Wrapper Application Generation for Semantic Web: An XWRAP Approach

A Thesis
Presented to
The Academic Faculty

by

Wei Han

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
2003

Wrapper Application Generation for Semantic Web: An XWRAP Approach

Approved by:

Professor Ling Liu, Adviser	Professor Santosh Pande
Professor Calton Pu, Co-adviser	Dr. Terence Critchlow (Lawrence Livermore National Laboratory)
Professor Olin Shivers	

Date Approved _____

To my supportive wife, Loretta, who has come all the way from San Francisco to cook for me.

To my dear parents, who have devoted all their effort to make me successful.

ACKNOWLEDGEMENTS

This dissertation contains the results of my Ph.D. research at GT. I am grateful to the many people who offered their help and their commentary to my work during these years. I would especially like to thank my advisor (Ling Liu) and co-advisor (Calton Pu) for giving me the opportunity to do this work, for the stimulating discussions that forced me to better explicate my ideas, for their constant support and consideration during my completion of the kind of research reported here, and for the freedom I experienced while working on this research project. I am greatly indebted to them for their guidance, encouragement and support, which I feel honored to have received during the past years.

I wish to thank Professor Pande, Shivers, and Dr. Critchlow for taking part in my PhD defense committee. Their critical reading and penetrating remarks have made the final revisions both necessary and effective.

I have also benefited by the comments on various portions of the work reported here from David, Wei Tang, Henrique, Dan, and the participants in the conferences and workshops in which I have participated.

I wish to take this opportunity to acknowledge the financial subsidies I received from the DOE, DARPA, NSF, and GT for their support of my research work and for the assistance in travel to various conferences and workshops across the US.

Last, but not least, I wish to give my thanks to my family, especially my wife, for their understanding and love. I thank all my friends and colleagues who helped me in one way or another and who made my experience in Atlanta a memorable one.

Since the manuscript went through a number of revisions, none of the people acknowledged should take responsibility for any remaining errors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The Semantic Web can be seen as an extension of the current web in which information is given well-defined meaning, enabling machines to make more sense of the Web. The grand vision of the Semantic Web is to make Web data machine-processable, and rests on the premise that this can be achieved by accommodating semantics of the Web data. For the Semantic Web to reach its full potential, there is a demand for technologies capable of capturing, extracting, exploiting, and utilizing semantics of Web data. Example technologies are languages for semantic annotation of Web documents, ontology, semantic interoperation of programs that have been developed totally independently, technologies and methodologies for describing, searching and composing Web Services.

One of the fundamental building blocks in these technologies is application generation of wrappers (executable programs), specialized in extracting and annotating interesting information from heterogeneous Web data sources. Most of the wrappers used in the Semantic Web (and Web services) today are programmed either manually or by wrapper generation systems supervised by human experts in wrapper production. With the rapid growth of Web data and the number of Web services available today, and the frequent changes of Web data sources in content, presentation, and delivery format, generating wrapper applications by hand or by intensive supervision does not scale. Several attempts have been made to explore automatic application generation techniques for generating wrapper programs on demand. It is widely recognized that application generation improves programming productivity and reduces human errors without sacrificing code efficiency. Most opportunities for application generation occur in domain specific applications. When a programming activity is well-understood and clearly defined, application generation can greatly reduce rote coding and tedious debugging.

This dissertation work is dedicated to develop a systematic framework and a suite of system-level facilities for automatic application generation. We refer to these generated

applications as wrappers, which perform transformation, semantic filtering, and semantic annotation of Web information. We present the XWRAP design philosophy, methodology, and the engineering techniques for information extraction application generation. We demonstrate the benefits and unique features of the XWRAP framework and XWRAP techniques for extracting and annotating Web data through three prototype systems:

1. XWRAP Original: a semi-automatic wrapper generation system for semi-structured Web data. XWRAP Original can generate wrappers for most of Web sites within a couple of hours, and it outperforms other wrapper generation systems in terms of the diversity of the Web pages it can handle.

2. XWRAP Elite: a Web based application generation system that supports fully automated information extraction and wrapper generation for object-rich Web pages. XWRAP Elite, to our knowledge, is the only wrapper generation system that allows users to generate wrappers in Java code and Java classes on the Web in minutes.

3. XWRAP Composer is the first application generation toolkit that supports information extraction and information aggregation for multiple pages from different Web sites. The unique features of XWRAP Composer are the interface and outerface description languages, and the Composer scripting language for encoding the workflow dependency, the flow control logic, and extraction methods of the wrapper programs and semantic annotation of the Web data to be wrapped.

XWRAP technology has been used and tested by over 500 registered users as of January 2003 and has generated more than 2000 wrappers. Compared to manual approach and other existing semi-automatic wrapper generators, XWRAP technology provides a number of competitive advantages: First, XWRAP Elite can generate wrappers in minutes with code quality and efficiency equivalent to Human experts. Second, XWRAP systems support a variety of transformations of Web data, including HTML to XML, XML to XML, plain text to XML, and a set of information filters. Third, XWRAP Composer technology is the only wrapper generator to date that is capable of extracting, aggregating, and

filtering information from multiple Web pages with workflow dependency. In comparison, most existing wrapper generation tools are limited to wrapping information from individual Web documents, one at a time. This dissertation also reports the extensive experiments conducted on XWRAP systems, showing the efficiency, trade-offs, and code quality of the XWRAP wrapper applications.

CHAPTER I

INTRODUCTION

There has been a lot of work on creating a new breed of distributed web services, typically composed from component software programs or so-called agents, such as supply-chain management and E-commerce. Becoming a commonly accepted data exchange standard on the Internet, XML allows distributed applications to be integrated into sophisticated services. However, much of the valuable information on the Web is and continues to be in the immediate future, in "human-oriented" HTML documents.

A popular approach to access Web pages by software programs is writing wrappers to encapsulate the access to sources and produce a more structured data, such as XML, to enable a number of applications such as electronic commerce [34, 53, 54, 25] to integrate. (This is sometimes called *screen scraping*.) A wrapper is a software program specialized to a single data source (such as a Web page), which converts the source documents and queries from the source data model to another, usually a more structured, data model [49]. Several projects have implemented hand-coded wrappers for a variety of sources, such as DIOM [48, 52], Garlic [24], TSIMISS [44], InfoSleuth [7], and Ariadne [33].

However, developing and writing such a wrapper cost a lot of effort due to the irregularity, heterogeneity, and frequent updates of the Web data and their presentation formats. Hand-coding wrappers becomes a major drawback in situations where it is important to gain access to new Web sources or frequently changing Web sources. We observe that, with a good design methodology, only a relatively small part of the wrapper code deals with the source-specific details, and the rest of the code is either common among wrappers or can be expressed at a higher level, more structured fashion.

There are a number of challenging issues in automation of the wrapper construction process.

- First, most Web pages are HTML documents, which are semi-structured text files

annotated with various HTML presentation tags. Due to the frequent changes in presentation style of the HTML documents, and the lack of semantic description of their information content, it is hard to identify the content of interest using common pattern recognition technology such as string regular expression specification used in LEX and YACC.

- Second, wrappers for Web sources should be more robust and adaptive in the presence of changes in both presentation style and information content of the Web pages. It is expected that the wrappers generated by the wrapper generation systems will have lower maintenance overhead than handcrafted wrappers for unexpected changes.

- Third, wrappers often serve as interface programs and pass the Web data extracted to application-specific information broker agents or information integration mediators for more sophisticated data analysis and data manipulation. Thus it is desirable to provide a wrapper interface language that is simple, self-describing, and yet powerful enough for extracting and capturing information from most of the Web pages.

With these issues and objectives in mind, we have started a research project called XWRAP, aiming at developing an adaptive framework for semi-automatic construction of wrappers for Web information sources, and implementing a number of prototype software system to demonstrate the validity and the usefulness of our proposed frameworks. The goal of the XWRAP development can be informally stated as the transformation of "difficult" HTML input into "program-friendly" XML output, which can be parsed and understood by sophisticated query services, mediator-based information systems, and agent-based systems.

The first two XWRAP systems are XWRAP Original and XWRAP Elite. XWRAP Original is an interactive wrapper generation system for semi-structured Web data. XWRAP Original can generate wrappers for most of Web sites within a couple of hours, and it outperforms other wrapper generation systems in terms of the diversity of the Web pages it can handle. XWRAP Elite is a Web based application generation system that supports fully automated information extraction and wrapper generation for object-rich Web pages. XWRAP Elite, to our knowledge, is the only wrapper generation system that allows users

to generate wrappers in Java code and Java classes on the Web in minutes.

We have observed that wrappers generated by wrapper generators, such as XWRAP Original or XWRAP Elite, usually perform well when extracting information from individual documents (single web pages) but are poorly equipped to extract information from multiple linked Web documents. An obvious reason for the inefficiency is due to the lack of system-wide support in the wrapper generators. For example, a wrapper in XWRAP Elite or Original can only perform information extraction over one type of Web document, while typical bioinformatics sources use several types of pages to present their information, including HTML search forms and navigation pages, summary pages for search results, and strictly formatted pages for detailed search results.

Information extraction over multiple different pages imposes new challenges for wrapper generation systems due to the varying correlation of the pages involved. The correlation can be either horizontal when grouping data from homogeneous documents (such as multiple result pages from a single search) or vertical when joining data from heterogeneous but related documents (a series of pages containing information about a specific topic). Furthermore, the correlation can be extended into a graph of work-flows. A multi-page wrapper not only enriches the capability of wrappers to extract information of interests but also increases the sophistication of wrapper code generation.

XWRAP Composer is the first application generation toolkit that supports information extraction and information aggregation for multiple pages from different Web sites. The unique features of XWRAP Composer are the interface and outerface description languages, and the Composer scripting language for encoding the workflow dependency, the flow control logic, and extraction methods of the wrapper programs and semantic annotation of the Web data to be wrapped.

Important to note here is that semi-automated wrapper construction is just one of the challenges in building a scalable and reliable mediator-based information integration system for Web information sources. The other important problems include resolving semantic heterogeneity among different information sources, efficient query planning for gathering and integrating the requested information from different Web sites, and intelligent caching

of retrieved data, to name a few. The focus of this paper is solely on wrapper construction.

The rest of this monograph is organized as follows. Chapter 2 surveys the state of art in information extraction and software generation, particularly in wrapper generation. Chapter 3 and Chapter 4 discuss our methodology and frameworks in XWRAP Original and XWRAP Elite. Chatper 5 studies the performance of XWRAP Elite and extends it into XWRAP Composer. Finally, I summarize my XWRAP research work in Chapter 6.

CHAPTER II

STATE OF THE ART IN INFORMATION EXTRACTION

2.1 Introduction

Due to the enormous amount of data surging on the Web, there has been much interest in retrieving and integrating the data. However, the Web currently does not support such information management tasks very well. Most Web data reside in HTML pages, which do not have a fixed structure. Web sources do not provide a query interface like the traditional database system. Furthermore, each Web source is autonomous, and so interoperability of the Web sources becomes a big challenge.

A popular approach to handling the problem is to write wrappers that encapsulate the access to Web sources. Several projects have implemented hand-coded wrappers for a variety of sources [13, 62]. However, developing and writing such a wrapper takes a lot of effort due to the irregularity and heterogeneity of the Web data. Hand-coding wrappers becomes a major drawback in situations where it is important to gain access to new Web sources or frequently changed Web sources. We observe that, with a good design methodology, only a relatively small part of the code deals with the source-specific details, and the rest of the code is either common among wrappers or can be expressed at a higher level in a more structured fashion. Some wrapper generation toolkits [4, 1, 50, 56, 75, 67, 68] have been developed to quickly build wrappers semi-automatically.

After the wrappers are generated, the maintenance issue becomes important. The Web data sources constantly update their information with (slight) changes in format, and the wrapper needs to be refined to fit the new format. Some wrapper generators adapt wrappers to new scenarios incrementally by feedback-based improvements [4, 32, 42, 68], while other generators using machine-learning algorithms need to rebuild the new wrapper [40, 1].

Collaborations of wrappers enable us to integrate information from multiple Web sources. Each Web page is treated as an information source, as if each page is a small database.

Wrappers collaborate with each other by connections between these small databases, such as hypertext-links in Web pages, index pages of Web Sites, and mapping tables of Web sources [61, 33].

While more and more wrappers have been developed for various sources, mechanisms and technology aiding the construction and sharing of wrapper programs are becoming essential for Web information manipulation. Wrapper repository systems [64, 41] allow wrapper developers to publish different wrappers online, so that other users can simply tailor the wrappers to different situations. Furthermore, such a repository system can be used for the empirical analysis of learning algorithms that generate extraction patterns.

In the following sections, we first illustrate the functionality of a wrapper in Section 2.2, and then address different technical issues related with the wrapper. Section 2.3 introduces the general approaches of application code generation. Section 2.4 discusses the methodology of automated wrapper generation. Section 2.5 addresses the maintenance of wrappers. Section 2.6 explains how to integrate the results from different wrappers. Finally, Section 2.7 introduces wrapper repository systems and some open issues for designing a wrapper repository.

2.2 Wrapper Functionality

The following examples will illustrate the functions of a wrapper involving answering queries about current stock quotes. An excellent data source is the eSignal site, which has an HTML page for each stock queried by the stock symbol. Figure 1 shows the page for Inter-tel, and Figure 2 shows a fragment of HTML source-code for this page. People may ask, "What is today's open price for Inter-tel, whose stock symbol is INTL?" In order to answer this question, the wrapper creates a remote connection call to the page for Inter-tel on the eSignal site. Then the wrapper extracts the stock information into a structured data format, for example, the XML format shown in Figure 3. Now we can issue a database-like query on the XML result using some XML query language.

However, we may not know the stock symbol for Inter-tel. Collaborations of wrappers through a mediator can facilitate solving this problem. A mediator is a program that

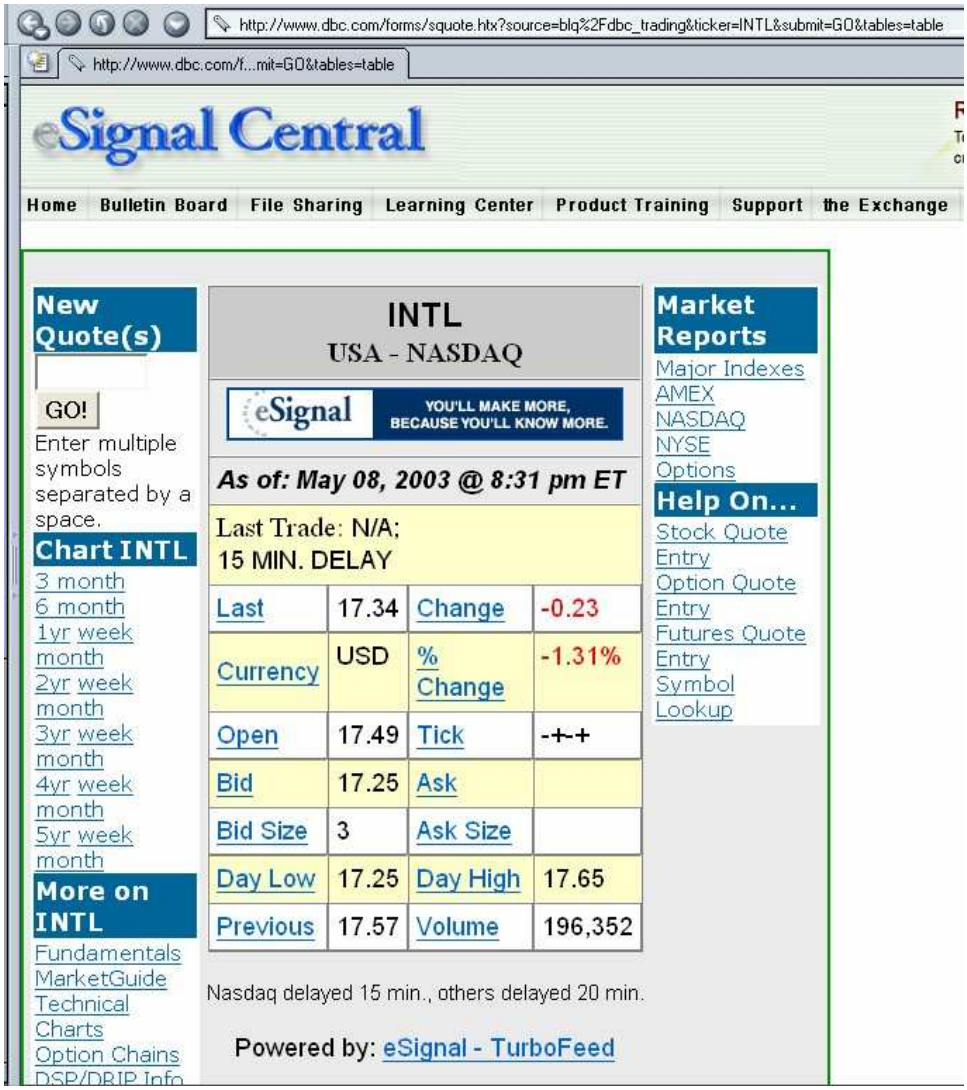

Figure 1: Stock Quotes for Inter-tel at www.dbc.com

integrates the results from different wrappers. For instance, another Web page at eSignal allows looking up stock symbols by company names. First, the mediator queries the wrapper for Stock-Symbol-Look-Up to retrieve the stock symbol by inputting the company name, *Inter-tel Corporation*; the mediator inputs the symbol to the wrapper for Stock-Quotes to reach the page for Inter-tel. Finally, the wrapper for Stock-Quotes returns Inter-tel stock information to the mediator.

```
<table>
...
<tr><td colspan=4 valign=top><font face="Arial" size="-1"><strong><i>
As of:  May 08, 2003 @ 8:31 pm ET</i></font></td></tr><tr BGCOLOR="#ffffcc">
<td colspan=4 align=left valign=top>Last Trade:  </b><font face="Arial" size="-1">
N/A;<br> 15 MIN. DELAY</font></td></tr>
<tr BGCOLOR="#ffffff">
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#last">Last</a></font></td>
<td align=left valign=top><font face="Arial" size="-1"> 17.34</b></td>
<td> <font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#change">Change</a></font></td>
<td align=left valign=top><font face="Arial" size="-1"><font color=#ff0000> -0.23</font>
</td></tr>
...
<tr BGCOLOR="#ffffff">
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#tradesize">Bid Size</a></font></td>
<td align=left valign=top><font face="Arial" size="-1">3</td>
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#tradesize">Ask Size</font></td>
<td align=left valign=top><font face="Arial" size="-1"> <tr BGCOLOR="#ffffcc">
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING#
daylow">Day Low</a></font></td>
<td align=left valign=top><font face="Arial" size="-1"> 17.25</font></td>
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#dayhigh">Day High</a></font>
</td><td align=left valign=top><font face="Arial" size="-1"> 17.65</font></td></tr>
<tr BGCOLOR="#ffffff">
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#prev">Previous</a></font></td>
<td align=left valign=top><font face="Arial" size="-1"> 17.57</font></td>
<td><font face="Arial" size="-1"><a href="/http2_data/glossary.htx?source=BLQ/DBC_TRADING
#volume">Volume</a></font></td>
<td align=left valign=top><font face="Arial" size="-1"> 196,352</font></td></tr>
</table>
```

Figure 2: A Fragment of HTML Code of Savannah Weather Page

2.3 Application Generation

Alleviating the sophistication of components in software development and the difficulty of software maintenance, application generation improves programming productivity and

```
<quote name="INTL">
   <Last>17.34</Last>
   <Change>-0.23</Change>
   <Currency>USD</Currency>
   <Percent_Change>-1.31%</Percent_Change>
   <Open>17.49</Open>
   <Tick>-+-+</Tick>
   <Bid>17.25</Bid>
   <Ask></Ask>
   <Bid_Size>3</Bid_Size>
   <Ask_Size></Ask_Size>
   <Day_Low>17.25</Day_Low>
   <Day_High>
   <Previous>17.57</Previous>
   <Volume>196,352</Volume>
</quote>
```

Figure 3: Extracted Intel-tel Stock Information in XML

reduces human errors without sacrificing code efficiency [17]. Most opportunities for application generation occur in domain specific applications. When a programming activity is well-understood and clearly defined, application generation greatly reduces rote coding and tedious debugging.

Figure 4 shows the basic processes to develop a program using an application generator. Starting with specifications that embed domain specific knowledge, it goes through a *front end*, a *translation engine*, and a *back end*. The front end is responsible for scanning the specifications, converting them to a more convenient **IR** (Intermediate Representation) *a*, such as a DOM tree for XML specifications, and verifying their syntax. The translation engine maps IR *a* to another IR *b* that is more suitable for generating executable code, such as a programming language. The back end produces a program by transforming IR *b* into an executable form.

The structure of an application generator looks very similar to that of a typical compiler. In fact, a compiler can be viewed as a special type of application generator. However, application generation research usually studies programs written in *domain specific languages (DSLs)* while a compiler is often for a general purpose programming language, such as C or

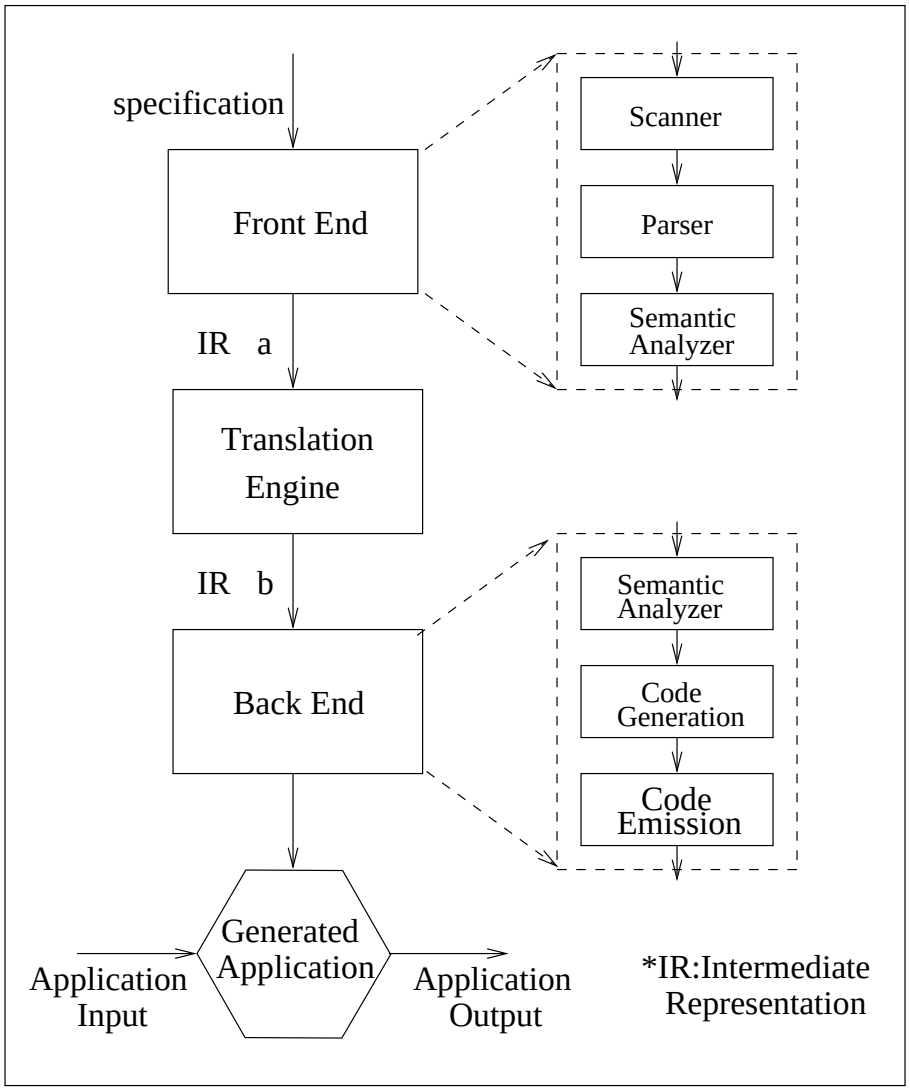

Figure 4: An Application Generator

Java. There is no strict definition for being "domain specific", but DSL often refers to programming languages for some specific tasks, such as workflow management, communication protocols, partial evaluations, etc.

While application generation offers advantages, it also imposes several challenging issues.

- Application generators are difficult to build because they require both the insight of an application domain and the programming skill to write parsers and language translators.

- Realizing the opportunity for an application generator often occurs late in a product

development cycle, when the incentive to build the application generator becomes less.

- The benefits of application generation are hard to measure. An application generator usually only applies to a few existing situations while it is uncertain how it can be used in how many other future situations. Thus, it is difficult to weigh the trade-off between the long-term benefits and the short-term cost to build the application generator.

- The performance of generated applications can differ up to several orders of magnitude depending if optimization is explored or not. The generated code is often so structured that it has a great potential to be optimized.

2.4 Automated Wrapper Generation

Many researchers have studied wrapper generation systems that semi-automatically generate wrapper programs. The key idea of wrapper generation is to discover wrapping rules to extract data of interest from the input and compose results to the output. In this document, we have surveyed wrapper generation projects and compared them from the following aspects: Program Generation Process, Data Extraction, and Results Composition.

2.4.1 Program Generation Process

From a program generation point of view, wrapper generators are mainly different in how they guide wrapper developers to locate data of interest and conclude the rules. There are five common techniques widely used in wrapper generators: scripting, GUI highlighting, machine learning, ontology, and modeling.

A scripting-style wrapper generator allows to manually write wrapping rules in a domain specific language. For example, W4F [68] requires wrapper developers to define extraction rules in HEL (HTML Extraction language), and then converts the rules into an executable wrapper. Such a scripting style is similar to coding programs in C while HEL is a much simpler language. W4F also provides some visual aids to obtain parameter values, such as the tree path of a particular HTML node in a Web document. Scripting-style wrapper generators often cannot scale in an environment with a lot of data sources; however, they

are also rich in functions to extract complex information. Araneus [6] adopts the scripting style.

A more automated approach is interactively highlighting information. A wrapper developer marks data of interest through a GUI, and then the wrapper generator produces rules to extract information by human supervision, heuristics or machine learning algorithms. Visually interactive programs tend to be easy to learn. However, they also involve much human effort. How to reduce as much human input as possible is a big issue. Systems using this approach include LIXTO [66], NoDoSE [1] and XWRAP Original [47], to name a few.

Machine learning algorithms are widely used in many wrapper generators. Given a set of training examples, the toolkit compares the similarities and discrepancies, searches for constants and variables in the examples, and determines how to locate them in the future. The deriving process, also called *Wrapper Induction*, often holds reasonable assumptions to interpret the constants and variables. For instance, data sections and sub-sections are often separated by mark-up tokens, such as HTML tags or punctuation signs. Ariadne [40], also called Stalker [58], generates a lexical analyzer that searches a page for tokens indicating the heading of a section using LEX, a lexical analyzer generator based on string regular expressions. For instance, given a stock-quotes page at eSignal, Ariadne is able to identify tokens of interest such as **Open** and **Big** by regular expressions. Then, Ariadne analyzes the nesting hierarchy of sections, determining which sections comprise the page at the top level and which sub-sections (if any) comprise other sections in the page. Other wrapper generation systems based on machine learning include SoftBot [36] and RoadRunner [16].

While machine learning facilitates wrapper developers to derive wrapping rules without having to manually analyze documents, a common disadvantage for machine learning algorithms is the reliance on training examples. First, the examples need to be representative enough to cover most scenarios. In addition, such reliance prevents wrappers to be generated on the fly. In many circumstances, it is hard or even impossible to obtain training examples.

Ontology is another useful aid for wrapper generation. Given a specific application domain, ontology can guide a wrapper generator to locate constants of interest in the input

and to construct data objects with them. David Embley from BYU has demonstrated the use of ontology to extract data objects (or records) in domain specific Web pages, such as car Web sites [19]. Integrated into XWRAP Elite, OMNI [9] has also made use of ontology to assist extracting information.

Modeling can also facilitate the generation of wrapping rules. Given a set of target data structures, such as lists, tuples, and paragraphs, modeling-based wrapper generators look for sections of data that conform to the data structures. This approach reduces a lot of human efforts if the structures of target data are known. NoDoSE, DEByE [2], LIXTO, and XWRAP Original choose this approach.

Table 1 shows how the five techniques are used in wrapper generation projects. A wrapper generator can use several techniques. Particularly, GUI highlighting often needs machine learning or modeling technique to derive extraction rules. W4F is the only exception. It uses GUI highlighting to look for tree-paths used in scripting.

We also rank the degree of automation in terms of the wrapper generation process. In general, the scripting style has the worst automation. GUI highlighting improves the automation process if accompanied with modeling or machine learning techniques. If a wrapper generator does not rely on highlighting to find data of interest, it has the best automation. BYU, XWRAP Elite [28], and RoadRunner are the three projects with the highest automation scores.

In Figure 5, we also see that XWRAP contains three components, XWRAP Original, XWRAP Elite and XWRAP Composer. XWRAP Original is a modeling tool. XWRAP Elite is a heuristics-based tool with an optional aid of ontology. It also has a scripting-style interface that allows wrapper developers to generate or revise wrappers in a domain specific language. XWRAP Composer is currently a scripting tool used to create sophisticated multi-page wrappers. Because XWRAP has adopted such a hybrid approach, it can semi-automatically generate wrappers for specific application domains with minimum human involvement and maintain the capability to extract complex information from various data sources in other domains. While XWRAP does not use machine learning, it also gets the benefit of generating wrappers on the fly for a particular domain because it does not

Projects	Scripting	High-lighting	Machine Learning	Use of Ontology	Modeling	Degree of Automation
Araneus	Yes	Yes	Yes	No	No	Poor
BYU	No	No	No	Yes	No	High
DEByE	No	No	No	No	Yes	Medium
Garlic	Yes	No	No	No	No	Poor
JEDI [31]	Yes	No	No	No	No	Poor
LIXTO	No	Yes	Yes	No	Yes	Medium
NoDoSE	No	Yes	Yes	No	Yes	Medium
RAPIER	No	Yes	Yes	No	No	Medium
Road-Runner	No	Optional	Yes	No	No	High
Softbot	No	Yes	Yes	No	No	Medium
SoftMealy [30]	No	Yes	Yes	No	No	Medium
SRV	No	Yes	Yes	No	No	Medium
STALKER/	Yes	Yes	No	No	No	Medium
Ariadne						
W4F	Yes	Partial	No	No	No	Poor-Med
WHIRL [15]	Yes	No	No	No	No	Poor
Whisk/ Crystal	No	Yes	Yes	No	No	Medium
WIEN [39]	No	Yes	Yes	No	No	Medium
XWRAP-Original	No	Yes	Partial	No	Yes	Medium
XWRAPElite	Yes	No	No	Yes	No	High
XWRAP-Composer	Yes	No	No	No	No	Poor

Figure 5: Methodologies used in wrapper generators

need training examples. XWRAP Elite is the first and only wrapper generator to produce wrappers online on the fly.

2.4.1.1 Data Extraction

Many wrapper generation systems use string regular expression as their main extraction technique, such as SoftBot and NoDoSE. The expressions can be inducted by machine learning algorithms or hand-written by developers. Because string regular expression can be applied to all text documents, these systems theoretically can often extract data from all string-based data sources. However, string regular expressions are often difficult to write or derive, hurting the scalability of the wrapper generation systems. They are also prone

to errors, sacrificing the accuracy of data extraction.

Some other wrapper generators try more domain-specific approaches that utilize inherent structural features of the input data type. For example, Natural Language Processing (NLP) is the main extraction technique in several early wrapper systems, such as RAPIER [11], SRV [21], and WHISK. These systems often use machine-learning algorithms based on linguistic patterns to learn extraction rules. Some systems can only extract a single record from a document. Hence they are called "single-slot" system.

Web documents are another domain many wrapper generators focus on. They often use the implicit tree structure of HTML documents such as W4F [68] and RoadRunner. W4F uses tree-path regular expression to locate data items. The tree structure presents information more naturally than a string format. Therefore it is easier to write and understand a tree-path regular expression than a string regular expression. Furthermore, the tree structure often offers some additional wrapping clues. RoadRunner compares two similar HTML trees and identifies the variable items. It speculates the variable items are often data of interest. Other HTML specific systems include XWRAP Elite and BYU.

Domain-specific approaches automate the wrapper generation process for suitable data input, but they also limit the types of documents the systems can handle. NLP-based wrapper systems usually don't work well on Web documents. HTML-aware toolkits often cannot generate wrappers for text-based reports for scientific experiments.

All the techniques above cannot extract information from multiple data sources or a data source with multiple possible formats. For example, a search for DNA sequences at PDB might return three possible answers, each of which has a particular format. The no-result answer gives a response indicating no matching DNA can be found. If the search finds exactly one DNA sequence, the answer will contain all detailed information about the sequence. However, if the search finds more than one DNA sequence, the answer will only include a list of sequence summaries, which contain links to detailed information of the sequences. Furthermore, the user may want to extraction detailed information for all matching DNA sequences. None of the data extraction techniques are sufficient enough in this scenario for lack of workflow-like capability.

XWRAP is the only wrapper generation system with such a workflow capability. First, XWRAP Original and XWRAP Elite semi-automatically generate wrappers for HTML documents. Second, XWRAP Composer provides a programming interface to integrate wrappers generated by other systems or hand-coded by developers. The integration language contains both pure data extraction commands and workflow-like control commands. The integration approach takes advantage of the useful structural information embedded in HTML tree representation, enabling extraction from more loose-structured data and multiple data sources. Furthermore, it tolerates other types of data input by allowing integrating wrappers generated from other systems or even hand-written ones.

	Input Format	Extraction Method/ Script Language	Single-slot / Multi-slot	Output Format
Araneus	HTML/Text	Editor	Multi-slot	XML, Text
BYU	HTML/Text	Ontologies	Multi-slot	Text
DEByE	HTML/Text	String Regular Expression	Multi-slot	XML
Garlic	HTML	String Regular Expression	Multi-slot	GDL
JEDI	HTML	JEDI	Multi-slot	XML, Text
LIXTO	HTML	ELOG	Multi-slot	XML, Text
NoDoSE	Semi-structured	String Pattern Analysis	Multi-slot	XML
RAPIER	Plain English	NLP	Single-slot	String
RoadRunner	HTML	Regular Expression	Multi-slot	XML, Text
Softbot	Mainly HTML	String Regular Expression	Multi-slot	Text
SoftMealy	HTML	String Regular Expression	Multi-slot	Text
SRV	Plain English	NLP	Single-slot	String
STALKER/Ariadne	HTML	String Regular Expression	Multi-slot	KQML/XML
W4F	HTML	Treepath Pattern	Multi-slot	Text
WHIRL	HTML	Regular Expression	Multi-slot	Text
Whisk/Crystal	Plain English	NLP	Single-slot	String
WIEN	HTML	String Regular Expression	Multi-slot	Text
XWRAPOriginal	HTML	Treepath Pattern	Multi-slot	XML
XWRAPElite	HTML	Tree Analysis/Ontologies	Multi-slot	XML
XWRAPComposer	Semi-Structured	Mixed	Multi-slot	XML

Figure 6: Comparison of Wrapper Generators on Input and Output Format

Figure 6 shows a comparison of wrapper generators in terms of data extraction techniques. Wrapper generators for plain English input usually adopt NLP algorithms; Wrapper generators for semi-structured text input (e.g. lab experiments) tend to rely on string regular expression, such as NoDoSE; Recent Web-page wrapper generators often utilize HTML tree structure to facilitate locating information, while early Web-page wrapper generators might still use string regular expression, such as Softbot.

2.4.1.2 Results Composition

After identifying interesting data in the source documents, wrappers output them in a more structured format. In order to communicate with the mediator or other applications, this format should satisfy the following requirements: It needs to be understood by various applications, such as the mediator. In addition, it must support the query capability so that some post-processing can be done on the extracted information, such as filtering.

XML (eXtensible Markup Language) [70] is a very good choice for such a unified data format. It is a self-describing language, so that XML data is easily interpreted in different applications. The query languages on XML data have been proposed and are being implemented both in academic and industrial areas, such as XML-QL and XQL. Furthermore, more applications and vendors are beginning to support XML at the present.

Figure 6 shows most current wrapper generators support XML output. However, wrapper generators designed before the emergence of XML adopted other semi-structured output format. In the TSIMMIS project, the Object Exchange Model (OEM) is chosen as the common data model exported by the wrappers. OEM is also a self-describing model, which captures the essential features of models used in practice, generalizing them to allow arbitrary nesting and to include object identity. An application can request OEM objects from a wrapper using the MSL query language. Due to the similarity between OEM and XML, we can treat OEM objects as XML objects as well.

Stalker chooses KQML [20], an agent communication language, to facilitate communication between the wrapper and a mediator. Garlic [12] describes data using GDL (Garlic Data Language), a variant of the ODMG's Object Description Language. However, these formats are only understood by certain applications, which limits their use.

In general, wrappers do not necessarily produce the schema of wrapping results. Some wrappers pre-define a result schema, and other wrappers output results in a self-describing language, such as XML, which can be understood without a schema. Pre-defining the result schema is not always possible. For example, referring back to the page for Savannah, not all the weather conditions appear on that page, such as Precipitation last hour. Schema

extraction techniques are often employed to summarize a near-accurate schema from previous extracted data. For instance, DataGuide [23] serves as dynamic schemas, generated from the database populated by wrapper results.

2.5 Wrapper Maintenance

It is important to regularly maintain wrappers because Web sources frequently update their information with (slight) changes in presentation and data structure. The presentation changes often prevent a wrapper from extracting information from the correct location. The changes in data structure usually cause errors in interpreting the source during result composition. Wrappers need to be refined to fit the new format.

Several research projects have been studying wrapper maintenance [37, 14, 43], and nearly all of them separate the issue into two components: *wrapper verification* and *wrapper repairing*. Wrapper verification detects a dysfunctional wrapper when the data source makes enough changes to cause wrapping errors. Wrapper repairing produces a rectified wrapper conforming to the new data format.

2.5.1 Wrapper Verification

N. Kushimick et al. [37] first introduced the notion of Wrapper Verification and proposed an algorithm RAPTURE to verify if a wrapper correctly extracted information. RAPTURE computes the similarity between a wrapper's output and previous correct output based on a set of syntactic features, such as the length or the fraction of punctuation of the extracted strings. It then combines the similarity values to derive an overall probability that the wrapper is correct. In particular, RAPTURE found that the HTML density feature was the most effective measure to identify a dynfunctional wrapper.

K. Lerman et al. [35] improved RAPTURE's results with a more fine-grained comparison. They also adopted a machine-learning solution that compared the wrapper results to correct result generated in the past. However, in addition to numerical comparison on syntactic features, [35] compares data patterns derived from DataProg algorithm. The data patterns are actually regular expressions of data items.

Chidlovskii [14] presented another exception-handling approach to detect wrapper errors. When a wrapper cannot locate or extract certain information during the wrapping process, the wrapper immediately throws an exception and triggers the recovery and maintenance routine. This approach relies on a close integration with the wrapper system and often requires extra effort from wrapper developers to avoid false exceptions.

2.5.2 Wrapper Repairing

After detecting a wrapper producing errors, we need to correct it. Regenerating the wrapper is always an option, but such a naive option does not scale and is often unnecessary since Web sites changes are usually incremental. A partial modification in the wrapper is likely sufficient to adapt to the new source format.

Landmark wrappers usually enjoy the benefits of partial modification. When the Web source changes its format, we simply need to test the old wrapper on some new sample documents. Because of the similarity between the old format and the new, adapting the wrapper is also very straightforward. For instance, [35] proposed an re-induction algorithm to semi-automatically repair a wrapper after detecting the wrapper produced incorrect data. The re-induction algorithm was able to regenerate content-based and landmark-based rules with a supervised interface. Ariadne, XWRAP(Original), W4F, and TSIMMIS [4, 50, 68, 32] use this strategy.

Chidlovskii [14] introduced an approach to automatically repair dysfunctional landmark wrappers. Assuming changes are small, his approach relies on redundant or alternative views for information extraction recovery. When a wrapper fails to extract a data item or landmark information, it will use the alternative views to search for the most possible matching item. The alternative views include grammatical rules, content features and backward wrappers.

Not all wrappers are landmark ones or can be easily translated into landmark ones. If a wrapper is generated by machine learning algorithms, we simply retrain the wrapper with the new documents. For example, Softbot has predefined six generic wrapper classes with adjustable parameters. Each wrapper class can extract information from one document

pattern and a majority (70 percent in total) of the real Web resources are covered by the six classes in their evaluation test [40]. Wrapper developers highlight interesting sections in many sample documents, and then the machine-learning algorithm adjusts the predefined parameters to find a combination of wrapper classes to extract the highlighted sections correctly. If such a combination is not available, the algorithm returns the best combination with the fewest mistakes. The developers can either correct the best combination manually or add more wrapper classes fitting new patterns to find a complete correct combination. NoDoSE [1] also adopts this strategy.

2.6 Integration of wrapper information

An important use of wrappers is to extend the query capabilities of a source. Collaborations of wrappers help us answer some queries that a single wrapper cannot handle. For example, people may ask, "How many cities in Georgia are raining today?" In order to answer this query, the wrapper needs to navigate all the pages for the cities in Georgia on the NWS (National Weather Service) site and then integrate results from each separate page. It becomes essential to understand the connections between those pages; in particular, we should know their exact locations before we retrieve each page.

Ariadne [33] wrappers navigate Web pages at a site through index pages. For example, NWS has an index page containing an image-map for each state. Figure 7 shows the image-map page for GA, and Figure 9 shows a fragment of the HTML source code of the index page. The index page serves as an information source listing all the URLs for each city page. A wrapper for this index page locates all the pages for cities in GA.

This strategy reuses the technology of the wrapper for a single page. However, it cannot work when an index page is not available or such an index page does not provide enough information. For example, referring back to the example query, "What is the temperature in Savannah today," we do not know the city code for Savannah, GA, and the index page does not allow the wrapper to locate the page for Savannah based on the city name. It is impractical to retrieve all the pages for the cities in GA and then look for the one for Savannah.

