Extraction and Integration of Data from Semi-structured Documents into Business Applications

Ph. Bonnet & S. Bressan

The Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02142

Extraction and Integration of Data from Semi-structured Documents into Business Applications

Ph. Bonnet
GIE Dyade, Inria Rhône-Alpes
655 Avenue de l'Europe
38330, Monbonnot Saint Martin, France
Philippe.Bonnet@dyade.fr

S. Bressan
MIT Sloan School of Management
50 memorial drive, E53-320
Cambridge, MA, 02139, USA
steph@context.mit.edu

1 Introduction

Collecting useful information along the information Autobahn is a fun *window shopping* activity which rapidly becomes frustrating. As the technology offers a broader logical connectivity, enables schemes for secure transactions, and offers more guarantees of quality, validity, and reliability, it remains difficult to manage the potentially available data, and to integrate them into applications.

Can I write a program which would assist me in planning my next business trip by retrieving relevant data about hotel prices, weather forecast, plane ticket reservations? How can I manage my investment portfolio without gathering and copying everyday, by hand, stock prices and analysts recommendations into spreadsheets?

On the one hand users can *browse* or *surf* the World Wide Web (under the name of which we liberally include all electronic information sources and protocols, e.g. WAIS, FTP, etc). Data is then embedded into HyperText Markup Language documents, Postscript, LaTeX, Rich Text Format, or PDF documents. These documents are meant for *viewer* applications, i.e application whose task is only to present them to end-users in a readable, possibly nicely laid out format. On the other hand, Electronic Data Interchange networks, which allow the integration of data into complex applications, remain specific to narrow application domains, operating systems, and most often proprietary applications.

Wiederhold, in [Wie92], proposed a reference architecture, the **mediation** architecture, for the design of heterogeneous information systems integration networks. As the reference for the *Intelligent Integration of Information* (I^3) DARPA program, it has been adopted by most American research groups and became a de facto standard for the research community. As several other projects (e.g. the SIMS project at ISI [AK92], the TSIMMIS project at Stanford [GM95], The Information Manifold project at AT&T [LSK95], the Garlic project at IBM [PGH96], the knowledge Broker project at Xerox [ABP96], or the Infomaster project at Stanford [GKD97]), the **Disco** project at GIE Dyade [TRV96, TAB$^+$97] and the **COIN** project at MIT [GBMS97, BGF$^+$97] implement this architecture.

The mediation architecture categorizes the mediation tasks into three groups of processes: the *facilitators*, the *mediators*, and the **wrappers**. The facilitators are application programming and user interfaces facilitating the seamless integration of the mediation network into end-users environments. The mediators provide the advanced integration services: query and retrieval over multiple sources, schematic federation, semantic integration, as well as the necessary metadata management. The mediator assumes physical connectivity and a first level of logical connectivity to the remote information sources. This initial connectivity is provided by wrappers. At the physical level a wrapper constitutes a gateway to a remote information source, being an on-line database, a program, a document repository, or a Web site. At the logical level, a wrapper masks the heterogeneity of an information source; it provides a mapping, suited to the mediation network, of the data representation, the data model and the query language.

Although the World Wide Web has departed from the initial raw distributed hypertext paradigm to become a generic medium for all kinds of information services including online databases and programs, most Web services, with the exception of Java and ActiveX based interfaces, provide information in the form of documents. The Web-wrapping challenge is to extract structured data from these documents. In fact, for large number of documents, it is possible to identify certain **patterns** in the layout of the presentation which reflect the structure of the data the document presents. These patterns can be used to enable the automatic identification and extraction of data. It is for instance the case when data is organized in Excel spreadsheet tables, HTML item-
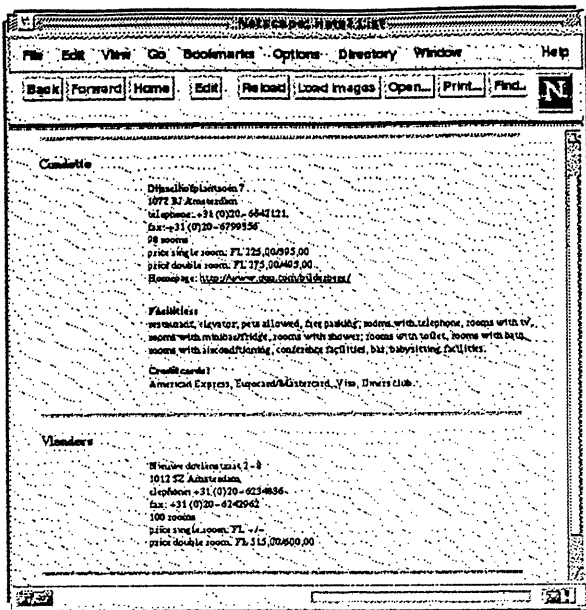
1

