10-23-2000

Dynamic data retrieval on the world wide web

Dmitriy Beryoza
Florida International University

DOI: 10.25148/etd.FI14051129

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

DYNAMIC DATA RETRIEVAL ON THE WORLD WIDE WEB

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Dmitriy Beryoza

2000

To: Dean Arthur W. Herriott
College of Arts and Sciences

This dissertation, written by Dmitriy Beryoza, and entitled Dynamic Data Retrieval on the World Wide Web, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Nagarajan Prabakar

Subbarao Wunnava

Raimund Ege

Maxim Chekmasov

Naphtali Rishe, Major Professor

Date of Defense: October 23, 2000

The dissertation of Dmitriy Beryoza is approved.

Dean Arthur W. Herriott
College of Arts and Sciences

Interim Dean Samuel S. Shapiro
Division of Graduate Studies

Florida International University, 2000

DEDICATION

I dedicate this thesis to my wife, Yulia; my parents, Tatiana and Alexander; and my brother, Sergei. Without their love, patience, support and constant encouragement the completion of this work would not have been possible.

ACKNOWLEDGMENTS

I wish to thank members of my committee, and especially my major professor, Dr. Naphtali Rishe, Dr. Maxim Chekmasov and Dr. Nagarajan Prabhakar, for their support, guidance, valuable advice, and patience. Dr. Maxim Chekmasov, Alexander Simanov, Yulia Pichugina, and Andrei Selivonenko provided valuable comments on the first version of this document. Special thanks go to all my colleagues at HPDRC who made this research project possible—Marina Chekmasova, Anna Mullary, Vladimir Mullary, Alexander Simanov, Andrei Kirienko, Andrei Selivonenko, Oksana Dyganova, Dmitry Tsyboulsky, Michael Baranovsky, Kiran Balakrishna, Rukshan Athauda and Eugene Kalenkovich. Additionally I would like to thank all students and research staff who helped us implement parts of the functionality—Juan Carlos Carrillo, Vishal Maru, Yulia Pichugina, Ashok Madala, Sunil Godavarthi, Alex Roque, Celestino Pena, Oksana Petrova, Mikhail Petrov, Marina Klimchuk, Rob Valenti, Ray Morejon, Pusheng Zhang, and Philip Bayer. I would also like to thank Theresa O'Connell and Maria Monteagudo for their invaluable administrative help on all stages of this project.

This research was supported in part by NASA (under grants NAG5-9478, NAGW-4080, NAG5-5095, NAS5-97222, and NAG5-6830) and NSF (CDA-9711582, IRI-9409661, HRD-9707076, and ANI-9876409).

ABSTRACT OF THE DISSERTATION

DYNAMIC DATA RETRIEVAL ON THE WORLD WIDE WEB

by

Dmitriy Beryoza

Florida International University, 2000

Miami, Florida

Professor Naphtali Rishe, Major Professor

Methods for accessing data on the Web have been the focus of active research over the past few years. In this thesis we propose a method for representing Web sites as data sources. We designed a Data Extractor data retrieval solution that allows us to define queries to Web sites and process resulting data sets. Data Extractor is being integrated into the MSemODB heterogeneous database management system. With its help database queries can be distributed over both local and Web data sources within MSemODB framework.

Data Extractor treats Web sites as data sources, controlling query execution and data retrieval. It works as an intermediary between the applications and the sites. Data Extractor utilizes a two-fold "custom wrapper" approach for information retrieval. Wrappers for the majority of sites are easily built using a powerful and expressive

scripting language, while complex cases are processed using Java-based wrappers that utilize specially designed library of data retrieval, parsing and Web access routines. In addition to wrapper development we thoroughly investigate issues associated with Web site selection, analysis and processing.

Data Extractor is designed to act as a data retrieval server, as well as an embedded data retrieval solution. We also use it to create mobile agents that are shipped over the Internet to the client's computer to perform data retrieval on behalf of the user. This approach allows Data Extractor to distribute and scale well.

This study confirms feasibility of building custom wrappers for Web sites. This approach provides accuracy of data retrieval, and power and flexibility in handling of complex cases.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

1 Introduction

1.1 Problem background

The explosive growth of the World Wide Web in the recent years has provided users worldwide with unprecedented volumes of information. This growth was fueled by the ease with which information can be published and accessed from anywhere in the world. Anyone with a browser and an Internet connection can access any document available on the Web. Data on the Internet is primarily published in HTML [HTML], a text format with rich hypertext presentation capabilities. This format, however, does not provide any mechanism for conveying to the user the semantics of the data that it presents.

There are several ways to find information on the Web. The simplest one is to browse the Web site that contains the data of interest. For example, if a person is interested in information about new and used car prices, she can go to Edmund's guide (http://www.edmunds.com) and read everything about a particular car model. This, of course, requires prior knowledge of the specific Web site.

The next level of querying for information is interaction with a search engine like Yahoo! (http://www.yahoo.com) or AltaVista (http://www.altavista.com). Here users specify keywords and the search engine tries to find Web sites that contain them. Sometimes, sites themselves provide simple search functions.

Even more control over querying is given by specialized search and interaction functionality. Complex queries that contain many different parameters can be posed. For example, when user is looking for airline tickets on Travelocity (http://www.travelocity.com), she can search available flights by dates and times, source and destination, airline, price and other factors. Often, such complex querying functionality is associated with other transactions, such as interacting with online services and purchasing products.

This variety of ways to look for information on the Web, unfortunately, does not give us a comparable variety in presentation formats. With few exceptions, this information is provided only for *visual consumption.* No convenient mechanisms exist for analysis and processing of the found data, users can only read what is presented in a Web page. There is no way for user to, for example, create complex queries to the travel agent's database—for each of such queries a special program would have to be implemented inside the agent's Web server. Even if the queries that are available meet user's needs there is no way to work with the data that they return. The stock quotes that are available on the financial Web sites most of the time cannot be imported in a spreadsheet for further analysis. The addresses of businesses from online Yellow Pages cannot be used in electronic address book or printed as mailing labels.

The sites that do provide data in an easy-to-process form, such as XML [XML] (a "cousin" of HTML that retains the semantics of data), are still rare. Although this is

expected to change in the future, large volumes of data are likely to remain in HTML for quite some time.

1.2 Problem statement

As we have seen the data on the Web is available to users through very simple interfaces that, nevertheless, do not allow close interaction with information. In the majority of cases user can only browse through data, not being able to define complex queries to it or feed it into databases or analytical tools. Because of these limitations data on the Internet is often underutilized.

There exists a strong need for developing a mechanism for accessing data that is scattered across Web sites and using it efficiently in a variety of applications, such as database management systems, spreadsheets, and analytical tools. As we will see in the next chapter this problem has interested researchers for several years. A number of approaches to building a mechanism that would connect Web sites and applications have been proposed both in academia and business, which only emphasizes the need for it.

In this research project we present a system for accessing data on the Web developed at High-Performance Database Research Center (HPDRC), a research center at Florida International University.

1.3 Dissertation overview

The rest of this thesis is structured as follows. Chapter 2 gives a brief insight into the currently existing methods for Web data retrieval. Various academic research projects are reviewed, along with business solutions and developer tools currently available on the market. Chapter 3 introduces the Multidatabase Semantic Object-Oriented Database System (MSemODB) being developed at Florida International University. It describes, in particular, the place of Web data sources in the MSemODB scheme. Web data source analysis and wrapper construction are covered extensively in Chapter 4. It also provides information on Data Extraction Library functionality created for simplification of wrapper design, as well as on the scripting language that allows speedy wrapper creation for simple data sources. Chapter 5 describes our approach to distributing data extraction functionality to the client side (throughout this work, we use *data extraction* interchangeably with *data retrieval*). Finally Chapter 6 summarizes the ideas introduced in this study and discusses avenues for further improvements and additional research that is planned in this project.

2 Related work

2.1 Theoretical research

Over the past several years many researchers have studied ways for collecting and processing data available on Web sites. A variety of methods for accessing such data have been proposed and two major directions in data extraction—*automatic* and *wrapper-based*—have emerged.