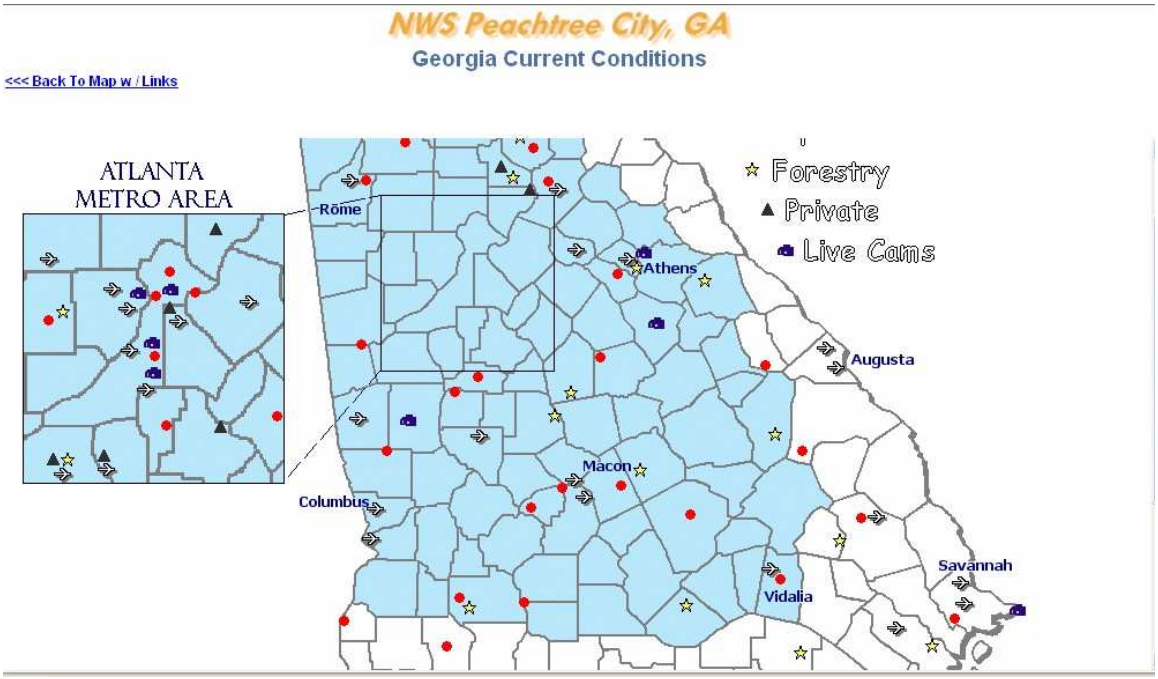

Figure 7: The Index Web Page for GA on NWS

We can solve this problem by integrating wrappers for other data sources, such as World Wide Airport and City Code Database (WWACCD), which list the city codes alphabetically by city names. As described in Section 2, the mediator queries the wrapper for the WWACCD site to obtain the city code by the city name, "Savannah, GA," and then the mediator uses the city code to generate the URL for the city on the NWS site.

When we integrate information across multiple Web sources, we need to resolve the common problem of connecting the corresponding entities. In this example, the "URL " for a city on the NWS s1te is related with the "city code" on the WWACCD site, and we want to represent their relationship. Ariadne selects a primary source for an entity's name and then provides a mapping table from that source to each of the other sources where a corresponding relationship between two entities in different Web sites is found. For example, if we choose the NWS site as the primary source, we will have a table that maps the entity "URL " on the NWS site to "city code" on the WW ACCD site. A mapping method describes the relationship between two entities; "city code" is a sub-string of "URL." Currently, such a mapping table is written by hand. The drawback of this approach is that

```
<MAP NAME="newcwa">
<!-- UGA Observations -->
...
<AREA SHAPE="circle" COORDS="316,126,5" HREF="http://www.griffin.
peachnet.edu/cgi-bin/GAEMN.pl?site=GARO&report=c" onMouseOver="
overliba('Rome')" onMouseOut="nd()">
<AREA SHAPE="circle" COORDS="312,268,5" HREF="http://www.griffin.
peachnet.edu/cgi-bin/GAEMN.pl?site=GARV&report=c" onMouseOver="
overliba('Roopville')" onMouseOut="nd()">
<AREA SHAPE="circle" COORDS="826,505,5" HREF="http://www.griffin
.peachnet.edu/cgi-bin/GAEMN.pl?site=GASA&report=c" onMouseOver="
overliba('Savannah')" onMouseOut="nd()">
<AREA SHAPE="circle" COORDS="745,418,5" HREF="http://www.griffin
.peachnet.edu/cgi-bin/GAEMN.pl?site=GAST&report=c" onMouseOver="
overliba('Statesboro')" onMouseOut="nd()">
<AREA SHAPE="circle" COORDS="536,679,5" HREF="http://www.griffin
.peachnet.edu/cgi-bin/GAEMN.pl?site=GAVA&report=c" onMouseOver="
overliba('Valdosta')" onMouseOut="nd()">
...
</MAP>
```

Figure 8: A fragment of the HTML source-code of the index page for GA on NWS

it cannot deal with more than three Web sites that are fully connected because we cannot find a primary source that involves all the mappings between two sources.

Araneus [61] gives a global view of Web sources by describing the scheme of a Web hypertext. Thus the wrapper can integrate information in different Web pages or Web sites by querying the global view. Araneus considers each Web page an object with an identifier (the URL) and a set of attributes, which include text, images, or links to other pages. Araneus navigates the Web following the links between different pages, and weaves Web pages to provide a global view of the information sources. However, Araneus cannot represent the relationship of Web pages with no hypertext links. For example, it cannot answer the example query about the temperature in Savannah, because there are no hypertext links between the WWACCD site and the NWS site.

2.7 Wrapper repository

A wrapper repository is a distributed metadata repository where users can contribute, download and share wrappers. The fundamental issue is a common interface for shared

wrappers. As the first wrapper repository, RISE [64] allows wrapper developers to register wrapper programs with their metadata and page model information. The metadata includes **Original Site**, **IE/WG by** (the author), **Project**, **Institution**, **Algorithm**, **Main Paper**, and **Other Papers**; the page model information contains the pattern of source documents, result schema, and sample pages. However, requiring the developers to manually describe the pattern and the result schema in a proprietary language, RISE is labor-intensive and does not scale. Furthermore, while users can download wrappers and test wrappers on sample pages, RISE does not allow running wrappers online through the repository system.

Based on a meta-data approach, WPstore [41] is a wrapper repository that supports running shared wrappers online. Its construction consists of two phases. The first phase includes the development of an extensible wrapper framework and a component library, called *component library*. This library hosts a collection of code components useful for constructing wrapper programs, and a collection of wrapper API functions, plug-ins, and meta-interfaces for applications to hook up with specific wrapper programs downloaded from the WPstore. The second phase involves setting up a distributed metadata repository, to which wrapper developers can contribute wrappers and the wrapper-users can download and run them.

There are several open issues in wrapper repository. The first issue is how to detect and handle broken wrappers. If Web sources change their content or presentation format, the wrapper relying on the earlier format is broken; the repository should facilitate the notification that a wrapper is broken. The second issue is how to manage new releases of a wrapper and what sort of version control facility should be provided. Wrappers need to be updated whenever the corresponding sites change their format, and the current wrapper cannot adapt the changes automatically. The wrapper developer may release the new version of a wrapper with better performance or more functionality. A version control mechanism would be needed to track the new release of the wrappers and facilitate users to decide whether an updated wrapper is available or should be upgraded.

2.8 Conclusion

With the fast growth of the Web, retrieving information from different Web sources for future integration becomes increasingly beneficial. However, the lack of program interfaces to accessing the Web information often poses difficulty for integration. A common approach to accomplish this task is to build a wrapper for each Web source and then query that source through the wrapper. In this chapter, we have overviewed the major issues of wrapper technology.

We have discussed different approaches of automated wrapper generation. Various wrappers analyze the source document by the string regular expression, the tree regular express) and the string length; the results of wrappers are composed in a unified format, such as XML, OEM or KQML, to facilitate communication with the mediator or other applications.

We have introduced the wrapper maintenance issue and have surveyed related research projects. Generated wrapper maintenance consists of two sub-issues: wrapper verification and wrapper repairing. Wrapper verification is discovering when a wrapper stops functioning correctly and wrapper repairing is the process to regenerate or correct the wrapper.

We have also showed how to integrate the results from wrappers for different pages or Web sites. Navigating Web pages through index pages or hypertext links connects Web pages into a global integrated view. When index pages or hypertext links are not available, some mapping tables need to be built manually.

Finally, we have addressed the wrapper repositories that aid the construction and sharing of wrapper programs. We have also observed that some challenging issues in this area have not been solved.

CHAPTER III

XWRAP ORIGINAL

3.1 Introduction

In this chapter, we describe the design and the development of XWRAP Original methodology, including the mechanisms, the strategies, and the declarative rule language for information extraction and the algorithms for code generation based on the information extraction rules. We will also report our initial development experience of the XWRAP Original generator system.

Our XWRAP Original approach is based on extensive experience with wrapper construction. First, we want to leverage on standards as much as possible, thus choosing XML as our output format. Second, we know that a modest participation from wrapper programmer can simplify the entire process significantly, thus adopting an iterative construction method. The development of XWRAP presents not only a software tool, but more importantly also, the methodology for developing an XML-enabled, feedback-based, interactive wrapper construction facility that generates value-added wrappers for Internet information sources.

The remaining of the chapter is organized as follows. We first describe the functionality of a wrapper and address various technical issues related with the construction of a wrapper in Section 3.2. Then we describe the methodology of automated wrapper generation in Section 3.3. We discuss the functional wrappers in XWRAP in Section 3.4 and address the implementation issues of XWRAP and the maintenance of the wrappers in Section 3.5. We conclude the paper with a report on our initial performance evaluation, a summary of related work, and a discussion on the futureresearch plans.

3.2 The XWRAP Original Framework

3.2.1 An Informal Classification of Wrappers

A *wrapper* is a software program specialized to a single data source, which converts the source documents and queries from the source data model to another, usually more structured data model. Wrappers are needed to translate between some new functionality (e.g., a mediator query language that supports queries beyond the keyword-based approach) and the original sources. In the case of HTML sources, a key role of a wrapper is to explore and capture the meta information implicitly defined by an HTML document (or a set of documents) and make the meta information explicit, so the information content can be extracted and interpreted correctly and automatically.

We divide wrappers into two classes of interest to us. The first class, called *data wrapper*, treats data sources just as data. The main function of data wrappers is to interpret the wrapped data on behalf of user programs, which operate themselves on data. Data wrappers pass through invocations implemented by the original sources as data strings. We discuss the design and implementation choices in Section 3.3. The second class, called *functional wrapper*, understands the operations and invocations (e.g., queries) on the data sources. Usually, functional wrappers provide a superset of data wrapper capabilities (or a higher level of abstraction) by adding additional query capabilities or some new interpretation or translation of the source data. For example, a wrapper written for an international electronic commerce application program may translate a dollar-denominated value into different local currencies by invoking a currency exchange service. While there are many interesting examples of functional wrappers, we have found the basic foundation to build them to be the same as data wrappers. Without a good data wrapper construction toolkit, it is difficult to build functional wrappers directly. Therefore, in the rest of this paper we first focus on the tools for data wrapper construction, with the understanding that they apply also to functional wrappers. We discuss the issues in design and implementation of functional wrappers in Section 3.4. The methodology and XWRAP software are designed for both classes of wrappers.

3.2.2 General Design issues

A wrapper framework consists of a generic code structure and a collection of code components that can be tailored to build specialized wrappers through source-specific code customization. A key component of a wrapper framework is the wrapper API and a library of code that implements this API. Specifically, a wrapper API implemented in Java usually includes the Java class hierarchy rooted at the **Wrapper** class.

- **Wrapper** class, representing wrapper objects. A wrapper object is an execution of the corresponding wrapper program. Therefore, each **Wrapper** object consists of a query object, a result-packaging object, a wrapper execution status object, and a wrapper exception handler. In **Wrapper** class, one also specifies the properties that are common to all types of wrappers, such as the version number, the software release date, the manufacture name, the coding language, the operational platforms, to name a few. Relational wrappers and HTTP wrappers are specialized classes of the **Wrapper** class.

- **Query** class, representing query objects and mechanisms for firing queries over remote site and fetching data of interest. HTTP queries are instances of **HTTPQuery** class, which can be implemented as a specialization of the **Query** class.

  Each HTTP query can be further classified as an object of **HTTPGetQuery** class or an object of **HTTPPostQuery** class.

- **WrapperOutput** class, representing the results to be returned to the caller of the wrapper program. A WrapperOutput object is generated through a series of transformation and filtering processes of the source document fetched from the original data source.

- **WrapperStatus**, indicating the status of a wrapper that is running.

  Different types of wrappers may need different methods for monitoring the execution status of a wrapper program. Thus, the **WrapperStatus** class can be specialized into **RelationalWrapperStatus**, **HTTPWrapperStatus**, and so forth.

- **WrapperException**, used to indicate errors that arise when firing a wrapper, such as timeout handling in the presence of network connection failure or slow links. **WrapperException** class is used to capture the errors and exceptions common to all wrappers, different types of wrappers may have specific exceptions and specific exception handling needs. Thus, both generic and specialized wrapper exception-handling methods are needed.

A major challenge in designing an extensible wrapper framework for wrapper construction is the identification of mandatory functionality of a wrapper and the clean separation of optional functionality of a wrapper from undesirable functionality. For example, should we consider sophisticated retrieval mechanisms, error handling, and the choice of streaming mode or blocking mode as mandatory wrapper functionality? Should performance, robustness, statistics, proxies, optimization be better treated as optional functionality?

Should we consider massaging data, sophisticated recovery strategies as undesirable wrapper functionality?

Based on our experience in building wrappers, the mandatory functionality of a wrapper should contain those capabilities that are crucial for achieving the basic goal of a wrapper. For example, to enhance the information extraction quality, it is necessary for a wrapper to provide sophisticated retrieval and filtering mechanisms, and simple error handling strategies. Examples of error handling strategies include handling timeouts with user-specified thresholds and providing a status method that returns the runtime status of the wrapper. Furthermore, to improve the responsiveness of a wrapper, both blocking mode and streaming mode of interaction between a wrapper and its applications should be provided. When a wrapper runs in the streaming mode, applications are able to fire a wrapper (e.g., by issuing a query) and receive a stream of returned data, rather than having to block the wrapper until the wrapper query is terminated.

To provide the streaming interface, the **Wrapper**'s `fire(...)` method returns a synchronized queue. The wrapper will run its own thread and write to the queue, and the application can read from it. Many applications may not take advantage of the streaming mode, so a much simpler blocking interface is provided.

In addition to the mandatory functionality, there are a number of optional functionality of a wrapper that are important and desirable, including performance statistics and mechanisms for wrapper query optimization. For example, many applications would need more detailed statistics information about the wrapper execution (e.g., clock times, bytes transferred, number of tuples or objects returned) than simply that the status is done or running. It is also desirable that a wrapper may act as a responsible optimizer that ensures that the applications will not throw dozens of queries per second to a site and especially allow an application-configurable inter-query delay.

In the current design of the Component Library [46], we consider the following functionality undesirable, primarily because of the complexity introduced.

- **Caching.** A wrapper should not try to do any caching on behalf of the applications that invoke the wrapper. Such functionality should be provided by the application that needs the caching, not the wrapper.
- **Data Mediation.** A wrapper should return the extracted content with minimal mediation. For example, a wrapper should not remove duplicated information or cannibalize attribute values. Such requirements should be met by the applications.
- **Failure recovery.** A wrapper should not try to provide any sort of sophisticated failure recovery strategy such as retry failed queries a few times. Such functionality is best handled by the application.

Several functionality are useful but are not considered in the current design of the Component Library, including the mechanisms that make wrappers able to learn to be more adaptive to changes at source sites, and the incorporation of complex data types. Whether or not such functionality is desirable remains a question that needs to be answered.

3.2.3 General Steps for Data Wrappers

Usually, a wrapper is written in a scripting language such as PERL or a programming language such as C or Java. A typical wrapper performs the following tasks to fulfill its mission:

- First, it accepts a query request from an application at the mediator level (so called a mediator application). Then it establishes a network connection to transfer data, for example, by sending an HTTP request to the remote web server to fetch the corresponding web page(s). The issue of caching in the wrapper is an orthogonal aspect that falls outside the scope of this paper.

- Second, it processes the retrieved page by differentiating useful data from junks and removing irrelevant data such as HTML tags, fancy advertisements around the page, and transforms the interesting portions of the web page into the target structure - a more structured format.

We divide this data processing pipeline into four tasks: (1) remote document retrieval, (2) syntactic structure normalization, (3) semantic association, and (4) structure transformation. A clean separation of these tasks allows the wrapper generation system to raise the abstraction level of the wrapper generator components by introducing declarative specification of the retrieval, extraction, and structure transformation semantics.

Remote document retrieval component accepts query request from an application at the mediator level (so called a mediator application). Then it establishes the network connection and issues an HTTP request to the remote web server through a `HTTP Get` or `HTTP Post` method, and fetches the corresponding web page.

Syntactic structure normalization component "corrects" the grammatical errors in the web page. Many web pages are designed for human browsing, so their structure may not fit the expectations of a source-document specific tree parser. In particular, irregularities (e.g., the omission of end-paragraph) and ambiguities cause difficulties for many source-document specific tree parsers, including variations of HTML parsers or XML parsers.

Semantic association component deals with declarative specification of information extraction logic in the form of extraction rules. The set of extraction rules should include rules for differentiating useful data from junk, and removing irrelevant data such as HTML tags, fancy advertisements around the page. For HTML pages, the extraction should take advantage of the HTML grammar as well as of regular expression patterns. The extraction

language should be expressive enough to capture the explicit and implicit structural semantics of the HTML documents. In principle, the set of information rules derived at this step will encode the wrapper developer's knowledge and interest as well as the extraction logic of the retrieved web page.

Structure transformation component is responsible for transforming the

the retrieved web page into a target structure, a more structured format (such as XML), by encoding the set of extraction rules explicitly into the target structure (such as XML tags) and the code generation script.

This separation brings more modularity and robustness into the wrapper programs generated by XWRAP. For example, by separating the information extraction from the structure transformation process, we can capture and specify extraction logic in a declarative language, independent of the underlying implementation of the structure transformation. Hence, the addition of new extraction rules or revision of the existing extraction rules in the presence of changes to the wrapped web site can be done without rewriting the code for structure transformation. The update of extraction rules will be incorporated into the next release of the wrapper program by simply re-running the structure transformation.

3.2.4 An Overview of the XWRAP Generator System

In the Continual Queries project at OGI [54, 53, 55], we have built an event-driven update monitoring system for Internet information sources. Motivated by our initial experience and frustration from writing wrapper programs by hand, we have developed an interactive software tool, called XWRAP, for semi-automatic generation of XML-enabled wrappers for Internet data sources [45]. The main objective of XWRAP is to wrap the source documents into XML format and provide content filtering capabilities using XWRAP XML query processor. A wrapper programmer (developer) may enter the URL that he/she would like to wrap. XWRAP will then interact with the wrapper developer by walking through the XWRAP wrapping process, consisting of a sequence of 7 steps:

1. **Enter URL** accepts the URL of the target data source to be wrapped and fires a remote fetch function to obtain the target source document.

2. **Source Normalization** parses and transforms the source document in plain text format to a tree structure. XWRAP normalizes each HTML document into an XML-tree with standard HTML tags.

3. **Semantic Token Extraction** accepts the semantic tokens (S-tokens) identified by the wrapper developer and generates a set of source-specific S-token extraction rules and a comma-delimited file, consisting of S-token name and S-token value pairs.

4. **Hierarchical Structure Extraction** takes the hierarchical structures identified by the wrapper developer through interactive clicks and highlights, and derives a source-specific hierarchical structure extraction (H-struct) rules written in the form of XML-template, an XML file with action semantics.

5. **Learn** is the step where XWRAP learns by going through the procedures necessary for generating the wrapper program through the generation of the XML representation of the source document. The input of the learning step includes the semantic token extraction rules, the source data, and the hierarchical structure extraction rules (XWRAP-script). It produces an XML representation for the source document fetched from the URL entered at the beginning in addition to the pseudo code for the wrapper program.

6. **Wrapper Program Construction** produces the executable wrapper program for the given data source based on the pseudo code generated from the **Learn** step.

7. **Wrapper Program Test** allows the developer to test the wrapper program generated by entering another URL of the same data source and invokes the **Learn** process after the source normalization. If the result produced is not satisfactory, an iterative process can start from the S-token extraction, performing incremental revision of the S-token extraction rules and H-struct extraction rules.

8. **Wrapper Program Release** provides final packaging of the wrapper program. Once the wrapper developer is satisfied with the test of his/her wrapper program generated by XWRAP, he/she can click the release button to exit the XWRAP generator.

Figure 9 shows an interactive session of the XWRAP generator system. National Weather Service is a very good data source providing weather information. We use as a sample page a URL on the National Weather Service site, which contains the current weather information for Savannah, GA. After a developer enters the URL, an HTML tree produced by a source specific parser will be displayed. Highlighting a string in the HTML page on the bottom, the developer can locate and select the corresponding node in the tree. The developer is specifying extraction rules for a region of the HTML table of current weather conditions. Specification values can be derived from the parse tree. For example, the regular expression of a selected node in the tree can be inserted into the text field of specification panel on the right.

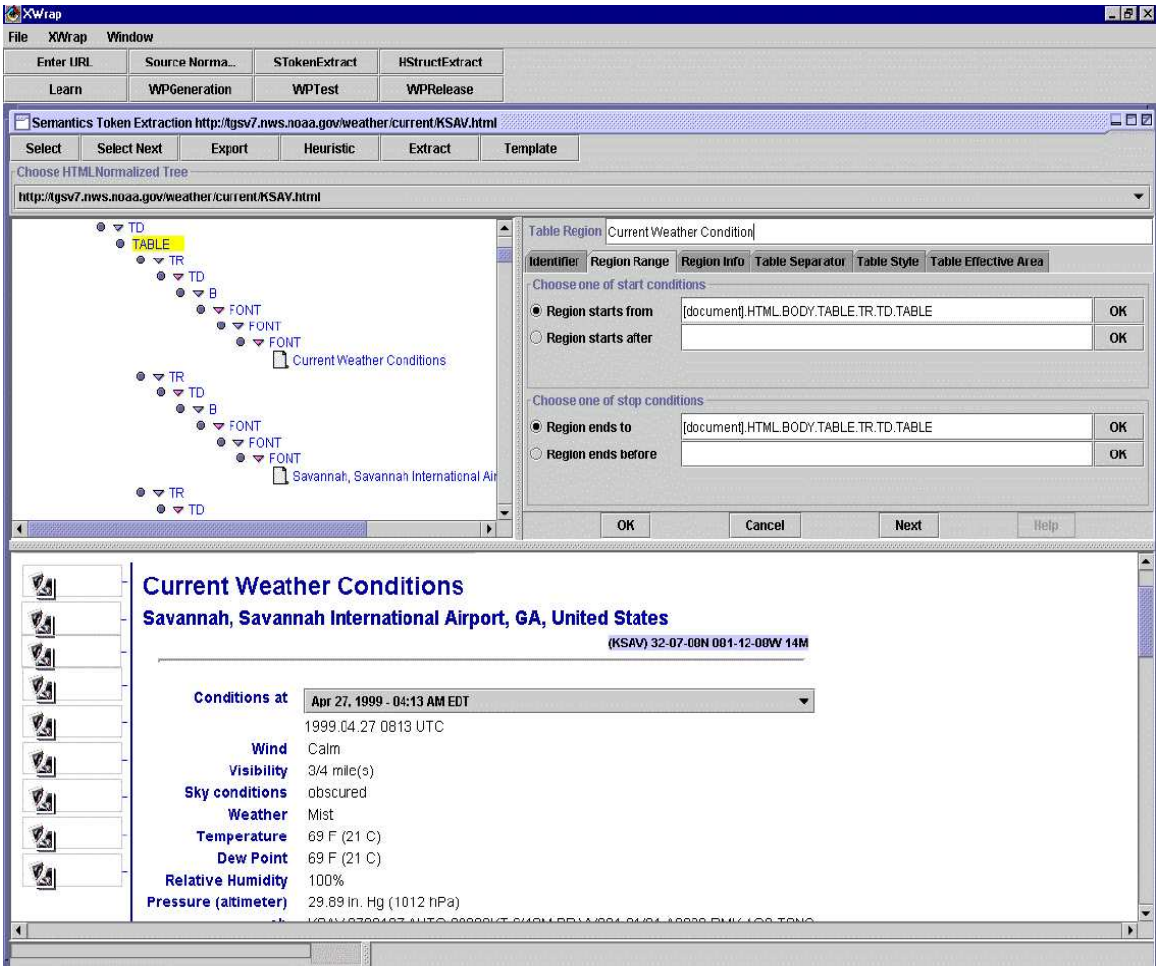

Figure 9: Screenshots of the XWRAP generator system

3.3 Data Wrapping Design in XWRAP

3.3.1 Architecture

The architecture of XWRAP for data wrapping consists of four components - Syntactical Structure Normalization, Information Extraction, Code Generation, Program Testing and Packaging. Figure 10 illustrates how the wrapper generation process would work in the context of data wrapping scenario.

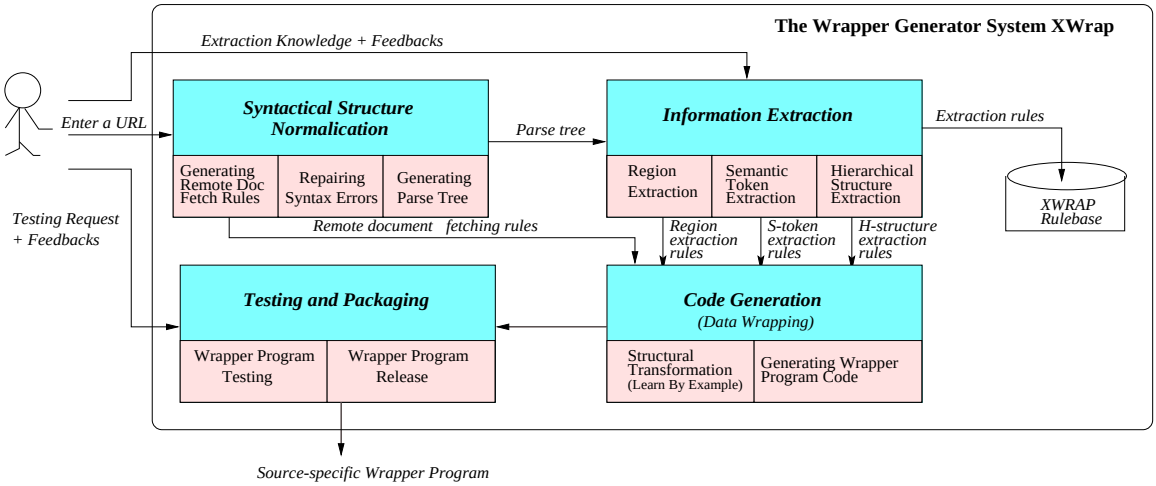

Figure 10: XWRAP system architecture for data wrapping

Syntactical Structure Normalization is the first component and also called Syntactical Normalizer, which prepares and sets up the environment for information extraction process by performing the following three tasks. First, the syntactical normalizer accepts an URL selected and entered by the XWRAP user, issues an HTTP request to the remote server identified by the given URL, and fetches the corresponding web document (or so called page object). This page object is used as a sample for XWRAP to interact with the user to learn and derive the important information extraction rules. Second, it cleans up bad HTML tags and syntactical errors. Third, it transforms the retrieved page object into a parse tree or so-called syntactic token tree.

Information Extraction is the second component, which is responsible for deriving extraction rules that use declarative specification to describe how to extract information

content of interest from its HTML formatting. XWRAP performs the information extraction task in three steps - (1) identifying interesting regions in the retrieved document, (2) identifying the important semantic tokens and their logical paths and node positions in the parse tree, and (3) identifying the useful hierarchical structures of the retrieved document. Each step results in a set of extraction rules specified in declarative languages.

Code Generation is the third component, which generates the wrapper program code through applying the three sets of information extraction rules produced in the second step. A key technique in our implementation is the smart encoding of the semantic knowledge represented in the form of declarative extraction rules and XML-template format (see Section 3.3.7). The code generator interprets the XML-template rules by linking each executable components with each type of rules. We found that XML gives us great extensibility to add more types of rules seamlessly. As a byproduct, the code generator also produces the XML representation for the retrieved sample page object.

Testing and Packing is the fourth component and the final phase of the data wrapping process. The toolkit user may enter a set of alternative URLs of the same web source to debug the wrapper program generated by running the XWRAP automated testing module. For each URL entered for testing purpose, the testing module will automatically go through the syntactic structure normalization and information extraction steps to check if new extraction rules or updates to the existing extraction rules are derived. In addition, the test-monitoring window will pop up to allow the user to browse the test report. Whenever an update to any of the three sets of the extraction rules occurs, the testing module will run the code generation to generate the new version of the wrapper program. Once the user is satisfied with the test results, he or she may click the release button (see Figure 11) to obtain the release version of the wrapper program, including assigning the version release number, packaging the wrapper program with application plug-ins and user manual into a compressed tar file.

The XWRAP architecture for data wrapping is motivated by the design guidelines discussed in Section 3.2, the desire for taking advantage of declarative language for specification of information extraction knowledge, the desire of exploring reusable functionality, and the

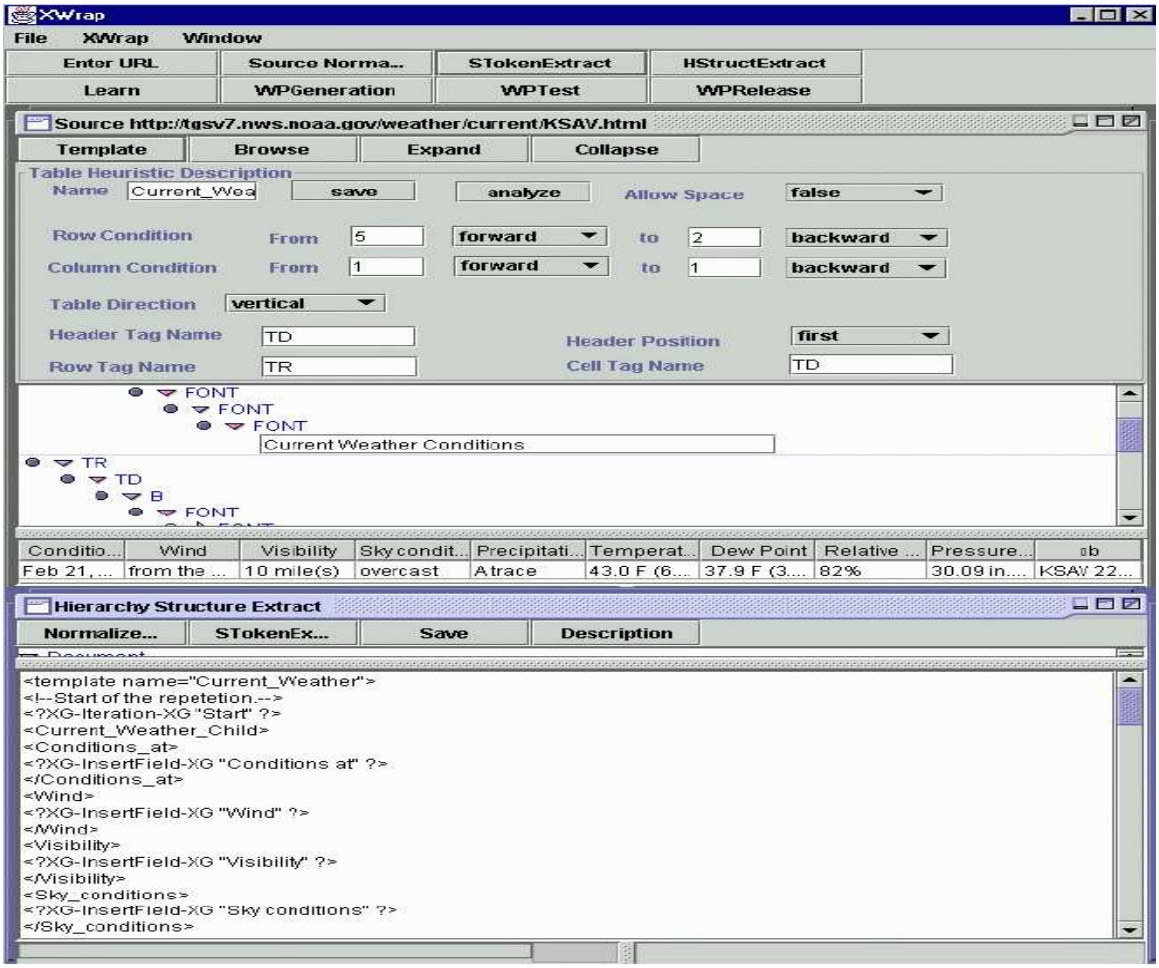

Figure 11: A screenshot of the Hierarchical Structure Extraction Window

design decision for separating data wrapping from functional wrapping. Due to the space restriction, we omit the discussion of XWRAP architecture for function wrapping in this paper.

3.3.2 Phases and Their Interactions

As the wrapper-generation process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the construction process as occurring in one single step. For this reason, we partition the wrapper construction process into a series of subprocesses called *phases*, as shown in Figure 12. A phase is a logically cohesive operation that takes as input one representation of the source document and produces as output another representation.

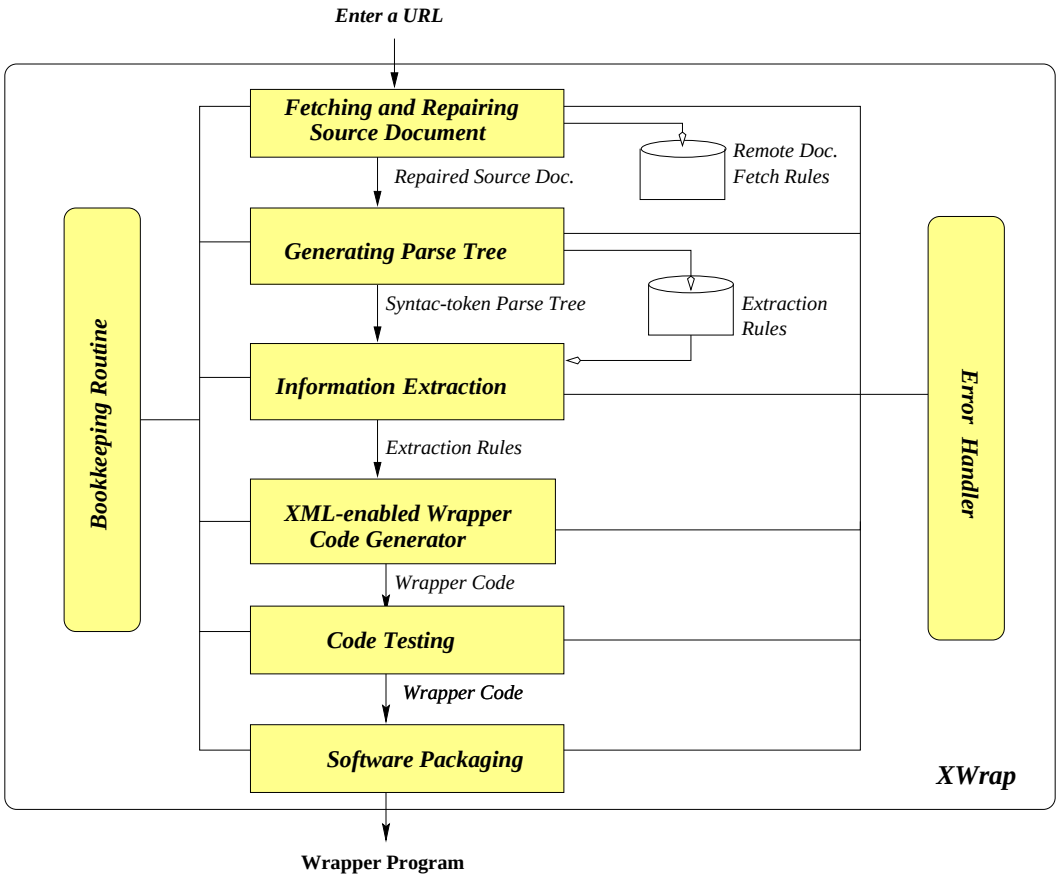

Figure 12: Data wrapping phases and their interactions

XWRAP goes through six phases to construct and release a wrapper. Tasks within a phase run concurrently using a synchronized queue; each runs its own thread. For example, we decide to run the task of fetching a remote document and the task of repairing the bad formatting of the fetched document using two concurrently synchronous threads in a single pass of the source document. The task of generating a syntactic-token parse tree from an HTML document requires as input the entire document; thus, it cannot be done in the same pass as the remote document fetching and the syntax reparation. Similar analysis applies to the other tasks such as code generation, testing, and packaging.

The interaction and information exchange between any two of the phases is performed through communication with the bookkeeping and the error handling routines.

The *bookkeeping* routine of the wrapper generator collects information about all the data objects that appear in the retrieved source document, keeps track of the names used by the

program, and records essential information about each. For example, a wrapper needs to know how many arguments a tag expects, whether an element represents a string or an integer. The data structure used to record this information is called a symbol table.

The *error handler* is designed for the detection and reporting errors in the fetched source document. The error messages should allow a wrapper developer to determine exactly where the errors have occurred. Errors can be encountered at virtually all the phases of a wrapper.

Whenever a phase of the wrapper discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic message. Once the error has been noted, the wrapper must modify the input to the phase detecting the error, so that the latter can continue processing its input, looking for subsequent errors. Good error handling is difficult because certain errors can mask subsequent errors. Other errors, if not properly handled, can spawn an avalanche of spurious errors. Techniques for error recovery are beyond the scope of this paper.

In the subsequent sections, we focus our discussion primarily on information extraction component of the XWRAP, and provide a walkthrough example to illustrate how the three sets of information extraction rules are identified, captured, and specified. As the syntactical structure normalization is a necessary preprocessing step for information extraction, a brief description of the syntactic normalizer is also presented.

3.3.3 Syntactical Structure Normalization

The Syntactical Structure Normalization process is carried out in two phases, as shown in Figure 12. The first phase consists of two concurrently synchronous tasks - remote document retrieval and syntax reparation. The second phase is responsible for generating a syntactic-token parse tree, of the repaired source document.

3.3.3.1 Fetching a Web Page

The Remote Document Retrieval component is responsible for generating a set of rules that describe the list of interface functions and parameters as well as how they are used to fetch a remote document from a given web source. The list of interface functions include the declaration to the standard library routines for establishing the network connection, issuing

an HTTP request to the remote web server through a `HTTP Get` or `HTTP Post` method, and fetching the corresponding web page. Other desirable functions include building the correct URL to access the given service and pass the correct parameters, and handling redirection, failures, or authorization if necessary.

For each wrapper, there is a set of retrieval rules. Each rule specifies the name of the rule, the list of parameters it takes, the built-in functions `GetURL` or `PostURL`, the type of the URL protocols like *http, file, and ftp*, the protocol-specific remote fetch method (such as `HTTP GET` and `HTTP POST`), and the corresponding URL.

XWRAP will automatically take care of packing the URL request parameters in the correct way as required by the `HTTP GET` and `HTTP POST`) protocol variants. In the case of a *PostURL* request, the correct construction of the parameter object needs to be deduced from the web form where the URL request originates. The `HTTP` specification requires that the `POST` parameters be submitted in the order they appear in the form of the page.

Assume we want to construct a wrapper for noaa current weather report web site, and the URL entered at the start of XWRAP is http://weather.noaa.gov/cgi-bin/currwx.pl?cccc=KSAV, asking for the current weather in Savannah. Figure 13 shows a remote document retrieval rule derived from the given URL. It uses the XWRAP library function `URLGet(...)`. The regular expression specified by `Match(K[A-Z]{3})` specifies that the location code is restricted to four capital alphabet characters, starting with the character "K".

When a web site offers more than one type of search capability, more than one retrieval rules may need to be generated.

```
Remote_Document_Fetch_Rules(XWrap-weather.noaa.gov)::
   GetURL(String location-code)
   {
      Protocol:  HTTP;
      Method:  GET;
      URL: http://weather.noaa.gov/cgi-bin/currwx.pl?cccc=$location-code;
      ParaPattern:  location-code, match(K[A-Z]{3});
   }
```

Figure 13: Example rules for fetching remote documents

In addition, all wrappers generated by XWRAP use the streaming mode instead of the blocking mode (recall Section 3.2). Namely, the wrapper will read the web page one block[1] at a time.

3.3.3.2 Repairing Bad Syntax

As soon as the first block of the source document is being fetched over, the syntax repairing thread begins. It runs concurrently with the Remote Document Retrieval thread, and repairs bad HTML syntax such as missing or mismatched end or start tags, end tags in the wrong order, illegal nesting of elements. This step also turns each bachelor tags into a pair of tags, which plays an important role in the region extraction process to be discussed in Section 3.3.5. Each type of HTML errors is described in a normalization rule. The same set of normalization rules can be applied to all HTML pages. Our HTML syntax error reparation module can clean up most of the errors listed in HTML TIDY [63, 71].

3.3.3.3 Generating a Syntactic Token Tree

Once the HTML errors and bad formatting are repaired, the clean HTML document is fed to a source-language-compliant tree parser, which parses the block character by character, carving the source document into a sequence of atomic units, called *syntactic tokens*. Each token identified represents a sequence of characters that can be treated as a single syntactic entity. The tree structure generated in this step has each node representing a syntactic token, and each tag node such as `TR` represents a pair of HTML tags: a beginning tag `<TR>` and an end tag `</TR>`. Different languages may define which is called a token differently. For HTML pages, the usual tokens are paired HTML tags (e.g., `<TR>`, `</TR>`), singular HTML tags (e.g., `<BR>`, `<P>`), semantic token names, and semantic token values.