Figure 1: Sample Web Page

ized lists, or Word structured paragraphs. Furthermore, as documents are updated or as new documents are added, a relative perennity of the patterns can be assumed. We will call such documents **semi-structured documents** [Suc97]. Of course, it is a paradox to try and extract data from a document, which, most likely, was generated from the content of a structured database in the first place. However, such a database, when it exists, is generally not available.

We focus in this paper on the pattern matching techniques that can be used to implement generic wrappers for semi-structured documents; we discuss how Prolog can support these techniques.

2 Generic Wrappers

Let us consider the example document rendered on figure 1. The document is an HTML page containing a list of hotels together with rates, addresses, and descriptions of facilities. It is one example of the documents served by a Web hotel guide for the city of Amsterdam. This Web service has been wrapped with the COIN Web wrappers, and is now accessible as a relational database exporting the relation "h-ams", which contains attributes such as Name, the name of the hotel, Single, the rate for a single room, or Double, the rate for a double room. A wealthy user can now look for hotels whose rates for single rooms are higher than 250 Dutch Guilder. Table 1 shows an example of a query in SQL and its results as processed and presented by the COIN Web wrappers.

Web wrappers can either be specific programs de-

select Name, Single from h-ams where Single>250;
as of Tue Aug 26 12:34:12 1997; cols=2,rows=7

Name	Single
Amstel InterContinental	725
Amsterdam Renaissance	332
De l'Europe	475
Golden Tulip Barbizon Centre	375
Grand Hotel Krasnapolsky	350
Hilton Amsterdam	360
Holiday Inn Crown Plaza	345

Table 1: Query Results (as returned by the COIN wrappers Web query interface)

signed and implemented for interfacing a specific set of documents, or generic components intended to be reused for a variety of sets of documents. For specific wrappers, the patterns to be identified are generally defined once and for all, and the data extraction procedure is an optimal implementation of the matching of these patterns. For **generic wrappers**, the patterns are parameters in a **specification**. The specification is a declaration of what data need to be extracted, how it can be extracted, and how it should be re-structured and presented.

Figure 2 shows the architecture of the Disco [ABLS97] and COIN [BL97] generic Web wrappers. Queries are submitted to the wrapper interface. They are formulated in the wrapper query language (Object-relational algebra for Disco, SQL for COIN). The query is compiled, together with the relevant specifications (the specifications of the documents needed to answer the query). The planner/optimizer generates an execution plan. The plan is executed by the executioner. The plan describes what documents need to be retrieved by the network access component, which patterns need to be matched from individual documents by the pattern matching component, and how individual data need to be combined and presented.

3 Pattern Matching

3.1 Example

The basic wrapping problem is the definition of patterns which can be used to automatically and efficiently identify and extract data from the documents. The Web service we consider in our running example serves HTML pages. An excerpt of the source code for the HTML document 1 is presented in figure 3. A list of hotels with rates, addresses, and descriptions of facilities is presented on a single page in an HTML table.