Although *fully automatic* extraction and labeling of data on arbitrary sites is currently beyond capabilities of computer science, assisted, learning-based data extraction has been quite successful in some systems [AD99, GM99, LN99]. In these examples systems guesses the site structures and ask users to label data fields that were found. Automated data extraction systems were created that are well tailored to specific domains [DEW97], use ontologies [ECJ99], combinations of heuristics [EJN99], and language-based parsing [MSS99].

Wrappers are interface modules that mediate between Web sites and clients that want to extract data from them. Wrappers are usually coded manually or generated through special wrapper-generating browsers, such as ones described in [LHB+99, NQL, SK98]. In our research we used wrapper-based approach because it gives the highest accuracy results and can be used to cover virtually any problem domain.

Wrappers use specialized scripting languages to define data extraction and labeling rules. Some of such languages are based on Prolog-like predicate logic [LSS96, Coh98, LD96, CDS+98], others are procedural [WEBL, NQL].

Through data extraction Web sites can be used as data sources in applications and database systems. To facilitate such integration wrapper languages are often based on well-studied SQL syntax [MMM96, AM98, BD99, BD99a, KS98, BDHS96, NS00, LC99]. In such systems queries to the Web are short and easy to maintain. They also extend SQL to provide good search, parsing and site navigation functionality.

Many wrapper languages have special features that simplify the process of data extraction. Some use regular expressions for data extraction and navigation through HTML documents [NS00, MMM96, AM98, AM97, HGC+97, DYKR00, KS98, Coh98, WEBL, NQL]. Because of the hierarchical nature of HTML many researchers [AM98, BDP00, EJN99, LN99, Coh98, LHB+99, NS00] use trees to represent Web documents. Such structures are simpler to search and navigate. Others [HGC+97] favor flat file representation because it is easier to store, requires no parsing, and is forgiving to document syntax errors.

Some of the systems [LSS96, AM98, AMM97] in addition to data extraction provide restructuring of documents. These systems can be used as standalone tools because presentation of results is a built-in feature. Form processing is used in some (but not all) wrapper projects [DEW97, BD99, KS98, DFKR99, FS99] in addition to simple

link navigation. Some projects [CM98, AD99] support data extraction from documents that are not HTML.

The systems and extraction methods listed above deal with formats that do not stroe information about semantics of data. In XML data comes already labeled and structured, and therefore pattern matching is usually not needed. There is, however, a need for document navigation and querying. As XML is becoming a language of choice for data exchange and software integration, new languages emerge that support these operations. Examples include XSLT [XSLT], XML-QL [DFF98], XML Query Language (XQL) [RLS98], and SgmlQL [HLM+97], XPath [XPATH], XPtr [XPTR] and others [BGL+99, CDTW00, CCS00, MS99, FSW+99].

2.2 Business solutions and data retrieval tools

Data extraction on the Web has not only been the topic of theoretical research—it is widely used in commercial applications. Some of the solutions were born as research projects and later found application in business.

Perhaps the most popular type of Internet data retrieval application is a *comparison-shopping agent*. Such agents dynamically collect prices across a multitude of Web stores and allow user to compare and shop. Examples include MySimon (http://www.mysimon.com), Yahoo! Shopping (http://shopping.yahoo.com), RoboShopper (http://www.roboshopper.com), and R U Sure (http://www.rusure.com)

Some comparison shopping systems were born as university research projects in Web data retrieval and then successfully applied to business needs. Good examples of this are Jango (http://jango.excite.com), which grew out of ShopBot research [DEW97]; GoTo Shopping (http://shop.goto.com/) (formerly Cadabra) which is based on Stanford "DDI" research project; and Junglee (now part of Amazon.com), which also came from Stanford research.

Another application of data retrieval is *auction shoppers*—systems that query Web-based auctions dynamically and bid for available products on behalf of user. Popular examples of these systems are Auction Watchers (http://www.auctionwatchers.com/), Auction Tracker inside R U Sure (http://www.rusure.com/) and Bidders Edge (http://www.biddersedge.com)

Some of the commercial data analysis and integration tools that are not related to Web data retrieval sometimes provide this functionality. Microsoft Excel is capable of retrieving a Web page and treating it as a spreadsheet through a feature called Web Query. Microsoft Access can import data into tables from HTML files. SuperNova data integration suite of products (http://www.supernova.com) is shipped with a GUI-based editor for building wrappers around Web sites. These wrappers then help present Web sites as relational data sources that could be queried for information.

Some of the tools available on the market were built specifically for dynamic data extraction, processing and integration with other data sources. Known examples of such tools are packages available from OnDisplay (http://www.ondisplay.com),

Cohera (http://www.cohera.com) and Liaison (http://www.liaison.com). They provide GUI editors for building wrappers around Web sites. Using these editors regions inside HTML pages can be marked and labeled as data fields. Data extraction tools collect data using one or more wrappers (doing so periodically or on demand) and channel collected data into existing applications. Some of them provide application programming interfaces to facilitate such integration, which allows enterprises to build custom applications that use Web sites as data sources.

A large number of software packages that make it easier to extract data from Web sites have appeared in recent years. Among them are Net Reaper (http://www.chimerasoft.com/chimerasoft/netreaper/), MergEm (http://www.sky.net/~floersch/htmltools.html#merge), Web Sifter (http://inventiveweb.com/prodwebsft.htm), SGrep (http://www.cs.helsinki.fi/~jjaakkol/sgrep.html), and MacroBot (http://www.ipgroup.com/macrobot/). A number of specialized programming languages, such as WebL [WEBL], Network Query Language [NQL], and W4F [W4F], were designed recently to simplify tasks of Web navigation and page content extraction. Some of the tools that were originally designed for text processing, such as Perl programming language [PERL] and OmniMark [OMNI], are also being used for Web data extraction.

2.3 Summary

As we have seen, there has been a significant number of projects, both research and commercial, that attempt to define efficient mechanisms for Web data retrieval,

extraction and processing. Some of the researchers propose custom-built wrappers, some concentrate on building the SQL-like query languages for extracting the information, others present the mechanisms for auto-discovery of data on the Web.

In this work we would like to define a system for data extraction from the Web that will satisfy the following requirements:

- *Power.* System must extract data from the vast majority (if not all) of data providers on the Internet. This, of course, requires development of powerful networking, site analysis and data extraction functionality.

- *Embeddability and database integration.* We must be able to integrate data extraction functionality with other applications. Additionally, tight integration with database management systems must be provided.

- *Portability and mobility.* System must operate on multiple platforms, and be compact for embedding in mobile data collection solutions.

- *Simplicity and maintainability.* Solution must be simple to use and maintain, because the format and functionality of data sources on the Web changes frequently, and related changes will have to be made in the system's setup.

In the rest of this work we describe our experiences developing such system.

3 Heterogeneous database approach

3.1 Introduction

In the previous chapter we have seen a review of existing methods for accessing data on the Web. The majority of these methods concentrate on extraction and purification of data, and channeling it to external applications and users. However, others ([HGI+95, BDKM99, MAM+98]) approach Web data extraction as a part of a bigger problem of *heterogeneous database integration.*

In our research we are also investigating ways of integrating data extraction into a heterogeneous database system.

3.2 MSemODB

The need exists for integration of the wide variety of heterogeneous databases available today. Such integration would let users access resources of multiple databases of different types and structures via a unified interface. It will also empower them to pose queries over a collection of different data sources. To date there has been a variety of research projects on the issues of heterogeneous database integration. Various researches have studied the issue of bringing together different types of relational and object databases as well as semi-structured and unstructured data sources. At HPDRC, this problem has also been investigated extensively. Our research resulted in the development of MSemODB - a heterogeneous database

management system. The general architecture structure of MSemODM is shown in Figure 3-1.