Example 3.1 Consider the weather report page for Savannah, GA at the national weather service site (see Figure 14) and a fragment of HTML document for this page in Figure 15. Figure 16 shows a portion of the HTML tree structure, corresponding to the above HTML fragment, which is generated by running a HTML-compliant tree parser on the Savannah

[1] A block here refers to a line of 256 characters or a transfer unit defined implicitly by the HTTP protocol.

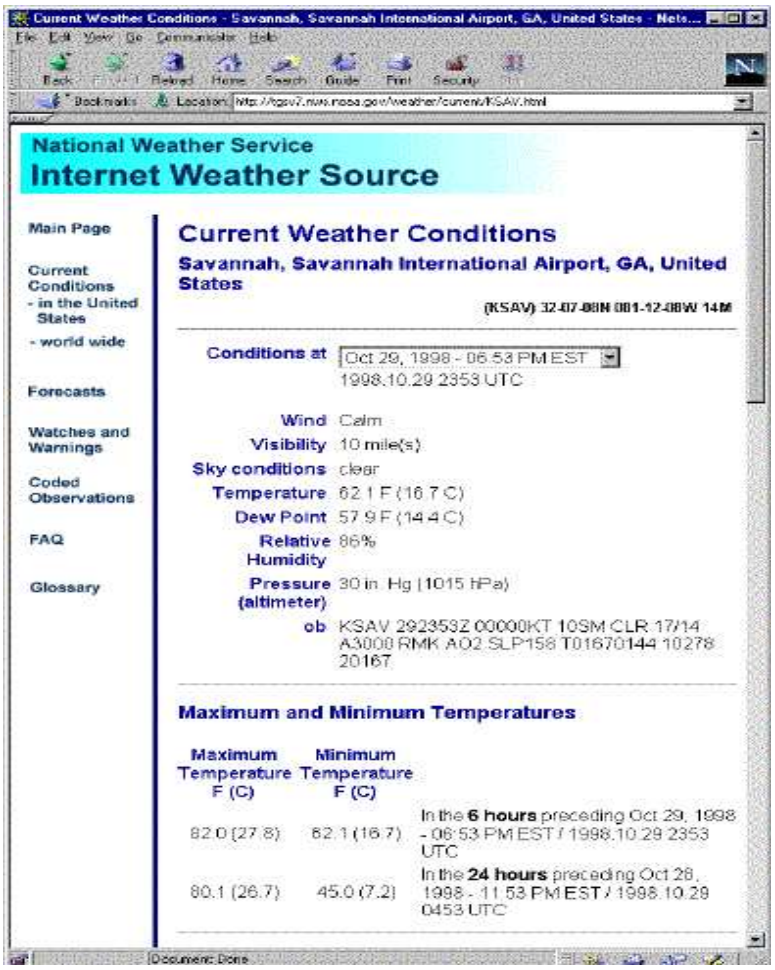

Figure 14: An example weather report page at the nws.noaa.gov site

weather source page. In this portion of the HTML tree, we have the following six types of tag nodes: `TABLE`, `TR`, `TD`, `B`, `H3`, `FONT`, and a number of semantic token nodes at leaf node level, such as `Maximum Tempature`, `Minimum Tempature`, `84.9(29.4)`, `64.0(17.8)`, etc.

Important to note is that every syntactic token parse tree is organized as follows. All non-leaf nodes are tags and all leaf nodes are text strings, each in between a pair of tags. XWRAP defines a set of tree node manipulation functions for each tree node object, including `getNodeType(node_id)`, `getNodeName(node_id)`, `getNodeId(String NN)`, and `getNodePath(node_id)`, in order to obtain the node type - tag node or leaf (value) node, the node name - tag name or text string, the node identifier for a given string, or the path expression from the root to the given node. We use dot notation convention to

```
<TABLE><TR><TD COLSPAN=3><H3><FONT FACE="Arial, Helvetica">Maximum and Minimum Temperatures</FONT>
</H3> </TD></TR><TR><TD ALIGN=CENTER BGCOLOR="#FFFFFF"><B><FONT COLOR="#0000A0"><FONT FACE=
"Arial,Helvetica">Maximum<BR>Temperature<BR>F(C)</FONT></FONT></B></TD><TD ALIGN=CENTER BGCOLOR=
"#FFFFFF"><B><FONT COLOR="#0000A0"><FONT FACE="Arial,Helvetica">Minimum<BR>Temperature<BR>F(C)
</FONT></FONT></B></TD><TD></TD></TR><TR><TD ALIGN=CENTER><FONT FACE="Arial,Helvetica">82.0(27.8)
</FONT></TD><TD ALIGN=CENTER><FONT FACE="Arial, Helvetica">62.1(16.7)</FONT></TD><TD><FONT FACE=
"Arial, Helvetica">In the <B>6 hours</B> preceding Oct 29, 1998 - 06:53 PM EST / 1998.10.29 2353
UTC</FONT></TD></TR><TR><TD ALIGN=CENTER><FONT FACE="Arial,Helvetica">80.1(26.7)</FONT></TD>
<TD ALIGN=CENTER><FONT FACE="Arial, Helvetica">45.0(7.2)</FONT></TD><TD><FONT FACE="Arial,
Helvetica">In the <B>24 hours</B> preceding Oct 28, 1998 - 11:53 PM EST / 1998.10.28 0453 UTC</FONT>
</TD></TR><TR><TD COLSPAN=3><HR SIZE=1 NOSHADE WIDTH="100%"></TD></TR></TABLE> ......
```

Figure 15: An HTML fragment of the weather report page at nws.noaa.gov site

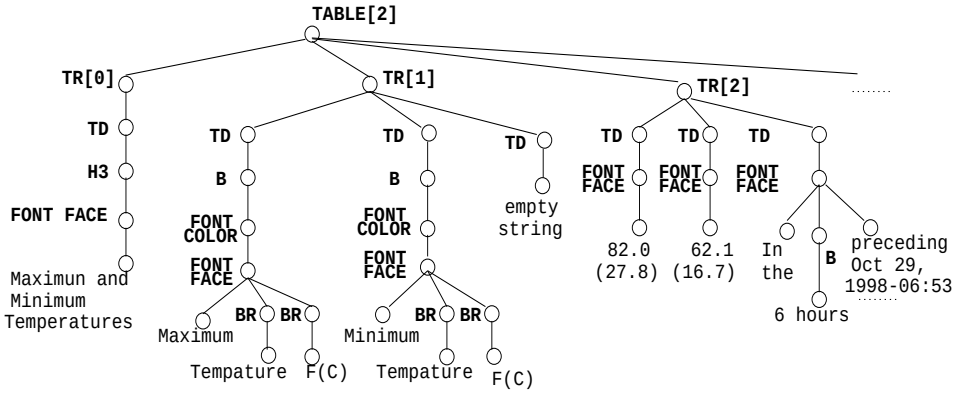

Figure 16: A fragment of the HTML tree for the Savannah weather report page

represent the node path. A single-dot expression such as nodeA.nodeB refers to the parent-child relationship and a double-dot such as nodeA..nodeB refers to the ancestor-descendent relationship between nodeA and nodeB.

Figure 17 presents a subset of the building blocks (or library functions) that the XWRAP information extraction rule language compiler utilizes to generate executable code.

3.3.4 Information Extraction Methodology

The main task of the information extraction component is to explore and specify the structure of the retrieved document (page object) in a declarative extraction rule language. For an HTML document, the information extraction phase takes as input a parse tree generated

Building Block Name	Function
getNodeAttribute(NodeID, AttributeName)	To get the attribute value provided NodeID and Attribute-Name
getNodeID(TreePath)	To obtain the unique identifier of the first node given a path expression
getNodeIDs(TreePathExpression)	To obtain a list of nodes in the tree given a path expression
getNextNodeID(TreePath, CurrentNodeID)	To obtain the id of the next node in the tree after the current node
getNodeName(NodeID)	To extract the node name (tag name)
getNodeType(NodeID)	To get the tag type, such as comment, tag, or text
getNodeValue(NodeID)	To get the concatenation value of all text string descendant
getNodePath(NodeID)	To derive the absolute tree path object of the node
getNodePathExpression(NodeID)	To derive the absolute tree-path regular expression of the node
.....	.....

Figure 17: Building Block Functions of XWRAP parse tree manipulation

by the syntactical normalizer. It first interacts with the user to identify the semantic tokens (a group of syntactic tokens that logically belong together) and the important hierarchical structure. Then it annotates the tree nodes with semantic tokens in comma-delimited format and nesting hierarchy in context-free grammar. More concretely, the information extraction process involves three steps; each step generates a set of extractions rules to be used by the code generation phase to generate wrapper program code.

- Step 1: *Identifying regions of interest on a page*

  This step is performed via an interactive interface, which lets the XWRAP user guide the identification of important regions in the source document, including table regions, paragraph regions, bullet-list regions, etc. The output of this step is the set of region extraction rules that can identify regions of interest from the parse tree.

- Step 2: *Identifying semantic tokens of interest on a page.*

  This step is carried out by an interactive program, called *semantic-token extractor*, which allows a wrapper developer to walk through the tree structure generated by the syntactic normalizer, and highlight the semantic tokens of interest in the source

document page. The output of this step is the set of semantic token extraction rules that can locate and extract the semantic tokens of interest and produce a comma-delimited file containing all the element type and element value pairs of interest.

- Step 3: *Determining the nesting hierarchy for the content presentation of a page.* This step is performed by the hierarchical structure extractor, which infers and specifies the nesting structure of the sections of a web page (document) being wrapped. Such hierarchical specification will be used for content-sensitive information extraction from the source document(s). The outcome of this step is the set of hierarchical structure extraction rules specified in a context-free grammar, describing the syntactic structure of the source document page.

Important to note is that, for structured data sources such as database sources or XML documents, the information extraction process can be conducted automatically, following the table schema or the XML tags. However, this is not the case for unstructured or semi-structured information sources such as HTML documents or text files, because semi-structured or unstructured data is provided with no self-describing properties. Therefore, our goal is to perform the information extraction with minimal user interaction.

In summary, the semantic token extractor analyses the parse tree structure of the source document and its formatting information, and guesses the semantic tokens of interest on that page based on a set of token-recognition heuristics. Similarly, the hierarchical structure extractor also uses the formatting information and the source-specific structural rules to hypothesize the nesting structure of the page. The heuristics used for identifying important regions and semantic tokens in a page and the algorithms used to organize interesting regions of the source page into a nested hierarchy are an important contribution of this work. We describe them in more detail below.

3.3.5 Region Extraction: Identifying Important Regions

The region extractor begins by asking the user to highlight the tree node that is the start tag of an important element. Then the region extractor will look for the corresponding end tag, identify and highlight the entire region. In addition, the region extractor computes

the type and the number of sub-regions and derives the set of region extraction rules that capture and describe the structure layout of the region. For each type of region, such as the table region, the paragraph region, the text section region, and the bullet list region, a special set of extraction rules are used.

3.3.5.1 Table Region Extraction

For regions of the type TABLE, Figure 18 shows the set of rules that will be derived and finalized through interactions with the user.

```
Region_Extraction_Rules(String source_name)::
   Tree_Path(String node_id, String node_path){
      setTableNode = node_id;
      node_path = getNodePath(node_id); }

   Table_Area(String node_id, String TN, String CN, Integer rowMax, Integer colMax){
      setRowTag(node_id) = ?TN;
      setColTag(node_id) = ?CN;
      rowMax = getNumOfRows(node_id);
      colMax = getNumOfCols(node_id); }

   Effective_Area(String node_id, String rowSI, String rowEI, String colSI, String colEI){
   setRowStartIndex(node_id) = ?rowSI;
      setRowEndIndex(node_id) = ?rowEI;
      setColStartIndex(node_id) = ?colSI;
      setColEndIndex(node_id) = ?colEI;
      getEffectiveArea(node_id); }

   Table_Style(String node_id){
      if (ElementType(child(child(node_id, 1), 1)) = 'Attribute'
         if ElementType(child(child(node_id, 1), 2)) = 'Attribute')
            setVertical(node_id) = 1, setHorizontal(node_id) = 0;
         else
            setHorizontal(node_id) = 1, setVertical(node_id) = 0; }

   getTableInfo(String node_id, String TNN, String TN, String TP){
      setTableNameNode(node_id) = TNN;
      TN = getTableName(TNN);
      TP = getNodePath(TNN); }
```

Figure 18: Extraction rules for a table region in an HTML page

The rule `Tree_Path` specifies how to find the path of the table node. The rule `Table_Area` finds the number of rows and columns of the table. The rule `Effective_Area` defines the effective area of the table. An effective area is the sub-region in which the interesting rows and columns reside. By differentiating the effective area from a table region, it allows us, for example, to remove those rows that are designed solely for spacing purpose. The fourth rule `Table_Style` is designed for distinguishing vertical tables where the first column stands for a list of attribute names from horizontal tables where the first row stands for a list of attribute names. The last rule `getTableInfo` describes how to find the table name by giving the path and the node position in the parse tree.

Example 3.2 Consider the weather report page for Savannah, GA at the national weather service site (see Figure 14), and a fragment of HTML parse tree as shown in Figure 16. To identify and locate the region of the table node `TABLE[2]`, we apply the region extraction rules given in Figure 18 and obtain the following source-specific region extraction rules for extracting the region of the table node `TABLE[2]`.

1. By applying the first region extraction rule, XWRAP can identify the tree path for `TABLE[2]` to be
   `HTML.BODY.TABLE[0].TR[0].TD[4].TABLE[2]`.

2. To identify the table region, we first need the user to identify the row tag `TR` and the column tag `TD` of the given region of the `TABLE[2]` node. Based on the row tag and column tag, the region extractor may apply the second extraction to deduce that the table region of `TABLE[2]` consists of maximum 5 rows and maximum 3 columns.

3. The extraction rule `Effective_Area` will be used to determine the effective area of the table node `TABLE[2]`. It requires the user's input on the row start index `rowSI = 2`, the row end index `rowEI = 4`, the column start index `colSI = 1` and the column end index `colEI = 3`. With these index information, the region extractor can easily identify the effective table region, the area that does not include the row for table name and the empty row.

4. By applying the rule `Table_Style`, we can deduce that this table is a horizontal table, with the first row as the table schema.

5. To determine how to extract the table name node, we need the user to highlight the table name node in the parse tree window (recall Figure 11). Based on the user's input, XWRAP can infer the path expression for the table name node is `TABLE[2].TR[0].TD[0].H3[0].FONT[0].FONT[0]`.

   Then by applying the fifth region extraction rule `getTableInfo`, we can extract the table name. Note that the function `getTableName(node_id)` calls the the following semantic token extraction rule to obtain the actual string of the table name (see Section 3.3.6 for details on semantic token extraction).

```
<ST_extract> ST_extract(String TN)
   <rule_exp>
     extract TN = TABLE[2].TR[0].TD[0].getChildNode(1).getNodeValue()
     where   TABLE[2].TR[0].TD[0].getChildNode(1).getNodeType() = 'tag'
     and     TABLE[2].TR[0].TD[0].getChildNode(1).getNodeName() = 'H3';
   </rule_exp>
</ST_extract>
```

   The path of this table name node can be computed directly by invoking `getNodePath(getNodeId('TABLE[2]'))`, which returns

   `HTML.BODY.table[0].TR[0].TD[4].TABLE[2].TR[0].TD[0].H3[0].FONT[0].FONT[0].`

It is important to note that the design of our region extraction rules is robust in the sense that the extraction rules are carefully designed to compute the important information (such as the number of tables in a page, the number of attributes in a table, etc.) at runtime. For example, let us assume that the nws.noaa.gov wrapper was constructed using the example page from a Portland weather report at a specific time, which happens to contain only three tables instead of the normal layout of four tables. The first table contains only 7 rows instead of the normal layout of 9 rows. When the very same wrapper runs to extract the page of Savannah, GA, our wrapper will automatically deduce that the page has four tables

and the first table has 9 rows, rather than assuming all the weather report at nws.noaa.gov obey the same format.

Furthermore, our region extraction rules are defined in a declarative language and therefore independent of the implementation of the wrapper code. This higher level of abstraction allows the XWRAP wrappers to enjoy better extensibility and ease in maintenance and in adapting to changes at the source.

3.3.5.2 Text Region Extraction

Another important region type is **Text Region**. Generally speaking, data items in a text region are less tightly connected with each other than data items in a table region. A typical text region contains a title text and a list of sub-regions. Each sub-region can be either a text region type or a table region type or a composite region again. Two issues are involved in extracting a text region correctly. The first issue is related to the identification and separation of a complex text region into primitive regions. The second issue is related to the identification of the title text from the body text in a text region.

Consider an example drawn from the CIA world fact book web site. Figure 19 shows two example pages, containing text regions. Both were obtained from the CIA world factbook web site. The two example pages describe information about USA and Taiwan. `Geography` is the first text region. It contains a list of sub-regions, called *paragraphs*, such as **Location** and **Geographic coordinates**. Each paragraph has either a value such as **Geographic coordinates** or a list of sub-regions. In this example, the text region `Geography` has a sub-region **Area**, which consists of four paragraph types again. They are **total**, **land**, **water** and **note**, and each has a corresponding value.

Similar to the table region extraction rules described in the previous section, the text region extraction rules are given in Figure 20.

The extraction rules shown in Figure 20 describe how to extract the Geography region. They check the region identifier condition first. The region extraction will proceed only when the identifier condition is true. In this example, the identifier condition is the following: First, the region contains a string constant identified as "Geography". Second, these rules

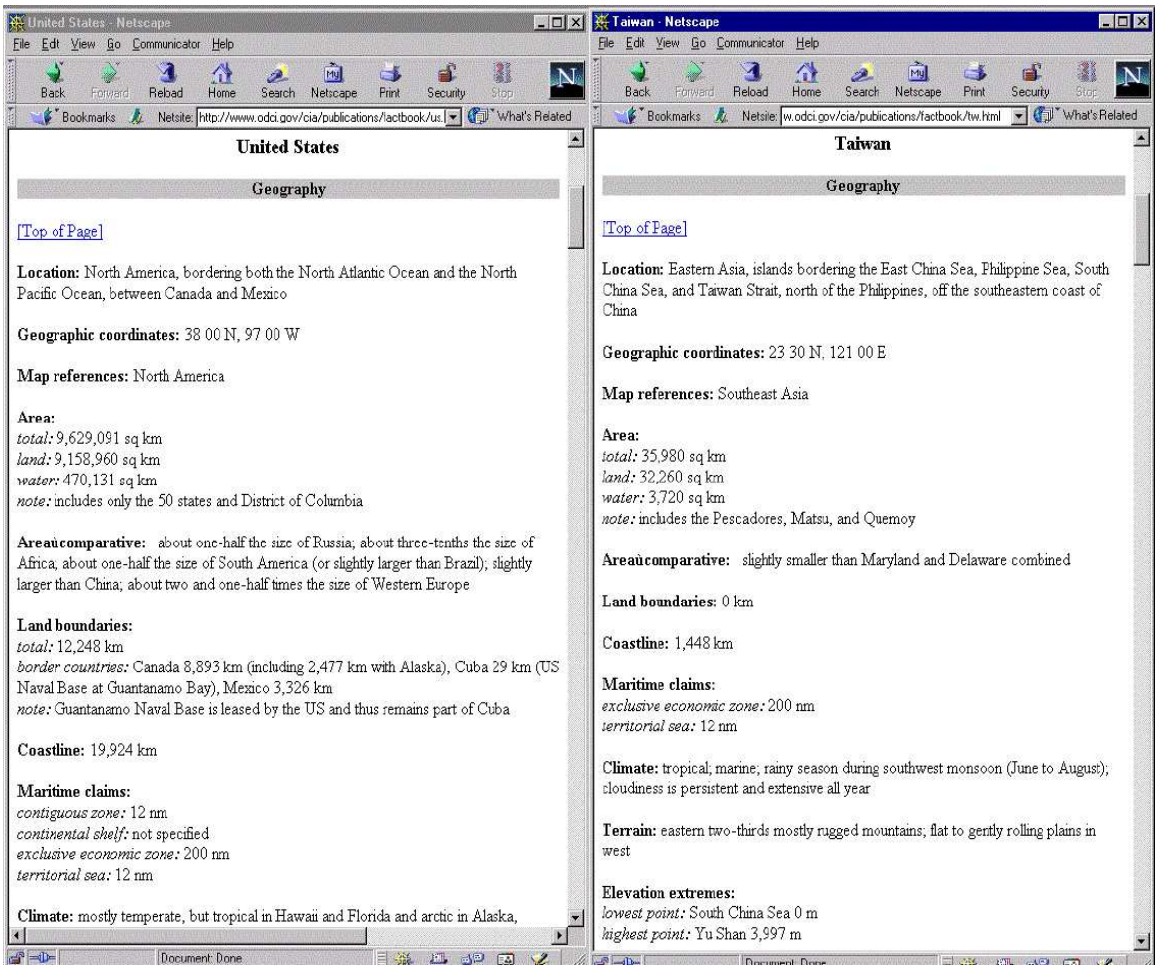

Figure 19: An example country introduction page on CIA

instruct the wrapper program to extract the region name. Third, the region range will be located. A Start node and an End node bound a region range. Finally, all the attribute-value pairs in the region range, such as "Map references: Southeast Asia." will be extracted.

Text regions are a lot simpler than table regions since the text regions contain less semantic layout information. However, text regions have higher irregularity. For example, in Figure 19 the text region of **Land boundaries** in the `United States` page has a list of sub-regions such as `total`, `border countries`, and `note`; whereas the text region of **Land Boundaries** in the `Taiwan` page from the very same source has no sub-regions but a simple value `0 km`. Such types of irregularities often cause difficulties in writing or generating a wrapper program for such a web source. Furthermore, text regions tend to have deeper

```
<region_extract type="text">
   <check_region_identifier>
      $regionIdentifierNode.$regionIdentifier Op($regionIdentifierValue);
   </check_region_identifier>

   <get_region_info>
      <ST_extract>
         <rule_exp>
            extract regionName = $regionNameNode.getNodeValue();
         </rule_exp>
      </ST_extract>
   </get_region_info>

   <search_region_range>
   regionNodes = searchNodes($startFromNode, $startAfterNode, $endToNode, $endBeforeNode);
   </search_region_range>

   <ST_extract>
      // get all the attribute-value pairs
      ...
   </ST_extract>
</region_extract>
```

Figure 20: Extraction rules for a text region in CIA pages

nested structure compared with table regions. Very often, people use nested text regions to express information content that has more hierarchical semantics.

Add a screen shot on the region extraction and a paragraph for illustration.

3.3.6 Semantic Token Extraction: Finding Important Semantic Tokens

In general each semantic token is a sub-string of the source document that is to be treated as a single logical unit. There are two kinds of token: specific strings such as HTML tags (e.g., `TABLE`, `FONT`), and semantic tokens such as those strings in between a pair of HTML tags. To handle both cases, we shall treat a token as a pair consisting of two parts: a token name and a token value. For a tag token such as `FONT`, the tag name is `FONT` and the tag value is the string between a beginning tag `<FONT>` and its closing tag `</FONT>`. A semantic token such as `Maximum and Minimum Temperature F(C)` or `Current Weather`

`Conditions` in between the start and end tags of the tag token `FONT` will be treated as either a name token or a value token, depending on the context or the user's choice. Similar treatment applies to the token such as `Savannah, Savannah International Airport, GA, United States`. In addition, a help function - `getNodeValue(node_id)` is provided for semantic token extraction rules. It extracts and concatenates all text strings from the leaf nodes of the subtree identified by the given `node_id`.

The main task of a semantic token extractor (S-token extractor for short) is to find semantic tokens of interest, define extraction rules to locate such tokens, and specify such tokens in a comma-delimited format[2], which will be used as input in the code generation phase. The first line of a comma-delimited file contains the name of the fields that denote the data. A special delimiter should separate both field names and the actual data fields. The XWRAP system supports a variety of delimiters such as a comma (,), a semicolon (;), or a pipe (|). To identify important semantic tokens, the S-token extractor examines successive tree nodes in the source page, starting from the first leaf node not yet grouped into a token. The S-token extractor may also be required to search many nodes beyond the next token in order to determine what the next token actually is.

XWRAP generates semantic extraction rules using templates. For each type of region, XWRAP pre-define a template of extraction rules. The template contains extraction rules with undetermined parameters. XWRAP asks wrapper developers some questions and the answers to those questions will be the values of the parameters. For example, XWRAP may ask, "is this table vertical or horizontal?"

Example 3.3 Consider a fragment of the parse tree for the Savannah weather report page shown in Figure 16. From the region extraction step, we know that the leaf node name `Maximum and Minimum Temperatures` of the left most branch `TR[0]` is the heading of a table region denoted by the node `TABLE[2]`. Also based on the interaction with the user, we know that the leaf nodes of the subtree anchored at `TABLE[2].TR[1].TD[0]` should be treated as a semantic token with the concatenation of all three leaf node names, i.e., the

[2] A comma-delimited format is also called delimited text format. It is the lowest common denominator for data interchange between different classes of software and applications.

string `Maximum Temperature F(C)`, as the token name; and the leaf nodes of the tree branch `TABLE[2].TR[2].TD[0]`, i.e., the string `84.9 (29.4)`, is the value of the corresponding semantic token. Thus a set of semantic token extraction rules can be derived for the rest of the subtrees anchored at `TR[3]` and `TR[4]`, utilizing the function `getNodeValue()`.

```
<ST_extract>
  ST_extract(String ST_name[], String ST_val[][])
  <!-- Start of the repetition -->
  <loop> integer rowNum = 3, 4
    <loop> integer columnNum = 0,1,2
      <rule_exp>
        extract ST_val[rowNum, columnNum] = ~TABLE[2].TR[rowNum].TD[columnNum].getNodeValue()
        where   ~TABLE[2].TR[1].TD[columnNum].getNodeValue() = ST_name[columnNum];
      </rule_exp>
    </loop>
  </loop>
</ST_extract>
```

By traversing the entire tree of the node `TABLE[2]` and applying the derived extraction rules, we may extract all the token values for each given token name in this region. Similarly, by traversing the entire tree of Savannah page, the semantic-token extractor produces as output a comma-delimited file for the Savannah weather report page. Figure 21 shows the portion of this comma-delimited file that is related to `TABLE[2]` node. The first line shows the name of the fields (the rows) that are being used. The second and third lines are two data records.

```
.......
Maximum Tempature F(C); Minimum Tempature F(C); TD
82.0(27.8);62.1(16.7);In the <B>6 hours</B> preceding Oct 29,
1998 - 6:53 PM EST / 1998.10.29 2353 UTC
80.1(26.7);45.0(7.2);In the <B>24 hours</B> preceding Oct 28,
1998 - 11:53 PM EST / 1998.10.28 0453 UTC
.......
```

Figure 21: A fragment of the comma-delimited file for the Savannah weather report page

3.3.7 Hierarchical Structure Extraction: Obtaining the Nesting Hierarchy of the Page

The goal of the hierarchical structure extractor is to make explicit the meaningful hierarchical structure of the original document by identifying which parts of the regions or token streams should be grouped together. More concretely, this step determines the nesting hierarchy (syntactic structure) of the source page, namely what kind of hierarchical structure the source page has, what are the top-level sections (tables) that form the page, what are the sub-sections (or columns, rows) of a given section (or table), etc.

Similar to the semantic token extractor, the hierarchical structure can be extracted in a semi-automatic fashion for a larger number of pages. By semi-automatic we mean that the task of identifying all sections and their nesting hierarchy is accomplished through minimal interaction with the user. The following simple heuristics are most frequently used by the hierarchy extractor to make the first guess of the sections and the nesting hierarchy of sections in the source document to establish the starting point for feedback-driven interaction with the user. These heuristics are based on the observation that the font size of the heading of a sub-section is generally smaller than that of its parent section.

- Identifying all regions that are siblings in the parse tree, and organizing them in the sequential order as they appear in the original document.
- Obtaining a section heading or a table name using the paired header tags such as `<H3>`, `</H3>`.
- Inferring the nesting hierarchy of sections or the columns of tables using font size and the nesting structure of the presentation layout tags, such as `<TR>`, `<TD>`, `<P>`, `<DL>`, `<DD>`, and so on.

We develop a hierarchical structure extraction algorithm that, given a page with all sections and headings identified, outputs a hierarchical structure extraction rule script expressed in an XML-compliant template for the page.

Figure 22 shows the fragment of the XML template file corresponding to the part of a

NWS weather report page shown in Figure 16. It defines the nesting hierarchy, annotated with some processing instructions.

```
......
<Maximum_and_Minumum_Temperatures>
<Description>Maximum and Minimum Temperatures</Description>
<!-- Start of the repetition -->
<?XG-Iteration-XG "Start"?>
  <Maximum_and_Minimum_Temperatures_Child>
     <Maximum_Temperature>
     <Description>MaximumTemperature F(C)</Description>
     <Value><?XG-InsertField-XG "Maximum Tempature"></Value>
     </Maximum_Temperature>

     <Minimum_Temperature>
     <Description>MinimumTemperature F(C)</Description>
     <Value><?XG-InsertField-XG "Minimum Temperature"></Value>
     </Minimum_Temperature>

     <TD>
     <Description></Description>
     <Value><?XG-InsertField-XG "TD"></Value>
     </TD
  </Maximum_and_Minumum_Temperatures_Child>
<?XG-Iteration-XG "End"?>
<!-- End of the repetition -->
</Maximum_and_Minumum_Temperatures>
......
```

Figure 22: A fragment of the hierarchical structure extraction rule for nws.noaa.gov current weather report page

The use of XML templates to specify the hierarchical structure extraction rule facilitates the code generation of the XWRAP for several reasons. First,

XML templates are well-formed XML files that contain processing instructions. Such instructions are used to direct the template engine to the special placeholders where data fields should be inserted into the template. For instance, the processing instruction **XG-InsertField-XG** has the canonical form of XG-InsertField-XG is `<?XG-InsertField-XG "FieldName"?>`. It looks for a field with a specified name "FieldName" in the comma-delimited file and inserts that data at the point of the processing instruction.

Second, an XML template also contains a repetitive part, called XG-Iteration-XG, which is necessary for describing the nesting structure of regions and sections of a web page. The XG-Iteration-XG processing instruction determines the beginning and the end of a repetitive part. A repetition can be seen as a loop in classical programming languages. After the template engine reaches the "End" position in a repetition, it takes a new record from the delimited file and goes back to the "Start" position to create the same set of XML tags as in the previous pass. New data is inserted into the resulting XML file.

Due to the fact that the heuristics used for identifying sections and headings may have exceptions for some information sources, it is possible for the system to make mistakes when trying to identify the hierarchical structure of a new page. For example, based on the heuristic on font size, the system may identify some words or phases as headings when they are not, or fail to identify phases that are headings, but do not conform to any of the pre-defined regular expressions. We have provided a facility for the user to interactively correct the system's guesses. Through a graphical interface the user can highlight tokens that the system misses, or delete tokens that the system chooses erroneously. Similarly, the user can correct errors in the system generated grammar that describes the structure of the page.

The XWRAP code generator generates the wrapper code for a chosen web source by applying the comma-delimited file (as shown in Figure 21 for the running example), the region extraction rules (as given in Example 3.2), and the hierarchical structure extraction rules (see Figure 22), all described using the XWRAP's XML template-based extraction specification language. Due to the space limitation, the details on the language is omitted here.

Finally, to satisfy the curiosity of some readers, we show in Figure 23 a fragment of the XML document transformed from the original HTML page by the XWRAP_nws.noaa.gov wrapper program, which was generated semi-automatically using XWRAP toolkit for the NWS web source.

```
......
<Maximum_and_Minumum_Temperatures>
<Description>Maximum and Minimum Temperatures</Description>
<Maximum_and_Minimum_Temperatures_Child>
   <Maximum_Temperature>
      <Description>MaximumTemperature F(C)</Description>
      <Value>82.0(27.8)</Value>
    </Maximum_Temperature>

   <Minimum_Temperature>
      <Description>MinimumTemperature F(C)</Description>
      <Value>62.1(16.7)</Value>
   </Minimum_Temperature>

   <TD>
      <Description></Description>
      <Value>
      In the 6 hours preceding Oct 29, 1998 - 06:53 PM EST / 1998.10.29 23:53 UTC
      </Value>
   </TD>
</Maximum_and_Minumum_Temperatures_Child>
<Maximum_and_Minimum_Temperatures_Child>
   <Maximum_Temperature>
      <Description>MaximumTemperature F(C)</Description>
      <Value>80.1(26.7)</Value>
    </Maximum_Temperature>

   <Minimum_Temperature>
      <Description>MinimumTemperature F(C)</Description>
      <Value>45.0(7.2)</Value>
   </Minimum_Temperature>

   <TD>
      <Description></Description>
      <Value>
      In the 24 hours preceding Oct 28, 1998 - 11:53 PM EST / 1998.10.28 0453 UTC
      </Value>
   </TD>
</Maximum_and_Minumum_Temperatures_Child>
</Maximum_and_Minumum_Temperatures>
......
```

Figure 23: A fragment of the XML document for the NWS Savannah weather report page

3.4 Functional Wrappers in XWRAP

3.4.1 Overview

Functional wrappers are primarily built on top of data wrappers by incorporating additional functionality such as richer query capabilities, or enhanced quality of services, or domain-specific functional requirements.

Data Wrappers v.s. Functional Wrappers

As discussed in Section 3.2, data wrappers are used for extracting information of interest from the web sources without providing additional functionality. A data wrapper can be seen as a data projection processor that performs a type of projections over the original data source. Generally speaking, the types of queries one can issue over a data wrapper should be at least equivalent to the types of queries the original web source can accept.

Consider the noaa current weather condition service web source. Queries such as "*tell me the current weather condition in Portland*" can be answered either by the web source or the CQ_noaa wrapper directly. The main difference lies primarily in the result page. The noaa web site may return the result with some advertisements on the page, while the CQ_noaa wrapper will return the weather information with the advertisements removed. It is also possible to design a data wrapper that returns only a portion of the web source. For instance, one may design a data wrapper using XWRAP, which only extract the high and low temperature, the sky condition and the wind information. However, the query results returned from a wrapper query will contain only those information content that are extracted (wrapped) by the designated data wrapper.

In contrast, a functional wrapper often adds more query capabilities to the original Web source. Consider the national weather services Web source nws.noaa.gov presented in Figure 14. The types of queries acceptable by the noaa web site are those search queries, each with a location specified by a preserved four-character string, such as `KPDX` for Portland. For more sophisticated queries, such as the selection query: "*tell me the detailed weather condition in Portland if the temperature drops below 32F*", a functional wrapper is required.

Although the initial work on database interoperability focused primarily on data wrappers, there is an increasing interest on functional wrappers for a number of reasons. First,

functional wrappers provide interoperability at higher levels than data interchange. For example, a functional wrapper layer is a natural solution for adding standard query interfaces (e.g., Z39.50) to a specialized information service building on an application specific native interface. Second, functional wrappers can provide higher levels of abstraction that facilitate the construction of mediators [48]. Third, functional wrappers may provide a more convenient view on the data source by adding selection and other forms of data manipulations on top of the original source capabilities.

Construction of Functional Wrappers

A functional wrapper differs from a data wrapper because of the new functionality, which introduces a second layer of wrapping, around the underlying data wrapper. When a query request arrives, the functional wrapper decomposes the request into two components: a filter query and the remote connection call for data fetching. The latter component is the same as in a data wrapper, and the processing proceeds the same way. After the returned data is transformed into the target structure, i.e., the end of data wrapper processing, the functional wrapper processing resumes. The filter query part of the wrapper request is evaluated against the target structure (which was built for this purpose), and returns only the matching results to the mediator application. An obvious example of such functional wrappers is to use XML-QL [18] as a tool to implement the content-filtering over XML documents produced by the XWRAP data wrappers.

In the following subsection, we discuss three types of functional wrappers that are currently under consideration within the Continual Queries project.

The in-depth research results specific to functional wrappers will be reported in upcoming papers.

3.4.2 Query Capability Enhancement

A popular way to use wrappers is to build functional wrappers that provide richer set of enhanced query capabilities than the original web source. It is well known that typical query capabilities that most of the Web data sources provide are keyword-based search with restrictions on a small and quite limited set of information properties. For instance, most

of online bookstores support only keyword-based search on title, authors, subject of books but no complex conditional search on price and publishing year etc. Therefore, complex search requests bounded by conditional projections and selections over the search results will need the assistance of wrappers and mediators.

Example:

Suppose that we want to build a weather integration agent (broker or mediator) that is able to report weather conditions in several cities when the weather change reaches a specified update threshold. Due to the fact that most of the weather sources provide limited query capability. Typically, one can only ask for weather information by given a city name. In order to build a more intelligent weather integration agent, we need to build a wrapper that is capable of performing projection and selection in terms of different weather conditions. Assume that we choose the noaa.gov weather service as the information source of interest. We can build a noaa wrapper in two phases. In the first phase, we build a data wrapper to the current weather report web source (maintained by the noaa.gov national weather service). Assume by design we want this data wrapper to be able to extract (project) all the current weather condition information and the maximum and minimum temperature information, but no requirement for 24-hour summary. Thus, the main task of this data wrapper is able to extract such information from each of the current weather report pages (HTML documents). An effective approach to building such a data wrapper is to transfer the HTML documents into a more structured data format such as XML. Figure 23 shows an example output of the XWRAP data wrapper for noaa.gov. In the second phase, a functional wrapper will be built on top of the data wrapper, which provides efficient processing of conditional selection queries such as "*send me the Portland weather report if the temperature is below 42F in Portland*". Furthermore, by building the weather integration mediator on top of the functional wrapper, we will be able to answer queries such as "*tell me the Portland weather report if the temperature is below 42F in either Astoria or NewPort or Salem or Aurora or Vancouver, Washington*". Interesting to note is that neither of the two types of queries can be handled (answered) if we reply on the source query capability provided at the weather report service web site (nws.noaa.gov).

3.4.3 Change Detection

Another important functional wrapper that is being built in the context of the Continual Queries project is the change detector for Web information sources. There are two types of change detectors. The first type is called *simple change detector*, capable of detecting simple changes for any Web page with limited change detection capabilities. The second type is called *advanced change detector*, capable of detecting and identifying sophisticated updates and representing changes in terms of a richer set of change operations such as insert, delete, update, move, copy, etc. However, the advanced change detectors require deeper understanding of the content of web pages and more sophisticated change detection processing over the content of web pages. Thus, it is important to have a wrapper wrap and transfer the web pages into more structured and preferably self-describing documents before running the change detection test.

Example:

Suppose that you want to monitor changes of certain keywords or phases in any web pages or being notified when a change occurs to a Web site but you do not care what types of changes were happened. Then a simple change detector would do the job for you.

However, if you are interested in monitoring particular types of weather condition updates from a Web source, such as the noaa.gov weather service Web source, then a functional wrapper with advanced change detection capability is acquired. Typically, such a functional wrapper is capable of monitoring almost any changes to the Web source you are interested in. For instance, the change detector for noa.gov weather service can watch the weather condition updates on wind, temperature, dew point, and pressure. It sends out an alert only when the update reaches the specified threshold on these specific weather properties (such as temperature drops by 10F).

3.4.4 Quality of Service

Another very interesting type of functional wrappers is the support for quality of service, such as certain response time guarantee, resolution guarantee, result size guarantee for information delivery. Ideally, each Web search request will be associated with, for instance, a

QoS response threshold. If the user's request will be served within the threshold, the request will be processed. Alternatively, one may associate each search request with a response measurement in various scales such as seconds or millisecond or percentage completed or percentage towards completion.

Currently we are actively looking into the techniques and commercial off the shell products that we could benefit from in developing continual query services with various degrees of QoS guarantee.

3.5 XWRAP Implementation Architecture and Related Issues

3.5.1 XWRAP System Architecture

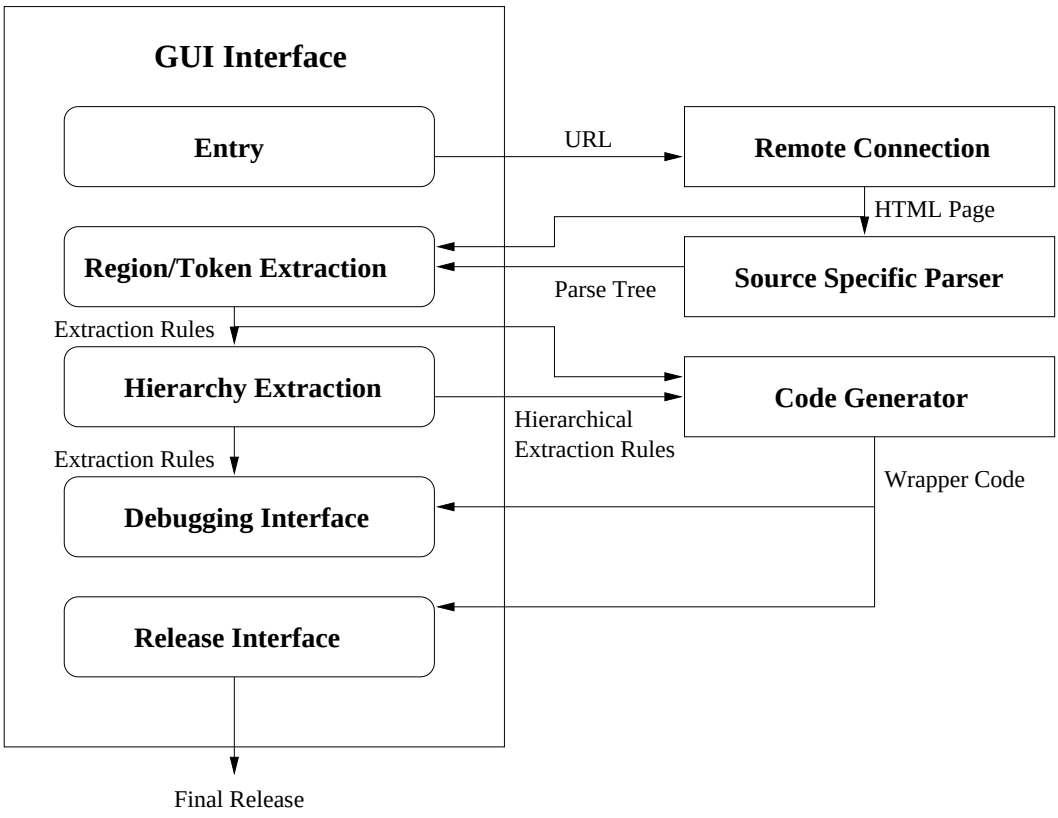

Figure 24: XWRAP Implementation Architecture

Figure 24 shows the XWRAP implementation structure. XWRAP consists of four components, GUI interface, Remote Connection, a source specific parser, Code Generator.