A simple piece of data to extract is, for instance,

2

Query   *Results*

```
Interface

Query
Compiler/
Interpreter

Planner/          Executioner
Optimizer
                  Pattern Matching
                  Network Access
Specification
Compiler/
Interpreter
```
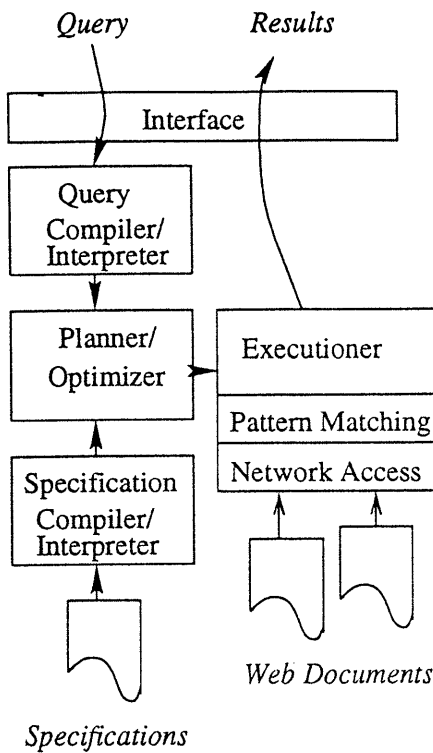
Web Documents

Specifications

Figure 2: Generic Web Wrapper Architecture

the name of the hotels in the list. Taking advantage of the text formatting, namely of the HTML tags, we decide to rely on the fact that hotel names are always between <STRONG> and </STRONG> tags. We also make sure that any text in such a block is a hotel name. As it is the case, a simple regular expression[1] matches the hotel names and tags and stores them in variables, called back references.

For instance the following is a Perl program that returns the list of hotel names as specified above ($d is a string that contains the source code of the HTML document):

```
<1> @r = ($d =~ /<STRONG>(.*?)<\/STRONG>/sg);
<2> foreach $e (@r)
<3>{print $e; print "\n";}
```

The string is matched with the regular expression on line <1>[2]. For each succefully matched substring, the portion corresponding to a part of the regular expression in parenthesis is stored in a variable, called a back reference. Several matches are obtained by the option s and g on line <1> indicating that the string is to be treated as a single sequence (as opposed to

---

[1] We use the Perl regular expression syntax unless otherwise specified

[2] ".*+" and ".*?" are, respectiveley, the greedy and lazy iteration quantifiers. For instance, ".*?" at the begining or at the end of a regular expression matches nothing and is therefore useless. For instance, ".*" at the beginning or at the end of a regular expressions matches as much as possible, and therefore limits the number of matches to at most one.

```
<TR><TD WIDTH="450" COLSPAN="8">
<IMG WIDTH="444" HEIGHT="3" vspace="15"
ALT="line" SRC="streep.gif"><BR>
<FONT SIZE="3">
<STRONG>Condette</STRONG>
</FONT><BR>
<IMG vspace="2" SRC="dot_clear.gif"><BR>
</TD></TR>
<TR><TD WIDTH="25"><BR></TD>
<TD WIDTH="425" COLSPAN="7">
<FONT SIZE="2">Dijsselhofplantsoen 7<BR>
1077 BJ  Amsterdam<BR>
telephone: +31 (0)20-6642121<BR>
fax: +31 (0)20-6799356<BR>
98 rooms<BR>
price single room: FL 225,00/395,00<BR>
price double room: FL 275,00/495,00<BR>
```

Figure 3: Sample Web Page (Excerpt of the HTML Source)

a sequence of lines), and that matching should be iterated, respectively. The list of back references is stored in the array @r. The loop (line <2>) prints the values in the cells of the array (line <3>). A Similar program can be written in Prolog, for instance, with a Definite Clause Grammar (DCG) (taking its input from a list of characters):

```
hotel_name(Name) --> anything(_),
    ["<","S","T","R","O","N","G",">"],
    anything(Name),
    ["<","/", "S","T","R","O","N","G",">"].
anything([]) --> [].
anything([X|L]) --> [X], anything(L).
```

The use of pattern matching as opposed to a complete parsing of the document introduce a form of robustness in our specification. Indeed, other parts of the document, which can be free text, will have little impact on the extraction as long as they do not contain a matching string. Therefore they can vary without compromising the behaviour of the wrapper.