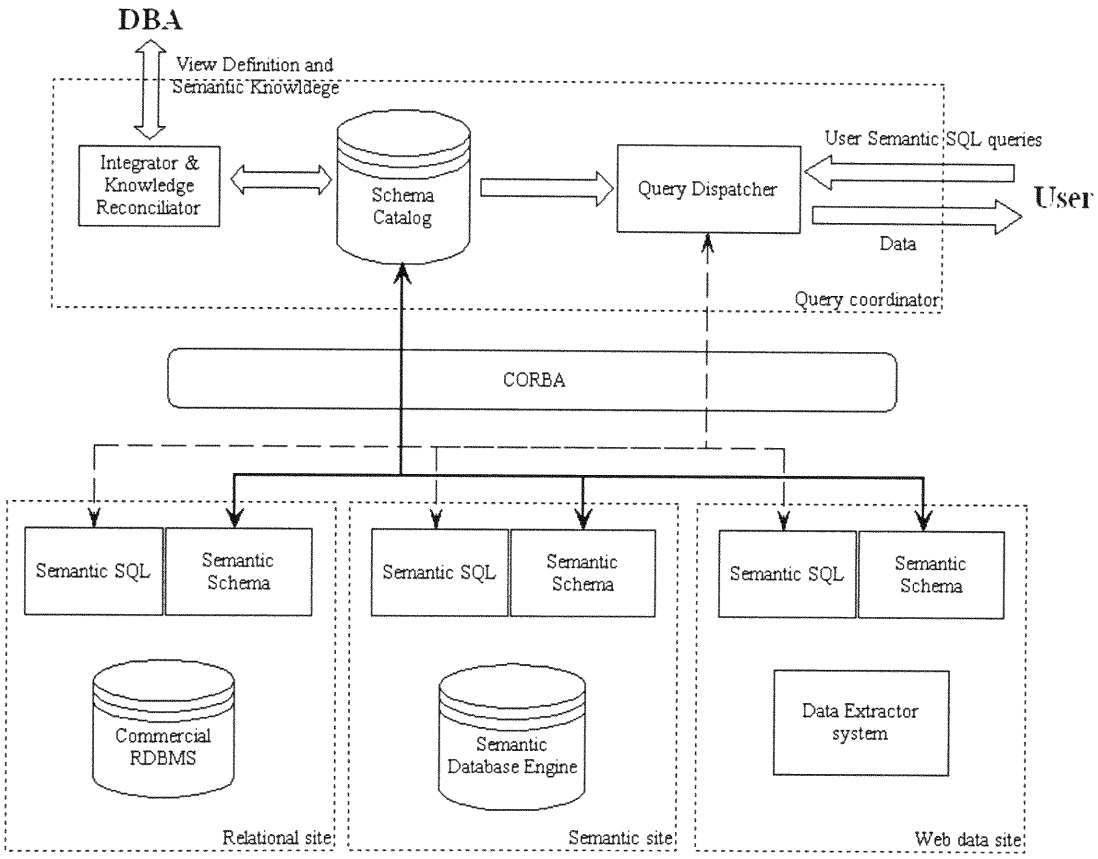

Figure 3-1 MSemODB architecture

MSemODB consists of several components. *Query Coordinator* that coordinates all operations in the system. *Sites*, each of them *wrapping* a particular type of data source, integrate these sources into the MSemODB system.

The system is using Semantic Binary Object-oriented Data Model (Sem-ODM) [Ris92] as a data representation standard and SQL query interface for communication between its components. Sem-ODM combines the simplicity of relational and power of object-oriented data models. A major advantage of this model is its ability to use standard SQL-92 query facility interpreted over Sem-ODM schemas (called Semantic SQL) in a variety of relational and object-oriented databases. This feature makes MSemODB compatible with a number of existing tools developed for standard SQL—a well-studied and popular query language.

The communication between components in the system uses CORBA technology, which is an efficient cross-platform and language-independent communication medium.

The main module that controls execution in the system and flow of data is *Query Coordinator*. Its function is to collect database schemas from all member databases and dispatch user queries to the appropriate database sites. It gives a common user interface to all the databases in the system. Through it users enter queries using common query language and view resulting datasets in a single data model. Query Coordinator consists of *Integrator & Knowledge Reconciliator*, *Schema Catalog* and *Query Dispatcher*. *Schema Catalog* collects schemas of individual relational, semantic and Web database sites, and coordinates them to resolve conflicts. Database administrator can use *Integrator & Knowledge Reconciliator* to manage and modify Schema Catalog, and to introduce new relations that are not apparent from mere collection of member schemas. *Query Dispatcher* optimizes and executes queries

using Schema Catalog. It decomposes queries posed by user into sub-queries based on the knowledge stored in the Schema Catalog and dispatches these sub-queries to appropriate sites for execution. When the results are available, it assembles them and presents them to the user.

The database sites are exposed in the system through their individual *Semantic SQL* and *Semantic Schema* modules. For the *Relational Site* a special knowledgebase and a reverse-engineering tool facilitate relational-to-semantic schema translation and storage. The majority of translation tasks are performed automatically. The database administrator can step in and make modifications and enhancements to the schema after the automatic conversion is completed. The Semantic SQL module of the Relational Site implements an algorithm which automates conversion from Semantic SQL queries to relational SQL queries. With this functionality, virtually any commercial RDBMS available on the market today can be integrated with MSemODB.

In the *Semantic Site* that wraps around Semantic Databases integration is far more natural. Semantic Object-oriented Database engine (Sem-ODB) (developed at HPDRC) already has Semantic Schema and Semantic SQL query facilities in implemented. This database system is a multi-platform, distributed, fully functional client-server implementation, that is suitable both for standard database applications and for large-volume data and spatial data applications.

The third site is built around a *Web data extraction system* that is the topic of this work. The purpose of this system, called *Data Extractor*, is to provide a reliable and efficient framework for data extraction from the Web. Using Data Extractor external applications pose queries to Web sites and extract data from them. Data Extractor presents extracted data in two-dimensional tables that can be further processed and returned to the user. This system is currently being used as a standalone application and development tool. A set of modules is being implemented to supply Semantic SQL and Semantic Schema functionality to the Web data site for its integration into MSemODB.

Further discussion and detailed description of MSemODB architecture and technologies that are used in its design can be found in [Ath00, RAYC00, RAYC00a, RYA+00, RYA+00a].

4 Web data sources

4.1 Introduction

We have seen many existing approaches to solving the problem of data retrieval on the Web. However, at HPDRC we felt that no single system has all the features necessary for adequate data extraction from the Web. In our work, the Data Extractor system, we attempted to integrate the advantages of the systems reviewed in Chapter 2.

4.2 Data Extractor

Data Extractor system provides our heterogeneous database system, MSemODB, with a mechanism for accessing data available on the World Wide Web. The Semantic SQL and Semantic Schema modules access Data Extractor through a standard interface that allows schema discovery, query initiation and data retrieval. Together with these modules the system works as an integral part of a MSemODB system, capable of executing SQL queries and returning result datasets.

Data Extractor system consists of several components (see Figure 4-1).