GUI interface allows wrapper developers to specify extraction rules and hierarchical rules interactively, and other components provide some functional support such as the remote network connection and the source specific parser.

GUI interface contains five wizards:

- **Entry**. The Entry wizard allows wrapper developers to specify a URL that they want to wrap, and it passes the URL to the remote connection component. The remote connect component will fetch the Web page and parse it using a source specific parser to produce an HTML tree. Both HTML document and HTML tree will be returned to the Region/Token Extraction wizard.

- **Region/Token Extraction**. This wizard helps wrapper developers to derive the extraction rules based on the HTML page and the HTML tree. Extraction rules will be sent to Hierarchy Extraction and Code Generator.

- **Hierarchy Extraction**. The Hierarchy Extraction wizard is useful tool to capture and specify the hierarchical structure of the source page being wrapped. It first deduces a sketch of the hierarchy information based on the region/token extraction rules and some heuristics. Then it allows wrapper developers to make necessary corrections and additions. The final hierarchical extraction rules will be sent to the XWRAP Code Generator.

- **Debugging Interface**. Code Generator generates a wrapper program based on region/token extraction rules and hierarchical extraction rules. The wrapper program will be passed to the Debugging interface. The Debugging interface allows wrapper developers to test the wrapper code using other URLs of the same Web source. If the wrapper doesn't work, it will print error report and feedback. The wrapper developer will then decide which steps to re-run accordingly.

- **Release Interface**. The Release component allows wrapper developers to release a wrapper code if they are satisfies with the test results on other URLs.

3.5.2 Wrapper Execution Model

We have discussed the research issues, the development, and the use of XWRAP wrapper generator system in previous sections. In this section, we discuss the execution model of the wrapper programs generated by XWRAP.

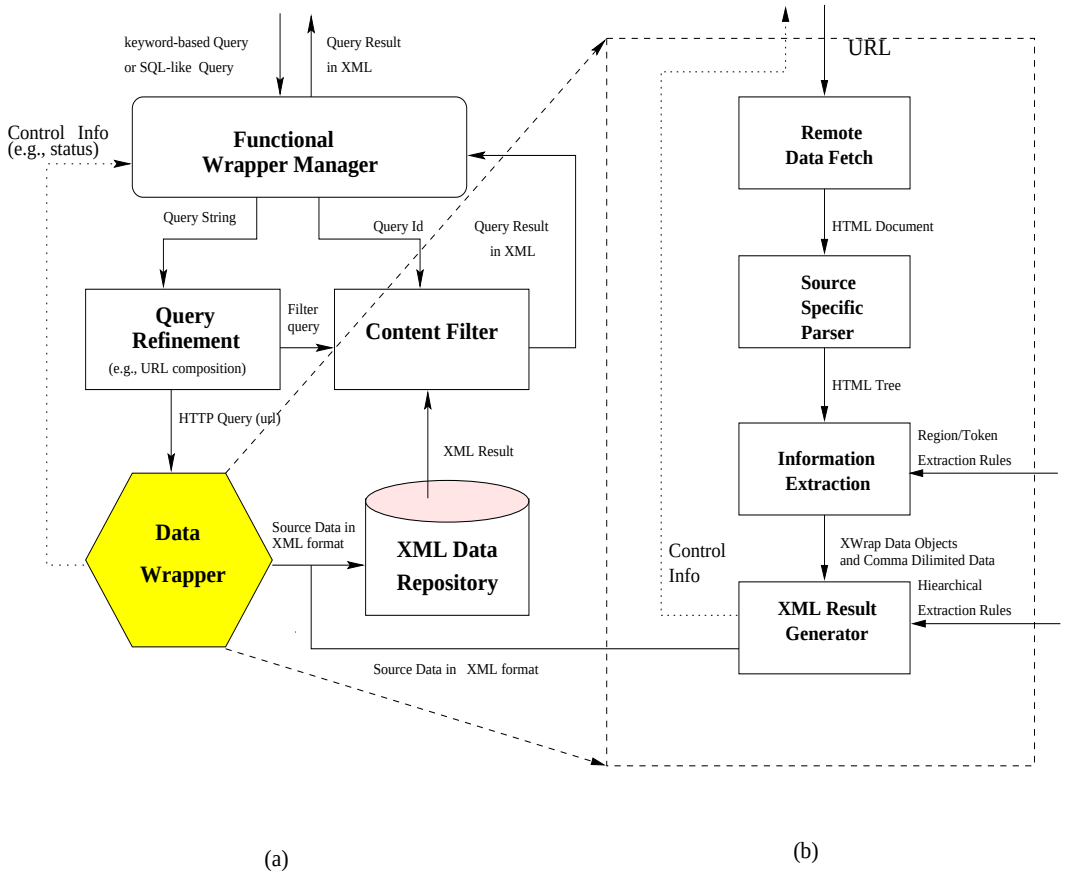

Figure 25: (a) Functional Wrapper Execution Model (b) Data Wrapper Execution Model

3.5.2.1 Data Wrapper Model

Figure 25 presents a sketch of the execution model of wrappers generated by XWRAP. A XWRAP-generated functional wrapper consists of four main components. The first component is the wrapper manager that coordinates the communication and data or control flow exchange between the wrapper components. The second component is the query refinement module that parses and transforms the external query requests into a HTTP fetch

command (URL) and a filter query. The third component is the data wrapper that transforms the source document into a more structured data document such as XML. The fourth component is a functional module that operates functional computation on the XML interpretation. For an advanced query wrapper, the content filter module is used, which runs the filter query over the output document of the data wrapper. The execution is initiated by a service call from external applications such as mediator or agent). The call goes through the query refinement and generates one HTTP query and one filter query. The HTTP query is sent to the data wrapper to get an XML interpretation of the whole source data, and then the functional component operates functional computation on the XML interpretation. The execution model of data wrappers will be discussed in Section 3.5.2.1.

Consider a query asks for "the current temperature in Portland, Oregon". Once this request is accepted by the wrapper manager, it will be passed to the query refinement module where the query is processed in two steps. First, the refinement module figures out the location symbol for the city "Portland". Then it forms a request for the weather report for location symbol `KPDX`. The latter query can be decomposed into two queries: an HTTP query and a filter query. The HTTP query is sent to the data wrapper for further processing and the filter query is run over the XML data to project the temperature information.

A distinct feature of our wrapper execution model is that we provide a clean separation of the data translation and the functional operation process. The data wrapper translates the source information of interest to a more structured format such as XML, and the functional component deals with additional functional operations. Separating data wrapping from functional wrapping allows us to group multiple wrapper queries over the exactly same web source, thus reducing the network connection cost and the remote data transfer cost.

In practice, it is quite often that many users are interested in the same Web source but in different subsets of data items. Instead of developing different wrappers for different users, we would like to use one data wrapper and multiple functional wrappers to improve the scalability, the performance, and the robustness of the wrapper.

Interesting to note is that wrappers in other systems [7, 25, 34, 38, 68] do not follow this model. They interleave the data translation and the functional operation process. For

example, consider the same query "what is the current temperature in Portland, Oregon now?" Their wrappers will only extract the temperature information from the Web page in the data translation process. Other wrappers may run faster than wrappers generated by XWRAP considering only one user's request. However, if many users share the interests on a same page, other wrappers will extract information from the same Web page multiple times, while wrappers generated by XWRAP retrieve data from the XML result from the shared data wrapper multiple times. We believe operations on structured data such XML data will be much faster than on original Web pages.

Another benefit of our execution model is that we can get the metadata information of the source data as a byproduct of data wrapping. Such schematic information tells the application what information this wrapper can provide, including the content description and query capability description. Such metadata information provides a useful means for mediators or broker agents to fuse, integrate, and manipulate data. Although we can get the self-describing metadata information for each individual page that is dynamically generated, it requires tremendous effort (if it is not impossible), to derive a generic "schema" for all the web pages of a given web source. For example, recall the weather page for Savannah, some weather conditions such as **Precipitation last hour** appear in earlier result documents may become unavailable in the current result page. This is a typical feature for most of the dynamically generated web pages. Such unpredictability makes it quite difficult to figure out all the possible properties based on induction by examples. Typically, the quality of example samplers is one of the most reliable factors for enhancing the robustness of the wrapper programs.

Figure 25(b) shows the execution model for a data wrapper. A data wrapper consists of four components: **Remote Data Fetch**, **Source Specific Parser**, **Information Extraction** and **XML Result Generator**.

- Remote Data Fetch takes a URL and locates the Web page and passes the page to the Source Specific Parser.

- The Source-specific Parser translates the page into a tree structure. We call it a parse

tree.

- The Information Extraction module identifies effective regions such as tables or text sections, and semantic tokens of interest according to the Region/Token extraction rules. The extraction result is recorded in a Comma Delimited format and sent to the XML Result Generator.

- The XML Result Generator constructs the XML output in terms of the region/semantic-token extraction rules and the hierarchical structure extraction rules.

- Finally, the XML result will be recorded in the XML data repository, so that different applications can share the result.

In current implementation, the four components run sequentially. A component starts execution only after the previous component finishes. For example, Information Extraction starts after the whole parse tree produced by the Source Specific Parser. Parallel execution improves the performance, but it also incurs much more complexity. It will be our future work.

3.5.3 Implementation Platform

XWRAP is developed in Java. The interface design package is Swing 1.1. Wrappers generated by XWRAP are also in Java. Our initial performance evaluation on XWRAP and wrappers generated was carried out on Microsoft Window NT Server 4.0. (See Section 3.6.)

3.6 Performance Measurements

3.6.1 Representative Web Sites

Due to the rapid evolution of the web, there are few agreed upon standards with respect to the evaluation of web pages. Existing standard benchmarks such as the SPECweb96, Webstone 2.X, and TPC-W impose a standard workload to measure server performance. Although it is an interesting challenge to collect a representative set of web sites for comparing the performance of web data source wrappers, that task is beyond the scope of this paper. For our analysis, we have chosen 4 web sites that are representative in our opinion:

1. NOAA weather site shown in Figures 11 and Figure 14. NOAA pages combine multiple small tables (vertical or horizontal) with some running text. Number of random samples collected: 10 different pages.

2. Buy.com, a commercial web site [www2.buy.com] with many advertisements and long tables. This is a web site with frequent updates of content and changes of format. It is an example of challenging sites for wrapper generators. Web pages used in our evaluation are generated dynamically by a search engine. Pages used include book titles that contain keywords such as "JDBC" and "college life". Number of random samples: 20 pages.

3. Stockmaster.com, another commercial site [www.stockmaster.com] with advertisements, graphs, and tables. This is an example of sites with extremely high frequency updates. Pages used in our evaluation are also generated dynamically, including stock information on companies such as IBM and Microsoft. Number of random samples: 21 pages.

4. CIA Fact Book (http://www.odci.gov/cia/publications/factbook), a well-known web site used in several papers [68, 3]. Although infrequently updated, it is included here for comparison purposes. Number of random samples: 267 pages.

3.6.2 Evaluation of Wrapper Generation

The first part of experimental evaluation of XWRAP concerns the wrapper generation process. Since the use of wrapper generator depends on many factors outside of our control, we avoid making any scientific claims of this evaluation result. The experiments are included so readers may gain an intuitive feeling of the wrapper generator usage.

We measured the approximate time it takes for an expert wrapper programmer (in this case a graduate student) to generate wrappers for the above 4 web sites. Since production-use wrappers are typically written and maintained by experienced professional programmers, this is a common case. The results are shown in Figure 26.

We already have several improvements on the GUI that should shorten the wrapper

generation process.

Data Source	Generation Time(minutes)	Revision (times)	Extraction Rules Length(lines)	XML Template Length(lines)
NOAA	40	2	114	153
CIA Factbook	25	1	237	23
Buy.com	16	0	102	46
Stockmaster	23	1	90	46

Figure 26: XWRAP Performance Results

Our initial experience tells us that the main bottleneck in the wrapper generation process is the number of iterations needed to achieve a significant coverage of the web site. The main advantage of our wrapper is the level of robustness. The wrappers generated by XWRAP can handle pages that have slightly different structure (such as extra or missing fields (bullets or sections) in a table (a text section) than the example pages used for generating the wrapper.

However, when the pages are significantly different from the example pages used in the wrapper generation process, the wrapper will have to be refined.

Since a wrapper is generated "by example", the choice of a simplistic example page would produce too simple a wrapper for more complex pages. Typically, as more complex pages are encountered, the wrapper is refined to handle the new situation. Ideally, one would find the most complex example web page of the site, and use it to generate the "nearly complete" wrapper for that site. This is a topic of ongoing research.

3.6.3 Evaluation of Wrapper Execution

Our current implementation has been built for extensibility and ease of software maintenance. Consequently, we have chosen software components with high functionality and postponed the optimization of data structures and algorithms to a later stage. One example of such trade-off is the use of Java Swing Class library to manage all important data structures such as the document tree. This choice minimizes the work for visualization of these data structures, which is more important than raw performance at this stage of XWRAP development.

All measurements were carried out on a dedicated 200MHz Pentium machine. The machine runs Windows NT 4.0 Server and there is only one user in the system. All the XWRAP software is written in Java. The main Java package used is Swing.

Data Source	Avg. vs. St. Dev.	Document Size(byte)	Document Tree Length	Result XML Size(byte)	Doc/XML
NOAA	Average	31135	1145	7593	4.1
	St. Dev.	465	23	42	0.1
CIA Factbook	Average	16115	834	18981	0.9
	St. Dev.	4503	188	5623	0.1
Buy.com	Average	44075	832	5172	9.6
	St. Dev.	11871	232	2014	3.4
Stockmaster	Average	21218	523	370	57.3
	St. Dev.	1137	32	11	2.4

Figure 27: Performance Statistics w.r.t. source document size and result XML size

Figure 27 shows the first characterization of web page samples. We see that NOAA and Stockmaster.com have high uniformity (low standard deviation) in document size, due to their form-oriented page content (standard weather reports and standard stock price reports). The CIA Fact Book has medium standard deviation in document size, since the interesting facts vary somewhat from place to place. The Buy.com pages have high variance in document size, since the number of books available for each selection topic varies greatly.

Also from Figure 27 we see that both the wrapper-generated document tree length and result XML file size are closely related to the input document size. We call wrappers that ignore a significant portion of the source pages (in this case, the advertisements in Buy.com and Stockmaster.com) *low selectivity* wrappers. In our case, Buy.com and Stockmaster.com are low selectivity due to heavy advertisement, and their Input-Doc-Size/Output-XML-Size ratio is high (9.6 and 57.3, respectively). Purely informational sites such as NOAA and CIA Fact Book tend to have high selectivity (4.1 and 0.9, respectively).

An expected, but important observation is about consistent performance of the wrappers, in terms of successfully capturing the information from source pages. First, form-oriented input pages such as NOAA and Stockmaster.com have high uniformity (low standard deviation) in the result XML file size. Second, for variable-sized pages in Buy.com

and CIA Fact Book, we calculated the correlation between the input document size and the output XML file size (from the data table not shown in the paper due to space constraints). The correlation is strong: 1.00 for Buy.com and 0.98 for CIA Fact Book. This shows consistent performance of wrappers in mapping input to output.

Data Source	Avg. vs. St. Dev.	Fetch Time(ms)	Expand Tree Times(ms)	Extraction Times(ms)	Generate Times(ms)	Total (ms)	Correlation Doc/Time
NOAA	Average	4391	8531	3841	1128	18520	0.45
	St. Dev.	1032	1055	228	116	1636	
CIA Factbook	Average	1907	11916	4709	3902	23043	0.93
	St. Dev.	265	3366	1175	1297	5776	
Buy.com	Average	6908	7777	2748	838	18909	0.66
	St. Dev.	4333	1553	1439	287	6602	
Stock-master	Average	1972	5489	1412	468	9973	0.35
	St. Dev.	489	453	497	121	1131	

Figure 28: Performance of Wrappers w.r.t. Fetch, Expand, Extract, and Result Generate time

Figure 28 shows the summary of execution (elapsed) time of wrappers. It is comforting that form-oriented pages (NOAA and Stockmaster.com) take roughly the same time (standard deviation at about 10% of total elapsed time) to process. This is the case for both a high selectivity site such as NOAA and a low selectivity site such as Stockmaster.com. For variable-sized pages in Buy.com and CIA Fact Book, we calculated the correlation between the input document size and total elapsed processing time: 0.66 for Buy.com and 0.93 for CIA Fact Book. The higher correlation of CIA Fact Book is attributed to its high selectivity (same input and output size), and lower correlation of Buy.com to its lower selectivity (input almost 10 times the output size). This shows the consistent performance of wrappers in elapsed time.

Figure 28 also shows that most of the execution time (more than 90%) is spent in four components of the wrapper: Fetch, Expand, Extract, and Generate. The first component, Fetch, includes the network access to bring the raw data and the initial parsing. Since we have no control over the network access time, the fetch time has high variance. This is confirmed by the lowest variance of the smallest documents (CIA Fact Book) and highest variance of largest documents (Buy.com).

The second component, Expand, consumes the largest portion of execution time. It is a utility routine that invokes Swing to expand a tree data structure for extraction. This appears to be the current bottleneck due to the visualization oriented implementation of Swing, and it is a candidate for optimization. The third component, Extract, also uses the Swing data structure to do the Information Extraction phase (Section 3.3.4). This phase does more useful work than Expand, but it is also a candidate for performance tuning when we start the optimization of the Expand component.

The fourth component, Generate, produces the output XML file. It is clearly correlated to the size of the XML file. Except for the extremely short results from Stockmaster.com (consistently at about 370 bytes), the execution time of Generate for the other three sources is between 5 and 6 bytes of XML generated per 1 ms.

3.7 Related Work

3.7.1 Wrapper/Mediator Environment

Wrappers have been developed either manually or with software assistance, and used as a component of agent-based systems, sophisticated query tools and general mediator-based information integration systems [74, 48, 51]. For instance, the most recent generation of information mediator systems (e.g., Ariadne [34], CQ [53, 54], Internet Softbots [38], TSIMMIS [22, 25]) all include a pre-wrapped set of web sources to be accessed via database-like queries. However, developing and maintaining wrappers by hand turned out to be labor intensive and error-prone, due to technical difficulties such as undocumented HTML/XML tags, plain text, ill-formed HTML, and subtle variations in the content (small to the human perception, but difficult for programs).

In addition, several projects develop tools to support enhanced Web search. For example, Araneus [6] provides an object model and a object specification language. The wrapper developers must first model the web sources of interest in terms of the Araneus object specification language by specifying the entities of interest and their properties. Based on the collection of object specifications, Araneus creates a global view of the Web sources of interest in a Web hypertext format. Araneus considers each Web page as an object with

an identifier (the URL) and a set of attributes. Attributes include text, images or links to other pages. Araneus navigates the Web following the links between different pages and Web pages are woven together to provide a global view of the information sources. However, Araneus cannot represent the relationship of Web pages with no hypertext links. Furthermore, the Araneus approach does not scale as it requires the wrapper developer to code the wrapper by hand using their object specification language.

3.7.2 Automated Wrapper Generators

The process of automated wrapper generation can be divided into two main steps: 1) analyzing the source page and 2) composing structured results. Analyzing the source page involves identifying sections and sub-sections of interest in a source page. Composing structured results involves producing a structured representation of the extracted information.

3.7.2.1 Analyzing Source Pages

In order to extract interesting sections from Web pages, wrappers need to analyze the source page's format to identify those sections first. Most wrapper generators assume sections and sub-sections are separated by mark-up tokens, such as HTML tags or punctuation signs. Ariadne [34] generates a lexical analyzer that searches a page for tokens indicating the heading of a section using LEX, a lexical analyzer generator based on string regular expressions. Then, Ariadne analyzes the nesting hierarchy of sections, determining which sections comprise the page at the top level and which sub-sections (if any) comprise other sections in the page. SoftBot [38] also adopts a similar methodology.

It is not easy to derive the string regular expression. Some projects analyze the source document based on a tree model. For example, XWRAP [45] uses a source specific parser to transform the HTML Web page into a tree. W4F [68] also transforms the source document into a tree structure using different parsers and then uses the path regular expression to highlight interesting nodes.

However, many text files do not generally contain the mark-up tags, and not all the structural elements are separated by markers (such as tags). NoDoSE [1], which focuses on extracting information from text files, analyzes the source document based on the string

length. Through a GUI interface, the user hierarchically decomposes the source document, outlining its interesting regions and describing their semantics. A mining component attempts to infer the grammar of the file from the information the user has input so far. The grammar rules describe the pattern of the source document by the string length of interesting regions. The drawback of this approach is that the string length is not always the primary feature of the source document.

3.7.2.2 Composing Structured Result

After identifying interesting data in the source documents, wrappers output them in a more structured format. In order to communicate with the mediator or other applications, this format should satisfy the following requirements. It needs to be unified in order to be understood by various applications, such as the mediator. In addition, it supports the query capability so that some post-processing can be done on the extracted information, such as filtering.

XML [70] (eXtensible Markup Language) is a very good choice for such a unified data format. XML is a self-describing language, so that XML data can be easily interpreted in different applications. The query languages on XML data are proposed and being implemented both in academic and industrial areas, such as XML-QL [18] and XQL [57]. Furthermore, more applications and vendors are beginning to support XML at the present.

In the TSIMMIS project, the Object Exchange Model (OEM) is used as the common data model exported by the wrappers. OEM is also a self-describing model, which captures the essential features of models used in practice, generalizing them to allow arbitrary nesting and to include object identity. An application can request OEM objects from a wrapper using the MSL query language. Due to the similarity between OEM and XML, one can treat OEM objects as XML objects. TSIMMIS wrappers export results through a set of templates. Unlike XWRAP, templates in TSIMMIS are composed manually.

Other formats for the wrapper results are also available. Stalker [59] chooses KQML, an agent communication language, to facilitate communication between the wrapper and a mediator. Garlic [24] describes data using GDL (Garlic Data Language), which is a variant

of the ODMG's Object Description Language. However, these formats are only understood by certain applications, which limits their usage.

Tova Milo and Sagit Zohar [56] use schema matching to simplify the wrapper generation, when both the source schema and the result schema are available. They observe that in many cases the schema of the data in the source system is very similar to the result schema. In such cases, much of the translation work can be done automatically based on the schema similarity. They define a middleware schema, and each data source to be used in their system needs a mapping of its data and schema to (or from) the middleware format. They develop an algorithm to match and translate the objects in the source with the objects in the result comparing the two instances of the middleware schema. While most Web pages are still in HTML, which do not have a schema, schema matching does not work. However, if more XML information appears on the Web, this approach will speed up the wrapper generation, since XML documents contain self-describing metadata that can be treated as schema information.

3.7.3 Inductive Learning Algorithms

Inductive learning, a well-studied paradigm in machine learning, can be used in automated wrapper generation. Induction is the process of reasoning from a set of examples to a hypothesis that generalizes or explains the examples. An inductive learning algorithm, then takes as input a set of examples, and produces as output a hypothesis. For example, provided as input a set of sample Web pages from a Web source and expected extracted results, an inductive algorithm for wrapper generators deduces as output hypothetical wrapper specifications for that Web source.

SoftBot [38, 36] developed a wrapper generation system using inductive learning technique. Several generic wrapper classes with adjustable parameters are pre-defined in the wrapper generation system. Each wrapper class can extract information from one document pattern. Wrapper developers highlight interesting sections in many sample documents, and then a machine learning algorithm will adjust those parameters to find a combination of wrapper classes to extract the highlighted sections correctly. If such a combination is not

available, the algorithm will return the best combination with the fewest mistakes. The developers can either correct the best combination manually or add more wrapper classes capture new patterns until a completely correct combination is found. SoftBot has developed six basic wrapper classes [36], and argued that a majority (70% in total) of the real Web resources can be covered by the six classes in their evaluation test.

NoDoSE also adopts the inductive learning technique. Using a GUI, the user hierarchically decomposes a plain text file, outlining its regions of interest and then describing their semantics. The task is expedited by a mining component that attempts to infer the grammar of the file from the information the user has identified so far.

A common weakness of the existing inductive learning algorithms developed for wrapper generation systems is the inflexibility to adapt to changes. Typically, the presentation layout changes or content reorganization can easily break the existing set of patterns used by the learning algorithms. Consequently, the wrappers generated by the old inductive learning algorithm may need to be rebuilt completely according to up to date sample documents. Therefore, without a convenient mechanism to create new patterns and an adaptive solution to seamlessly incorporate new patterns into the learn algorithms, it would be very hard to repair and maintain the wrappers generated using these learning algorithms.

3.8 Conclusion

We have presented a framework for designing intelligent wrapper generators. The framework classifies wrappers into two categories, data wrappers and functional wrappers. Wrapper developers build data wrappers using a declarative rule-based language. Inductive learning algorithms and GUI interface help the developers to derive the rules. Functional wrappers are built on top of data wrappers. Functional processing components operate on the XML output of the data wrappers. Different functional wrappers can share a data wrapper to improve the performance.

The contributions of the framework presented in this paper are the following:

- A clean separation of tasks of building wrappers that are specific to a Web source from the tasks that are repetitive for any source.

- A declarative rule-based language to specify data wrappers.
- Inductive learning algorithms and GUI interfaces facilitate wrappers developers to derive wrapper patterns.

CHAPTER IV

XWRAP ELITE

4.1 Introduction

While XWRAP Original greatly improves wrapper generation process and reduces the wrapper generation time to hours, it is still often inadequate to support overwhelmingly increasing Web sources due to its inevitable interactive with wrapper developers. In this chapter, we propose a systematic approach to build an near-automated system for wrapper construction for Web information sources. Called **XWRAP Elite**, the system builds on our previous work in [50]. The goal of XWRAP Elite is to provide a tool for the easy transformation of human-oriented HTML into machine-readable and semantically meaningful XML. Implemented by wrappers, this transformation enables integration of new XML-based applications, such as Web service, mediator-based information systems, and agent-based systems. A main challenge is automatically generating the wrappers, which minimizes human involvement and associated costs. Our main contribution consists of a set of algorithms and heuristics that balance the requirement of semantic input with the need to automate the information extraction and wrapper generation process.

This is not the first time the problem of automatic information extraction from a Web document has been addressed. [5, 26] discover objects of interest manually. They first examine the documents and find the HTML tags that identify the objects, and then write a program to extract them. [1, 68, 3, 34, 65, 36, 38, 69] are semi-automatic wrapper generators whose approaches rely primarily on the use of syntactic knowledge, such as specific HTML tags, to identify objects.

XWRAP Elite approach differs from these proposals in two distinct ways. First, we clearly separate a wrapper into three components: object extraction, element extraction and output tagging (see Section 4.2). Once a Web document is fetched, XWRAPElite automatically extract data objects from the document and then separate the objects into

elements by heuristics. It also marks the data output in XML with the tagging rules input by a user. The clean modularization allows each component of a wrapper be customized into other applications where only some component of the wrapper is needed. Furthermore, we want to leverage on standards as much as possible, thus choosing XML as our output format. The development of XWRAPElite presents not only a software tool but also the methodology for developing an XML-enabled, feedback-based, interactive wrapper construction facility that generates value-added wrappers for Internet information sources.

Second, we automated the process to extract data at a fine-grained element level, while most of the existing approaches require the wrapper developers to write information extraction rules by hand using a domain-specific language or to input their knowledge through an interactive GUI. The automation of data extraction offers a number of advantages over existing approaches:

1. XWRAPElite can generate wrapper code on demand. It can generate a wrapper for a given web site through analysis of just a few documents (often 1 or 2 is sufficient), making it easy for other applications to integrate an data extraction component at run time.

2. XWRAPElite presents a scalable and robust solution to data extraction and wrapper code generation. It can generate Java wrapper code in a couple of minutes without requiring manual input of human knowledge. Users are given the option to provide domain XML tag names to the objects extracted and their elements, which are meaningful to specific domains or applications..

3. XWRAPElite had built-in facilities, such as micro-feedback mechanisms, which allow wrappers to be re-generated on demand as the data source being wrapped, especially the content regions being wrapped, have changed significantly.

The rest of the chapter is organized as follows. Section 4.2 gives an overview of the XWRAP Elite architecture. Section 4.3 introduces some basic concepts of object extraction and Section 4.4 describes the automated object extraction procedure. Section 4.5 describes

how elements are extracted for individual objects. Section 4.6 describes how elements are semantically tagged. Section 4.8 concludes the chapter.

4.2 System Architecture

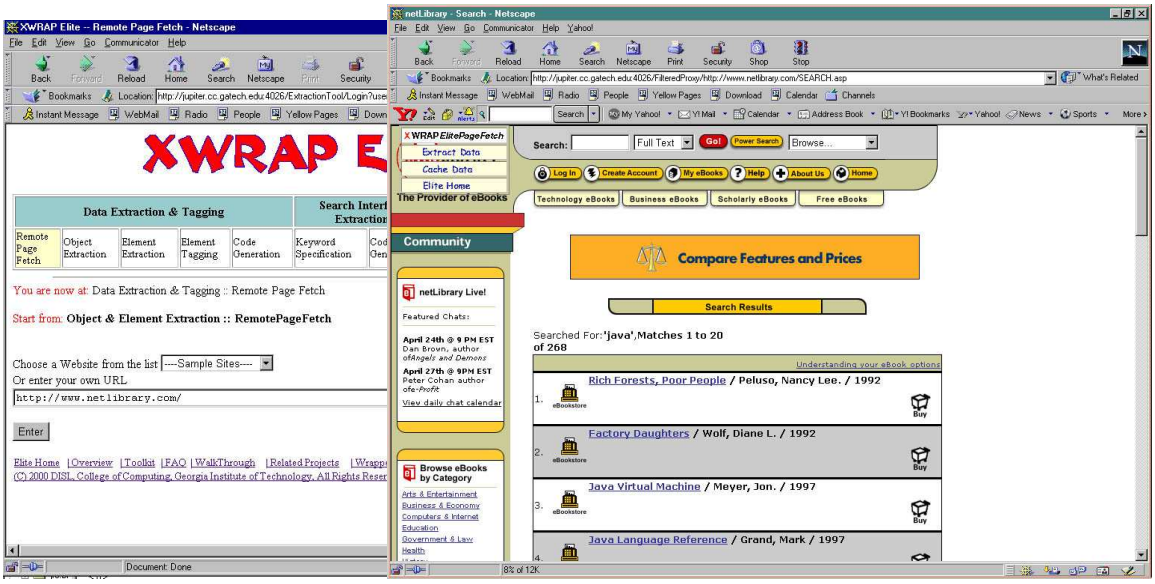

Figure 29: XWRAP Elite System Architecture

XWRAPElite is the first on-demand wrapper code generation system that allows users to generate Java wrappers on the Web with zero learning curve. It also provides advanced data extraction and quality tuning capabilities to allow experienced wrapper developers to further enhance the data extraction quality through element discovery, element enumeration, and element alignment. The first prototype system of the XWRAPElite consists of six main components as shown in Figure 29.

- Client Manager – it takes the URL entered by the user and invoke the XWRAPElite *steering module*.
- Steering Controller – The main task of the steering controller is to record the user's browsing path starting from the URL entered. Once the page containing the data content of interest, it will invoke the data extraction component.
- Data Extraction – is one of the key components. It applies two phases of data

extraction algorithms to discover and extract the objects of interest. The content-rich subtree identification phase consists of several algorithms to discover and locate the query answer subtrees in a page. The object extraction phase consists of algorithms for object separator identification and element separator identification.

- XML Mirroring – This component is responsible for tagging the objects extracted and the elements separated using either system-default tag names or user-supplied and domain-specific tag names that are semantically meaningful for the subsequent applications.

- Search-Interface Extraction – Any wrapper code generator needs to have both the data extraction component and the search interface extraction component. The former generates code that can perform data extraction tasks and the latter generates code that can mimic the search interface of the website being wrapped, allowing the wrapper users to enter the search keywords and/or search conditions in the same way as they would do at the target source site.

- Code Packaging – This component as the name suggested, bundles the code for data extraction and the code for search interface extraction into a coherent source code package and produces both the Java source code and the executable binary code.

- Testing Module – Once the wrapper code is generated, the testing component will validate the wrapper quality by testing the code again a new keyword search, for example, and to check if the wrapper can actually extract and tag the data content correctly. It usually gives the success rate, precision and recall information as a measure of the code quality. Other statistics information can also be provided in this stage, including the average response time for a wrapper, the Quality of Service support, and the maximum number of objects that need to be extracted and passed over to the next program.

To help the readers on how to use the XWRAPElite system, we below gives a walk through of the initial steps. Figure 30 contains two screen shots. The left one is a screenshot

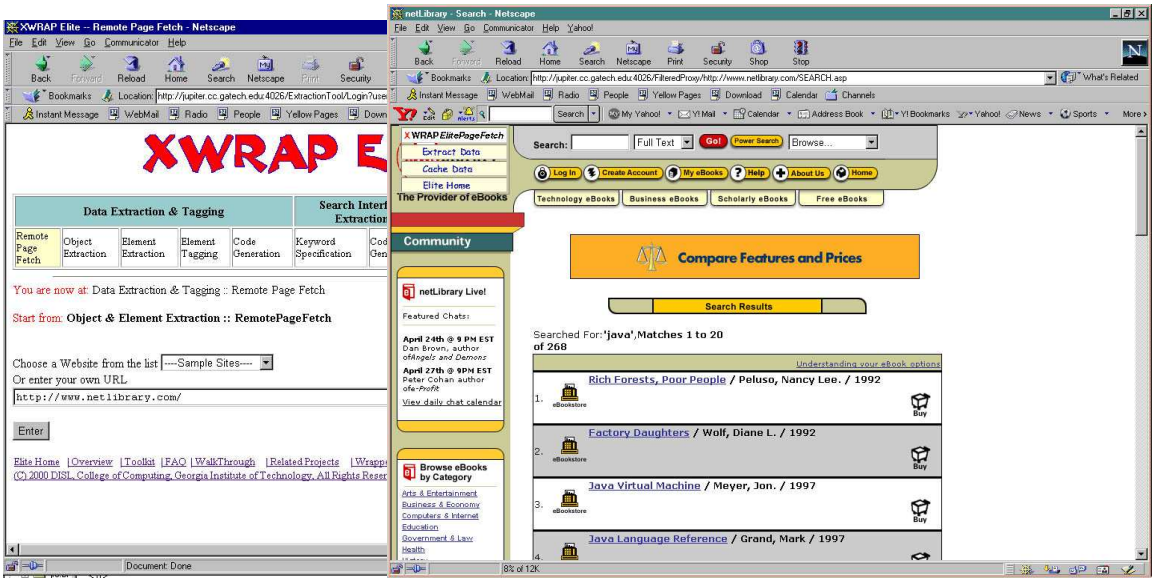

Figure 30: XWRAP Elite Screen Shots – Get Started

of the front user interface of the first prototype of XWRAP Elite. A user first enters an URL of the Web site of interest. By clicking on the `Enter` box on the page, the XWRAPElite client manager installs the steering controller on the website located by the URL (see the control panel on the top-left corner of the Web page shown in the right screen shot). Now the user can surf the Web site by navigating hypertext links and filling in HTML forms. The steering control service enables the XWRAP Elite to record the browsing path of the user, which will be used to generate location code that is capable of locating the target page that contains the data content to be extracted. When the user arrives at a Web page that contains the content he wants to wrap, he can click on the *Extract Data* button on the steering control panel at the upper left corner, which will feed the Web page to XWRAP Elite data extraction engine and used as a sample document for page layout analysis and extraction code generation. Figure 31 gives a closer view of the page that the user is interested in wrapping. As you can see, the page contains the query answer region that starts with the first book titled "The Design Pattern Java Workbook", the advertisement region in the left side of the page, and the company logo and services listing of the service provider, Barnes&Noble.com in this case.

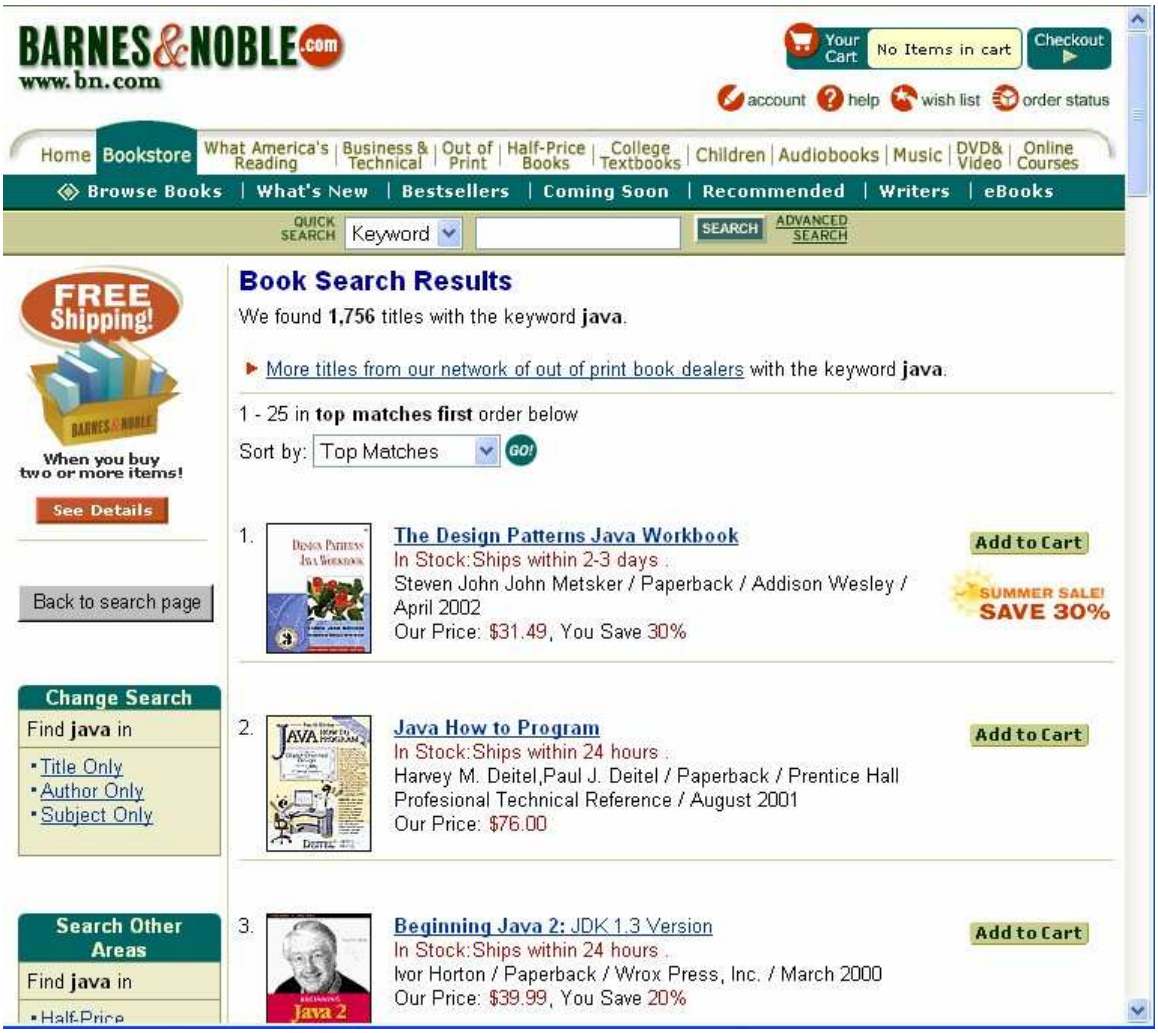

Figure 31: bn.com Search Result http://www.bn.com – on July 16, 2002

When the user clicks on the data extraction button at the steering controller panel on the page (see the right screen shot of Figure 30), the data extraction engine is invoked. It first performs remote page fetch, and then fires the object extraction, element extraction, element tagging, and generates the Java code of the data extraction component of the wrapper for Barnes&Noble.com. Figure 32 shows a screenshot after the execution of the object Extraction task.