In order to extract additional data about the individual hotels, for instance, the name, together with the (low and high) rates for single and double rooms, we need to use a more complex pattern with three back references. In Perl, the pattern could look like:

```
<STRONG>(.*?)<\/STRONG>.*?\
single room: FL (.*?)/(.*?)<BR>\n\
price double room: FL (.*?)/(.*?)<BR>
```

As we use more complex expressions, relying on more elements in the structure of the document to anchor our patterns, the robustness of the specification decreases.

Regular expressions seem to be a good candidate for a pattern specification language. However, as

3

we shall see, we need to understand their expressive power to decide wether they fit our needs in a sufficient number of situations. It must be clear, for instance, that a regular expression cannot understand the notion of nested blocks. In addition, we should not be confused by the fact that `<STRONG>`, `</STRONG>` is an HTML block. Indeed, only a particular control of the backtracking in the pattern matching of our Perl program (part of it is expressed by the ? after the *) will prevent the matching of substrings of the form `<STRONG>.<\/STRONG>.<\/STRONG>`. A regular expression language with negation (regular sets are closed under difference) could be used to exclude the `<\/STRONG>` sequence from the innermost match.

In summary, the pattern description language needs to allow the synthetic (concise and intuitive) and declarative expression of the patterns. It needs to be expressive enough to cover a satisfying class of patterns. Together with its robustness these qualitative properties will allow to handle documents whose content vary and will ease the maintenance of wrappers. Wrappers are most likely to be used as just-on-time services for the extraction of data. The extraction process needs to be efficient and avoid the pitfall of combinatorial complexity of the general pattern matching problem. As we shall see, this issue is mainly related to the control of backtracking.

Several tools and languages provide support for parsing and pattern matching[3]. Several unix commands such as grep, or ls, and many text editors, such as ed, or emacs support find- or replace-functions based on regular expressions pattern matching. SNOBOL4[4], Perl, or Python provide a native support for regular expressions and pattern matching. Regular expression pattern matching tools are announced for JavaScript. C (with its compiler-compiler tools Lex, Flex, Yacc, and Bison, or regular expression libraries such as the gnu regex),and Java (with JLex, Jacc, or regular expression libraries) also provide the necessary support for standard parsing and pattern matching. However, these tools do not offer the necessary flexibility in the control of the backtracking and are not as easily customizable as Prolog DCG. In the next sections we will study pattern matching from the point of view of its implementation with Prolog DCG.

3.2 Back to the Basics

A **formal language** [HU69] L is a set of words, i.e. sequences of letters or tokens, from an alphabet A (the empty sequence is noted ϵ). A language described in extension, by the enumeration of its elements, is of little practical use. Instead, languages can be described by generators or recognizers. A **generator**, for instance a grammar, is a device which describes the constructions of the words of the language. A **recognizer** is a device which, given a word of the language, answers the boolean question wether the words belongs to the language. In addition, a **transducer** is a device which translates a word of the language into a structure (e.g. compilers are transducers).

If a string s is a sequence of elements of the alphabet A, string pattern matching can be formally defined by the recognition of word w of a formal language L as a prefix of the string. If . stands for the language of words composed of a single element of A, $L*$ for the language of sequences composed by concatenation of any number of words of L, and $L_1 L_2$ for the language of sequences composed by the concatenation of any pair of words of L_1 and L_2, respectively, then the matching of a pattern L is the recognition of the language $. * L$ as a prefix of the string s.

A grammar, or *generative grammar*, is mainly a set of productions rules (grammar rules) whose left and right-end side are composed of non-terminal symbols (variables) and terminals (letters or tokens). Chomsky has proposed a four layer hierarchy of languages corresponding to four syntactically caracterizable sets of grammars: *the type 0* languages (almost anything), *the context sensitive* languages (e.g. Swiss German, cf. [GM89], or $a^n b^n c^n$), *the context free* languages (e.g. block languages, most programming languages), and the *regular* languages (language generated by regular expressions).

There are three main categories of recognizers: automata and machines, transition network, and definite clause grammars. In all categories we find a one to one mapping between types of recognizers, language classes in the Chomsky hierarchy, and grammar types. The recognizers as automatic devices or decision procedure give a framework to study not only how to implement parsers for a language but also for studying the inherent complexity of a language and therefore the cost of its parsing. DCG is a powerful formalism: DCG have the syntax of generative grammars but they correspond to Prolog programs which can serve as both generators and recognizers. In addition, the adjunction of arguments to the grammar terms allows the DCG to implement transducers.