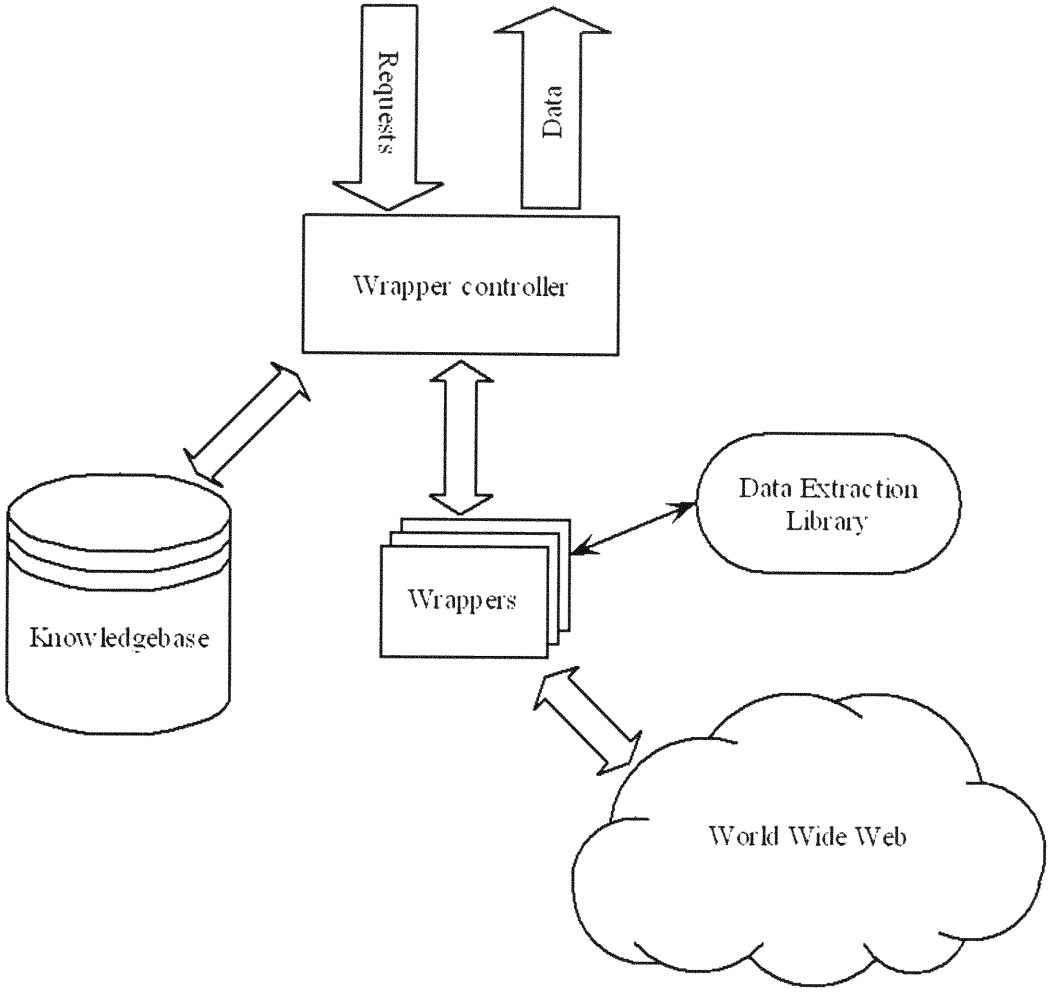

Figure 4-1 Data Extractor system structure

- *Wrapper Controller.* This is the main component of the system whose responsibility is to control the execution of all other parts of the system. It is the entry point for communications with the Data Extractor from the outside. It loads, executes and controls behavior of wrappers and redirects data that they generate to the user. It accesses the knowledgebase to become aware of the configuration and schema changes in the system.

- *Knowledgebase*. This module stores system configuration information. It contains data on what wrappers are available, where they can be loaded from, what parameters are required to execute them, and what kinds of data they generate.

- *Wrappers*. Wrappers are lightweight modules that execute in response to user requests. They extract data from the Web sites and return it to the user in an easy-to-process form.

- *Data Extraction Library*. This library contains extensive network access and HTML processing functionality. This functionality allows wrappers to traverse Web sites and extract data from HTML pages.

Data Extractor can be implemented as *standalone*, *embedded,* or *mobile* solution. As a *standalone* server it serves clients through a simple browser-based user interface, executes user queries, and delivers raw or processed data directly to the user. When *embedded* inside another application (as it is the case with MSemODB framework) Data Extractor acts as a data provider for that application. Lightweight *mobile* implementation of Data Extractor as a Mobile Data Retrieval Agent that is delivered to the user and executed at her computer is discussed in Chapter 5.

Let us now describe the process of Web site analysis and wrapper development.

4.3 Wrapper construction

As it was mentioned above, wrappers in Data Extractor execute on behalf of users and extract data from the Web sites. Wrappers essentially simulate a user working with the site through *the Web browser*. They fill out and submit forms, "click" on links, find data of interest inside of pages. To support this behavior special functionality was developed that emulates browser interaction with the Web site. It allows us to create and play back a "scenario" of user navigation through the site.

The process of creating a wrapper for a Web data source is a multi-step process:

4.3.1 Source selection

Surprisingly, selecting a Web site as a data provider in some applications might become a complicated task by itself. Cases when only a single Web site is a source of necessary information are actually quite rare. Stock quotes, airline schedules and weather information—the kinds of information that are needed in business applications—are usually provided by dozens of sites on the Web. As a result, selecting a source of information often becomes the first step in generating a wrapper.

There are many factors that have effect on decision to select a particular Web site as an information provider:

Registration and access control

Some of the sites today require user registration in return for providing information for free. This might cause problems for wrapper authors. The following must be considered:

a) *Registration overhead.* It is required that the human operator registers at the Web site and maintains the username and password in the knowledgebase or the wrapper itself. Upon registering the contact information has to be provided and username and password has to be kept current. This takes valuable time away from site analysis.

b) *Data depth.* Registration introduces an extra step in data extraction. Commonly a login form containing a username and password has to be filled out and submitted. With slower connections waiting for the form to be submitted and for user to be authenticated could last several seconds, delaying data retrieval.

c) *Secure Sockets Layer (SSL).* Most of the Web sites that require authentication also require data transfer after login to be done through SSL protocol. For reasons explained elsewhere in this work it was decided against implementation and deployment of an SSL-based solution. It is best to avoid such sites.

d) *Site overload.* The wrapper is likely to access the Web site much more frequently than the regular user. Some site operators may regard such behavior as an abuse

of service that they provide. When the wrapper operates through identity based on username and password it is much easier for the site maintainer to see that too many requests come from a particular user and terminate or restrict this user's access rights. It has to be noted that a carefully designed wrapper system will disallow information source overload through use of scheduling and load balancing algorithms. From our experience account termination most often occurs because of violation of internal quotas on number of Web site accesses set for users by the site owners. Actual site overload because of heavy wrapper use occurs much less frequently. This makes account termination more of a decision based on management and business policies, rather than on abuse.

Generally it is best to avoid sites that require registration because in our experience wrappers for such sites have proved to be burdensome to maintain effort required.

Web site owners' objections

Some of the Web sites owners are seriously concerned about outsiders using their information on a large scale. Even though the site access might be free, the high-volume access to the site and use of data by wrappers and related applications might (in the owner's opinion) constitute a copyright violation. Even though copyright and other legal claims in such cases are often without merit, it is wise to consult an attorney when creating wrappers for commercial use of information.

Site overload (a problem touched on in previous section) is also a concern. A "misbehaving" or, in other words, poorly written wrapper can cause significant problems for the Web site, flooding it with requests. In some cases such wrapper can effectively shut the site down, preventing legitimate users from accessing it. Special care should be taken when creating wrappers for slow sites. They must always be thoroughly tested before being used heavily.

Another problem that sometimes causes objection from Web site operators is *loss of banner revenue*. Wrappers are usually designed to ignore everything on the Web site except for the data inside HTML, and that includes advertisement banners. On a large scale this could cause profitability problems for the site, because for most sites on the Internet revenue from banner advertisements is the primary source of income. One could argue that Web crawlers that collect data for search engines also ignore banner advertisements. However search engines drive user traffic to Web sites and therefore it is beneficial for Web sites to allow them access to pages. Wrappers, on the other hand, only collect information from sites without bringing users to them. To avert complains of Web site maintainers it might be a good idea to give proper credit to the site, or even promote it. This can be done in the application that uses data from the site.

Cost

Although it is rare today, some sites do charge for access to information. Examples of this are Web sites that sell information record-by-record (e.g. address information),

and Web sites that charge subscription fees for access. In order to effectively access these services in most cases one would need automated payment system integrated with the wrapper system. Such system would need to track uses of data and correctly process payment information. In some cases it would also need to be integrated with a billing system that would pass the costs for accessing the data onto user. Significant costs associated with these data sources would make free use of data impossible and limit the widespread use of such wrappers. When situation allows, it is best to avoid pay-per-use and subscription sites as sources of information.

Web site performance and availability

If given a choice of Web sites of varying performance it is logical to give preference to the site that is the most responsive. This will assure that the wrapper, which by definition is slower than a local relational data source, would perform at the top speed. Additionally, the better performance of the Web site - the less intrusive and disruptive the wrapper is. High-performance sites like Yahoo! and Amazon.com, capable of handling millions of users, are less likely to be slowed down by wrappers.