During the object extraction process, the Object Extraction module displays the list of objects as they are being extracted. Upon the completion of the extraction, it outputs the list of objects identified as shown in Figure 32 and the statistics on the object extraction

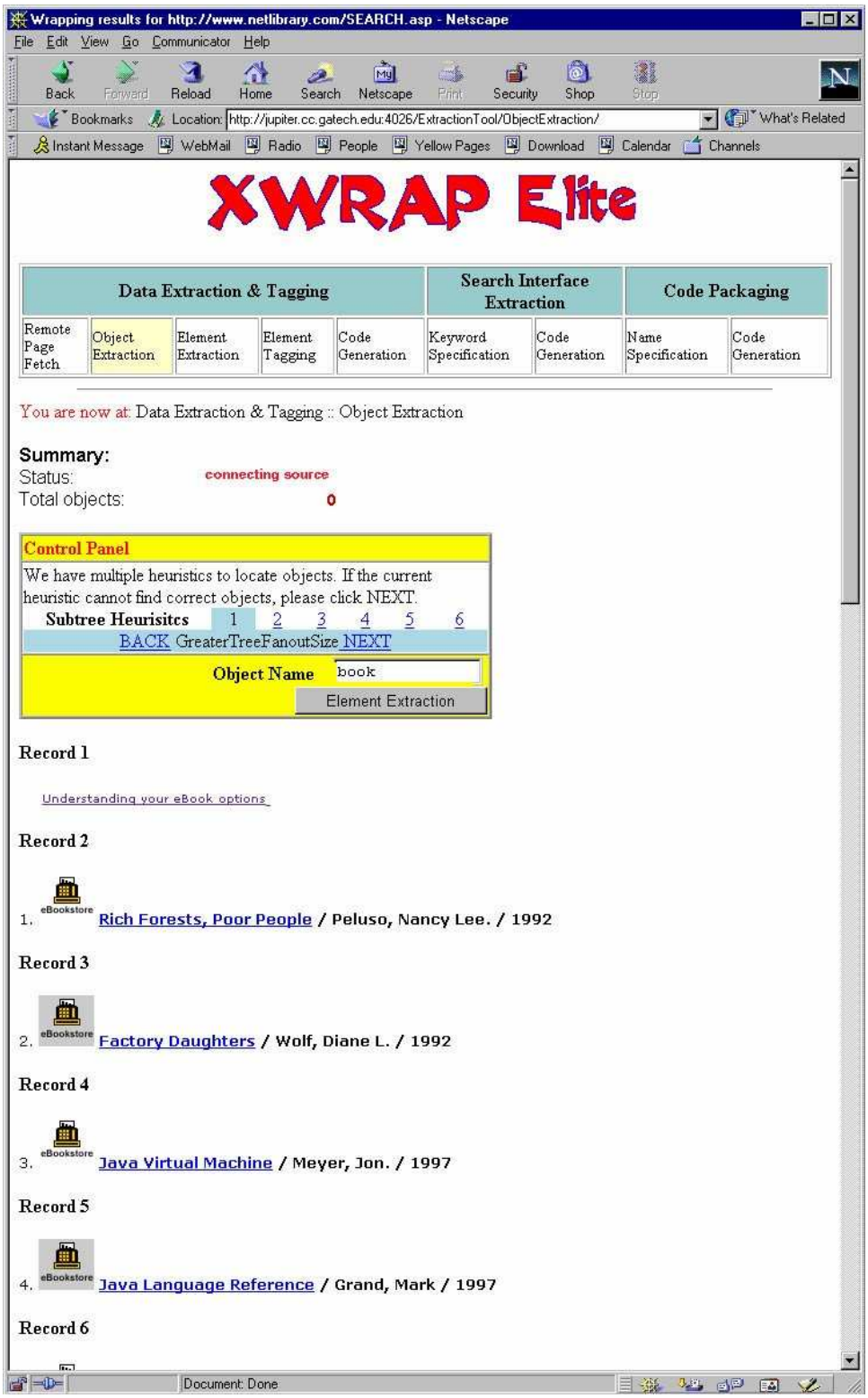

Figure 32: XWRAP Elite Screen Shots – Object Extraction

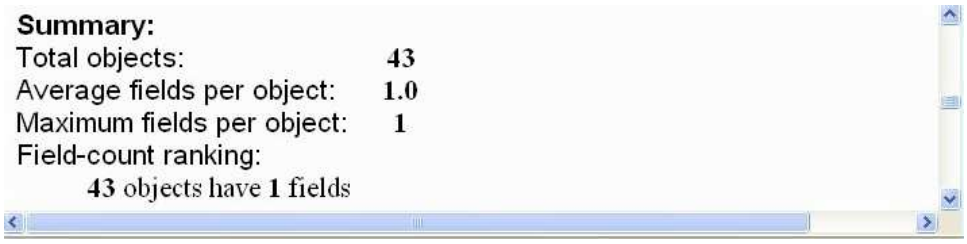

Figure 33: XWRAP Elite Screen Shots – Object Extraction Statistics

performed, such as the total number of objects extracted and the amount of time took by the object extraction module and so on (see Figure 33).

We will defer the element extraction and tagging as well as search interface extraction and code generation to the subsequent sections.

In summary, XWRAP Elite takes an URL and a sample document obtained by surfing from the given URL as input, and generates a wrapper program which is capable of extracting the content-rich query answer objects from the document and convert the HTML objects into XML data with semantically meaningful tags in four steps. As byproduct, the XWRAPElite code generator also output the data extracted for the sample document with XML tagging. This XML result can be used to validate the quality and the correctness of the wrapper code in terms of data extraction and element alignment.

Figure 34 shows the data extraction engine of the XWRAP Elite system and how each code module of a wrapper is generated. We can view the data extraction task as a three-phase process. In the first step, the engineer parses the sample document into an HTML tree. After filtering out the advertisement and graphics, it locates meaningful content region, discover the object boundary separator that can separate the content region into objects and then extracts the data objects. We call this step the *Object Extraction* component. The technical detail of this component is explained in Section 4.4. Next, the data extraction engine decomposes the objects into elements. This is the *Element Extraction* component, which involves element separator tag discovery, location, and element separation. We will describe the technical details in Section 4.5. The third phase focuses on element alignment and element tagging. In order to correctly tag elements of each object, we need to handle

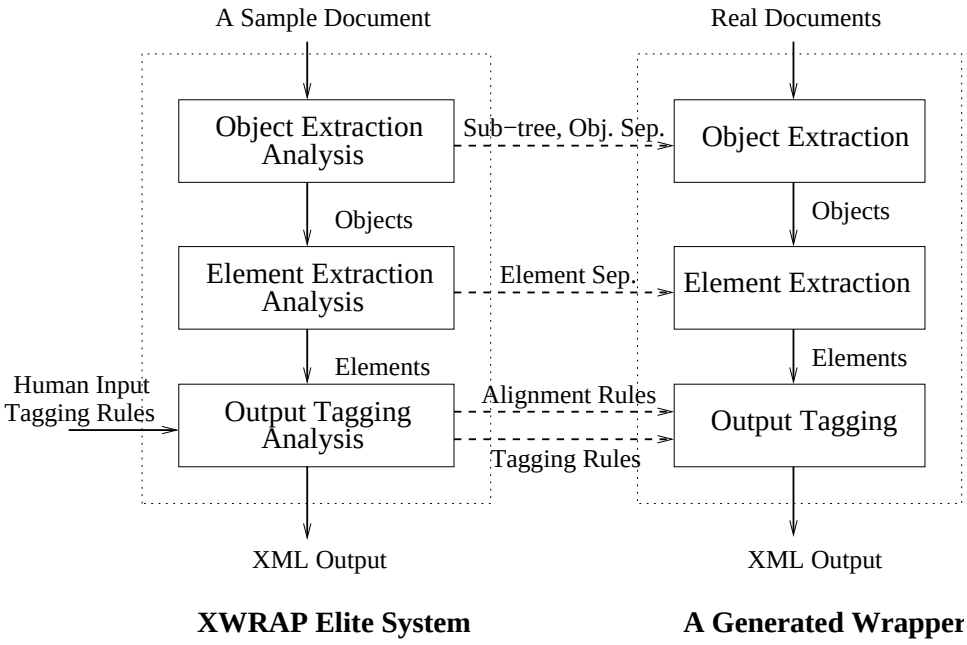

Figure 34: XWRAP Elite System Architecture

the situations where some objects have more elements than others and when some element is missing in a given object, simply using the appearance ordering of the elements will create errors in tagging the elements. Therefore, this phase needs to apply different element alignment techniques such as using type information to combine the elements across different objects into logical groups such as tables, numerical numbers, URL links, etc. Once the element boundaries are correctly verified by element alignment rules and confirmed by the user (usually a wrapper developer), the element tagging process starts. As an option, the user can input tagging rules by assigning a semantically meaningful element name to each group. Otherwise the system default tag names are used. This process, called *Output Tagging*, in XWRAPElite (see Figure 31) is described in Section 4.6. The Search Interface Extraction component will generate code that ideally allows the wrapper to provide the same interface capability as the source being wrapped. Most of today's wrapper technology can only offer a subset of the source interface search functionality. So does XWRAPElite. Upon generating the data extraction code and the source search interface extraction code, the Code Packaging component will package all the parts together into a wrapper that converts HTML data extracted from the original websites into well-defined XML documents.

In the next section we give a review of the basic concepts and the object extraction algorithms used in XWRAPElite system [9] and then we discuss the element extraction, alignment, tagging and code packaging in Section 4.3.

4.3 Basic Concepts and Object Extraction Overview

A well-formed Web document can be modeled as a *tag tree*. All the internal nodes of the tag tree are called *tag nodes*, denoting the part of the document enclosed by a start tag and its corresponding end tag, including all characters in-between. A tag node is labeled by the name of the start tag. The leaf nodes of the tag tree are *content nodes*, identifying the content data (text) between a start tag and its corresponding end tag or between an end tag and the next tag in the web document, such as numbers, strings, or encoded MIME types. A leaf node is labeled by its content. An example tag node is <*Title*> Home Page </*Title*> in which the label of the tag node is <*Title*> and the text string Home Page is a leaf node.

Any node in the tree can be uniquely identified by the path expression from the root node to the node. Therefore, in subsequent sections such a path expression sometimes refers to the node. Consider Figure 35. The path from the root node *HTML* to the *Title* node goes through the *Head* node. We use the dot notation to express the path as HTML[1].Head[1].Title[1]. The numbers in the brackets after each node indicate the order of the child in the tree. Similarly, the path from *HTML* to *Body* is HTML[1].Body[2].

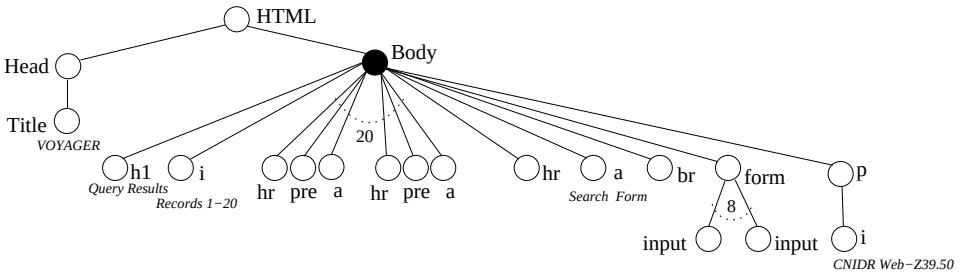

Figure 35: Tree Representation for Library Of Congress search results page.

Let $\mathcal{T} = (V, E)$ be a tag tree of a web document D where $V = V_T \cup V_C$, V_T is a finite set of tag nodes (internal nodes) and V_C is a finite set of content nodes (leaf nodes); $E \subset (V \times V)$,

representing the directed edges. We call a subtree anchored at node u, $u \in V$, a *minimal subtree* with the property P, if it is the smallest subtree that meets the following condition: There is no other subtree, such as $subtree(w), w \in V$, that satisfies both the property P and the condition that u is an ancestor of w. In Figure 35 there are two subtrees that contain all of the *hr* nodes, the subtree anchored at *HTML* and the subtree anchored at *Body*. The subtree anchored at *Body*, as shown in Figure 36, is the minimal subtree that contains all of the *hr* nodes.

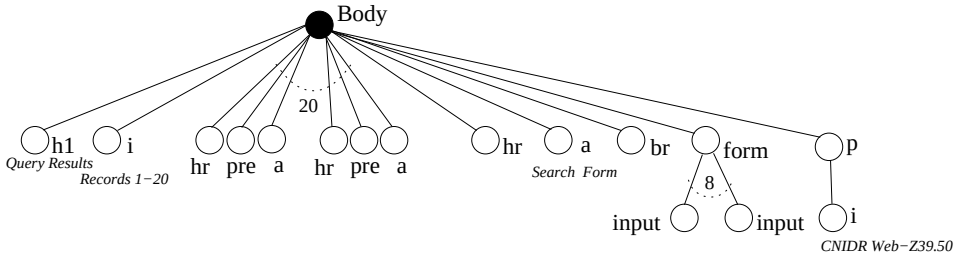

Figure 36: Minimal Subtree from Figure 35 with candidate object separator tags highlighted

In addition to the notion of minimal subtree, the following concepts are used frequently in the subsequent sections to describe the object extraction algorithms.

- $parent(u)$: The parent node of u is defined as $parent(u) = \{w | w \in V, (w, u) \in E\}$. The root node of a tree $\mathcal{T}$ is the only node that does not have a parent node.
- $children(u)$: Given a node u, $children(u)$ refers to the set of child nodes of u. $children(u) = \{w | w \in V, (u, w) \in E\}$. This definition says that a node w is a child node of u if and only if there exists an edge $(u, w) \in E$.
- $fanout(u)$: For any node $u \in V$, we use $fanout(u)$ to denote the cardinality of the set of children of u. $fanout(u) = \|children(u)\|$ if $u \in V_T$ and $fanout(u) = 0$ if $u \in V_C$.
- $nodeSize(u)$: For any node $u \in V$, if $u \in V_C$, i.e., u is a leaf node, then $nodeSize(u)$ denotes the content size in bytes of node u. Otherwise, u is a tag node, i.e., $u \in V_T$ and $fanout(u) > 0$. We define $nodeSize(u)$ as the sum of the node sizes of all the leaf nodes reachable from node u, i.e. $nodeSize(u) = \sum_{v_i \in children(u)}(nodeSize(v_i))$. For any node $u \in V$, we define the size of the subtree anchored at node u, denoted by

$subtreeSize(u)$, as the node size of u. I.e., $subtreeSize(u) = nodeSize(u)$.

- $tagCount(u)$: For any node $u \in V$, if $u \in V_C$ is a leaf node, then $tagCount(u) = 0$. Otherwise, $u \in V_T$ is a tag node and $tagCount(u) = 1 + \sum_{v_i \in children(u)}(tagCount(v_i))$. $tagCount(u)$ refers to the total number of tag nodes of which u is an ancestor.

4.4 Object Extraction

As shown in Figure 37, there are three steps to extracting data objects from an HTML document: document preparation, locating primary content, and object separation.

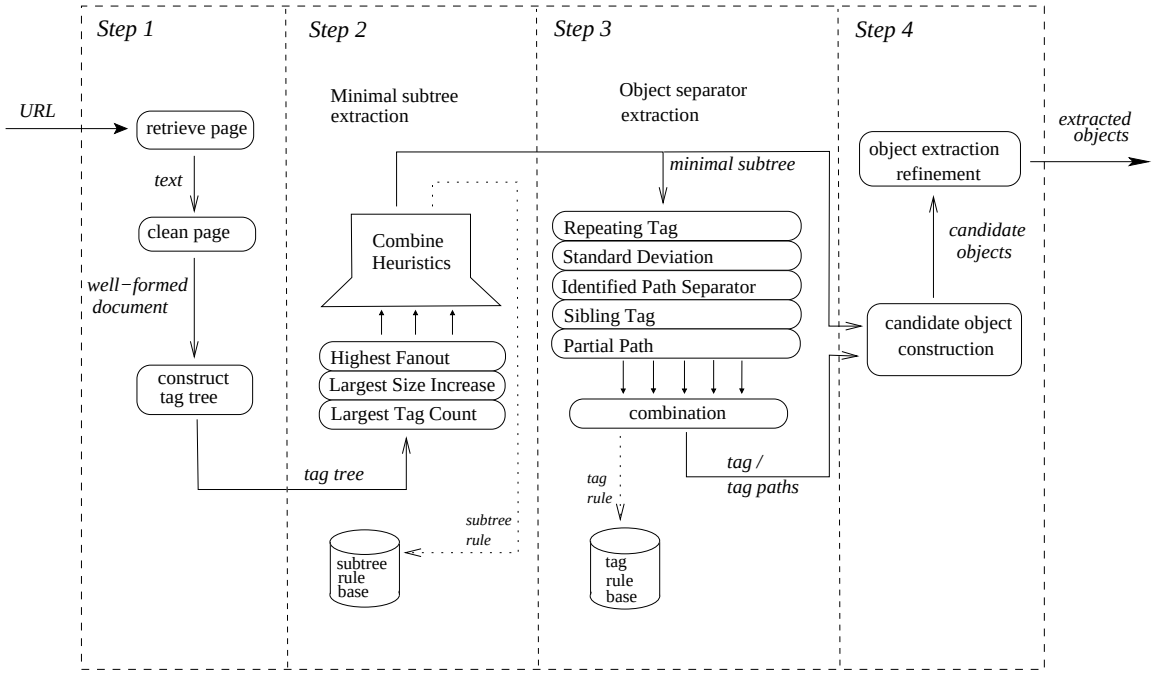

Figure 37: Object Extraction

Document preparation prepares the HTML text for object extraction. It takes a raw HTML page and performs the following two tasks: 1) The page is cleaned using a syntactic normalization algorithm, which transforms the given page into a well-formed document following rules similar to well-formed XML documents; 2) the document is converted into a tag tree representation based on the nested structure of start and end tags. There are many standard tools that can convert plain HTML text into a well-formed document, such as Tidy [63] from the W3C. Once the document is verified to be well formed, creating its

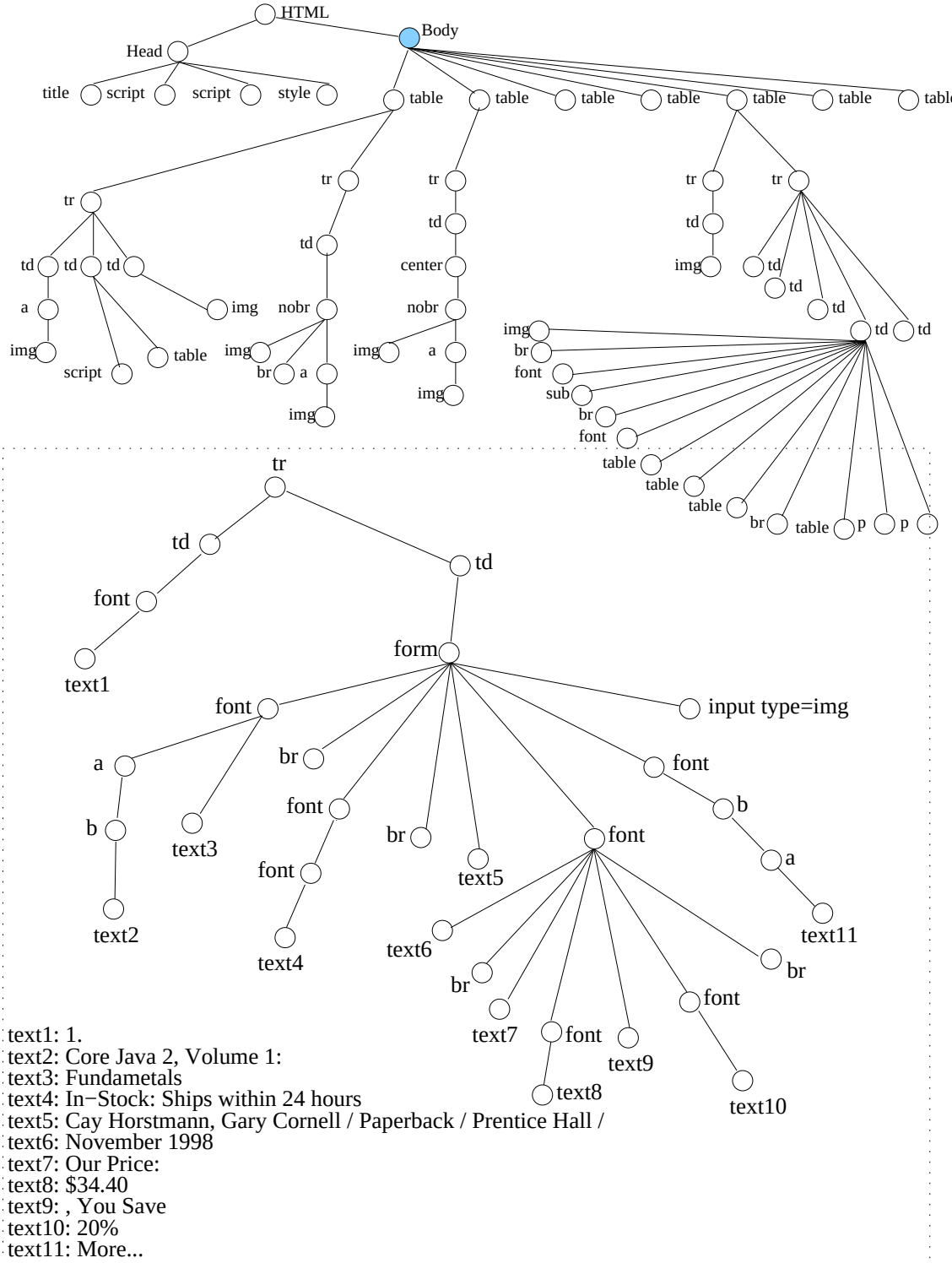

Figure 38: Tag Tree for bn.com, Figure 31

tree structure is straightforward. Figure 31 shows a screenshot of a Barnsandnobles' Web page, and Figure 38 displays the corresponding tree structure.

The second step is to find the primary content. As noted earlier, most web pages are designed for human browsing. In addition to the primary content region, web pages often contain other information such as advertisements, navigation links, and so on. Therefore, given a tree structure for a web document, the task of identifying the primary content region is reduced to the problem of locating the sub-tree containing the relevant data objects (typically the meaningful text). We call this task the *Object-rich Minimal Sub-tree Identification*, which will be discussed in Section 4.4.1.

After the object-rich subtree extraction process, the problem of extracting objects in a web page is reduced to the problem of finding the right object separator tag in the chosen subtree. This problem is divided into two steps. First, we decide which tags in the chosen subtree are to be considered *candidate object separator tags*. Since we are primarily interested in the content, we consider only the child nodes in the chosen subtree as *candidate tags*. Secondly, we identify the best object separator tag in the set of candidate tags that separate the objects most effectively. We will discuss the details in Section 4.4.2. Figure 32 shows a screenshot after Object Extraction.

4.4.1 Object-rich Minimal Subtree Identification

The main task of the object-rich minimal subtree identification is to locate the minimal subtree from an arbitrary Web page, which contains all the objects of similar structure in that page. The success rate of the minimal subtree identification algorithms is critical to the accuracy of the entire object extraction process. We have discussed the main ideas and motivation for object-rich minimal subtree identification algorithms in the previous section. In this section we first describe in detail the three individual minimal subtree identification algorithms: the highest fanout (HF) algorithm, the largest tag count algorithm (LTC), and the largest size increase (LSI) algorithm. We also show that, for a given Web page, the highest ranked subtree by one algorithm may not agree with the highest ranked subtree by another algorithm. In other words, each of the three algorithms may successfully identify

the correct subtree in some Web pages but fails in other Web pages. Such disagreement happens not only on different Web sites but also on different Web pages of the same Web site. The latter is primarily due to the fact that different search requests to the same Web site often result in different numbers of objects returned in a page. When the number of objects in the data object region is relatively small compared to the rest of the page, it is likely that one of the algorithms may fail in terms of either fanout or tag count or size difference between the subtree and each individual object (also referred to as size increase in short). To resolve this type of disagreement, in this section we also introduce a combined algorithm. It compares fanout, tag count, and size increase of all subtrees and chooses the minimal subtree that has higher fanout, larger tag count, as well as larger content size and size increase. Examples are provided throughout the discussion.

4.4.1.1 The Highest Fan-out Subtree Identification algorithm (HF)

The HF identification algorithm is the simplest of the three. It ranks all nodes of a tag tree by their fan-out and chooses the highest fan-out node as the minimal subtree. This algorithm works well for Web pages that have almost no advertisement or navigational regions. However, many dynamic Web sites today provide more than the search result objects. For example, most e-commerce sites want to provide brand-recognition as well as a consistent and highly evolved look-and-feel for their Web sites. These Web pages are likely to contain several navigational aids and other page elements that may not be directly related to the content of the query results. In such cases, the highest count algorithm does poorly.

Consider the Web page and its tag tree in Figure 39. The objects that we are interested in extracting from this example Web page are obviously the search results, namely the twelve news items marked by the twelve tables at the right side of the tree. Obviously, the correct minimal subtree is the subtree anchored at the tag node $html/body/form[2]$. However, by HF the highest ranked subtree is the one anchored at the tag node $html/body/form[2]/table[1]/tr/td[2]/font$, even though it does not contain any of the news items. The list of top four ranked subtrees by HF and their fanout values are given in Table 1 and Table 2 respectively.

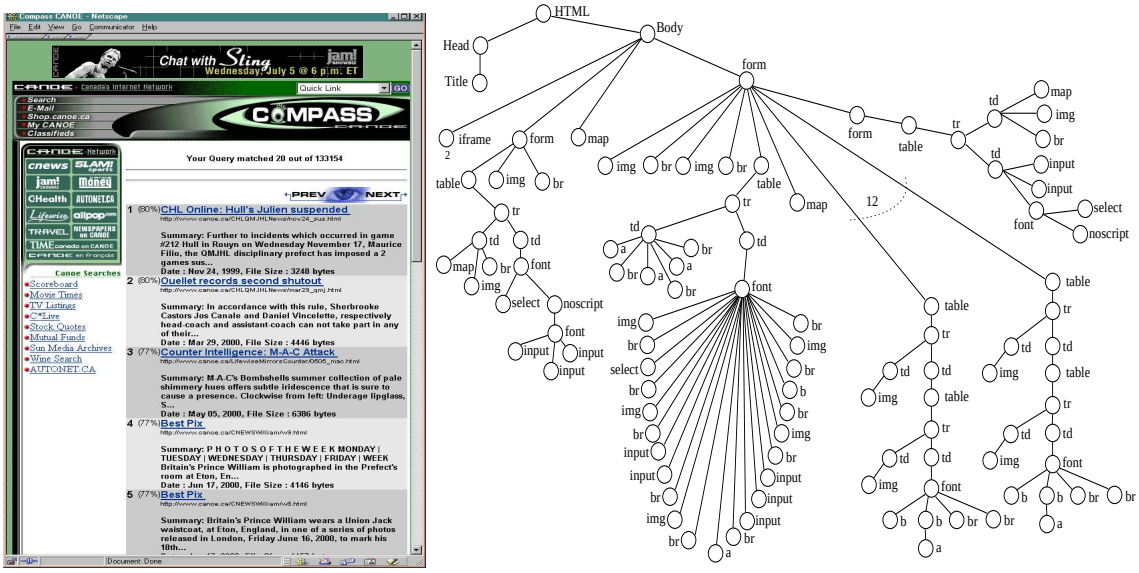

Figure 39: Canoe.com Search Result http://www.canoe.com – on July 2, 2000 – Web page and tag tree

Another situation where the HF algorithm may fail is when the number of navigational links is larger than the maximum number of query results displayed on a single page.

Table 1: Comparing HF, LSI, and LTC on canoe.com tag tree in Figure 39

Rank	HF	LSI	LTC
1	*html/body/form[2]/ table[1]/tr/td[2]/font*	**html/body/form[2]**	**html/body/form[2]**
2	**html/body/form[2]**	*html/body/form[2]/ table[1]/tr/td[2]/font*	*html/body/form[2]/ table[1]/tr/td[2]/font*
3	*html/body/form[2]/ table[1]/tr/td[1]*	*html/body/form[2]/ table[1]/tr/td[1]*	*html/body/form[2]/ table[1]/tr/td[1]*
4	*html/body*	*html/body/form[2]/ table[2]/tr*	*html/body/form[2]/ table[2]/tr*

4.4.1.2 The Largest Tag Count Subtree Identification Algorithm (LTC)

The LTC algorithm is motivated by the observations that data objects typically contain several mark-up tags and that a subtree of the highest fan-out may not necessarily have the largest tag counts. Given a Web page and its tag tree, the LTC algorithm produces a ranked list of object-rich subtrees in two steps. First, we rank all subtrees in ascending order by the total number of tags they have. Obviously, when comparing a subtree anchored

Table 2: Statistics used by the HF, LSI, and LTC algorithms on canoe.com tag tree in Figure 39

Rank	Fanout (HF)	Size Increase (LSI)	Tag Count / HACC (LTC)
1	23	5214	244 / 13
2	19	1802	24 / 10
3	6	366	7 / 3
4	4	267	15 / 2

at node u with another subtree anchored at an ancestor of u, the ancestor will always have more tags. Hence, in the second step we walk down the ranked list and re-examine those subtrees that have ancestor relationship. The subtree that has the highest appearance count of a child node tag will be ranked higher than the other subtree. Concretely, for each subtree, say T_i, in the ranked list, we compare it with every other subtree, say T_j, in the list. If $T_i \Longrightarrow^* T_j$, i.e., they have an ancestor relationship, then we compute the highest appearance count of the child node for both T_i and T_j, i.e., $HACC(T_i)$ and $HACC(T_j)$. If $HACC(T_j) > HACC(T_i)$, then T_i and T_j will exchange their ranking positions in the ranked list. Otherwise T_i will be compared with the next subtree after T_j in the ranked list. The process continues until all the subtrees are re-examined.

Recall the Web page and its tag tree in Figure 39. Compare the two subtrees *html/body* and $html/body/form[2]$ in Figure 39. The child tag $form$ in the subtree *html/body* has the highest appearance count of 2, whereas the child tag *table* in the subtree $html/body/form[2]$ has the highest appearance count of 13. Therefore, LTC ranks the subtree $html/body/form[2]$ higher than the subtree *html/body*. Table 1 lists the top five ranked subtrees obtained by HF and LTC separately. The LTC algorithm ranks the correct minimal subtree as its number one choice, whereas the HF algorithm fails.

The LTC algorithm implies that the more tags that are in a particular subtree, the more likely it will contain the data objects. However, there are cases in which the LTC algorithm fails. Typically, LTC fails when there is a node that has the largest tag count but it is not on the same tree branch as the correct minimal subtree. LTC also fails when there is a node on the same branch as the minimal subtree, which has a child node that has a higher appearance count than any of the children nodes of the correct minimal subtree. Consider

Figure 40: nbci.com Search Result http://www.aw.com – on October 15, 2000

the Web page from the **nbci** Web site shown in Figure 40 and its tag tree in Figure 41. *html/body/blockquote* is the correct minimal subtree which contains only the ten search results. However, by LTC the node *html/body* is the highest ranked subtree. LTC fails in this case because there exists a child node in *html/body*, which has a higher appearance count of 13 than any of the children nodes of the correct subtree *html/body/blockquote* (see Table 3 and Table 4).

Table 3: Comparing HF, LSI, and LTC on nbci.com tag tree in Figure 41

Rank	HF	LSI	LTC
1	*html/body*	**html/body/blockquote**	*html/body*
2	*html/body/table[7]/ tr/td/font/b*	*html/body*	*html/body/table[4]/ tr/td*
3	**html/body/blockquote**	*html/body/table[4]/ tr/td*	**html/body/blockquote**
4	*html/body/table[4]/ tr/td*	*html/body/table[7]/ tr/td/font/b*	*html/body/table[7]/ tr/td/font/b*

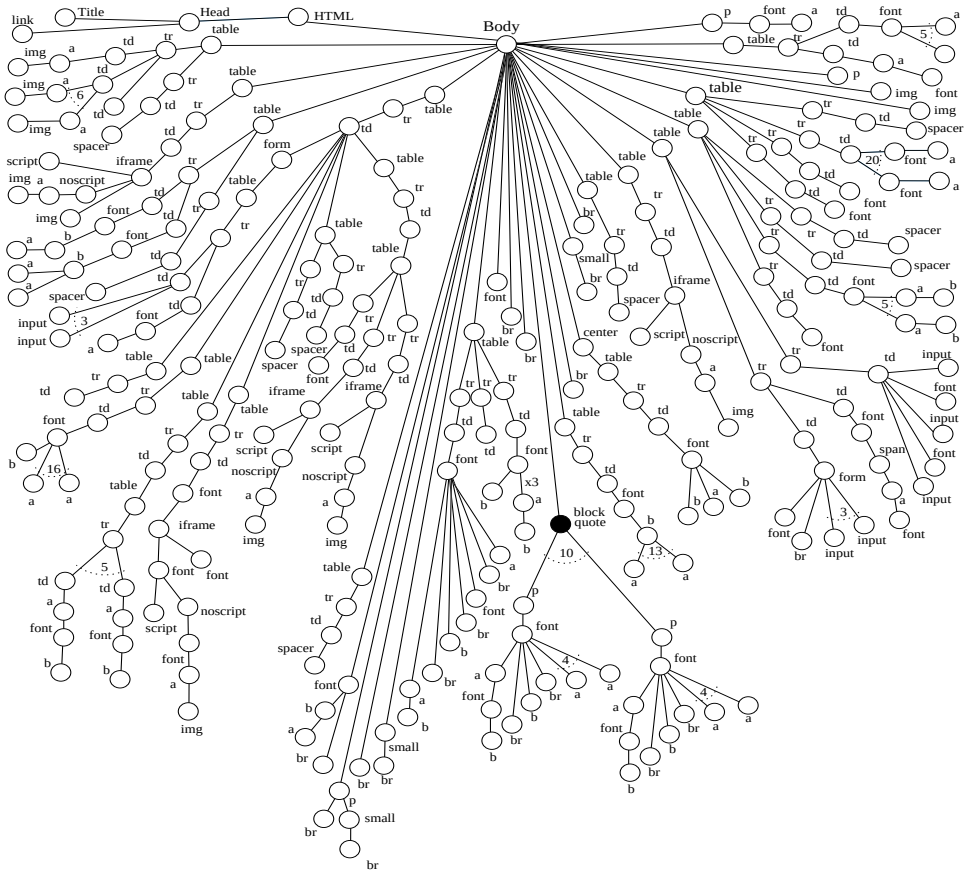

Figure 41: Tag Tree for nbci.com, Figure 40

4.4.1.3 The Largest Size Increase Subtree Identification Algorithm (LSI)

The LSI algorithm takes a tag tree of an arbitrary Web page and ranks all of its subtrees in two consecutive steps. First, it ranks all subtrees by their content size in byte. Since an ancestor of a node will always have a larger content size, in the second step we re-examine the ranking order between ancestors and their descendants, and re-rank ancestor/descendant pairs by the difference between their subtree size and the size of their largest child node. LSI promotes the fact that the greater the difference, the more likely the subtree is the minimal subtree. Recall the Web page in Figure 40 and its tag tree in Figure 41. By LSI the highest ranked subtree is *html/body/blockquote*. It is the minimal subtree that contains all the news objects in the page. However, both LTC and HF fail on this example. Table 3 shows the top four ranked subtrees by HF, LTC, and LSI. The statistics we use to determine the LSI rankings are given in Table 4 and Table 5.

Table 4: Statistics used in comparing HF, LSI, and LTC on nbci.com tag tree in Figure 41

Rank	Fanout (HF)	Size Increase (LSI)	Tag Count / HACC (LTC)
1	30	5614	333 / 14
2	22	3310	78 / 6
3	10	282	73 / 9
4	7	129	12 / 11

Table 5: Statistics used in calculating LSI on nbci.com tag tree in Figure 41

Rank	Subtree	Subtree Size	Largest Child	Largest Child Size	Size Difference
1	**html/body/blockquote**	6400	*html/body/blockquote/p[4]*	786	5614
2	*html/body*	9710	**html/body/blockquote**	6400	3310
3	*html/body/table[4]/ tr/td*	758	*html/body/table[4]/ tr/td/table[2]*	476	282
4	*html/body/table[7]/ tr/td/font/b*	144	*html/body/table[7]/ tr/td/font/b/a[10]*	15	129

The LSI algorithm is motivated by the following observations. First, when the highest fan-out algorithm fails, it usually fails on navigation menus in Web pages, which typically contain only links and descriptive link names. In contrast, the minimal subtree containing the data objects returned from a search is much larger in terms of the number of bytes. Second, the minimal subtree that contains the set of data objects of interest may not have the highest fanout or the largest tag count, but in most cases it will have a much larger size than any subtree of which it is not a descendant. Our experience shows that LSI has the highest success rate comparing to HF and LTC. However, there are cases where HF or LTC succeeds but LSI fails. Typically, when the data objects in a page are small in size and there are relatively too few search results in the page, the size difference between the object-rich minimal subtree and its largest child could be smaller than the size difference of a non-minimal subtree node and its largest child.

4.4.1.4 The Combined Algorithm and Its Performance

We have discussed three individual subtree identification algorithms.

They all produce a ranked list of subtrees and then choose the highest ranked subtree as the "correct" minimal subtree. All three algorithms work independently toward the same goal – finding the minimal object-rich subtree that contains all of the data objects. However,

each of the three algorithms uses a completely different criterion for subtree identification (such as fanout, tag count, or content size). Consequently, they do not always agree on their highest ranked choice and each of the algorithms is successful for only a subset of the Web pages. One way to improve the accuracy of object-rich minimal subtree identification algorithms is to develop a combined algorithm that finds the best way to combine the three independent algorithms. The goal is to make the combined algorithm successful for a much larger set of Web pages.

A well-known approach for combining evidences from two or more independent observations is to use the basic laws of probability [29]: Let $P(A)$ be the probability associated with the result of applying algorithm A over a Web page, and $P(B)$ be the probability associated with the result of applying algorithm B over the same Web page. The formula $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ will produce the compound probability $P(A \cup B)$ for locating the correct minimal subtree in this Web page. For example, if the probability factors that a subtree is the minimal subtree in a Web page are 78%, 63%, and 85%, then the compound probability for that subtree is 89% ($78\% + 63\% + 85\% - 78\% \times 63\% - 78\% \times 85\% - 63\% \times 85\% + 78\% \times 63\% \times 85\% = 89\%$). We refer to this compound probability the theoretical maximum success rate.

To combine the three individual algorithms, there are four ($\sum_{i=0}^{3} C(3,i) - 4 = 4$) possible combinations in addition to the trivial case of none and the three individual algorithms. To determine which of the combinations produces the best overall result for most of Web pages, we need to measure how well a combination algorithm performs. Concretely, we need to know the probability distribution of each of the three algorithms. This can be obtained by measuring the performance of each algorithm over a set of test Web pages. In addition, we need to know both the theoretical maximum success rate and the observed success rate. Then we determine the success rate for each of the four combinations.

Performance Measures of Individual Algorithms

To understand the performance of the three individual algorithms on different types of Web pages, we conducted a series of experiments on over 1,000 Web pages from 25 different Web sites. An empirical probability distribution for the success rate of each individual algorithm

is listed in Table 6. Whenever an algorithm is applied to a Web page, it produces a ranking of subtrees in the Web page. For each rank there is an associated probability (success rate). This probability is empirically determined by applying the individual algorithm over the set of test Web pages and calculating the percentage of times the correct subtree appears at each rank for each algorithm.

Table 6: Probability rankings for subtree identification algorithm

Algorithm	Rank 1	2	3	4
HF	0.63	0.32	0.01	0.02
LTC	0.65	0.17	0.17	0.00
LSI	0.91	0.09	0.00	0.00

Performance Measure of the Combined Algorithms

For each of the four possible combinations, a combined algorithm for subtree identification is implemented. To determine which combination is the best in general, we provide both the theoretical maximum success rate and the observed success rate. The former is obtained by using the probability formula, which computes the compound probability for each subtree in every Web page from our test set, based on the observed probability at each rank of each algorithm (given in Table 6). The later results from measuring the success rate of our implementation of the four combined algorithms. Concretely, each of the combination algorithms uses probabilities from the individual algorithms to rank each subtree in the Web page. The probability that a particular subtree is the minimal subtree is computed by combining the probability assigned to the subtree from each individual algorithm using the probability formula. For a particular Web site, we determine the percentage of times a combination algorithm correctly chooses the minimal subtree from all test pages of that site. Finally, the observed success rate is calculated by averaging the normalized percentage numbers from each Web site in our test list.

Table 7 shows the success rates of all combination algorithms over the test data. To conveniently represent a combination, each algorithm is abbreviated by a one letter acronym: **HF** by **H**, **LTC** by **T**, and **LSI** by **S**. Thus, **HTS** stands for the combination of the **HF**, **LTC**, and **LSI** algorithms. While some of the combinations performed equally well on the

Table 7: Success rates for subtree identification combinations on test data

Algorithm	Success	Combo	Experimental Success	Theoretical Maximum
HF	0.63	HT	0.65	0.87
LTC	0.65	HS	0.91	0.97
LSI	0.91	TS	0.91	0.97
		HTS	0.96	0.99

25 test Web sites, the combination of all three algorithms is the best choice. This is because each of the algorithms may fail in different circumstances, and using a combination of two algorithms is usually insufficient when the algorithms disagree with one another.

4.4.2 Object Separator Extraction

There are several ways to choose the object separator tags. One may consider each node in the chosen subtree a candidate separator or just the child nodes of the chosen subtree as the candidate separators. Based on the semantics of the minimal object-rich subtree, it is sufficient to consider only the child nodes in the chosen subtree the candidate separator tags.

Covering a wide range of possible mechanisms for discovering object separators, the first prototype of Omini supports five separator tag identification heuristics, each of which independently ranks the candidate object separators. The standard deviation heuristic (SD) and the repeating pattern heuristic (RP) were first proposed in [19]. The SD heuristic ranks the candidate tags based on the standard deviation of sizes between two tags. The RP heuristic ranks the candidate tags based on the difference between the counts of a pair of tags and the counts of a single tag. The partial path heuristic (PP) and the sibling tag heuristic (SB) are introduced in Omini. The former is inspired by the observation that multiple instances of the same object type often have the same tag structure. The latter is based on the observation that the objects identified by highest count sibling pairs are more likely to be of the same object type than the highest count single tags. The identifiable path separator heuristic (IPS) is an extension of the IT (Identifiable Tag) heuristic proposed in [19]. Instead of using the same list of pre-determined and ranked candidate tags for every tag tree, a different list is used based on the subtree that is chosen.

In the subsequent sections we will describe each of the five individual heuristics in detail. Then we will discuss a hybrid method to combine the rankings of the five heuristics for selecting the correct object separator.

4.4.2.1 Standard Deviation Heuristic (SD)

The SD heuristic measures the standard deviation of the distance (in terms of the number of characters) between two consecutive occurrences of a candidate tag, and then ranks the list of candidate tags in ascending order according to their standard deviation. It is based on the observation that the multiple instances of the same object type in a web document are approximately the same size.

Consider the tag tree for the Library of Congress web page shown in Figure 35. From the subtree extraction step, the subtree anchored at the node HTML[1].Body[2] is the chosen minimal subtree as shown in Figure 36. Among the set of child node tags, some tags have a lot more high counts than others do. For example, the tag *hr* appears twenty-one times, the tag *a* appears twenty-one times, and the tag *pre* occurs twenty times. We refer to these tags as the highest count tags. The standard deviation in distance is calculated between two consecutive occurrences of hr tag, between two consecutive occurrences of pre tag, between two consecutive occurrences of a tag, and so on. Table 8 shows the top three candidate tags by applying the SD algorithm to the Library of Congress example in Figure 36. It ranks the candidate tags in ascending order by the standard deviation in distance, with the smallest standard deviation first.