A simple way to implement a generic pattern matcher for any language L using DCG is given in [O'K90]:

```
match(Pattern) --> Pattern | ([_], match(Pattern)).
```

---

[3]A practical survey of regular expression pattern matching in standard tools and languages can be found in [Fri97]

[4]SNOBOL was the first language to provide regular expression macthing and backtracking [GPI71]!

4

The pattern `Pattern` is to be replaced by the actual root of the DCG recognizing the language L.

Unfortunately, the problem of matching a pattern is combinatorial by nature when its output is all the matching substrings and their respective positions. For regular expressions it is quadratic. For instance, matching the regular expression a+ in a string of n a can be done in about $\frac{(n^2+n)}{2}$ different ways. If the question compels a boolean answer: "does there exist a match?", the complexity can be dramatically reduced by only looking for the first match. This is however not satisfactory if the goal of the matching is to return back references. A usual compromise is to look for matching substrings which do not overlap: if s is a sequence composed of two strings s_1 and s_2, $s = s_1 s_2$, and s_1 is a word of the language $.*L$, the matching is recursively iterated on s_2. This standard behaviour of string pattern matching algorithm corresponds to the following Prolog program (which becomes slightly more complicated if we add provision for the management of back references):

```
matches(P) --> match(P),!,result(P).
result(P) --> [].
result(P) --> matches(P).
```

In regular expressions, there are two other sources of non determinism: the alternation (|) and the iteration *. The non determinism of these constructs is absolutely clear when the expressions are translated into DCG rules; It correspond to backtracking opportunities. For instance, an alternation of the form a|b becomes:

```
pattern --> [a] | [b].
```

An iteration of the form a*, resp. a*?, becomes the following DCG `greedy_star`, resp. `lazy_star`:

```
greedy_star --> ([a], greedy_star) | [].
lazy_star   --> [] | ([], lazy_star).
```

There are three main techniques to reduce the non-determinism of a DCG: indexing, elimination of the ϵ-transitions, and state combination. Indexing consists in preventing the DCG from backtracking on the reading of input. For instance, the DCG for a|b can be rewritten:

```
pattern --> [X], pattern(X).
pattern(a) --> [].
pattern(b) --> [].
```

ϵ-transitions do not read any token from the input. They only indicate possible continuations. For instance, in the DCG given in the next section for the automaton on figure 4, states `state2` and `state5` can be merged. Finally, by combining states which have the same input one can reduce the non-determinism, and eventually eliminate it for regular languages. The drawback is the possible combinatorial growth of the number of states as new states are created from the power set of the original set of states.

It is possible to use DCG to implement efficiently generators, recognizers, and transducers. Regular expressions can be optimized and compiled into efficient DCG. We shall use these devices for implementing pattern matching tools in Prolog.

4 Prototyping Generic Wrappers In Prolog

The Disco and the COIN projects are sharing a Prolog-based prototyping platform for the experimentation and testing of Web wrappers. The platform is implemented in ECL^{i}PS^e Prolog. The Web wrapper prototyping environment follows the general wrapper architecture illustrated in figure 2. The query compiler, the planner optimizer and the relational operators of the executioner have been adapted from the COIN mediator components. An overview of the design and Prolog implementation of these components is given in [BFM+97]. The network communication layer of the executioner, necessary to access documents over the World Wide Web is build from the authors ECLiPSe HTTP library [BB96].

Of interest in this paper are the libraries developed for the compilation and execution of pattern description language. These libraries implement standard algorithms for the construction, optimization, and simulation of regular expressions. The regular expressions are transformed into Finite State Automata, the automata are optimized and compiled into DCG. The DCG can be combined with non regular parsers.