HTML quality

The quality of HTML documents from which data is extracted can sometimes be an issue. Although the best effort is made by the Data Extraction Library algorithms to parse any HTML, some of the sites contain too many syntax errors. The only way for such HTML to be processed is without building a parsed tree, by treating HTML page

as a sequence of markup elements. Although this approach gives an alternative way of extracting data it is extremely hard to deal with. To summarize, such sites cannot easily be handled with technology that we describe and alternatives should be used, if available.

Mining depth

An important criterion for Web site selection is *mining depth.* This is an estimate of how many Web pages wrapper has to go through in order to retrieve the complete data set. This metric can be composed of two numbers. The first number is usually constant—it is the number of pages that the wrapper has to go through before getting to the page that contains data. On these users enter login information and fill out query forms. Second number is usually variable—it represents the number of HTML pages that wrapper must traverse to collect all data returned by a query. This number depends on the query that was posed through the form and sometimes can be manipulated through "results per page" parameter offered by some of the sites.

Requests for pages and downloading of these pages are significantly slower operations compared to parsing and data extraction. Consequently, the less mining depth is, the better potential performance of the wrapper is (wrapper performance depends on a set of other factors too, of course, such as site performance and accessibility). It is better to select the Web site that has the minimal mining depth.

Richness and volume of data

Some of the sites provide more useful or relevant kinds of data than the others, or provide bigger volumes of data. With all other conditions being equal it is better to give preference to the sites that provide the most complete data sets.

Data locality

As it was already stressed, minimization of the number of pages that have to be accessed for data extraction is one of the key factors to increasing wrapper performance and robustness. Minimizing the number of data pages to be retrieved and processed, is not, unfortunately, the only issue to be solved. In a significant number of cases pages that contain data do not have complete data. Let's consider a hypothetical example of an airline reservation system Web site. When user requests information about all tickets available for a given destination, a set of pages is returned that contains available flights and tickets. However, some crucial information might be missing, such as number of stopovers for a given flight, or an aircraft type. This information might be available on a separate page to which the results page gives a link. The problem is that to access this additional page we have to make separate request to the Web site. This request must be made *for every flight.* Therefore, for a total of x flights distributed over y HTML pages of results wrapper has to make $(y + x)$ page downloads rather than just y page downloads. The extra x represents the page with miscellaneous flight information that we have to retrieve for every flight. Usually the number of result pages y is comparatively small, but the total number of

items x on those pages is significantly bigger, which causes a sharp increase in the number of page downloads. This is a drawback that drastically decreases wrapper performance. It is best to avoid sites that require this many requests for data retrieval.

Data presentation formats

Some of the Web sites use Java applets, plug-ins and ActiveX controls to deliver information to the user. Technology that we are describing is not yet capable of extracting information from these browser applications. Sites that exclusively use these methods to deliver information should be avoided when selecting data providers.

4.3.2 Tree representation of HTML

Before describing the actual Web site analysis let us introduce a data structure that is used in Data Extractor project to represent HTML documents. When the Data Extractor project was developed it was tempting to represent HTML as flat files in memory. Flat files are easy to store and do not require complex handling algorithms for processing.

However we decided that the tree data structure is more appropriate for storing an HTML file. HTML is hierarchical in nature and, if the file is syntactically correct, it can be represented in a form of a tree. Tree representation has several advantages over flat files:

- *Convenient structure.* Traveling to pre-determined portions of the document stored in a tree is easier than it is in structure-less flat file.

- *Data delineation.* Clear division of HTML into separate regions simplifies extraction of semantics from it and defining data boundaries.

- *Simplified searching.* Searching for particular pieces of text and data can be performed in isolated regions of the HTML file.

The idea of storing HTML in a form of a tree was used in many research projects and software packages. One standard for creating and processing such trees is described in [DOM].

There are, of course, some disadvantages to storing HTML files in trees. One of the concerns is that time and amount of memory used to process and represent the tree are greater than for flat files. Another problem is the low syntax quality of HTML available on the Web, which makes it hard to build correct trees.

To determine how the HTML file can be presented in a form of a tree let us consider an example of a simple HTML file in Figure 4-2.

```
<HTML>
  <HEAD>
    <TITLE>Simple page</TITLE>
  </HEAD>
  <BODY>
    <H2>Hello, World!</H2>
    <!-- Simple comment -->
    <P>This is a very simple HTML page.</P>
  </BODY>
</HTML>
```

Figure 4-2 Sample HTML document

Despite its simplicity this file contains all major elements of a common Web page. These include tags, text and comments. In HTML, most tags have an opening and closing part. Some of the closing tags can be omitted or, for some types of tags, do not exist at all. The tags that can have closing parts can contain other tags and elements, forming hierarchical parent/child relationships. Text and comments cannot have children elements associated with them. Based on this information we can build a tree structure from the HTML text in Figure 4-2. This structure is shown in Figure 4-3.

We begin building a tree with a *Root* element. There is no equivalent to Root in HTML and we introduce it in order to simplify tree operations and make them more uniform. Because the HTML tree can have many sibling elements at the top, such tree would be hard to manage without a single Root element. An empty HTML file will have just the Root element as its tree.

As we parse HTML we are continuously adding elements to the tree. The elements that are located on the same level are considered siblings and are put into sibling

nodes in the tree. An element that is a child of another element becomes its child in a tree.

We start with *HTML* tag and put it directly under the Root node as it encloses all other elements in the tree. We replace the combination of an opening and closing tags with a single node in the tree that becomes the parent of the elements that these opening and closing tags surround.

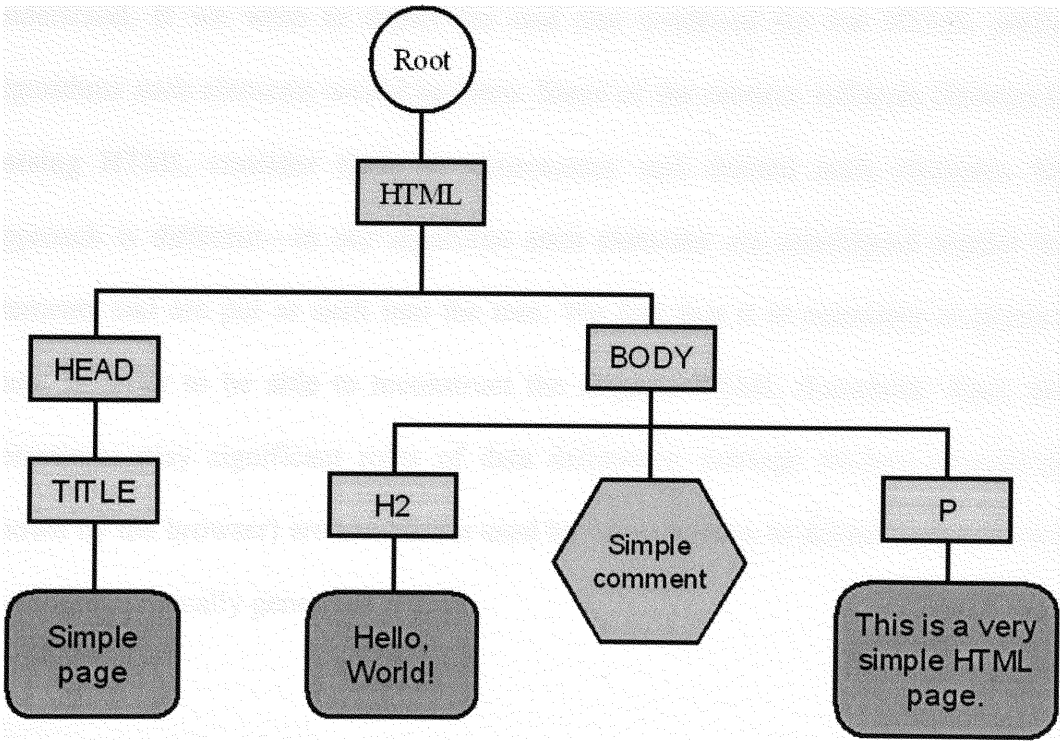

Figure 4-3 Tree representation of Simple HTML document

HEAD and *BODY* tags are children of *HTML* and are placed as such in the tree. As we continue, tags *TITLE*, *H2* and *P* are placed in their proper locations. Comments and pieces of text become tree *leaves* (nodes without children) and children to the

tags that surround them. Comment is located at the same level of the tree as *H2* and *P* because it is a sibling of those elements in the HTML source.