Table 8: Standard Deviation for tags from the minimal subtree in Figure 36

Rank	Tag	Standard Deviation
1	hr	114
2	pre	117
3	a	122

4.4.2.2 Repeating Pattern Heuristic (RP)

The RP heuristic chooses the object separators by counting the number of occurrences of all pairs of candidate tags that have no text in between. It computes the absolute value of the difference between the count for a pair of tags and the count for each of the paired tags alone and then ranks the candidate tags in ascending order by this absolute value. The reasoning behind this heuristic is that a single tag may mean many things, but that a pattern of two or more tags is more likely to mean one thing only. When there are no such pairs of tags in the chosen subtree, the RP heuristic produces an empty list of candidate tags. This simply means that the RP heuristic cannot identify the object separator tag amongst the candidate tags.

Table 9: Repeating tag ranking for minimal subtree from Figure 38

Tag Pair	Pair Count	Difference
table, tr	13	0
img, br	2	0
map, table	1	0
form, table	1	0
br, img	1	1
br, table	1	1

Consider the subtree $HTML[1].Body[2].table[5].tr[2].td[4].table[1]$ in Figure 38. Table 9 shows a ranked list of all tag pairs in descending order by the pair counts and the difference between the pair count and the individual tag count.

4.4.2.3 Identifiable Path Separator Tag heuristic (IPS)

The IPS heuristic ranks the candidate tags of the chosen subtree according to the list of system-supplied IPS tags. The IPS tags are tags identified by the system as the most commonly used object separator tags for different types of subtrees in Web documents. The idea behind this heuristic is the following. First we observe that Web documents, generated either by hand, by authoring tools, or by server programs, often consist of multiple presentation layouts within a single page, each defined by some specific type of HTML tag. For example, a web page may contain a table marked by table tag table, a list marked by

the list tag ul or ol, and a paragraph marked by the tag p. Secondly, each presentation layout tends to use a regular structure. For example, a table tends to use the row tag tr and the column tag td to define rows and columns of the table, and a list tends to use the list item tag li to define the list structure. Therefore, for every presentation layout (i.e., a subtree type), there are a few tags that are consistently used for separating objects within the subtree.

Based on these observations and the set of Web documents we have tested, we have created a list of object separator tags for each type of subtree as shown in Table 10. The full list of object separators is composed of all the identified tags for each type of subtree listed in Table 10, with duplicates removed.

Table 10: A Table of Object Separator Tags

Subtree	**Tag List**
body	table,p,hr,ul,li,blockquote,div,pre,b,a
table	tr,b
form	table,p,dl
td	table,hr,dt,li,p,tr,font
dl	dt,dd
form	table,p,dl
ol	li
ul	li
blockquote	p

The next step is to determine the rankings of these commonly used object separator tags. Table 11 lists the distribution of all object separator tags we observed in our tests of 50 web sites with a composite of over 2000 web pages. Based on the experimental results of the web sites tested, we have produced the following ranking of the full list of IPS tags. Such an ordered list is called the $IPSList$:

$\{tr,table,p,li,hr,dt,ul,pre,font,dl,div,dd,blockquote,b,a,span,td,br,h4,h3,h2,h1,strong,em,i\}$.

For tags of the same rank, the order is arbitrary.

Table 11: Object Separator Probability

Tag	% of time used as object separator
tr	34
table	18
p	10
li	8
hr	6
dt	6
ul	2
pre	2
font	2
dl	2
div	2
dd	2
blockquote	2
b	2
a	2

4.4.2.4 Sibling Tag Heuristic (SB)

The SB heuristic counts pairs of tags that are immediate siblings in the minimal subtree, and ranks all pairs of tags in descending order by the number of occurrences of the pair. For pairs of tags that have equal occurrences, the ranking follows the order of their appearances in the Web document. For example, in the fragment <a> ... ... <c> ... </c> , the sibling pairs are (a,b) and (b,c) and each occurs only once. Table 12 ranks sibling pairs from the Library of Congress tag tree in Figure 35 and bn.com tag tree in Figure 38. The sibling tag heuristic is inspired by the observation that given a minimal subtree, the object separator tag is expected to appear as many times as there are objects.

4.4.2.5 Partial Path Heuristic (PP)

The PP heuristic lists all the paths from a candidate node to any other node reachable from the candidate node in the chosen subtree, and counts the number of occurrences of each identified path. The list of candidate tags is then ranked in descending order by the count of all the identified paths. If two paths have an equal count, then the longer path will rank higher than the shorter one because the longer one indicates more structure. It

Table 12: Tag ranking for SB heuristic on Figure 38 and 35

	bn.com		Library of Congress	
Rank	**pair**	**count**	**pair**	**count**
1	table,table	11	hr,pre	20
2	img,br	2	pre,a	20
3	br,img	1	a,hr	20
4	br,table	1	h1,i	1
5	table,map	1	i,hr	1
6	map,table	1	hr,a	1
7	table,form	1	a, br	1
8			br,form	1
9			form,p	1

is interesting to note that if there are no paths with a length more than one, such as in Figure 35, this heuristic reduces to simply choosing the tag with the highest count. The main idea behind this heuristic lies in the observation that the multiple instances of the same object type often have the same tag structure.

Table 13 shows the PP rankings for the example web document in Figure 38 and the Library of Congress page in Figure 35.

Table 13: Tag ranking from partial path rankings for Figure 38 and Figure 35

	bn.com		Library of Congress	
Rank	**tag**	**count**	**tag**	**count**
1	td	355	hr	21
2	tr	227	a	21
3	img	176	pre	20
4	font	146	form	8

4.4.3 Determining the Correct Object Separator Tag: The Combined Algorithm

We have discussed each individual heuristic and its algorithm to produce a ranked list of candidate tags. Each of the five individual algorithms works independently towards the same goal – finding the right object separator from the set of candidate tags. As a result, each heuristic chooses its own highest ranked tag as the object "correct" separator. However,

as we observed from the discussions in the previous sections, these five heuristics may not always agree on the same highest ranked choice.

Table 14: Probability rankings for object separator heuristics

Heuristic	Rank 1	2	3	4	5
SD	0.78	0.18	0.10	0.00	0.00
RP	0.73	0.13	0.00	0.00	0.00
IPS	0.40	0.46	0.13	0.07	0.00
PP	0.85	0.06	0.02	0.00	0.00
SB	0.63	0.17	0.12	0.06	0.03

To understand the performance of these individual heuristics on different types of web pages, we have conducted a series of experiments on more than 500 web pages from 15 different web sites (see [10] for further detail). An empirical probability distribution for the success rate of each individual heuristic is listed in Table 14. For each heuristic, we first calculate the number of times the heuristic chooses a correct object separator tag at a particular rank. Then for each web site, we compute the success rate of the given heuristic by normalizing the number of times the correct separator tag is chosen by the number of pages tested over a particular web site. The success rate for each heuristic over the 15 web sites (shown in Table 14) is then calculated by averaging the normalized numbers from each web site.

A natural way to improve the accuracy of finding the correct object separator in a web document is to consider a hybrid approach to combine these independent heuristics. A well-known approach for combining evidences from two or more independent observations is applying the basic laws of probability [29]: Let $P(A)$ be the probability associated with the result of applying heuristic A over a web document, and $P(B)$ be the probability associated with the result of applying heuristic B over the same web document. The formula $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ produces the compound probability $P(A \cup B)$ for locating a correct object separator tag in this web document. For example, if the probability factors that a tag tr is an object separator in a web document are 78%, 63%, and 85%, then the compound probability for tag tr is 99% ($78\% + 63\% + 85\% - 78\% \times 63\% - 78\% \times 85\% -$

$63\% \times 85\% + 78\% \times 63\% \times 85\% = 99\%$).

To combine the five individual heuristics, there are 26 possible combinations in addition to the trivial case of none and the five individual heuristics. Table 15 lists the results of all combinations. To conveniently represent a combination, each heuristic is abbreviated by a one letter acronym: HC by H, SD by S, RP by R, IPS by I, PP by P, and SB by B. Thus, RSIPB stands for the combination of the RP, SD, IPS, PP, and SB heuristics. Based on the set of testing web documents, the combination of all five heuristics performs the best.

Table 15: Success rates for heuristic combinations on test data

Combo	Success	Combo	Success	Combo	Success
IB	0.61	RB	0.73	RI	0.75
RS	0.78	SI	0.78	SB	0.78
RIB	0.80	RSB	0.84	SIB	0.84
RPB	0.85	RP	0.85	SP	0.85
IPB	0.85	IP	0.85	PB	0.85
RSI	0.86	RIPB	0.86	RIP	0.86
RSP	0.88	SIPB	0.89	SIP	0.89
SPB	0.89	RSIP	0.92	RSIB	0.92
RSPB	0.92	RSIPB	0.98		

4.5 Element Extraction

After individual objects have been extracted from a page, the next step, called Element Extraction, identifies the elements inside the objects. A data object typically consists of several elements that are separated by a group of element separators. An element separator could be either an HTML tag, such as <td> in an HTML table, or a plain-text delimiter, such as the slash sign in "Cay Horstmann,Gary Cornell / Paperback / Prentice Hall /".

We have found that data objects from the same content region in a Web page are often homogeneous, sharing the same structure and group of element separators. This allows us to decompose these data objects into elements of the same or similar data types after the common group of element separators is located. In XWRAPElite, we develop a number of techniques for discovering and locating similar patterns in tag and text separation across objects extracted from a Query-Answer page, and then compile a group of separators that

are suitable for most of the objects.

4.5.1 Element Separation

There are several reasons why the object extraction methodology used in Section 4.4 for identifying and locating object separators cannot be directly applied to finding the group of element separators. First, objects are similar to each other in a Query-Answer page, while elements in different objects can vary widely in the total number of elements. As a result, the same type of elements may have different ordering across different objects. So it is important to have smart element alignment rules that can find the correct alignment relationship among the elements within an object. Secondly, the object extraction algorithms [9] assume that there exists a single HTML tag marking the boundary between objects in dynamically generated query-answer pages. This assumption is not applicable to element separation. For example, text delimiters are often as element separators in many dynamically generated query-answer pages.

In XWRAPElite two different approaches are developed for element separator identification problems. One is the mechanism for locating element tag-separators and another is the algorithms for identifying element text-separators. We then combine these two approaches to determine the correct element separators that can actually locate the boundaries between semantic meaningful elements within an object. We discover and locate the element tag-separators using a number of heuristic algorithms, including highest count tag, combination pattern, and standard derivation. To obtain text-separators, we extract text strings from objects and then analyze them with a string-based heuristic.

4.5.1.1 Identifying Element Tag Separators

To obtain a basic group of element tag-separators, we build HTML tag-trees for all objects extracted from a page. We start the element extraction from an object that is most representative of the set of objects extracted. For instance, if we have a total of 20 objects, among which 90 percent objects have similar size in bytes with only two exceptions: one object is quite smaller and another object is relatively large. It is quite likely that the exceptions found are outliers produced by the object extraction algorithms and should be

considered as errors. Given a tag tree that corresponds to the chosen representative data object, element tag-separators decompose the tree into sub-trees that contain semantically meaningful elements of interest. For instance, correct element tag-separators should not break any semantically meaningful element into pieces. Consider the tag tree in the lower part of the Figure 38. It shows that <font> is a correct element tag-separator but <b> is not since the tag node **b** only contains a part of the book title, "Beginning Java 2: JDK 1.3 Version". The main challenge is how to automatically discover element separators with high success rate.

In XWRAPElite, we use an incremental approach to finding the correct element separators. We first use a simple heuristic to get a basic set of tags that are commonly used as tag-separators in query-answer objects of a dynamically generated page. Then we apply three tag-tree based algorithms to prune those tags that are not good element tag-separators for the given object.

Concretely, we first look for all commonly used element tag-separators that are often used as element separators in dynamically generated objects, such as
 and <a>. Then we apply the following tag-tree based algorithms to find the best candidate tag-separators for element separation in the given object. The three ag-tree based algorithms are described as follows.

- **Highest Count Tag Heuristic** This heuristic considers tag occurrences. Elements are usually displayed uniformly or in a similar style. Often the same HTML tag is used to repeatedly as separator between elements of an object. This observation tells that the more frequently a tag occurs, the more likely it is used as an element separator. The highest count tag algorithm takes the tag-tree of the chosen object and returns the list of tags with the highest appearance count ranked the first.

- **Combination Pattern Heuristic** Another common phenomenon observed from the query-answer objects in dynamically generated pages is the fact that a combination of tags may appear together repeatedly in between multiple elements within a data object. The most common combination is the multiple appearances of adjacent tag

pairs, either as a parent-child pair or as a sibling pair. We call this type of tag pairs the *Combination Pattern*. The Combination Pattern algorithm explores such patterns in the tag-tree of the given object. If a tag frequently appears together with another tag or a tag combination pattern, then it is very likely that this tag should be considered as a candidate element separator. We compute the appearance frequency of a tag in a combination pattern by the difference between the tag occurrences and the combination pattern occurrences. The smaller the difference is, and the higher the tag's combination pattern frequency is, the more likely the tag is a good candidate element separator.

- **Standard Deviation Heuristic** This third heuristic takes a tag's occurrence pattern into account. Assuming elements are similar, a tag-separator often appears in a regular pattern, which is implied by the standard deviation of the interval between two consecutive occurrences of the same tag. The smaller the standard deviation of a tag's interval, and the more regularly the tag occurs, the more likely the tag is a separator.

4.5.1.2 Identifying Element Text Separators

Elements in plain text strings are usually separated by some symbols, such as " " and "/". We have built a seed list of commonly used text-separators based on our experiences and the Web documents we have studied. The current list includes the following symbols: slash ("/ "), colon (": "), space (" "), semicolon ("; "). Perhaps surprising to Unix programmers, line separators, such as "\n" are not included in our commonly used text-separator seed list. HTML treats the line separators as a simple space, so the line separators are often in the middle of an element.

However, we have observed that these commonly used text-separators sometimes also represent math signs or time formatting marks. For example, a slash can be used for the division operation (i.e. $15/3 = 5$), and a colon can separate time units (i.e. "15:30 p.m."). So we do not consider a slash (or a colon) a separator when there are digits on both sides of the slash (or the colon.)

We search text-separators in an object as follows. If a symbol on the seed list appears in the object and there are no digits occurring on both sides when the symbol is a slash or a colon, the symbol is placed into a group of candidate text-separators. After all the symbols in the seed list are examined, the final group contains all the text-separators for the object.

4.5.1.3 Determining the Best Element Separators

For each Web document examined by XWRAPElite object extraction component, a set of query-answer objects are returned. After obtaining element tag-separators and element text-separators for each object, we leverage them to build a representative element separator group, which only contains separators that appear in most objects. In the first prototype of XWRAPElite, we use the following algorithm to build the representative group. First, we union element separator groups for all objects into one big separator list. Second, for each separator on the list, we check if it appears in at least two groups. Removing the separators that only appears once from the list, we avoid accidental separators, such as the colon in the book title in Figure 31. The remaining sepators on the list form the representative group. Table 16 demonstrates an example induction of representative separators based on a primitive element separation study on four objects. In this example, it is clear that `td` is the tag element separator and slash "/" is the text element separator.

Table 16: Building a representative separator group

	Tag Separator	**Text Separators**
Object 1	td	slash("/"), colon (":")
Object 2	td, i	slash("/")
Object 3	td	slash("/")
Object 4	td	slash("/")
Representative	td	slash("/")

Figure 42 shows a screenshot of the applying element separation over an example source URL. Note that the XWRAPElite system prints out statistics at each step of the code generation process. In the element separation step, the statistics printed out include the total number of objects extrated, the average number of elements (fields) per object and the maximum number of elements (fields) per object.

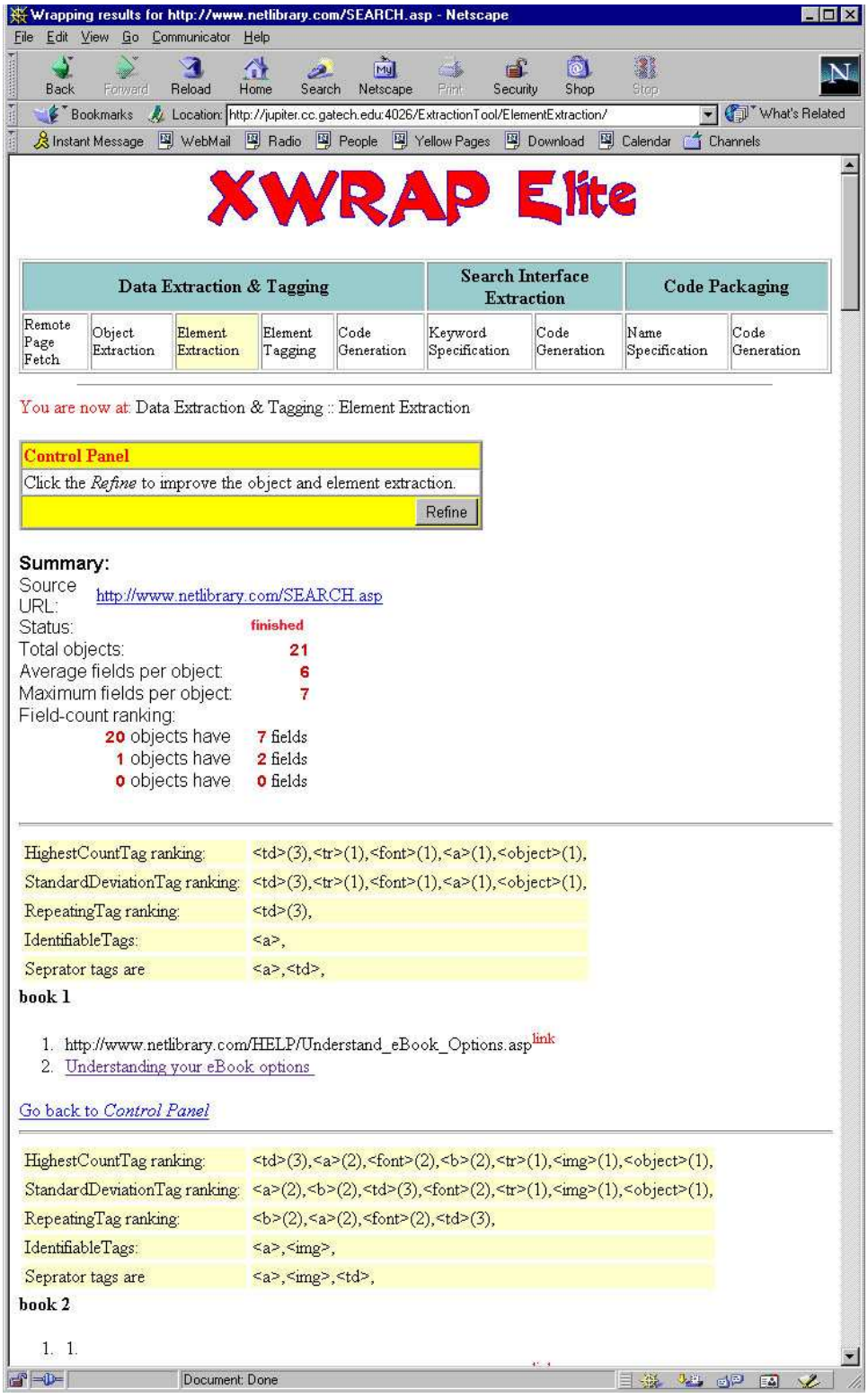

Figure 42: XWRAP Elite Screen Shots – Element Extraction

4.5.2 Refining Object Extraction Results Through Element Separation

In Section 4.4, we have found that some extracted objects are not of interest to the user. For example, we may extract an object as "*The results are:*" because it is on the first row of an HTML table and the rest of the rows are data objects of results. Many data-free objects often have much fewer or many more elements than objects of interest have. This observation has prompted us to remove some false objects by their element counts.

Object pruning is based on a boundary parameter that indicates the element counts of the objects of interest, which are bounded by the **Maximum Element Count** and the **Minimum Element Count**. Any object that has more elements than the Maximum Element Count or fewer elements than the Minimum Element Count will be considered a unwanted object.

We automatically estimate the element count boundary using the algorithm described as follows. After separating objects into elements, we group the objects by their element counts. Then we rank the groups by their size and choose the top three groups with the most objects. We assume the largest element count from the three groups to be the upper bound of the element count range and the least element count from the three groups to be the lower bound of the range. We allow developers to manually modify the upper and lower bounds as they see appropriate.

One may ask why the top three groups are considered. This is based on our experience and the test websites we have used. It would be interesting to see whether this number works for different types of websites and if not how to tune this number according to different types of websites.

```
10  objects have  10  elements
9   objects have  12  elements
8   objects have  8   elements
3   objects have  1   element
2   objects have  20  elements
1   objects have  9   elements
1   objects have  3   elements
```

Figure 43: Object Pruning Example

Figure 43 shows a ranking of groups by element counts. We look for the element count boundary within the top three groups, which contain ten, nine and eight objects respectively. So the maximum element count is twelve and the minimum element count is eight. We discard three objects that only have one element, two objects that have twenty elements and one object that has three elements. There is one object that has nine elements; however, we do not omit it because its element count falls in the range of (8, 12).

As shown in Figure 44, users can also manually change the element count boundary by clicking on the refine button in Figure 42.

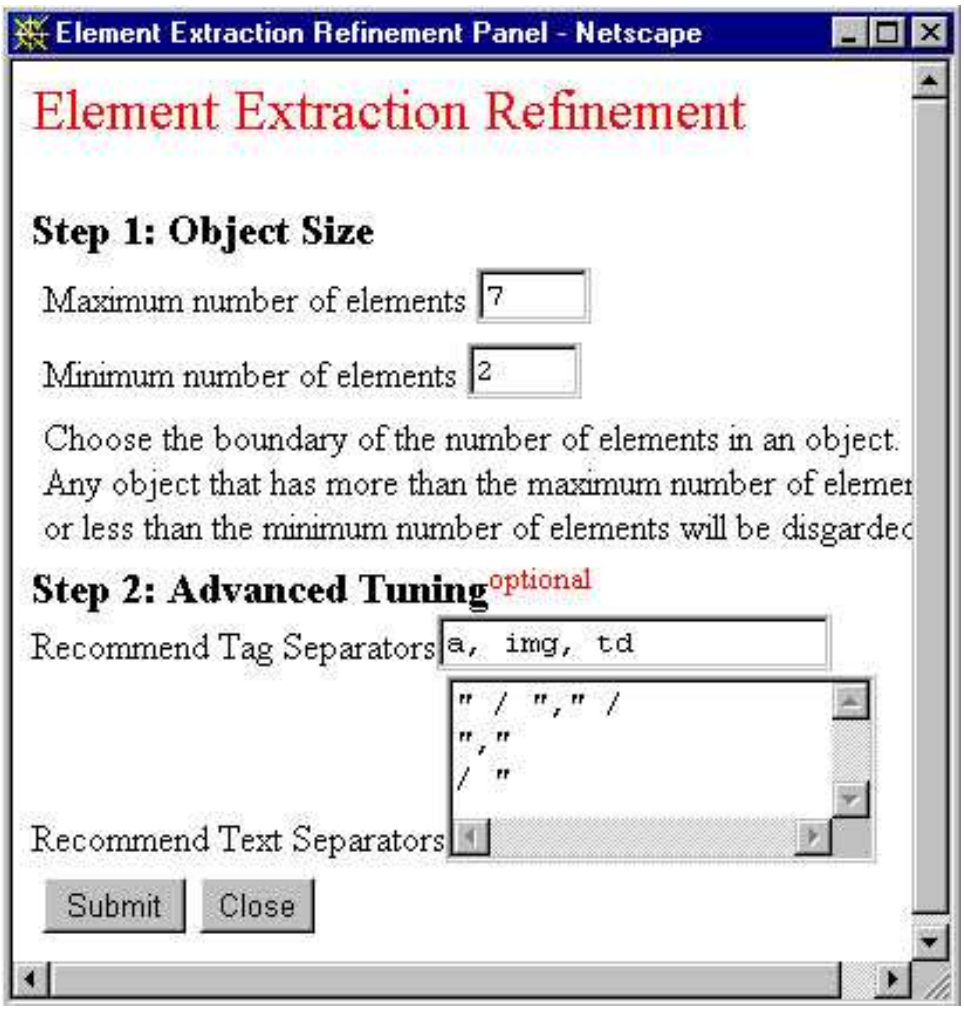

Figure 44: XWRAP Elite Screen Shots – Object Pruning

4.6 Element Alignment and Output Tagging

After Element Extraction, all the objects extracted can also be separated by elements. However, these data objects are still strings of text containing HTML tags. They are semantically meaningful only through human interpretation through browsing and are not understood by any machine-readable programs. In order to enable the object extraction and element extraction results to be fed into the machine-readable programs, it is important to transform the extracted data into a machine-readable semantic information. One way to achieve this objective is by marking objects and elements with semantically meaningful XML tags. This process is called **Output Tagging**. It allows domain experts to mirror the output of data extraction results with a pair of XML glasses. By default, the XWRAPElite system will use the word "record" to tag the objects extracted and "element" to tag the elements identified.

A common approach to identify data elements is using string regular expression matching. However, accurate regular expressions are always very difficult to obtain automatically, especially when elements are similar. An inaccurate regular expression will misidentify data elements. Another approach to identify elements is based on the appearance order of elements within an object. It works well for certain application domains, such as laboratory-experiment logs, but it cannot be applied to other areas. Furthermore, if an optional element does not occur in an object, it will cause the misidentification of the rest of the elements in the object.

Our approach is a hybrid one that combines regular expressions with the element appearance order. We initially assign an index number to each element according to the order in which it appears. Then we automatically generate some regular-expression patterns to help us align the index numbers in case an element is missing. Elements with the same number will be tagged in the same name. As we addressed earlier, it is difficult to automatically generate fine-grained regular expression for each data element. However, although the generated regular expressions are not always accurate or detailed enough to identify a particular element, they are good for the aligning purpose. For example, a price value in an American Web site always starts with '$', so a URL link starting with "http://" cannot

be a price element.

Order	Object1	Object2
1	<a href="...">	<a href="...">
2	Core Java2, Volume 1: Fundamentals	Java How To Program
3	In-Stock: Ships with 24 hours	In-Stock: Ships with 24 hours
4	Cay Horstmann,Gary Cornell	Harvey M. Deitel,Paul J. Deitel
5	Paperback	Paperback
6	Prentice Hall	Prentice Hall
7	November 1998	August 1999
8	Our Price:	Our Price:
9	$34.40	$64.75
10	You Save	<a href="morelink">
11	20%	more
12	<a href="morelink">	<img src="...">
13	more	
14	<img src="...">	

Figure 45: Elements From Two Objects Before Alignment

Figure 45 shows elements separated from the two objects in Figure 31. Object1 has 14 elements, while two elements are missing in Object2. The 10th element in Object2 doesn't match the 10th element in Object1. However, the 10th element in Object2 has the same type (i.e. Hypertext link) as the 12th element in Object1, and the 11th element (i.e. "more") in Object2 has the same value as the 13th element in Object2. So we can align the 10th, 11th, 12th elements in Object2 with the 12th, 13th, 14th elements in Object1, which is shown in Figure 46.

Figure 47 is a screenshot of the control panel for managing element alignment and inputting element names. Users can manually type element names and reset the alignment rules for better precision.

4.6.1 Discovering Regular Expression Pattern

Regular expression patterns can be strongly-matched and weakly-matched. A strongly-matched pattern normally implies a high possibility that two elements should be aligned if

Order	Object1	Object2
1	<a href="...">	<a href="...">
2	Core Java2, Volume 1: Fundamentals	Java How To Program
3	In-Stock: Ships with 24 hours	In-Stock: Ships with 24 hours
4	Cay Horstmann,Gary Cornell	Harvey M. Deitel,Paul J. Deitel
5	Paperback	Paperback
6	Prentice Hall	Prentice Hall
7	November 1998	August 1999
8	Our Price:	Our Price:
9	$34.40	$64.75
10	You Save	
11	20%	
12	<a href="morelink">	<a href="morelink">
13	more	more
14	<img src="...">	<img src="...">

Figure 46: Elements From Two Objects After Alignment

they both match the pattern, while a weakly-matched pattern indicates that two elements should not be aligned if only one of them matches the pattern.

Constant-value elements, such as "Review" or "Our price", are often strongly-matched patterns. Some elements contain constant strings with some variable numbers, for example, "ends in 5 hours 10 minutes." If we discard the variables, ("5" and "10" in this case,) the elements have a constant-value-like pattern. Constant-value and constant-value-like elements appear in a substantial ratio of objects so that we can easily recognize them from statistics by counting their appearances.

We automatically recognize four types of weakly-matching patterns, which correspond to the following four different classes of elements: hypertext links, images, dollar values (any element starting with "$" and followed by numbers, such as "$45.00"), and common text strings (the rest of the elements). These patterns help us avoid mismatching elements. For example, in Figure 45, the 10th element in Object2, which is a hypertext link, should match the 12th element in Object1 instead of the 10th element in Object1.

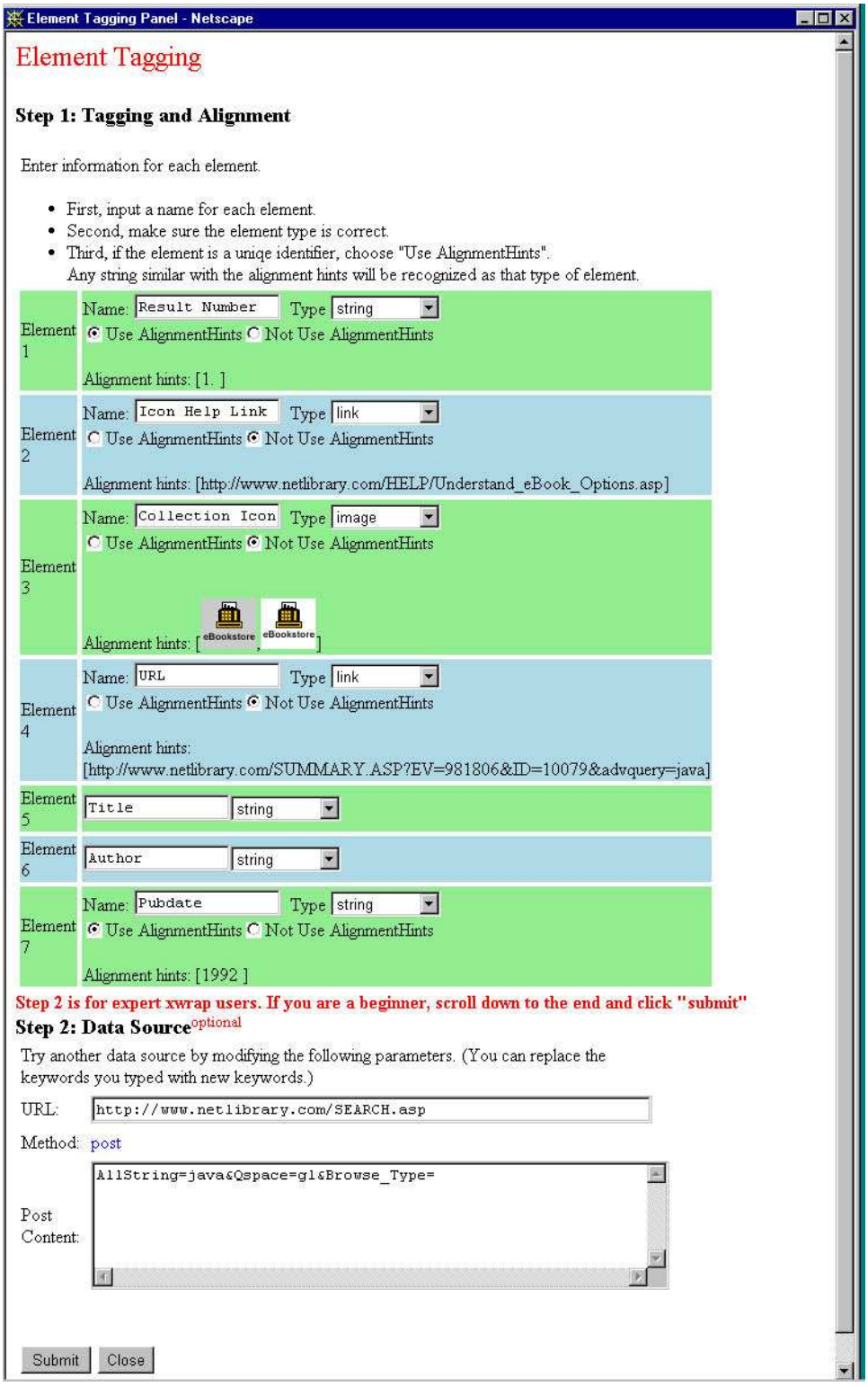

Figure 47: XWRAP Elite Screen Shots – Element Alignment and Tagging

4.6.2 Element Alignment

Element alignment matches the elements of an extracted object to the elements in the largest-element-count object. The basic idea is to assign each element an index, which is the order of its matching element in the largest-element-count object. Elements with the same index should share the same element name.

Given an element E in an object extracted and the object O with the largest-element-count, we find a matching element in O for an element E in the following three steps.

- *Searching Strongly-Matching Element*: If E is constant-value-like and it has a strong-matching element in the largest-element-count object, assign the index of the matching element to E. If there are multiple strong-matching elements, choose the one with a closest index location. If there is no matching element or E is not constant-value-like, go to the next step.

- *Searching Weakly-Matching Element Backward*: If an element in the largest-element-count object, whose index is smaller than or equal to the index of the element that is immediately before E, shares a weakly-matching pattern with E, we assume it is the matching element. If there are multiple matching elements, we choose the one with the largest index. If there is no matching elements, E's matching element is "unknown".

4.6.3 Code Packaging

After XWRAP Elite applies Object Extraction and Element Extraction, it can easily produce code for data extraction. However, in order for the code to be conveniently accessed, we package it with a search interface. Instead of asking for the URL location of input documents, a search interface contains a URL pattern that contains placeholders and automatically constructs a URL from users' keywords input by replacing placeholders with the keywords. Figure 48 shows a screenshot of generating a search interface.

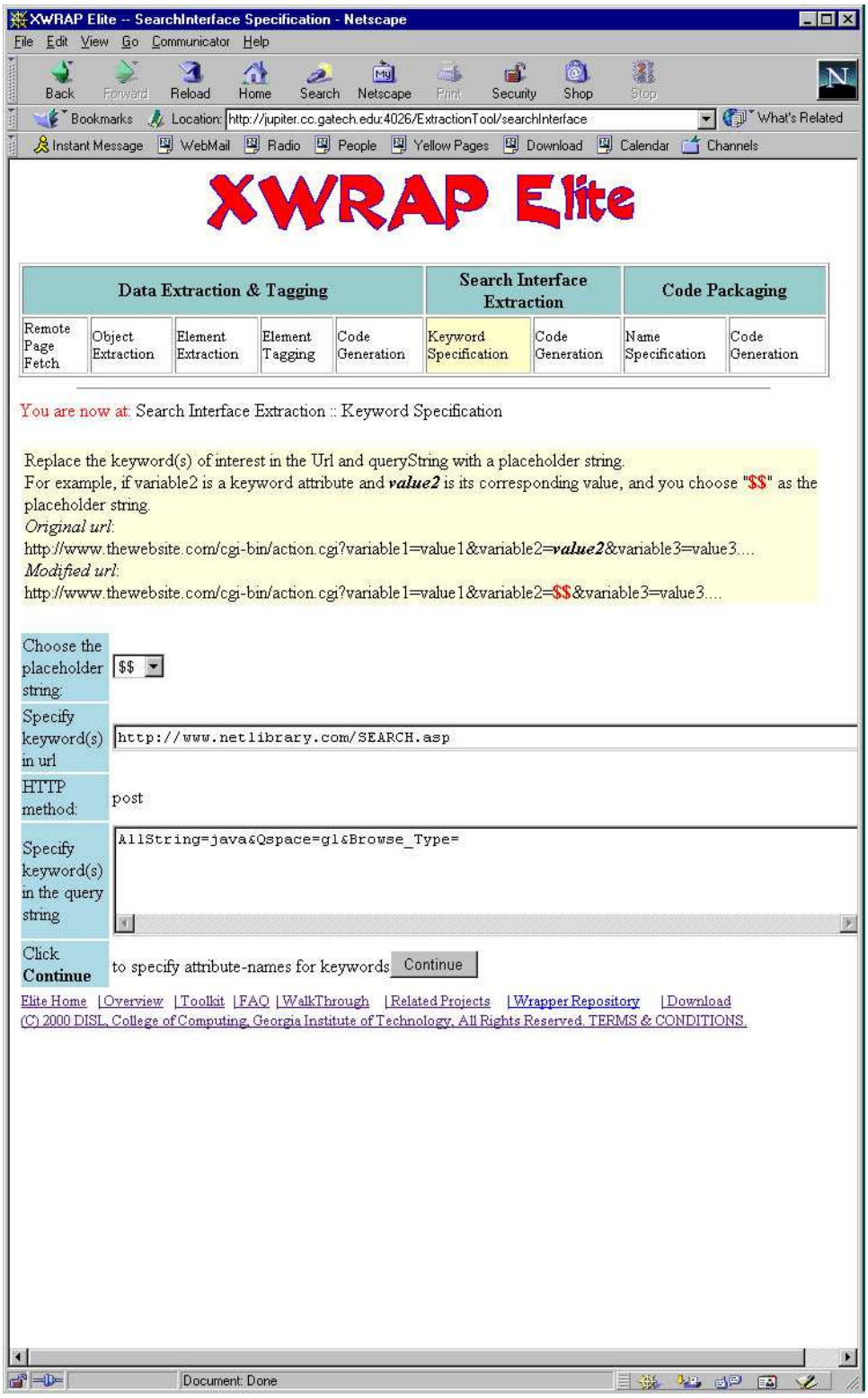

Figure 48: XWRAP Elite Screen Shots – Search Interface

4.7 Performance Evaluation

Our performance experiments examine both XWRAPElite toolkit and generated wrappers. For XWRAPElite, We study the process of wrapper generation. How fast can we build a wrapper? How many revisions do we make to achieve a high quality result? For generated wrappers, we evaluate their execution performance and accuracy of the extracted results.

4.7.1 Experiments On Wrapper Generation

Choosing 20 Web sites with diverse sophistication, an expert wrapper developer has utilized XRWAPElite to build wrappers for the Web sites. Figure 49 shows that all wrappers have been produced in less than 20 minutes. The time to generate a wrapper is loosely positively correlated to total number of elements the wrapper extracts from a page. In our experiments, when total elements are less than 100, wrapper generation costs less than ten minutes. Otherwise, it takes more than ten minutes to generate a wrapper. That is mainly because the developer has to spend more time in reviewing the extracted results during the generation process.

4.7.2 Experiments On Wrapper Quality

It is important to assure that the generated wrappers possess reasonable quality. Wrapper quality lies in two aspects, extraction accuracy and execution performance. A good wrapper should achieve high extraction accuracy within reasonable execution time. The set up of our wrapper quality tests is described as follows:

- All documents are stored locally.
- There are 10 to 50 documents per site.
- We run the wrappers on a Sunw Utltra-4 without other substantial workload.
- The OS is Solaris 5.7 and JVM version is 1.4.

4.7.2.1 Extraction Accuracy

Extraction accuracy is usually measured by *precision* and *recall*. Precision refers to the percentage of correct items in results while recall refers to the percentage of correct items

Web Sites	Wrapper Generation Time (m)	Total Elements	Revision (times)	Doc Size (KB)
buy.com	15	447	2	31.648
chapters	15	417	3	43.01
barnes	12	350	0	14.019
booksense	13	280	6	78.784
clinicalTrials	17	250	2	51.704
signpost	10	174	1	14.542
cbcconsumer	10	150	1	20.665
cnet2	20	144	2	18.689
infoseek	15	110	3	15.934
powells	12	107	3	25.759
canoe	10	100	2	3.714
coolservlets	10	75	0	14.261
mysql	10	72	3	68.274
contesting	10	54	1	23.811
linuxhacker	10	50	0	7.182
cancerlit	8	50	0	2.754
edusearch.de	8	48	1	27.098
eanimals	10	42	1	20.972
bangladesh.net	10	24	2	12.242
clockstop	10	20	1	4.251

Figure 49: Wrapper Generation Time Experiments

that are extracted versus total correct items in original data source. For example, Figure 50 demonstrates the object extraction accuracy of XWRAPElite wrappers, whose precision and recall both reach at least 80 percent for all Web sites. The definitions of object extraction precision and recall are described as follows:

- *Object Extraction Precision = (correctly extracted objects)/(total extracted objects)*
- *Object Extraction Recall = (correctly extracted objects)/(total correct objects in the documents)*

Among the 20 Web sites we have tested, 18 have achieved perfect precision and 10 have gained perfect recall scores. There are fewer perfect recall scores because we give a higher priority to precision in case of a conflict between precision and recall. For example, when a false object contains similar number of elements as true objects, we often choose to prune out the false object to maintain high precision even though some true objects might be

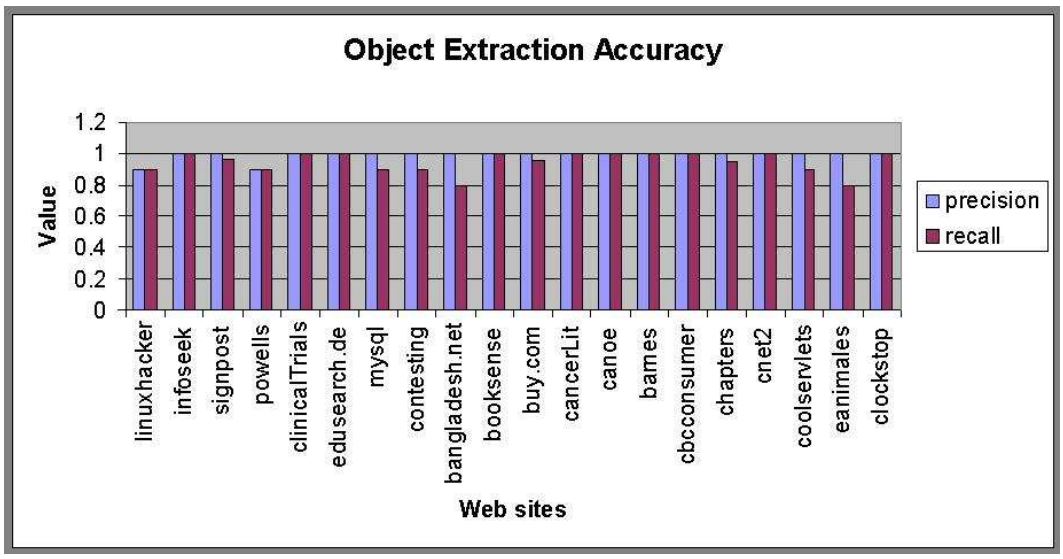

Figure 50: Object Extraction Accuracy

eliminated as well.

