for Computer Simulation, pp. 5-12
- [CGH⁺94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom: The Tsimmis Project: Integration of Heterogeneous Information Sources. In *Proc. IPSJ Conf.*, 1994, Tokyo, Japan, October, IEEE Press.
- [CUP96] CUP - Java Based Constructor of Useful Parsers. http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/, status May 1996.
- [GHI⁺95] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J., Widom: Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. *Proc. AAAI Symposium on Information Gathering*, 1995, pp. 61-64.
- [JavaCC97] JavaCC - Parser Generator in Java. <http://www.suntest.com/JavaCC/>, status Mar. 1997.
- [KnAm96] C. A. Knoblock, J. L. Ambite: Agents for Information Gathering. *To appear in Software Agents*, J. Bradshaw ed., AAAI/MIT Press, Menlo Park, CA.
- [McMa96] C. McManis: Looking for lex and yacc for Java ? You don't know Jack. (Jack is old name of JavaCC). <http://www.javaworld.com/javaworld/jw-12-1996/jw-12-jack.html>, status Dec. 1996.
- [MRT70] F. Mosteller, R. Rourke, G. B. Thomas: *Probability with Statistical Applications*. Addison-Wesley, 1970.
- [ODL93] K. Obraczka, P. B. Danzig, S.-H. Li: Internet Resource Discovery Services. *IEEE Computer* 26:9, 1993, pp. 8-22.
- [PMG⁺95] A. Puder, S. Markwitz, F. Gudermann, K. Geihs: AI-based Trading in Open Distributed Environments. *Proc. 3rd Int. IFIC TC-6 Conf. on Open Distributed Processing*, Feb. 1995, Brisbane, Australia. Chapman and Hall.
- [PuBu96] A. Puder, C. Burger: New Concepts for Qualitative Trader Cooperation. *Proc. Int. Conf. on Distributed Platforms*, 1996.
- [QRS⁺95] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, J. Widom: Querying Semistructured Heterogeneous Information. *Proc. Int. Conference on Deductive and OO Databases*, 1995.
- [SEK⁺92] M. F. Schwartz, A. Emtage, B. Kahle, B. C. Neuman: A Comparison of Internet Resource Discovery Approaches. *Computing Systems* 5:4, 1992, pp. 461-493.