The tree that we see does not, however, fully represent the HTML file that is shown in Figure 4-2. Some pieces of the HTML, namely spaces and carriage returns, are missing, while they are, technically, text elements. The decision not to include them in the tree depicted in Figure 4-3 was made for the sake of simplicity, because addition of 9 new text elements would overcrowd the figure and make it hard to understand. If we were to depict the real tree produced by our HTML parsing algorithms such elements would be there. Some of the modern software libraries for parsing HTML consider them as unnecessary and discard such elements. Our approach is different—in our algorithm such elements are considered regular text elements and are put as such into the tree. We feel that it is necessary to preserve them in order to be able to reconstruct the original HTML document. Also, they sometimes play significant roles of data delimiters: carriage returns (though not shown by the browser) are sometimes used by programmers to delimit output of data in programmatically generated HTML.

4.3.3 Web site analysis

Once a Web site is selected for data retrieval (but before a wrapper is created for it) a thorough analysis of the site must be performed. Success or failure of a particular wrapper depends primarily on how well was the site analysis done. Good analysis

usually makes the subsequent programming effort easier and the resulting code is more efficient, less bulky, and easier to maintain. The task of wrapper construction can be sometimes split between two professionals: an *analyst* that studies and documents the site in a special report, and a *programmer* that follows analysts' report and implements the wrapper. To save time and effort the same person usually performs both tasks. Therefore, when referring to "analyst" we will assume a person doing both analysis and implementation.

Programming is the most tedious and time-consuming task of the two. Eventually we hope to eliminate the programming step from the majority of wrapper construction tasks with the help of a GUI tool. Such tool would allow analyst to generate a wrapper by highlighting text portions inside the Web site, and recording link clicks and form fills.

There are many properties and features of the Web site that the analyst has to identify in her research. All of them will influence how the resulting wrapper will behave and how effective it will be:

Starting page/deep linking

Identifying the starting page where the data retrieval process should begin is very important. The seemingly simpler way of getting to data pages by following the path from the home page is not always the best one. In this case a lot of unnecessary pages have to be retrieved and analyzed before wrapper even retrieves the first data page.

The page that contains the query form may be buried deep inside the site. Therefore, the better way of doing this in some cases is to find learn the URL of the page that contains data or the form and retrieve that page directly, without going from the home page first. This technique is commonly called *deep linking*. This will primarily work for so-called *static* URLs that do not change because of context or parameters.

Sometimes a wrapper can deep link to the URL that is *semi-static*. This means that portions of the URL can be modified depending on the wrapper operating parameters and the resulting URL will point directly to the data page. This, for example, will work well for retrieving information from the Yahoo! Weather site.

The straightforward approach to finding out weather for Miami, Florida when going from the home page is:

a) Retrieve Yahoo! Weather home page (http://weather.yahoo.com)

b) Inside the page locate form entitled "Search by City or Zip Code"

c) Fill out the form with the string "Miami,FL" and submit it

d) Download the resulting page, parse it and extract the weather information

However, we notice that the resulting page URL is:

```
http://weather.yahoo.com/forecast/Miami_FL_US_f.html
```

This gives us a clue to how the city searches are done on this site. One could easily substitute "Seattle_WA" for "Miami_FL", and receive weather for Seattle instead. The resulting wrapper could be built to implement such substitution based on the parameters supplied at run time. This approach certainly involves certain dangers, as it is less robust than submitting the form. The server application that processes the form might be doing some elaborate error checking, for example, checking the name of the city and substituting alternative names if the city is not found.

Dynamic URLs with well-studied structure can be modified too in some cases. We call URLs *dynamic* when they contain changing parameters, or are otherwise changing depending on the circumstances. When the nature of changes is well known, substitutions of portions of the URL may allow deep linking. Let us consider Excite search engine. When searching for topic "dogs" we get a page with a list sites related to dogs. One can find by analyzing URL of the result page that the query string is associated with form parameter "search":

`http://search.excite.com/search.gw?search=dogs`

By substituting the parameter value with a different value (e.g. "cats") we will get the result page with a list of links related to cats. Here by changing the parameters in the URL we are effectively deep linking into that page.

When the form and its parameters are known, its submission can be simulated in software. Instead of loading the page that contains the form, finding it, filling it out

and submitting it wrapper can compose the appropriate HTTP GET or POST request and submit it to the server (in-depth discussion of HTTP protocol can be found in [HTTP]). This approach is certainly faster, but less reliable, because the form's hidden parameters are more likely to change than its regular parameters, and the wrapper will have to be changed more frequently when form submission is simulated.

In some cases deep linking is not possible—for example, when a *session* has to be established between user's Web browser and a server of the data provider. Sessions are established through one of the following three mechanisms: *persistent HTTP connection*, *cookies* and *URL rewriting*.

Persistent HTTP connection is a mechanism introduced in HTTP 1.1 specification, which allows the Web site to respond to multiple requests over a single TCP/IP connection. This connection uniquely identifies the client. With such connections deep linking is hard or impossible to do, because in order to establish connection user has to go to the site's home page and log in.

Cookies are small pieces of textual information that Web site can save in user's browser and then retrieve them when necessary. Through cookies, login or shopping basket information can be saved at client side. Because such information usually arrives and is retrieved in a certain order deep linking or skipping steps in this process is not always possible.

Finally, servers use *URL rewriting* to associate a session ID with the user. By embedding this ID in all pages and URLs that it sends to the client, server can always use the URL to tell what user sent it. Sometimes instead of session ID some other identifier, such as a user name, is used. Through this identification mechanism all transactions are associated with a particular user. One example of URL rewriting is used in Web-based e-mail system NetAddress. When the user is logged in to check her e-mail, a dynamic session ID is assigned to her and becomes part of the URL:

```
http://www.netaddress.com/tpl/Door/314IBGSZC/Welcome
```

The ID in this case is the combination of digits and letters "314IBGSZC". This ID is valid only for a small period of time—until the login expires or user logs out of her e-mail account. After that the URL becomes inaccessible.

Because the mechanism for allocating and distributing session IDs is located on the server there is no way to reliably simulate it in the wrapper. Therefore, deep linking to query results on sites with session IDs embedded in links is usually not possible.

Deep linking and skipping start pages helps lower the wrapper work time and the number of page downloads. Also, it sometimes helps lower the frequency of wrapper failures because the path from the entry point to the actual data becomes shorter, thus decreasing the number of places where something could change if the site layout is modified. On the other hand, deep linking might sometimes be the cause for wrapper failure. Rather than following the path that user usually follows by clicking links and

filling out forms, wrapper attempts to take a shortcut and does so based on knowledge of inner workings of the Web site which could change without warning. Web site changes are inevitable in most cases and whether deep linking will prevent wrapper from becoming outdated or hasten its expiration depends on a particular case.

Page identification

Before the data extraction can actually take place wrapper has to make sure that the page containing data was actually retrieved. Sometimes the page that is expected does not arrive. This happens because of many reasons. Network and site problems may generate HTTP error pages in response to the legitimate request. Site structure and layout changes could also cause generation of error pages. Finally, the query that the wrapper poses to the Web site might bring no results.

For reliable wrapper execution it is important to check for all of these signs before attempting to perform data extraction. HTTP errors are easily detectable and wrapper should have no problem with halting execution when it detects one. Error pages that are generated by the site are harder to handle, because the wrapper has to check for presence, or, on the contrary, for absence of certain features in the document. Checking for known error messages, absence or presence of known tables, links and forms could serve as good indications of whether the data was successfully delivered. Informational messages, such as "No records found" or "123 sites found", serve as good evidence of presence or absence of data.