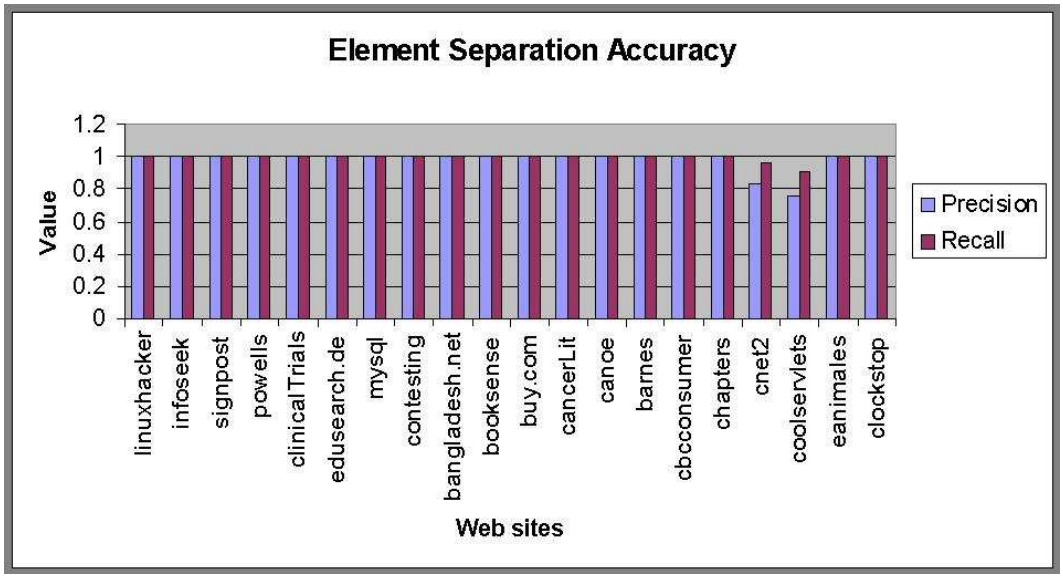

Figure 51: Element Separation Accuracy

Extraction accuracy of XWRAPElite wrapper at the element level is even better than the object level. We have conducted two experiments before and after element tagging respectively. The following formulas define precision and recall at the two stages.

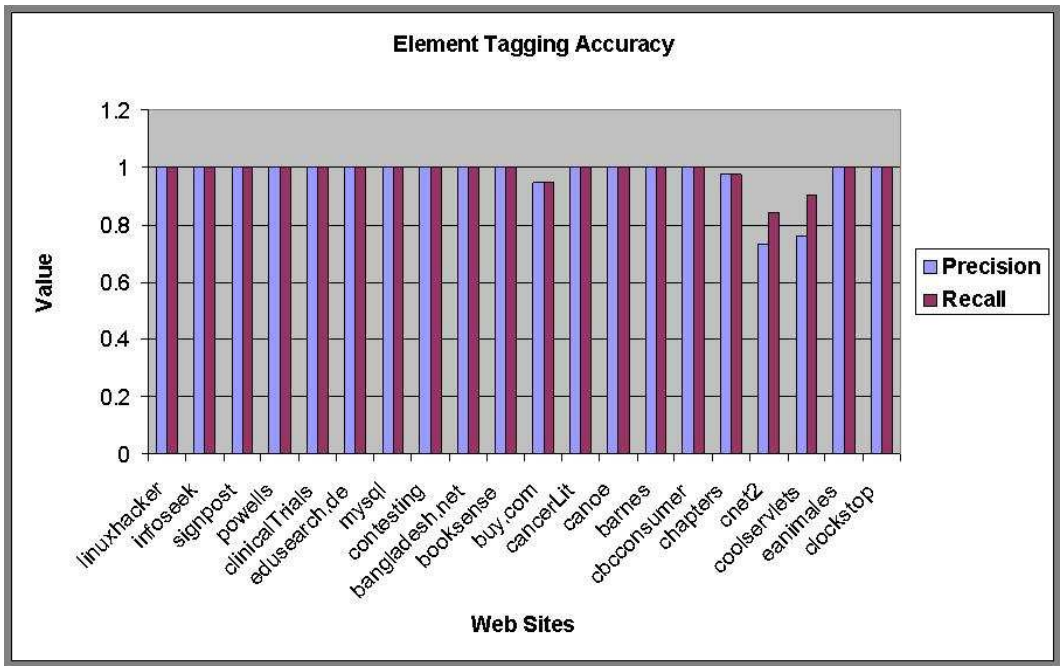

Figure 52: Element Tagging Accuracy

- *Element Separation Precision = (correctly extracted elements)/(total separated elements)*
- *Element Separation Recall = (correctly extracted elements)/(total elements in all correct objects)*
- *Element Tagging Precision = (correctly labeled elements)/(total labeled elements)*
- *Element Tagging Recall = (correctly labeled elements)/(total elements in the correct objects)*

Figure 51 shows there are 18 out of the 20 Web sites achieve perfect precision and recall after element separation.

Errors in element separation can propagate to element tagging. Figure 52 shows 16 Web sites maintain the double-perfect scores after element tagging. Among the remaining four Web sites, one of them does not make any error in tagging correctly extracted elements, but wrongly extracted elements contributes to the inaccuracy after element tagging. The tagging inaccuracy also comes from alignment errors. For example, when an object contains

unbounded number of elements, the alignment rules are likely to fail. Figure 53 presents an alignment failure in an object from CNET when two search engine links appear while there is only one search link in most other objects. In fact, the number of search engine links is unbounded. Relying on the knowledge of the maximum number of elements, XWRAPElite's alignment rules can cause tagging mistakes.

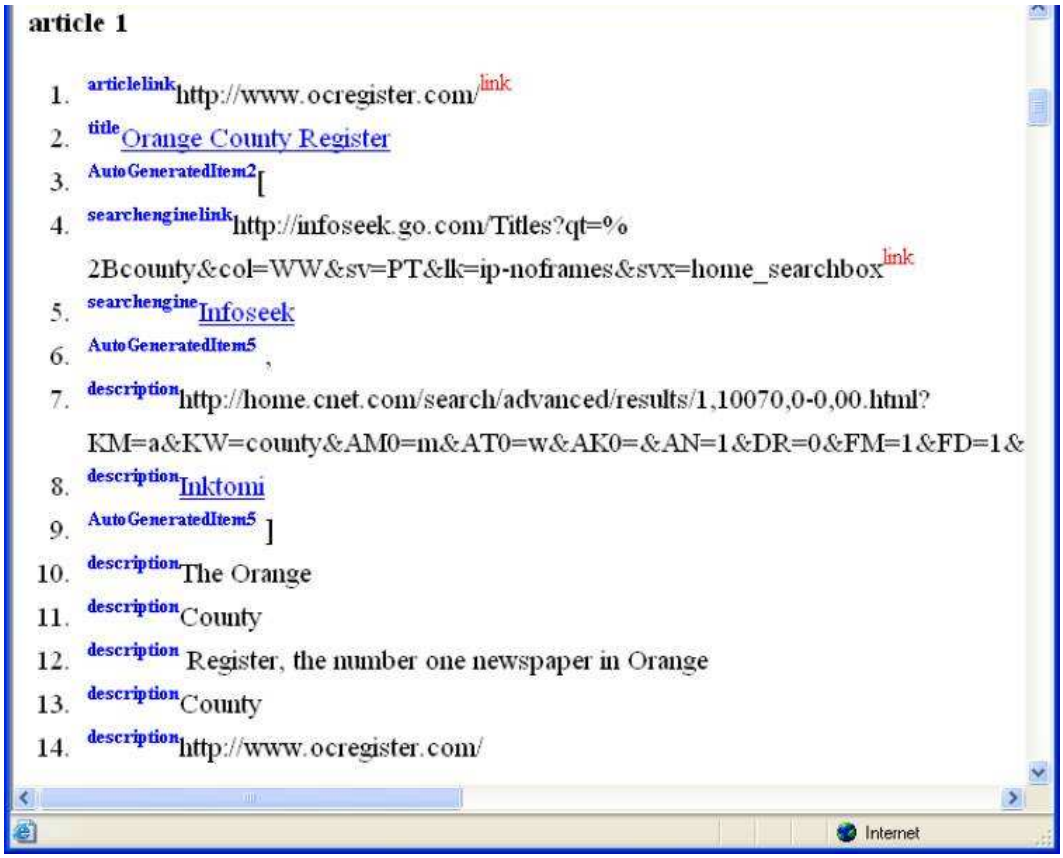

Figure 53: Element Tagging Fragment for CNET

The overall accuracy of XwrapElite wrappers is derived from the following formula:

- *overall precision = object precision * element tagging precision*
- *overall recall = object recall * element tagging recall*

The calculated results are shown in Figure 54. 70 percent of Web sites produce perfect precision and 45 percent gain perfect recall. All Web sites manage to achieve at least 0.7 at precision and 0.8 at recall.

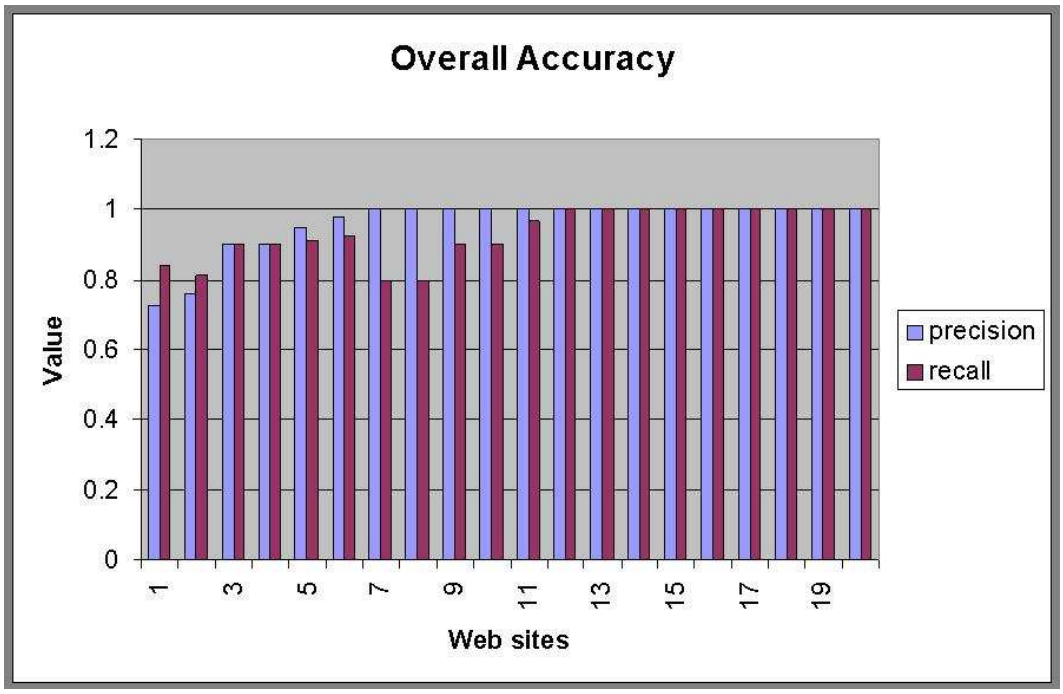

Figure 54: Overall Accuracy

4.7.2.2 Wrapper Execution Performance

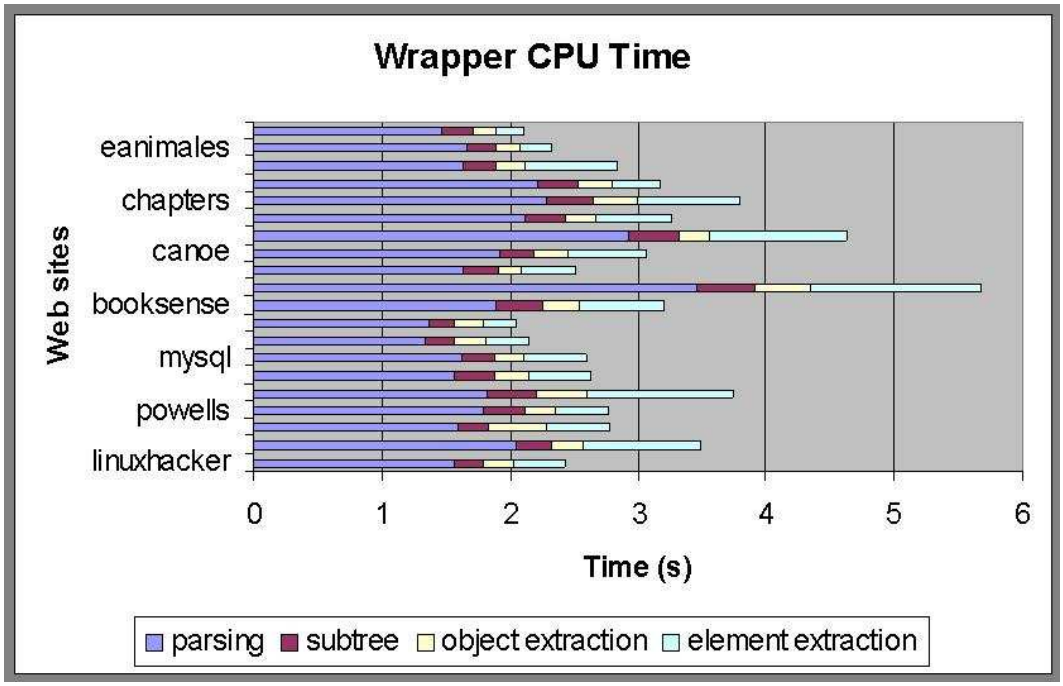

Figure 55: Wrapper CPU Time

The purpose of experiments on wrapper performance is to show the efficiency of generated wrappers and the factors that affect the efficiency. Figure 55 shows wrappers execution ranges from 2 to 5.7 seconds cpu time. In order to understand the variance of wrapper performance, we break down wrapper execution into four stages: parsing, identifying subtree, extracting objects and extracting elements. Element tagging is included in element extraction stage since tagging does not cost much time.

We have observed that parsing is the bottleneck of wrapper execution because it is quite expensive to build an HTML tree. Figure 56 reveals that parsing time is loosely linear to document size. It is not strictly linear because HTML parsing also requires syntax correction, such as resolving interleaving HTML tag pairs. The cost of syntax correction carries a degree of variance due to the randomness of HTML syntax errors in a document.

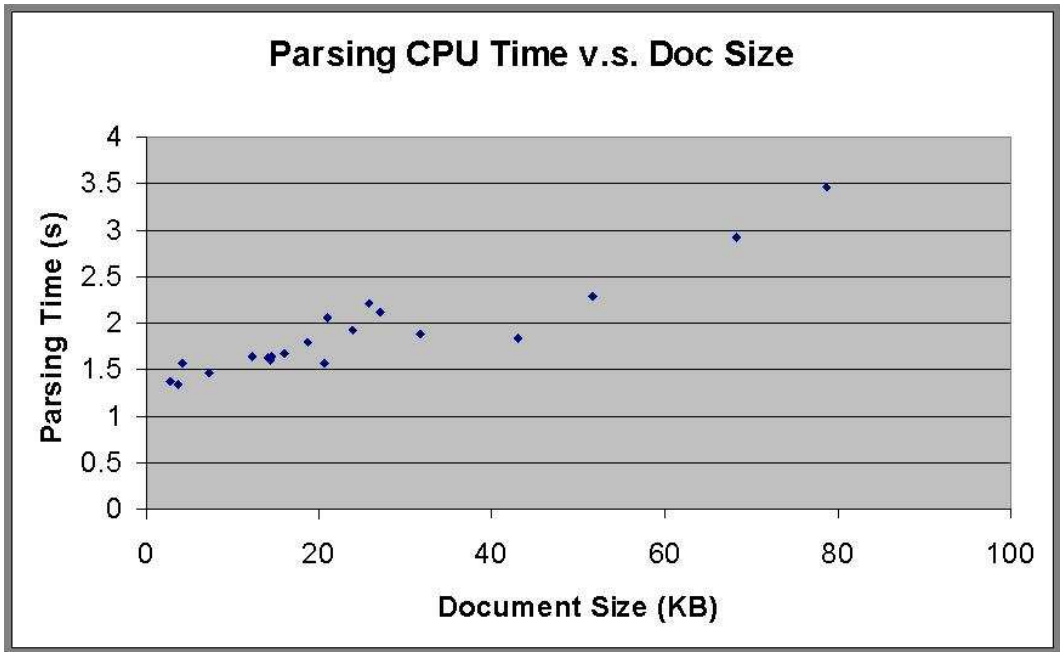

Figure 56: Parsing CPU Time v.s. Doc Size

Figure 57 demonstrates the correlation between subtree extraction time and the number of nodes in a document. Each dot represents the ratio of the CPU time of extracting the subtree vs. the average number of nodes per document for each Web site. The ratio is close to linear because the subtree extraction algorithm goes through each tree node to

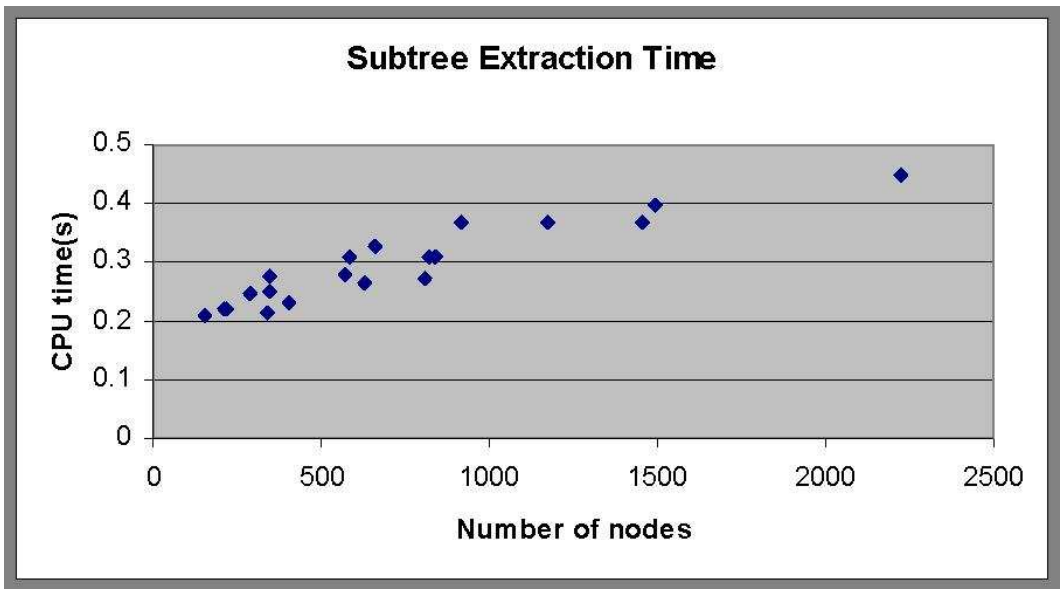

Figure 57: Subtree Extraction Time v.s. Number of Nodes

determine a ranking list. The variance comes from the topology of the tree structure. A more balanced tree yields often less variant cost on each node and produces less collective cost for the whole tree.

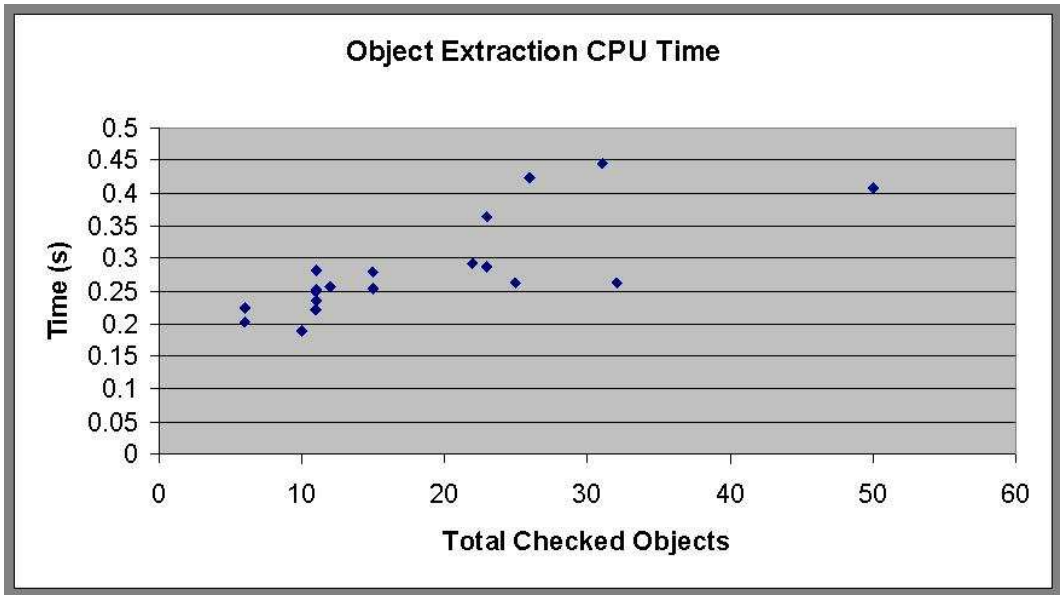

Figure 58: Object Extraction Time v.s. Total Checked Objects

We have also conducted experiments to find out the main performance factor of object

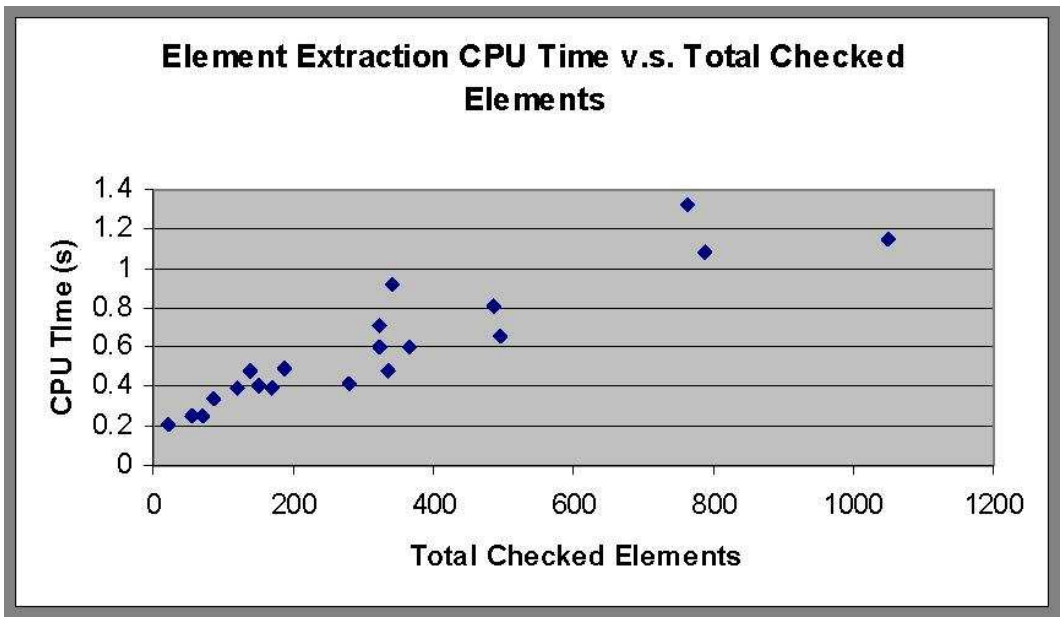

Figure 59: Element Extraction Time v.s. Total Checked Elements

extraction and element extraction. Particularly, we have sought to establish the correlations of object extraction time vs. total checked objects and element extraction time vs. total checked elements. However, Figure 58 and Figure 59 show that such correlations do not hold while total checked objects (or elements) certainly affect the execution time to certain degree. We believe object extraction and element extraction are also affected by other factors simultaneously, such as the number of element separators in element extraction. A more fine-grained break-up is needed to study all factors in the future work.

4.7.3 Comparison With Manual Wrappers

The quality of manually written wrappers often depends on the experience and skills of the wrapper developer. So it is hard to compare XWRAPElite wrappers to manual wrappers. In this section, we use an example-based approach to get a ball-park feeling. Particularly, we show that XWRAPElite offers the advantage of robustness over manual wrappers.

4.7.3.1 XWRAPElite Wrapper v.s. Regular-Expression-Based Manual Wrapper

We build a manual wrapper and an XWRAPElite wrapper for the same Web site eSignal Central. Figure 60 shows a sample page that displays stock information for Inter-tel.

Figure 60: A Web Page at eSignal Central

Based on string regular expression matching, the manual wrapper looks for an string identifier immediately before the stock info table and then extracts each stock attribute by substring matching. The performance comparison in Figure 61 comes to no surprise that XWRAPElite wrapper is easier to generate while it runs slower than the manual wrapper.

	XWRAPElite Wrapper	Manual Wrapper
Generation Time	25 minutes	over 4 hours
Lines of Code	54	139
Precision	1.00	1.00
Recall	1.00	1.00
Elapse Time(s)	2.7	0.076
CPU Time(s)	2.7	0.07

Figure 61: Performance Comparison Between XWRAPElite Wrapper And Manual Wrapper for eSignal Central

Figure 62 shows the XWRAPElite wrapper is more robust to possible changes in data source than the manual wrapper based on string regular expression. Whenever the changes involve text modification in identifiers, the manual wrapper will fail to work because it relies on the identifiers to locate data while the XWRAPElite wrapper can still function well because its extraction is based on the structure of the data source. The XWRAPElite wrapper might fail when the element order changes because its tagging rules are mainly based on element order.

Changes in	XWRAPElite Wrapper	Manual Wrapper
Subtree Location	Adaptable	Adaptable
Subtree Identifier Text	Adaptable	Non-adaptable
Element Identifier Text	Adaptable	Non-adaptable
Number of Elements	Adaptable	Adaptable
Element Order	Possibly Adaptable	Adaptable

Figure 62: Robustness Comparison Between XWRAPElite Wrapper And Manual Wrapper for eSignal Central

4.7.3.2 XWRAPElite Wrapper v.s. Tree-Based Manual Wrapper

Another common way to manually write wrappers is to utilize the HTML tree structure. However, instead of studying the statistics pattern of tree nodes as XWRAPElite does, it locates identifier nodes by tree-path. Figure 63 shows a sample page of Cooking.com, for which we have built a manual wrapper based on tree structure and an XWRAPElite wrapper.

As expected, Figure 64 shows that XWRAPElite saves much time in wrapper generation

Figure 63: A Web Page at Cooking.com

while the generated wrapper achieves similar or comparable performance in both extraction accuracy and wrapper execution. Particularly, a manual wrapper based on tree structure does not run much faster than an XWRAPElite wrapper because parsing HTML into a tree is the common bottleneck.

On the other hand, the XWRAPElite wrapper presents a better adaptability to changes in data source as shown in Figure 65. In general, when the original Web site slightly changes the tree structure of the identifier nodes, such as the location, the manual wrapper tend to fail because the tree path it uses is not valid any more.

	XWRAPElite Wrapper	Manual Wrapper
Generation Time	15 minutes	12 hours
Lines of Code	202	298
Precision	0.99	1.00
Recall	0.99	1.00
Elapse Time(s)	3.2	2.7

Figure 64: Performance Comparison Between XWRAPElite Wrapper And Manual Wrapper for Cooking.com

Changes in	XWRAPElite Wrapper	Manual Wrapper
Subtree Location	Adaptable	Non-adaptable
Subtree Identifier Text	Adaptable	Adaptable
Element Identifier Text	Adaptable	Adaptable
Number of Elements	Adaptable	Possibly adaptable
Element Order	Possibly Adaptable	Non-adaptable

Figure 65: Robustness Comparison Between XWRAPElite Wrapper And Manual Wrapper for Cooking.com

4.8 Conclusion

We have described XWRAP Elite, a tool for automatically generating wrappers for Web information sources. XWRAP Elite analyzes HTML documents a fine-grained object and element level to generate most of the wrapper code automatically. In contrast, many existing approaches require the wrapper developers to write information extraction rules by hand using a domain-specific language or to input their knowledge through an interactive GUI.

The automation of data extraction offers a number of advantages. XWRAP Elite is a scalable solution in the sense that creating a new wrapper costs much less time. The user only needs to input object and element names. It facilitates wrapper maintenance since web page changes are accommodated typically by running XWRAP Elite again. We are currently investigating techniques that support the automated detection and update of wrappers when target web pages change.

CHAPTER V

XWRAP COMPOSER

5.1 Introduction

Automatically extracting data from Web sites has been an important task for the entire life of the Web. Although there have been many advances in automated information exchange, there remains many example applications that can benefit from automated techniques to extract data from a variety of complex Web sites.

We have observed that wrappers generated by wrapper generators, such as XWRAP Original or XWRAP Elite, usually perform well when extracting information from individual documents (single web pages) but are poorly equipped to extract information from multiple linked Web documents. An obvious reason for the inefficiency is due to the lack of system-wide support in the wrapper generators. For example, a wrapper in XWRAP Elite or Original can only perform information extraction over one type of Web document, while typical bioinformatics sources use several types of pages to present their information, including HTML search forms and navigation pages, summary pages for search results, and strictly formatted pages for detailed search results.

Information extraction over multiple different pages imposes new challenges for wrapper generation systems due to the varying correlation of the pages involved. The correlation can be either horizontal when grouping data from homogeneous documents (such as multiple result pages from a single search) or vertical when joining data from heterogeneous but related documents (a series of pages containing information about a specific topic). Furthermore, the correlation can be extended into a graph of work-flows. A multi-page wrapper not only enriches the capability of wrappers to extract information of interests but also increases the sophistication of wrapper code generation.

XWRAP Composer is a semi-automated wrapper generation system that generates wrappers capable of extracting information from multiple heterogeneous Web documents.

By encoding wrapper developers' knowledge in Interface Specification, Outerface Specification and Extration Script, XWRAP Composer integrates single-page wrappers into a composite wrapper capable of extracting information across multiple interconnected pages. The enhancement module allows the generated composite wrapper to be used as a WSDL Web service [73] or a *Ptolemy* actor [8] for future integration.

In the following sections, we will discuss our design and development work on XWRAP Composer using a scientific data integration scenario as the application environment. Describes the actual scenario used by a biology scientist in his research, Section 5.2 explains the role of multi-page data extraction is a desirable and important requirement for scientific applications. Section 5.3 presents the XWRAPComposer design and illustrate how the XWRAPComposer approach addresses the cross-page data collection problems through the bioinformatic application scenario. Finally, we describe the status of the XWRAPComposer system development and discuss the future work in Section 5.4.

5.2 Application Scenario

One excellent example that demands multi-page data extraction is scientific data integration. The Internet and the World Wide Web (Web) have become one of the most popular means for disseminating scientific data from a variety of disciplines. For example, vast and growing amount of life sciences data reside today in specialized Bioinformatics data sources, many of them are accessible online with specialized query processing capabilities. The Molecular Biology Database Collection, for instance, currently holds over 500 data sources, not even including many tools that analyze the information contained therein. Bioinformatics data sources over the Internet have a wide range of query processing capabilities. Typically, many Web-based sources allow only limited types of selection queries. To compound the problem, data from one source often must be combined with data from other sources to give scientists the information they need.

For instance, Figure 66 illustrate a possible task a biologist might performs. He first uses a program called *Clusfavor* to cluster genes that changed significantly in a micro-array analysis experiment. After extracting all gene ids from the Clusfavor result, he feeds them

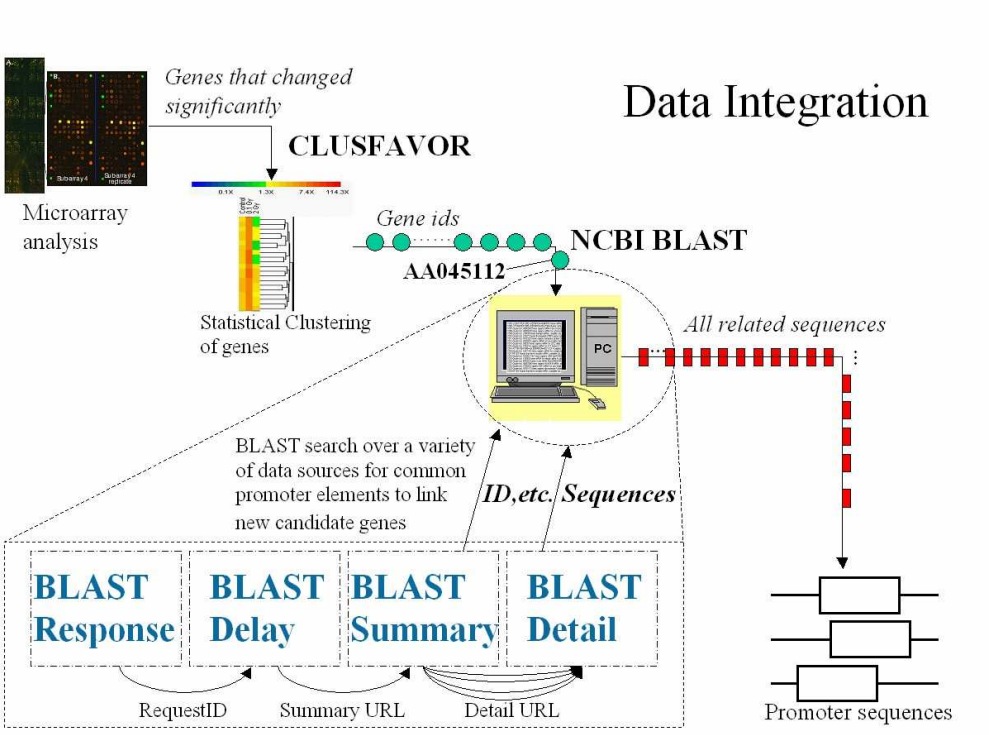

Figure 66: A Scientific Data Integration Example Scenario

into NCBI Blast Web site, which searches all related sequences over a variety of data sources. The returned sequences will be further examined to find promoter sequences. Each of the phases described above can also be decomposed into smaller tasks. For example, the search in NCBI Blast actually involves four steps. Due to the long waiting time for serving a BLAST search on the NCBI website [60], the NCBI Web server will first return a response page with a request ID. When the biologist asks for the results by the request ID, the Web site will presents a delay page if the results are not yet ready to display. Once the search results are delivered, they are displayed in a summary page that contains a summary of all genes matching the search query condition and links to detail pages of each found gene. If the summary page does not include detailed information that the biologist is interested in, he has to visit each detail page.

A critical challenge for providing system-level support for scientists to achieve such data integration tasks is the problem of locating, accessing, and fusing information from a

rapidly growing, heterogeneous, and distributed collection of data sources available on the Web. This is a complex search problem for two reasons. First, as the example in Figure 66 shows, work-flow dependency is often involved. Scientists today have much more complex data collection requirements than ordinary surfers on the Web. They often want to collect a set of data from a sequence of searches over a large selection of heterogeneous data sources, and the data selected from one search step often forms the filter condition for the next search step, turning a keyword-based query into a sophisticated search and information extraction workflow. Second, such complex workflows are manually performed daily by scientists or data collection lab researchers (computer science specialists). Figure 66 only shows one possible workflow example. Automating such complex search and data collection workflows presents three major challenges.

- Different websites use different query-answer control logics to present the answer pages to search queries.
- Cross-page data extraction has more complex extraction logic than the single page extraction system. Different applications require different sets of data to be extracted by the cross-page data extraction engine. Typically, only portions of one page and the links that lead the extraction to the next page need to be extracted.
- Data items extracted from multiple pages require associating with semantically meaningful naming convention, which often provided by the domain scientists who issued such cross-page extraction job.

In the next section we describe the XWRAPComposer approach to address these issues. Examples are provided to illustrate the XWRAPComposer design.

5.3 Solution Approach

We have developed a methodology and a framework for extraction of information from multiple pages connected via web page links. The main idea is to separate what to extract from how to extract, and distinguish information extraction logic from query answer control logic. The control logic describes how many different ways a query could be answered from a

given web site. The data extraction logic describes the cross page extraction steps, including what information is important to extract at each page and how such information is used as a complex filter in the next search/extraction step. We use interface and outerface description to describe what need to be extracted and use Composer Extraction Script language to describe the control logic and extraction logic and to implement the output alignment and tagging of data items extracted based on the outerface specification.

The XWRAPComposer system can be seen as a multi-page data extraction and code generation engine. It takes the following three types of input files and produces the executable code in Java. Our first prototype release of the XWRAPComposer also supports generating WSDL web services or Ptolemy wrapper actors upon request.

- Interface Specification: The schema of the incoming data that the wrapper takes as input. It defines the source location and the query interface for the wrappers to be generated.
- Outerface Specification: The schema of the result that the wrapper outputs. It defines the type and structure of objects extracted.
- Composer Extraction Script: The Composer extraction script code describes how to glue component wrappers. It uses a domain-specific scripting language for specifying the correlation between the fragments of information. It also shows how the result is composed from the incoming data.

Figure 67 presents an architecture sketch of the XWRAP Composer toolkit. XWRAP-Composer compiler takes three inputs, interface specification, outerface specification and extraction script, and compiles them into a Java wrapper, which can be further extended into either a Web service with wrapping capability or a Ptolemy wrapper actor for future integration. The compilation process of the XWRAPComposer includes generating code based on control logic and extraction logic as well as generating the correct output alignment and semantically meaningful tagging based on the outerface specification. In the subsequent sections we will give an overview of the design ideas for interface, outerface and

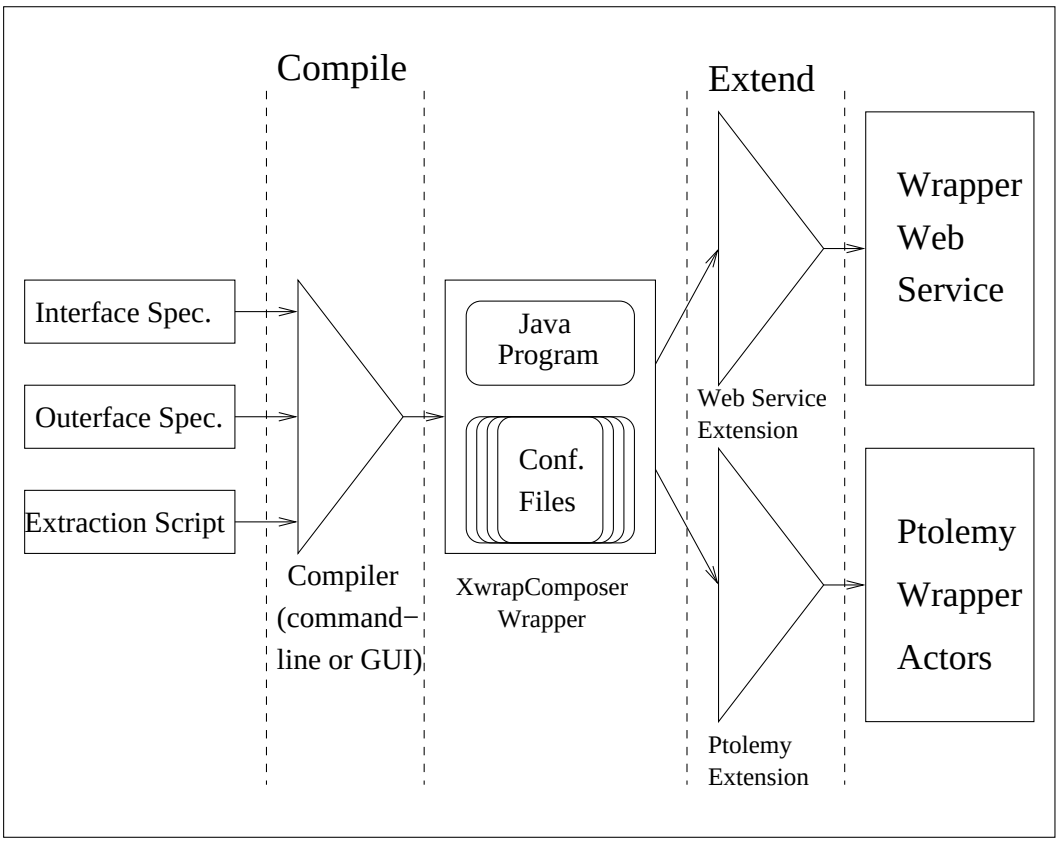

Figure 67: Structure Of XwrapComposer Toolkit

composer script descriptions and illustrate each of them using a concrete example developed for the bioinformatics scenario described in Section 5.2.

5.3.1 Interface and Outerface Specification

The design of the XWRAP Composer Interface and Outerface Specification serves two important objectives. First and for most, it will ease the use of XWRAP wrappers. Second, it will also facilitate the XWRAP Composer wrapper code generator system to generate Java code. Therefore, some components of the specification may not be directly useful for the users of these wrappers. The current XWRAP Composer implementation describes the input and output schema of the composite wrapper in XML Schema as the interface and outerface specification.