A Finite State Automaton is encoded as a DCG. For instance, the automaton of figure 4, representing the regular expression a(a|ab)*a, corresponds to (e.g.) the following DCG:

```
state0 --> [a], state1.
state1 --> state3 | state5 | ([a], state2).
state2 --> state5.
state3 --> [a], state4.
state4 --> [b],state5.
state5 --> state1 | ([a], state6).
state6 --> [].
```

A procedure compiles the regular expression into the DCG representation. Various procedures can the be applied to optimize the automaton: elimination of the ϵ-transitions, elimination of redundant or inaccessible states, determinization. Finally, the automaton is given a name and is compiled as a DCG.

Most importantly, we allow the token classes used in the regular expressions to correspond to other DCG. This simple mechanism enables the combination of regular and non regular parsers into regular
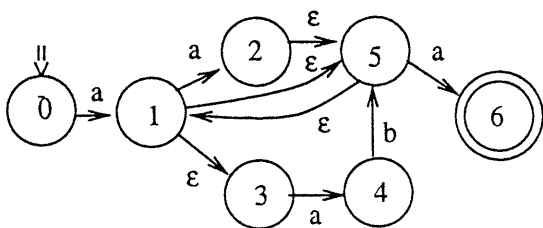
Figure 4: Finite State Automaton

expressions. It also enables the re-use of existing tokenizers (e.g. a tokenizer for HTML) by modularizing the code.

Finally, although we have concentrated on string pattern matching, we would like to remark that a simple modification of the input stream of the DCG allows to manipulate Directed Acyclic Graphs (DAGs). Let us assume, for instance, that the DAG composed of the two paths abcd and aefd is represented by the nested list structure [a,[[b,c],[e,f]],d]. We only need to trivially modify the 'C'/3 procedure in order for the DCG to parse paths in the DAG. Such a simple extension can be very useful for Web documents as argued in [PK95]. The nested structure can not only represent itemized lists or tables but also acyclic networks of hypertext documents linked by their hyperlinks.

5 Commercial and Prototype Wrappers

In this section we present five different Web wrappers. They are representative of three different approaches. EdgarScan is a specialized wrapper designed, programmed and tuned for specific documents of a specific application. The COIN and the Disco Web wrappers are generic wrappers. They compromise with the expressive power of the specification language to protect its declarativeness and minimize the programming task. TSIMMIS, and OnDisplay combine the declarative specifications with programming hooks to create a wrapper development environment. In fact many more projects (e.g [CBC97, KNNM96]) are developing wrapper development environemnts as a set of libraries for retrieving, parsing, and extracting data from documents. The set of ECLiPSe tools we have developed for the prototyping of wrappers fall into the latter category.

5.1 EdgarScan

EdgarScan [Fer97] is an application developped at the Price Waterhouse Technology Centre for the efficient and accurate automatic parsing of financial statements. In the United States, the Securities and Exchange Commission (SEC) has made compulsary for US corporation to file a variety of financial documents. This documents are stored and publicly available in the EDGAR database. Guidelines for what information is to be presented are stricter than guidelines for how information should be presented. The disparity of accounting practices and the specificity of each situation has prevented a structuration that can easily be used to automatically process the content of the filings. In these filings, accounting tables are structured summary of the main figures of a company. They are of highest interest for the financial analysts. Even in those tables, most data are accompanied by footnotes in plain english commenting on the definition of individual items, columns or rows, and necessary for a correct interpretation of the data. EdgarScan is specialized in the parsing of accounting tables. It extracts (using a C regular expression pattern matching library) and reconcile (using Prolog-based expert system) the data in the tables.

5.2 Centerstage

Centerstage [Ond97] is a commercial wrapper development and deployment tool kit. It operates as a Plug-in of the Internet Explorer. The kit not only comprises tools for the wrapper definition but also facilities to export data in a variety of desktop applications such as Excel, Word, etc. Additional server and driver components allow to interface the resulting data with Web or ODBC front-end applications. The wrapper definition and testing environment is based on a rich point-and-click interface: the user select the data to be extracted on a example Webpage and composes a pattern "by example". The pattern is then compiled into a wrapper agent, i.e. a JavaScript program which is able to extract the data. The agent program can be further edited and refined by the user. Our tests have shown that the pattern matching technique used by Centerstage is similar to the à la Perl regular expression pattern matching.