Location markers

One of the ways to locate data inside HTML is to search for it with reference to some visible *location marker* (or "landmark" [BD99, BDP00, MMK99]) inside the document. Location marker is any element of HTML that is uniquely identifiable. It could be a piece of text, an image, link, comment—any HTML element that could be found by searching the HTML tree. In addition to being unique, this marker has to be located close to the data that we are extracting. Once wrapper finds such marker inside the document, it could then "move" through the tree relative to marker's location in order to find data.

Let us consider an example of Ebay online auction system (http://www.ebay.com). When we are interested in a particular type of auction items we want to extract all information associated with them. That means extracting item titles, numbers, prices, bids, closing time, etc (see Figure 4-4).

However, as we can see, there is a large number of useless HTML elements—advertisement banners, menus, titles and other elements—that have to be skipped before we get to the actual auction data.

Useful data in this Web page starts in a table after the text "All items". This text is unique and is not likely to appear before that position (at least with that particular letter case and as a lone text element and not part of a sentence). We can use this

knowledge to instruct wrapper to travel to this piece of text inside the HTML, and then to retrieve auction data from the table that immediately follows it.

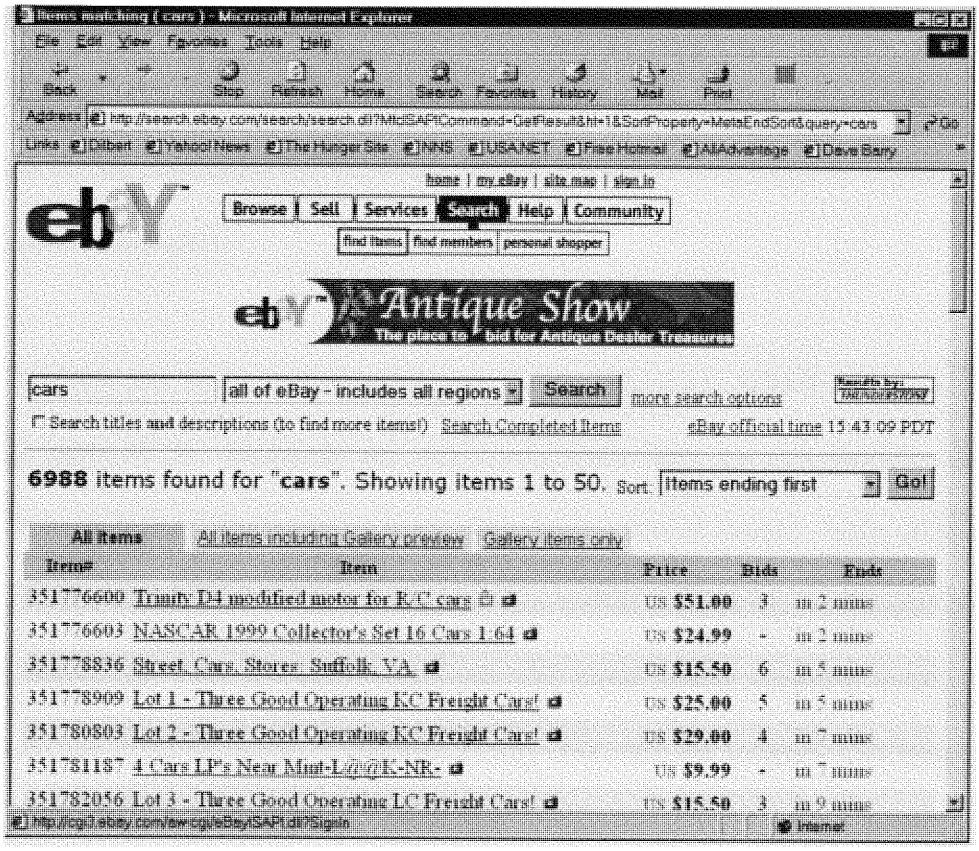

Figure 4-4 Sample of Ebay auctions

Before using this technique it is important to make sure that location marker is truly unique. In cases when a unique marker is hard to find, not one but a combination of HTML elements can be used as a marker.

Data identification markers

Finding data is made easier through the use of *data identification markers*. These markers are unique characteristics of an HTML document that point to data elements and delimit distinct data records. Such markers are usually unique only inside some part of an HTML document and they must be used together with location markers and other search techniques.

These markers can exist in many different forms. However, HTML elements that highlight parts of the document are most commonly used for this purpose. Such elements include tags that specify fonts, paragraphs, record breaks, table cells and colors. Some of the HTML elements that are not visible in the browser are useful for data identification. For example, comments and nonprintable symbols inside text records do not manifest itself to the user in any way, but the wrapper can split data into records using these elements as delimiters.

Let us consider a page from Zip2 (http://www.zip.com) Web site that shows records for department store addresses in San Francisco (see Figure 4-5). This page contains sets of titles, addresses and telephones of stores.

The data from the first record on this page is shown in bold in Figure 4-6. Extraction of this information is easy if you search for it using data identification markers that surround it. Wrapper can be instructed to extract store's title from the link that points to the details about the store. Street address and city and state information is located

next and can be extracted into separate records using BR tag as separator. Finally, the phone number can be identified as bold text in the adjacent table cell.

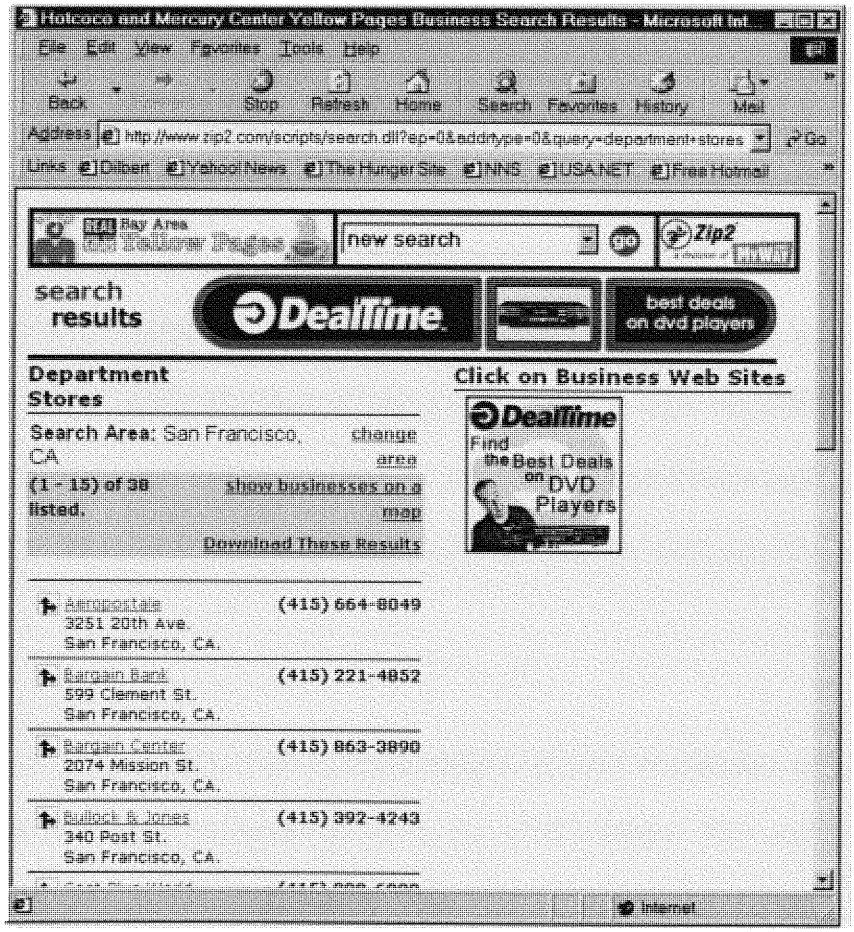

Figure 4-5 Zip2 address information

With the help of these features, data can be identified and extracted even when Web site structure changes, as long as such changes do not modify the page layout significantly.

```
<TD  ALIGN=LEFT VALIGN="TOP">
<FONT FACE="verdana,helvetica,arial" SIZE="1">
<A
HREF="http://www.zip2.com/scripts/search.dll?ep=3&id=495825812&pg=&totdup=&errp=&busco
b=&btl=on&query=department+stores&qcity=San+Francisco&qstate=CA&sic=531102&ck=&ccity=S
an+Francisco&cstate=CA&adrVer=-1&ver=d3.0">Aeropostale</A><BR>
3251 20th Ave.<br>
San Francisco, CA.</FONT><BR>
</TD>
<TD ALIGN="RIGHT" VALIGN="TOP"><nobr><B><FONT FACE="verdana,helvetica,arial"
SIZE="1">(415) 664-8049</FONT></B></nobr></TD>
```

Figure 4-6 Source of Zip2 address item

Tree search

One of the most powerful ways of locating data in HTML documents is searching. We ca search for a variety of pieces of information, such as tags, tag attributes, their values, text elements and comments. For the purposes of a particular application we can search for exact strings or substrings, searches can be case-sensitive or case-insensitive.