Concretely, the interface specification describes the wrapper name and which web site needs to be wrapped by giving the source URL and other related information. The outerface

```
<XCwrapper name="XC_BlastN_Summary" sourceURL=
 "http://www.ncbi.nlm.nih.gov/blast/Blast.cgi?PAGE=Nucleotides">
 <interface><!-- input schema in XML Schema -->
  <xsd:element name="input" type="xsd:string">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="select_db" type="string"/>
     <xsd:element name="query_sequence" type="string"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </interface>
 <outerface><!-- output schema in XML Schema -->
  <xsd:element name="resultDoc">
   <xsd:complexType>
    <xsd:element name="output">
     <xsd:complexType>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
       <xsd:element name="homolog">
        <xsd:complexType>
         <xsd:sequence>
          <xsd:element name="geneid" type="string"/>
          <xsd:element name="description" type="string"/>
          <xsd:element name="length" type="int"/>
          <xsd:element name="score" type="string"/>
          <xsd:element name="expect" type="string"/>
          <xsd:element name="identities" type="string"/>
          <xsd:element name="strand" type="string"/>
          <xsd:element name="link" type="string"/>
          <xsd:element name="beginMatch" type="int"/>
          <xsd:element name="endMatch" type="int"/>
          <xsd:element name="alignment" type="string"/>
          <xsd:element name="beginMatch" type="string"/>
         </xsd:sequence>
        </xsd:complexType>
       </xsd:element>
      </xsd:choice>
     </xsd:complexType>
    </xsd:element>
    <xsd:attribute name="docLocation" type="string"/>
    <xsd:attribute name="docType" type="string"/>
    <xsd:attribute name="createdBy" type="string"/>
    <xsd:attribute name="creationDate" type="string"/>
   </xsd:complexType>
  </xsd:element>
 </outerface>
</XCwrapper>
```

Figure 68: An Example Of Interface And Outerface Specification – NCBi Summary

specification describes what data items are of interest and the semantically meaningful names to be used to tag those data items. Figure 68 shows a concrete example for the XWRAPComposer interface and outerface schema for the NCBI BLAST summary wrapper.

5.3.2 Extraction Script

The XWRAPComposer script usually contains three types of root commands, document retrieval, data extraction and post processing. The document retrieval commands construct a file request or an HTTP request and fetch the document. The data extraction commands extract information from the fetched document. The post processing commands allow adding semantic filters to make the output conform to the outerface specification.

The general usage of commands is as follows:

```
Generate <object id> ::  <command name> (<input id>) {
   Set <property1 name> { <value> } [more value]
   Set <property2 name> { <value> }
   /* if the command is data extraction.  */
   [extraction code]
}
```

Where <object id> is the id of the output object from the command, <input id> is the id of the input object. Both input and output objects are XML nodes. For example, FetchDocument returns the content of a Web page, which is a text node in XML. Each command has a set of built-in properties to specify. <value> can be a string value, enclosed by a pair of quotes, such as *"this is a string value"*, or an XPath expression, enclosed by a pair of brackets, such as *[detailLink/text()]@<xpathroot>*. The value of "xpathroot" should be either <input id> or <object id> generated from previous commands.

If the command is data extraction, such as extractLink and extractContent Extraction code, the detail extraction code needs to be specified as well. The main command type for the extraction code is grab functions. XWRAPComposer also provides miscellaneous commands for control flow, process management and Boolean comparison. In order to output XML data more flexibly, an XSL style sheet may be applied to any XML object using the ApplyStyleSheet command.

Table 17 shows a list of commands that are currently supported in the first release of the XWRAPComposer toolkit [27].

Table 17: Supported XWRAPComposer Extraction Root Commands

Command	Category
ConstructHttpQuery	Document Retrieval
ReadFile	Document Retrieval
FetchDocument	Document Retrieval
ExtractLink	Data Extraction
ExtractContent	Data Extraction
GrabSubstring	Grab Function
GrabXWrapEliteData	Grab Function
GrabConsecutiveLines	Grab Function
GrabCommaDelimitedText	Grab Function
ContainSubstring	Boolean Comparison
While...Do...	Control Flow
If...Then...	Control Flow
ApplyStyleSheet	Post Processing
Sleep	Process Management

5.3.2.1 Commands

(1) Document Retrieval Command

We support the following Document Retrieval Commands in the first release of the XWRAP-Composer toolkit.

ConstructHTTPQuery

This command constructs a HTTP query that contains three components, URL, queryString and HTTP method. It has four properties, URL, queryString, httpMethod and vars. The first three properties are actually templates with placeholders, "$$". The last property is a list of strings to replace the placeholders.

Example:

```
Generate genbankSummaryPage ::  ConstructHttpQuery {
  Set inputSource {
    Set url { "http://www.amazon.com/book-search.cgi" }
    Set queryString { "keyword=$$&author=$$&start=1" }
    Set httpMethod { "post" }
    Set vars {
    /* the sub property names are only for reading.  */
    /* We replace the placeholders by the order of the vars.  */
       Set keyword { "java" }
       Set author { "john" }
       }
    }
  }
}
```

The result will be

```
<inputSource>
   <URL>http://www.amazon.com/book-search.cgi</URL>
   <queryString>keyword=java&author=john&start=1</queryString>
   <httpMethod>get</httpMethod>
<inputSource>
```

ConstructFileQuery

This command constructs a file request, which contains only one property, file name.

FetchDocument

This command takes a file request or an HTTP request as input and returns the content of the file or the Web page. It does not have any properties.

(2) Data Extraction

Currently we support two types of data extraction commands. They are used for extracting links or extracting content.

ExtractLink

This command indicates to extract an HTTP request from the input. It usually needs to extract URL, queryString and httpMethod. If queryString and httpMethod are not extracted, the default values will be used. The default httpMethod is "get" and queryString

is "". The return object of ExtractLink can be the input of FetchDocument.

ExtractContent

This command indicates to extract content from the input, which contains all kinds of data. The extraction method needs to be specified with GrabFunctions and XML construction commands as well as other commands.

(3) Grab Function

The Grab function is designed to facilitate the text parsing and string analysis during the extraction process. We support the following five grab functions.

GrabSubstring

Assuming the input is a text node, this commands extracts a substring by two properties, beginMatch and endMatch. It will return the string between beginMatch and endMatch. If there are multiple result strings, we only choose the first one. **GrabXWrapEliteData** This command applies an XWRAPElite wrapper to the input. The input should be a text node that represents the content of a Web page. The properties are generated by XWRAPElite toolkit. It allows modification for fine-tuning.

GrabConsecutiveLines

This command is to extract consecutive text lines from the input. It has three properties, beginMatch, endMatch and matchingMethod. The default matching method is to match the first string of the beginning line and the ending line with beginMatch and endMatch properties. However, we might need some domain specific matching method in some cases. Then we can use external functions as shown in NCBiDetail example.

GrabCommaDelimitedText

This command extracts comma-delimited data into XML. It has the following properties.

- *LineDelimiters:* The delimiters to separate data into rows. The default is the system line separators.
- *Delimiters:* the delimiters to separate data in a row to a list of cells. The default is comma.
- *StopStrings:* String that will be ignored.

- *Filters:* It contains two sub properties, minColCount and maxColCount. We will filter out all the rows whose column numbers are not in the range of [`minColCount`, `maxColCount`] inclusively.

- *RowOutput:* It specifies how to output the tabular data for each row in XML.

(4) Boolean comparisons

ContainSubstring This command returns a Boolean value, which indicates if the input contains a substring. It has one property, compSubstring.

Example:

```
ContainSubstring (answerPage) {
   Set compSubstring { "This page will be automatically updated in"}
}
```

The example demonstrates a command that checks if the text value of answerPage contains a string of "This page will be automatically updated in".

(5) Control Flow

While Do This command checks the conditions in the while clause, while the conditions are true, repeat the do clause. The while clause contains Boolean commands and the do clause contains other extraction-related commands.

(6) Post Processing

ApplyStylesheet This command applies a style sheet to the input XML. It has a property, StyleSheetFile.

(7) Process Management

Sleep This command pauses the process for certain time.

Usage: Sleep "<number of milliseconds>"

5.3.2.2 Example Extraction Scripts

Figure 69 shows an extraction script example for NCBi Summary. Given a sequence from the input, we first contruct a NCBi Blast search URL. Script *Set variable* { *[text()]* indicates the sequence is in the input with the XPath, "text()". The first *FetchDocument* retrieves the NCBi Blast response page that contains a request ID. We extract the ID and construct

the URL of search results. The control-flow command *while...do...* periodically invokes the second *FetchDocument* to retrieve the result page until the results are delivered. Finally we use *GrabXWRAPEliteData* to extract useful data from the result page.

5.3.3 Code Generation And Extensions

As Figure 67 shows, XWRAPComposer compiles interface, outerface and extraction script into an XWRAPComposer wrapper, which contains an executable Java program and a set of configuration files. The configuration files include the input and output schema obtained from interface and outerface, resource files used in the data extraction phase such as XSLT files. Furthermore, because Java programs are usually not convenient for application integration, wee have provided two extensions to XWRAPComposer wrappers.

- **Web-service Enabled Wrappers**

  Web services [72] have become a growing trend to access remote applications in data and application integration. In order to make the XWRAPComposer to generate WSDL-enabled wrapper services, one of the extensions we have made to the toolkit is to enable Composer generated wrappers to act as Web services. We carry out this task in two stages. First, we encapsulate an XWRAPComposer wrapper into a general Web service servlet. The servlet automatically extracts the input from a SOAP request, feeds it into the wrapper, and inserts the wrapping results in a SOAP envelope before sending back to the user. Second, WSDL generator can automatically generate Web service description by binding the wrapper's interface and outerface with the servlet configuration. Figure 70 shows the extensions added to the XWRAPComposer to produce wrappers as WSDL web services.

- **Ptolemy Wrapper Actors**

  Ptolemy [8] is a modeling tool to assemble concurrent components, providing a friendly GUI environment to connect task components and govern the interaction between them. The task components have to conform Ptolemy's actor interface in order to be integrated in Ptolemy. We have implemented such an actor interface for generated

```
/* Start constructing wrapper ncbisummary.  */
WrapperName "ncbisummary";

/* Contruct the URL for NCBi Blast search */
Generate blastSummaryPage ::  ConstructHttpQuery (input){
 Set inputSource {
  Set url {"http://www.ncbi.nlm.nih.gov/blast/Blast.cgi?QUERY=$$&..."};
  Set queryString { };
  Set method {"get"};
  Set variable { [text()] } ;
 }
}
Generate blastSummaryData ::  FetchDocument (blastSummaryPage) {}
Generate recordid ::  ExtractContent (blastSummaryData) {
 GrabSubstring {
  Set BeginMatch {"The request ID is <input name=\"RID\" size=\"50\"
type=\"text\" value=\"};
  Set EndMatch {"\" >" };
 }
}
Generate answerurl ::  ConstructHttpQuery (recordid){
 Set inputSource {
  Set url {"http://www.ncbi.nlm.nih.gov/blast/Blast.cgi?FORMAT_PAGE
_TARGET=Format_page_31680&RESULTS_PAGE_TARGET=Blast_Results_for_31680
&RID=$$&SHOW_OVERVIEW=on...&AUTO_FORMAT=Semiauto"};
  Set queryString { };
  Set method { "get" };
  /* The first recordid is the input id.*/
  Set variable { [text()] };
 }
}
Generate answerPage ::  FetchDocument(answerurl) {}
While {
 ContainSubstring(answerPage) {
  Set compSubstring {"This page will be automatically updated in"};
 }
} Do {
 Generate answerPage ::  FetchDocument(answerurl) {}
 /* Pause for 10 seconds.  */
 Sleep {
  set inverval {"10000"};
 }
}
Generate output ::  ExtractContent (answerPage) {
 GrabXWRAPEliteData {
 /* The following properties should be generated from a XWRAPELite tool.  */
 ...
 }
}
```

Figure 69: Extraction Script Example For NCBi Summary

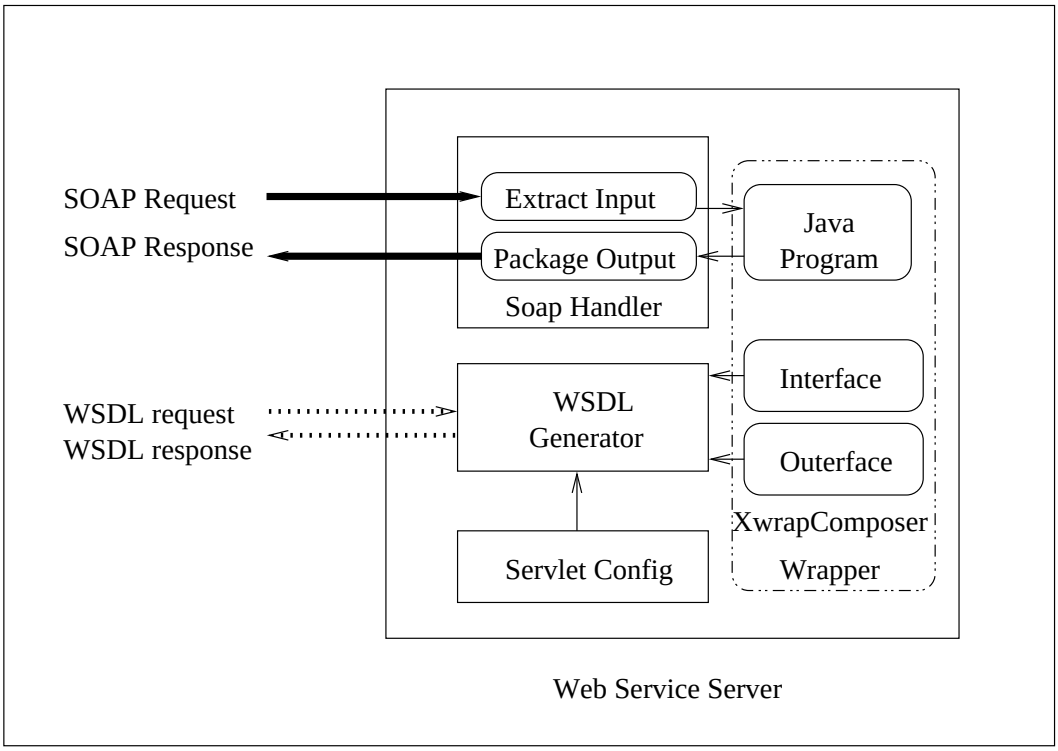

Figure 70: Web-service Enabled Wrappers

XWRAPComposer wrappers to be used in Ptolemy.

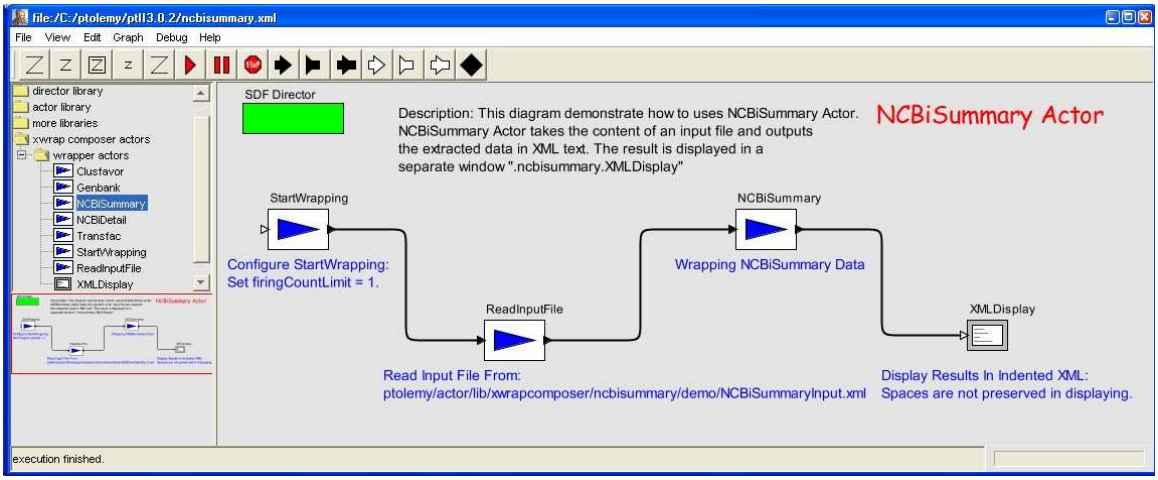

Figure 71: Ptolemy Wrapper Actor Example – NCBi Blast Summary

Figure 71 and Figure 72 demonstrate two ptolemy diagrams that are composed by XWRAP Composer wrapper actors from the left frame. After clicking on the play

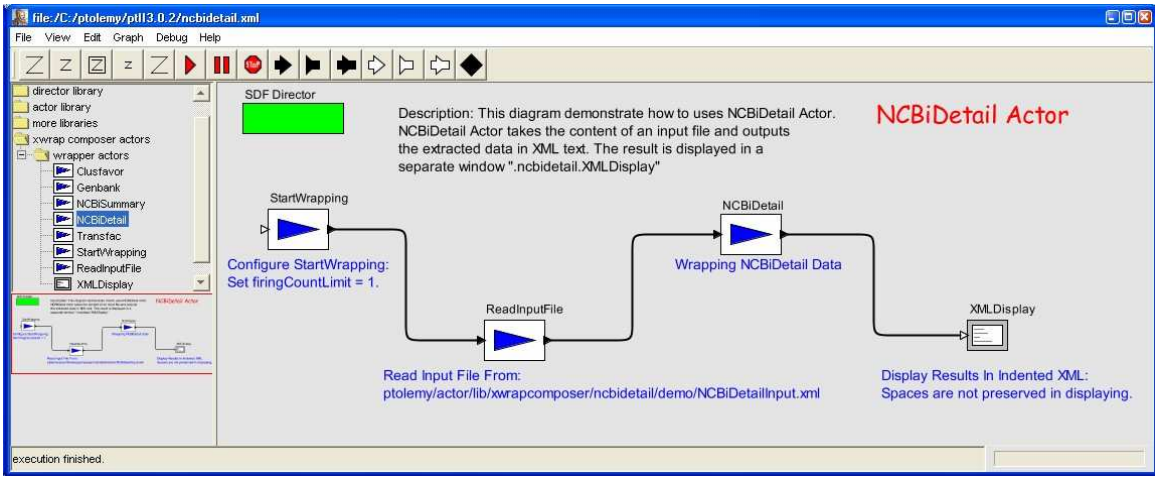

Figure 72: Ptolemy Wrapper Actor Example – NCBi Blast Detail

button on the top, **StartWrapping** triggers **ReadInputFile** to read a gene id from a specified input file. The gene id will then be sent to **NCBiSummary** Wrapper Actor and the wrapper actor performs the wrapping function upon the gene id input and returns ids of related genes as results. Shown as Figure 73 and Figure 74, **XMLDisplay** will pop up a window to present the results.

5.4 Conclusion and Future Work

The XWRAPComposer development advances the state of art research in information extraction and wrapper generation along two dimensions: (1) XWRAP Composer is the first wrapper generation system that is capable of generating multi-page wrappers. (2) XWRAP Composer is the only wrapper generation system that promotes the distinction between information extraction logic from the query-answer control logic, which makes the wrapper generation system much more robust against changes in the provider's web site design or infrastructure.

The future work of XWRAPComposer lies in three dimensions.

- Improving the XWRAPComposer GUI in Ptolemy that allows an XWRAPComposer wrapper can be generated in Ptolemy by composing a few task actors.

- Extending the Composer code generator's capability to allow wrappers to be generated

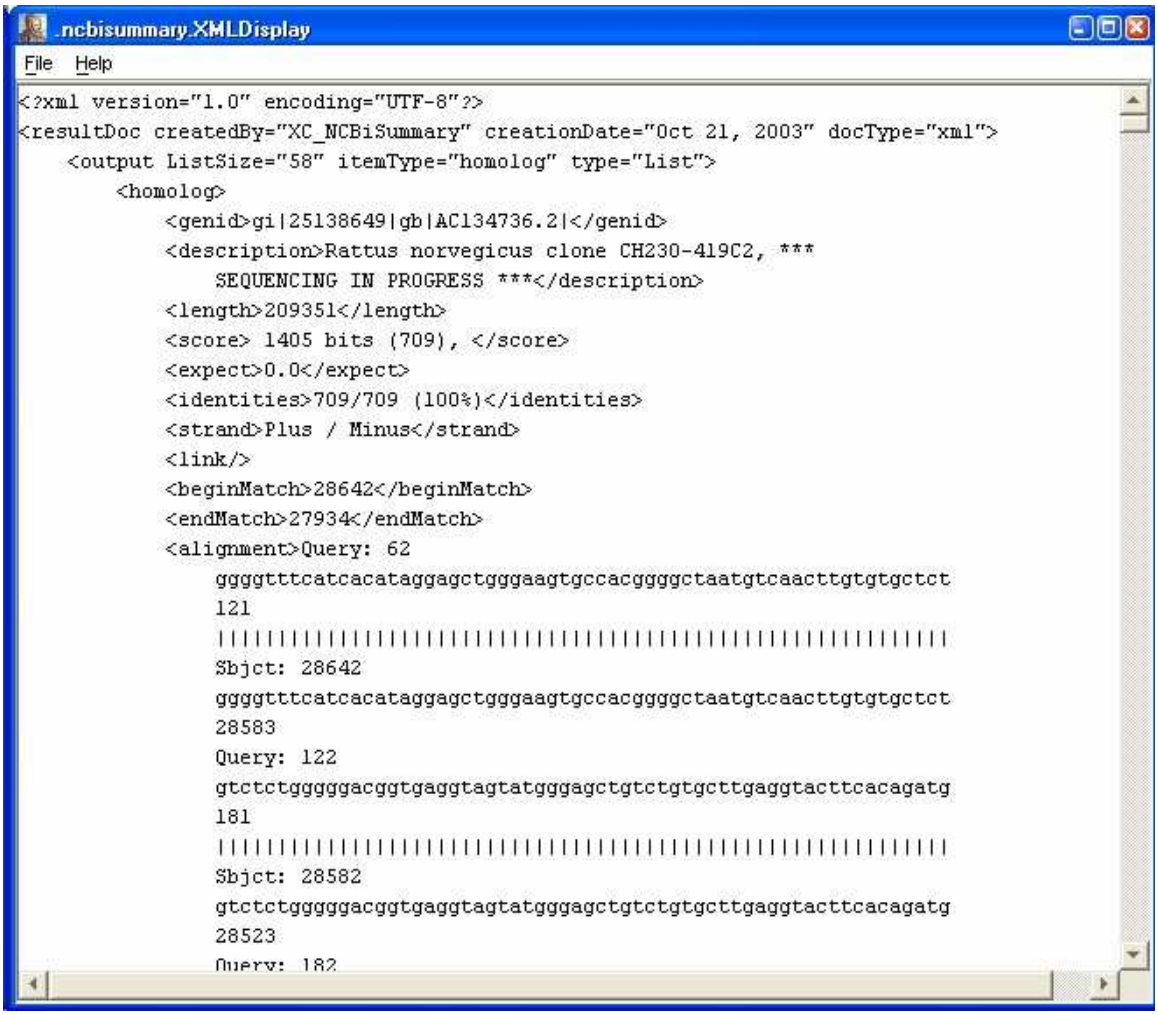

Figure 73: Ptolemy Wrapper Actor Result Example – NCBi Blast Summary

for more variety of scientific data sources. Currently we only provide capability for five examples.

- Exploring data provenance techniques for scientific data integration. We are interested in both the use of data provenance information to facilitate the scientific data integration process, improving scientific data integration quality as well as extracting data provenance information from the vast amount of data contents provided in many scientific data sources.

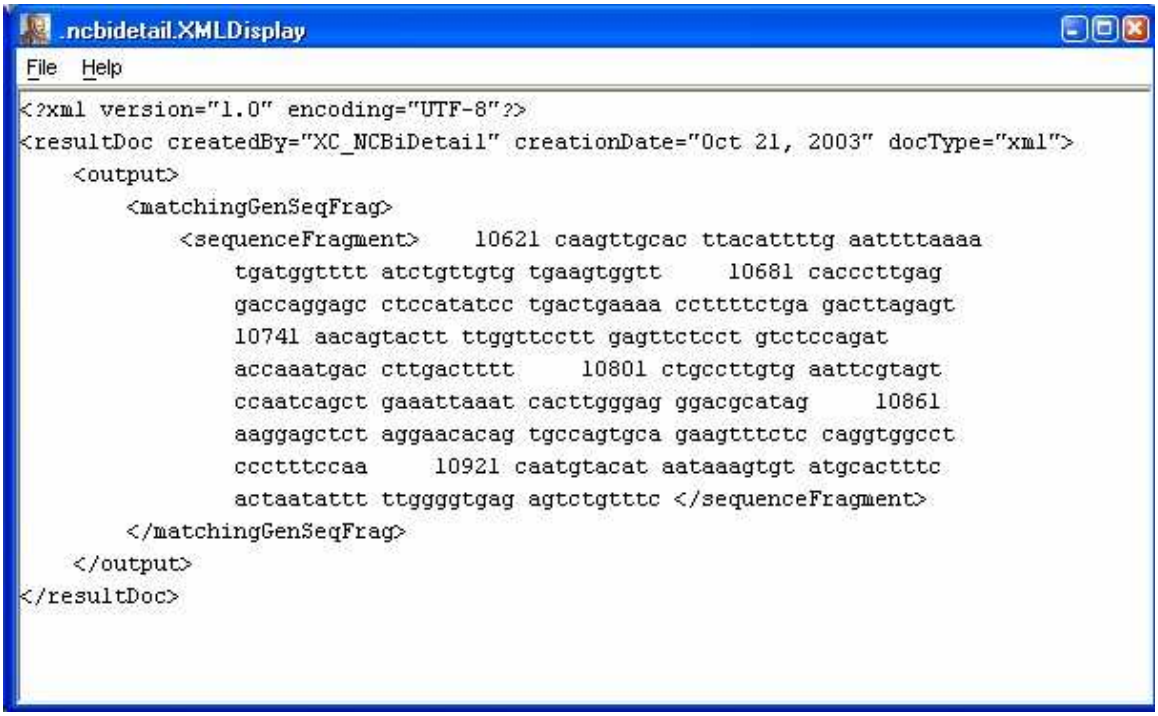

Figure 74: Ptolemy Wrapper Actor Result Example – NCBi Blast Detail

CHAPTER VI

CONCLUSION

With the fast growth of the Web, retrieving information from different Web sources for future integration becomes increasingly beneficial. However, the lack of program interfaces to accessing the Web information often poses difficulty for integration. A common approach to accomplish this task is to build a wrapper for each Web source and then query that source through the wrapper. Research have been studying wrappers from several perspectives, including wrapper generation, wrapper maintenance, and data integration as well as wrapper repositories.

In this dissertation, I have presented XWRAP framework for designing intelligent wrapper generators. The framework classifies wrappers into two categories, data wrappers and functional wrappers. Wrapper developers build data wrappers using a declarative rule-based language. Inductive learning algorithms and GUI interface help the developers to derive the rules. Functional wrappers are built on top of data wrappers. Functional processing components operate on the XML output of the data wrappers. Different functional wrappers can share a data wrapper to improve the performance.

The contributions of the XWRAP framework presented are the following:

- A clean separation of tasks of building wrappers that are specific to a Web source from the tasks that are repetitive for any source.
- XWRAP Original's declarative rule-based language to specify data wrappers, and its inductive learning algorithms and GUI interfaces to facilitate wrappers developers to derive wrapper patterns.
- XWRAP Elite's near-automation in data extraction for Web information sources. Analyzing HTML documents at fine-grained object and element level to generate most of the wrapper code automatically, XWRAP Elite minimizes the human involvement

in wrapper generation.

XWRAP technology has been used and tested by over 500 registered users as of January 2003 and has generated more than 2000 wrappers. Compared to manual approach and other existing semi-automatic wrapper generators, XWRAP technology provides a number of competitive advantages: First, XWRAP Elite can generate wrappers in minutes with code quality and efficiency equivalent to Human experts. Second, XWRAP systems support a variety of transformations of Web data, including HTML to XML, XML to XML, plain text to XML, and a set of information filters. Third, XWRAP Composer technology is the only wrapper generator to date that is capable of extracting, aggregating, and filtering information from multiple Web pages with workflow dependency. In comparison, most existing wrapper generation tools are limited to wrapping information from individual Web documents, one at a time. This dissertation also reports the extensive experiments conducted on XWRAP systems, showing the efficiency, trade-offs, and code quality of the XWRAP wrapper applications.

REFERENCES

[1] ADELBERG, B., "Nodose: A tool for semi-automatically extracting structured and semi-structured data from text document," *In Proceeding of ACM SlGMOD Conference*, 1998.

[2] ALBERTO H. F. LAENDER, B. R.-N. and DA SILVA., A. S., "Debye - data extraction by example."

[3] ASHISH, N. and KNOBLOCK, C. A., "Semi-automatic wrapper generation for internet information sources," in *Proceedings of Coopis Conference*, 1997.

[4] ASHISH, N. and KNOBLOCK, C. A., "Wrapper generation for semi-structured internet sources," *SIGMOD Record*, vol. 26(4), pp. 8–15, December 1997.

[5] ATZENI, P. and MECCA, G., "Cut and paste," *Proceedings of 16th ACM SIGMOD Symposion on Principles of Database Systems*, 1997.

[6] ATZENI, P., MECCA, G., and MERIALDO, P., "Semi-structured and structured data in the web: going back and forth," *Proceedings of ACM SIGMOD Workshop on Management of Semi-structured Data*, pp. 1–9, 1997.

[7] BAYARDO, R., BIHRER, W., BRICE, R., CICHOCKI, A., FOWLER, G., HELAL, A., KASHYAP, V., KSIEZYK, T., MARTIN, G., NODINEA, M., RASHID, M., RUSINKIEWICZ, M., SHEA, R., UNNIKRISHNAN, C., UNRUB, A., and WOELK, D., "Semantic integrationof information in open and dynamic environments," *Technical Report MCC-INSL-088-96, MCC Austin, Texas*, 1996.

[8] BERKELEY, "Ptolemy group in eecs," *http://ptolemy.eecs.berkeley.edu/*, 2003.

[9] BUTTLER, D., LIU, L., and PU, C., "A fully automated object extraction system for the world wide web," in *Proceedings of the 2001 International Conference on Distrubuted Computing Systems (ICDCS'01)*, (Phoenix, Arizona), pp. 361–370, May 2001.

[10] BUTTLER, D., LIU, L., and PU, C., "Omini: An Object Mining and Extraction System for the Web," *Technical Report, Sept. 2000. Georgia Tech, College of Computing*, 2000.

[11] CALIFF, M. E. and MOONEY, R. J., "Relational learning of pattern-match rules for information extraction," in *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, (Menlo Park, CA), pp. 6–11, AAAI Press, 1998.

[12] CAREY, M., HAAS, L., SCHWARZ, P., and ET AL., "Towards heterogeneous multimedia information systems: the garlic approach," in *IEEE Int. Workshop on Research Issues in Data Engineering*, 1995.

[13] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom., J., "The tsimmis project: integration of heterogeneous information sources," *10th Meeting of the Information Processing Society of Japan*, pp. 7–18, 1994.

[14] Childloveskii, B., "Automatic repairing of web wrappers," *3rd International Workshop on Web Information and Data Management*, pp. 24–30, 2001.

[15] Cohen, W., "A web-based information system that reasons with structured collections of text," *Proceedings of Autonomous Agents AA-98*, pp. 400–407, 1998.

[16] Crescenzi, V., Mecca, G., and Merialdo, P., "Roadrunner: Towards automatic data extraction from large web sites," *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy. pp109-118*, 2001.

[17] Czarnecki, K. and Eisenecker, U. W., *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D., "XML-QL: A Query Language for XML," *http://www.w3c.org/TR/1998/NOTE-xml-ql-19980819*, 1998.

[19] Embley, D. W., Jiang, Y., and Ng., Y.-K., "Record-boundary discovery in web-documents," in *Proceedings of the 1999 ACM SIGMOD*, (Philadelphia, Pennsylvania, USA), June 1999.

[20] Finin, T., Fritzson, R., McKay, D., and McEntire, R., "KQML as an Agent Communication Language," in *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)* (Adam, N., Bhargava, B., and Yesha, Y., eds.), (Gaithersburg, MD, USA), pp. 456–463, ACM Press, 1994.

[21] Freitag, D., "Machine learning for information extraction in informal domain," *PhD Thesis, Dept. of Computer Science, Carnegie Mellon University*, 1998.

[22] Garcia-Molina, H. and et al., "The TSIMMIS approach to mediation: data models and languages (extended abstract)," in *NGITS*, 1995.

[23] Goldman, R. and Widom, J., "Dataguides: Enabling query formulation and optimization in semistructured databases," in *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases* (Jarke, M., Carey, M. J., Dittrich, K. R., Lochovsky, F. H., Loucopoulos, P., and Jeusfeld, M. A., eds.), pp. 436–445, Morgan Kaufmann, 1997.

[24] Haas, L., Kossmann, D., Wimmers, E., and Yan, J., "Optimizing queries across diverse data sources," in *The International Conference on Very Large Data Bases*, 1997.

[25] Hammer, J., Brennig, M., Garcia-Molina, H., Nesterov, S., Vassalos, V., and Yerneni, R., "Template-based wrappers in the tsimmis system," in *Proceedings of ACM SIGMOD Conference*, 1997.

[26] Hammer, J., Garcia-Molina, H., Cho, J., Aranha, R., and Crespo, A., "Extracting semi-structured data from the web," *Proceedings of Workshop on Management of Semi-structured Data*, pp. 18–25, 1997.

[27] Han, W., "Xwrapcomposer," *http://www.cc.gatech.edu/projects/disl/XWRAPComposer/*, 2003.

[28] Han, W., Buttler, D., and Pu, C., "Wrapping web data into xml," *SIGMOD Record*, 2001.

[29] Higgins, J. J. and Keller-McNulty, S., *Concepts in Probability and Stochastic Modeling*. Duxbury, 1995.

[30] Hsu, C. and Dung, M., "Wrapping semistructured web pages with finite-state transducers," *To appear in the Proceedings of the Conference on Autonomous Learning and Discovery CONALD-98*, 1998.

[31] Huck, G., Fankhauser, P., Aberer, K., and Neuhold, E. J., "Jedi: Exchanging and synthesizing information from the web," *Coopis*, 1998.

[32] J. Hammer, M. Brennig, H. G.-M. S. N. V. V. and Yemeni, R., "Template-based wrappers in the tsimmis system," *In Proceedings of ACM SIGMOD Conference*, 1997.

[33] Knoblock, C., Minton, S., Ambite, J., Ashish, N., Margulis, J., Modi, J., Muslea, I., Philpot, A., and Tejada, S., "Modeling web sources for information integration," *In Proceeding of AAAI 1998.*, 1998.

[34] Knoblock, C. A., Minton, S., Ambite, J. L., Ashish, N., Modi, P. J., Muslea, I., Philpot, A., and Tejada, S., "Modeling web sources for information integration," in *Proceedings of AAAI Conference*, 1998.

[35] Kristina Lerman, S. M. and Knoblock, C., "Wrapper maintenance: A machine learning approach," *Journal of Artificial Intelligence Research*, vol. 18, pp. 149–181, 2003.

[36] Kushmerick, N., "Wrapper induction for information extraction," *PhD Thesis, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04*, 1997.

[37] Kushmerick, N., "Wrapper verification," *World Wide Web Journal*, pp. 3(2):79–94, 2000.

[38] Kushmerick, N., Weil, D., and Doorenbos, R., "Wrapper induction for information extraction," in *Proceedings of Int. Joint Conference on Artifical Intelligence (IJCAI)*, 1997.

[39] Kushmerick, N., Weld, D., and Doorenbos, R., "Wrapper induction for information extraction," *Proceedings of 15th International Concerence on Artificial Intelligence*, pp. 729–735, 1997.

[40] Kushrnerick, N., "Wrapper induction for information extraction," *Ph.D dissertation*, 1997.

[41] L. Liu, C. Pu, W. H.-D. B. and Tang, W., "Building an extensible wrapper repository system – a metadata approach," *MetaData conference*, 1999.

[42] L. Liu, C. Pu, W. H.-D. B. and Tang, W., "A xml-based wrapper generator toolkit for internet information sources," *ACM SIGMOD conference*, 1999.

[43] Lerman, K. and Minton, S., "Learning the common structure of data," *Proceedings of AAAI*, 2000.

[44] Li, C., Yerneni, R., Vassalos, V., Garcia-Molina, H., Papakonstantinou, Y., Ullman, J., and Valiveti, M., "Capability based mediation in tsimiss," in *Proceedings of ACM SIGMOD Conference*, 1997.

[45] Liu, L., Han, W., Buttler, D., Pu, C., and Tang, W., "An XML-based Wrapper Generator for Web Information Extraction," in *ACM SIGMOD International Conference*, June 1-4, Philadelphia 1998.

[46] Liu, L., Han, W., and Pu, C., "Building An Extensible Wrapper Repository System: A Metadata Approach," in *Proceedings of the 3rd IEEE Metadata Conference*, Bethesda, Maryland, April 6-7 1999.

[47] Liu, L., Han, W., and Pu, C., "An xml-enabled data extraction toolkit for web sources," *Information Systems Journal*, 2001.

[48] Liu, L. and Pu, C., "An adaptive object-oriented approach to integration and access of heterogeneous information sources," *DISTRIBUTED AND PARALLEL DATABASES: An International Journal*, vol. 5, no. 2, 1997.

[49] Liu, L., Pu, C., and Han, W., "XWrap: An Extensible Wrapper Construction System for Internet Information Sources," in *Technical Report, OGI/CSE, Feb.*, 1999.

[50] Liu, L., Pu, C., and Han, W., "XWrap: An XML-enabled Wrapper Construction System for Web Information Sources," *Proceedings of the International Conference on Data Engineering*, 2000.

[51] Liu, L., Pu, C., and Lee, Y., "An adaptive approach to query mediation across heterogeneous databases," in *Proceedings of the International Conference on Coopertive Information Systems*, (Brussels), June 19-21 1996.

[52] Liu, L., Pu, C., and Richine, K., "Distributed query scheduling service: An architecture and its implementation," *Special issue on Compound Information Services, International Journal of Cooperative Information Systems (IJCIS)*, vol. 7, no. 3, 1998.

[53] Liu, L., Pu, C., and Tang, W., "Continual queries for internet-scale event-driven information delivery," *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[54] Liu, L., Pu, C., Tang, W., Biggs, J., Buttler, D., Han, W., Benninghoff, P., and Fenghua, "CQ: A Personalized Update Monitoring Toolkit," in *Proceedings of ACM SIGMOD Conference*, 1998.

[55] Liu, L., Pu, C., Tang, W., and Han, W., "Conquer: A Continual Query System for Update Monitoring in the WWW," *International Journal of Computer Systems, Science and Engineering*, 1999. Special Issue on WWW Semantics, edited by Dan Suciu and Letizia Tanca.

[56] MILO, T. and ZOHAR, S., "Using schema matching to simplify heterogeneous data translation," *In Proceedings of 24th International Conference on Very Large Data Bases*, pp. 122–133, 1998.

[57] MURATA, M. and ROBIE, J., "Observations on structured document query languages," 1998.

[58] MUSLEA, I., MINTON, S., and KNOBLOCK, C., "Stalker: Learning extraction rules for semistructured, web-based information sources," *AAAI-98 Workshop on AI and Information Integration*, pp. 74–81, 1998.

[59] MUSLEA, I., MINTON, S., and KNOBLOCK, C., "Wrapper induction for semistructured, web-based information sources," *In the Proceedings of the Conference on Autonomous Learning and Discovery CONALD-98*, 1998.

[60] NCBI, "National center for biotechnology information – blast databases," *http://www.ncbi.nlm.nih.gov/BLAST/*, 2003.

[61] P. ATZENI, G. M. and MERIALDO, P., "To weave the web," *In Proceedings of the 2nd VLDB conference*, 1997.

[62] R. AHMED, E. A., "The pegasus heterogeneous multidatabase system," *IEEE Computer*, vol. 24(12), pp. 19–27, 1991.

[63] RAGGETT, D., "Clean Up Your Web Pages with HTML TIDY," *http://www.w3.org/People/Raggett/tidy/*, 1999.

[64] RISE, "Rise: A repository of online information sources used in information extraction tasks," *[http://www.isi.edu/ muslea/RISE/index.html] Information Sciences Institute / USC*, 1998.

[65] ROBERT B. DOORENBOS, OREN ETZIONI, D. S. W., "A scalable comparison-shopping agent for the world-wide web," *Agents*, pp. 39–48, 1997.

[66] ROBERT BAUMGARTNER, SERGIO FLESCA, G. G., "Visual web information extraction with lixto," *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy. pp119-128*, 2001.

[67] ROTH, M. T. and SCHWARZ, P., "Don't scrap it, wrap it! a wrapper architecture for legacy data sources," *In Proceedings of the 2 VLDB Conference*, 1997.

[68] SAHUGUET, A. and AZAVANT, F., "WysiWyg Web Wrapper Factory (W4F)," *Proceedings of WWW Conference*, 1999.

[69] SODERLAND, S., "Learning to extract text-based information from the world wide web," *Proceedings of Knowledge Discovery and Data Mining*, 1997.

[70] "Extensible markup language (XML) 1.0," tech. rep., W3C, 1998.

[71] W3C, "Reformulating HTML in XML," *http://www.w3.org/TR/WD-html-in-xml/*, 1999.

[72] W3C, "Web services," *http://www.w3c.org/2002/ws/*, 2002.

[73] W3C, "Web services description language (wsdl) version 1.2 part 1: Core language," *http://www.w3c.org/TR/wsdl12/*, 2003.

[74] WIEDERHOLD, G., "Mediators in the architecture of future information systems," *IEEE Computer*, March 1992.

[75] YANNIS PAPAKONSTANTINOU, ASHISH GUPTA, H. G.-M. and ULLMAN, I., "A query translation scheme for rapid implementation of wrappers," *In International Conference on Deductive and Object-Oriented Databases*, 1995.

VITA

Wei Han was born and grew up in Hangzhou, P. R. China, which is famous for sceneries and beautiful women. After high school, he went to Tsinghua University in Beijing, where he received Bachelor of Science in Automation. To pursue a different culture experience, he came to the US for graduate study. After obtained Masters of Science in Computer Engineering and Science at Oregon Graduate Institute, he moved to Georgia Institute of Technology in 1999.