5.3 TSIMMIS

The Data model of the TSIMMIS mediation network [GM95] is the Object Exchange Model (OEM). OEM is a complex object model which allow the flexible manipulation of complex data without the constraint of a rigid schema. The TSIMMIS wrappers (in [Suc97]) extract OEM objects from Web documents. The specification consist in a sequence of commands of the form [variables, source, pattern], which indicate the construction of an OEM object as the results of the matching of the pattern on the

6

source string. The variables can be used as sources for other commands, implicitly defining a nested object. The source can be the result of any primitive or function of the underlying language (Python) which produces a string (e.g. http-get, split). Interestingly, the commands are similar to (regular) grammar rules. The programmer can sometimes choose between regular expressions pattern matching and rule programming depending on the structure she wants to confer to the OEM object. Unfortunately, the semantics of the command language is not clearly defined (e.g. w.r.t backtracking). The semantics of the regular expression pattern matching is the one of the implementation language Python.

5.4 COIN

The COIN web wrappers [BL97] provide a relational interface to Web services. Web and ODBC front-ends are available and used in COIN mediation prototype [BGF+97].

The relational schema chosen by the wrapper designer to be the interface of a particular service is defined in terms of individual relations corresponding to the data extracted from a set of documents defined by the pattern of an HTTP message (method, URL, object-body). The semantics of the relation exported is defined under the Universal Relation concept: attributes of the same name are the same. When a query (in SQL) incomes, the wrapper creates and executes a plan accessing the necessary pages. A relation can be composed of both static and dynamic documents (e.g. document generated by cgi-programs and accessible through forms) in a very natural way (by automatically "filing the forms" with data from the query or extracted from other pages), making the COIN wrappers uniquely powerful.

To each relation exported by a given site, correspond a specification file. The file contains for each set of documents patterns described in a simple regular expression language. The COIN pattern definition language is currently limited to plain regular expressions, however because it is simple, it allows the very rapid development and maintainance of simple but practical wrappers.

The COIN wrappers have been prototyped in Prolog and Perl, they are currently ported to Java.

5.5 Disco

Disco wrappers, called adapters, provide an object-relational interface to programs, services and documents [ABLS97]. They are used in the Disco mediation prototype [TAB+97]. The extraction of information from a semi structured document is performed in the pattern matching component of the

Disco adapter. A stand-alone prototype, InfoExtractor (in [Suc97]), was developed to validate and refine the Disco information extraction approach: a document abstract model is used in which the document is hierarchically partitioned into regions which contain concepts of interest.

For each document, the configuration describes an abstract model of the document, i.e the hierarchical partitioning of the document into regions, concepts and subconcepts. To a concept is associated a regular expression and a set of weighted keywords. If the set of keywords is not empty, the match is accepted only if a function of the weights is above a given threshold. This feature provides a unique combination of pattern matching and information retrieval techniques and increases the robustness of the wrappers. The output of the pattern matching component is a tree of concepts and subconcepts.

The Disco wrappers have been prototyped in Prolog and Perl, they are now implemented in Java.

6 Conclusion

We believe in and we will contribute to the development of a network of Web wrappers offering a structured access to semi-structured Web documents. The application domains which can benefit by the just-on-time availability of data are numerous. As an illustration of the potential, the directories and catalogs registered in the Yahoo thesaurus already contain large amounts of useful information for professional and individuals waiting to be automatically integrated into decision support processes.

Although in many cases, ad-hoc solutions will need to be programmed, we believe that a trade-off can be found for the definition of a generic tool for the easy development of Web wrappers. The generic Web wrapper architecture we have described relies on a pattern description language and on pattern matching techniques. Prolog and Logic programming in general, as platforms for the implementation of symbolic manipulation programs, seemed to be a perfect match. Indeed, ECL^{i}PS^e proved to be a very good tool for the rapid prototyping and experimentation of Web wrappers. However, to reach the efficiency requirements of industrial strength code, designers and programmers are needed who have high programing skills [O'K90]. This is for instance the case for the efficient implementation of the parsing and pattern matching algorithms. At this level of fine tuning, the suitability of Prolog versus C or even Java is questionned. Furthermore, some limitations of most existing Prologs, such as the lack of support for concurrent programming, is a serious drawback for the implementation of network applications. Neverthe-

less, as a rapid prototyping tool, Prolog brings a competitive advantage in a situation where development cycles are constantly shrinking due to the pressure of competition in a global distribution infrastructure.

References

[ABLS97] R. Amouroux, Ph. Bonnet, M. Lopez, and D. Smith. The design and construction of adapters for information mediation. (Submitted), 1997.

[ABP96] J.M. Andreoli, U. Borghoff, and R. Pareshi. The constraint-based knowledge broker model: Semantics, implementation, and analysis. *J. of Symbolic Computation*, 1996.

[AK92] Y. Arens and C. Knobloch. Planning and reformulating queries for semantically modelled multidatabase. In *Proc. of the Intl. Conf. on Information and Knowledge Management*, 1992.

[BB96] S. Bressan and Ph. Bonnet. The ECLiPSe http library. In *Proc. of the Intl. Conf. on Indutrial Applications of Prolog*, 1996.

[BFM+97] S. Bressan, K. Fynn, S. Madnick, T. Pena, and M. Siegel. Overview of the prolog implementation of the context interchange mediator. In *Proc. of the Intl. Conf. on Practical Applications of Prolog*, 1997.

[BGF+97] S. Bressan, C. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The context interchange mediator prototype. In *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data*, 1997. (demo track).

[BL97] S. Bressan and T. Lee. Information brokering on the world wide web. In *Proc. of the Webnet World Conference*, 1997. (To be published).

[CBC97] B. Chidlovskii, U. Borghoff, and P.-Y. Chevalier. Towards sophisticated wrapping of web-based information repositories. In *Proceedings of the 5th International RIAO Conference*, Montreal, Canada, 1997.

[Fer97] D. Ferguson. Parsing financial statements efficiently and accurately using c and prolog. In *Proc. of the Intl. Conf. on Practical Applications of Prolog*, 1997.

[Fri97] J. Friedl. *Mastering Regular Expressions*. O'Reilly and Associates, Inc., 1997.

[GBMS97] C. Goh, S. Bressan, S. Madnick, and M. Siegel. Context interchange: New features and formalisms for the intelligent integration of information. Technical Report CISL WP97-03, MIT Sloan School of Management, 1997.

[GKD97] M. Genesereth, A. Keller, and O. Duschka. Infomaster: An information integration system. In *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data*, 1997. (demo track).

[GM89] G. Gazdar and C. Mellish. *Natural Language Processing in Prolog*. Addison Wesley, 1989.

[GM95] H. Garcia-Molina. The TSIMMIS approach to mediation: Data models and languages. In *Proc. of the Conf. on Next Generation Information Technologies and Systems*, 1995.

[GPI71] R. Griswold, J. Poage, and Plonsky. I. *The SNOBOL4 programming Language*. Prentice Hall, 1971.

[HU69] J. Hopcroft and J. Ullman. *Formal languages and their relation to automata*. Addison-Wesley, 1969.

[KNNM96] Y. Kitamura, H. Nakanishi, T. Nozaki, and T. Miura. Metaviewer and metacommander: Applying www tools to genome informatics. In *7th Workshop on Genome Informatics*, 1996.

[LSK95] A. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 1995.

[O'K90] R. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[Ond97] Ondisplay. centerstage. http://www.ondisplay.com, 1997.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. of the Intl. Conf. on Parallel and Distributed Information Systems*, 1996.

[PK95] K. Park and D. Kim. String matching in hypertext. In *Proc. of the Symp. on Combinatorial Pattern Matching*, 1995.

[Suc97] D. Suciu, editor. *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, 1997. http://www.research.att.com/ suciu/workshop-papers.html.

[TAB+97] A. Tomasic, R. Amouroux, Ph. Bonnet, O. Kapistskaia, H. Naacke, and L. Raschid. The distributed information search component (disco) and the world wide web. In *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data*, 1997. (demo track).

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous database and the design of Disco. In *Proc. of the 16th Intl. Conf. on Distributed Computing Systems*, 1996.

[Wie92] G. Wiederhold. Mediation in the architecture of future information systems. *Computer*, 23(3), 1992.