Because we are dealing with a tree structure additional types of searches are possible. Searching can be done in the entire tree or in any of its subtrees. This means that the search can start from the root element or from any element inside the tree structure and only affect descendants of that element. This is a useful property, because it allows us to localize search to specific logical parts of the HTML document, ignoring the rest. Additionally, we can search either in a subtree or *linearly*. In linear search the document is treated like a flat stream of HTML elements. Search starts from a given position in the document and continues until its end or until the element is found. This is different from subtree search, because subtree search finishes when the element is found or when every node in the subtree is visited by the search routine.

When the root element is the starting point for the search there is no difference in behavior for subtree and linear searches.

The decision to select either type of search depends on application needs. For some applications locating data inside isolated portions of the document is important (e.g. for searching inside tables). For other cases (like searching for location markers) it is easier to think of the document as of a flat file and search linearly.

Paths

The simplest yet the least reliable (in terms of long-term stability of the wrapper) way to locate data inside a page is by specifying a *path* to it. A path is a set of nodes we can traverse to reach the node we are looking for. Paths in HTML trees usually start from the root node.

As an example let us consider the path for reaching the text element "This is a very simple HTML page" in a tree in Figure 4-3. In order to reach it we start from the root of the tree and go to child number 1 at the root of the tree—tag *HTML*. After that we travel to child number 2 of tag *HTML*—tag *BODY*, and then to child number 3 of tag *BODY*—tag *P*. Finally we go to child number 1 of tag *P*—the text we are looking for. This trip through the tree can be recorded as an ordered set {1,2,3,1}. It represents a path from the root of the tree to the element of interest inside the document. Such sequences of steps can be easily followed in software, if the appropriate data structures and mechanisms for tree traversing are available.

Unfortunately, this approach is the leading cause of wrapper failure in the event of site changes. If a single node in the path is changed the whole path becomes invalid, requiring wrapper modifications.

It might be tempting to use paths as the only method for locating data inside HTML because of their simplicity. Paths are also much easier to implement in tools that assist analyst or developer in building wrappers. It is better, however, to concentrate on searching techniques, or combine searching with short paths that do not originate at the root node, because this will improve wrapper robustness.

Multipage data

The absolute majority of sites that provide large volumes of data dispense it in portions, showing it to user through sets of linked pages. There are multiple reasons for that: shorter pages transmit faster, they are easy to read, consume less memory on the client browser, and require less effort to be displayed. There is always a chance that user will find the data she is looking for on the first few pages without having to retrieve them all, thus minimizing load on the server. Finally, the more pages users view the more advertisement banners can be shown.

Being a convenience to users this often becomes an inconvenience for wrapper analysts. It is obviously easier to retrieve data from a single page than from a set of linked pages.

An Internet search engine like AltaVista (http://www.altavista.com) is a good example of a site that produces multi-page results from which data could be collected. AltaVista data is Web site descriptions, URLs and other information generated in response to search query. Data entries are displayed on multiple linked pages. User browses the result page, trying to find the Web site entry that is relevant to her query. If such entry is not found, user goes to the next page of results (see Figure 4-7).

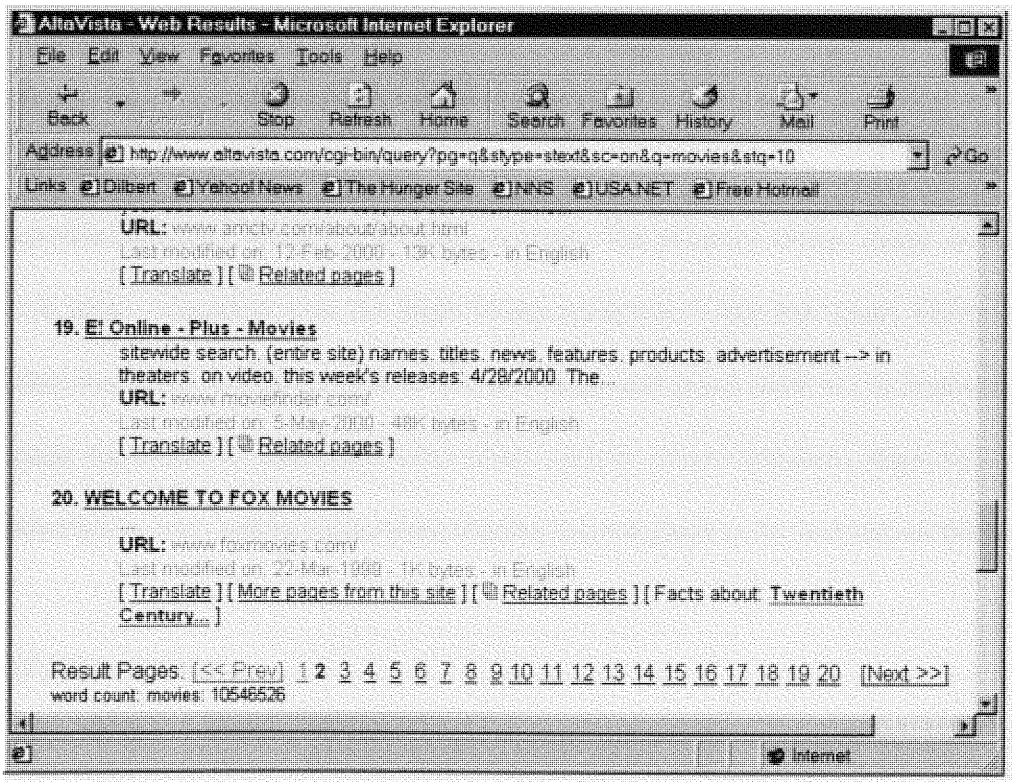

Figure 4-7 AltaVista search results

Data extraction of multi-page results is done continuously. When the first page of results is processed, wrapper follows the link to the next page. When that page is loaded, data extraction is done on it and the whole process repeats. The data that is

extracted is returned as a single data table or data stream to the process that executes the wrapper.

Techniques similar to those used for data identification are used for identification of links to pages in result sets. Such links usually appear either as HTML links or HTML forms (in the form of buttons). Wrapper simply follows the link or submits the form, and extracts the data from the resulting page, repeating the process over and over. Data extraction can be performed until all data is retrieved or, in cases when data sets are very big, until some threshold is reached.

It is important to know when to stop data retrieval. The absence of the "Next" link is usually a good indication that the page being analyzed is the last page. However sometimes such link can be present at the last page, and point back to the beginning of the page sequence. In such cases other clues, such as record numbers, page titles or other descriptive text, can be indications that the current page is the last one in the set.

Parallel page retrieval

Taking advantage of *parallel page retrieval* technique can significantly increase wrapper performance. Quite often the Web site that provides data displays it on multiple pages. Links to several such pages are sometimes accessible from the index or summary page. One example of this is the air flight schedule in an online travel agency Expedia (http://www.expedia.com). Such schedule is shown to user when she

enters a trip departure and destination cities and travel dates. See Figure 4-8 for an example.

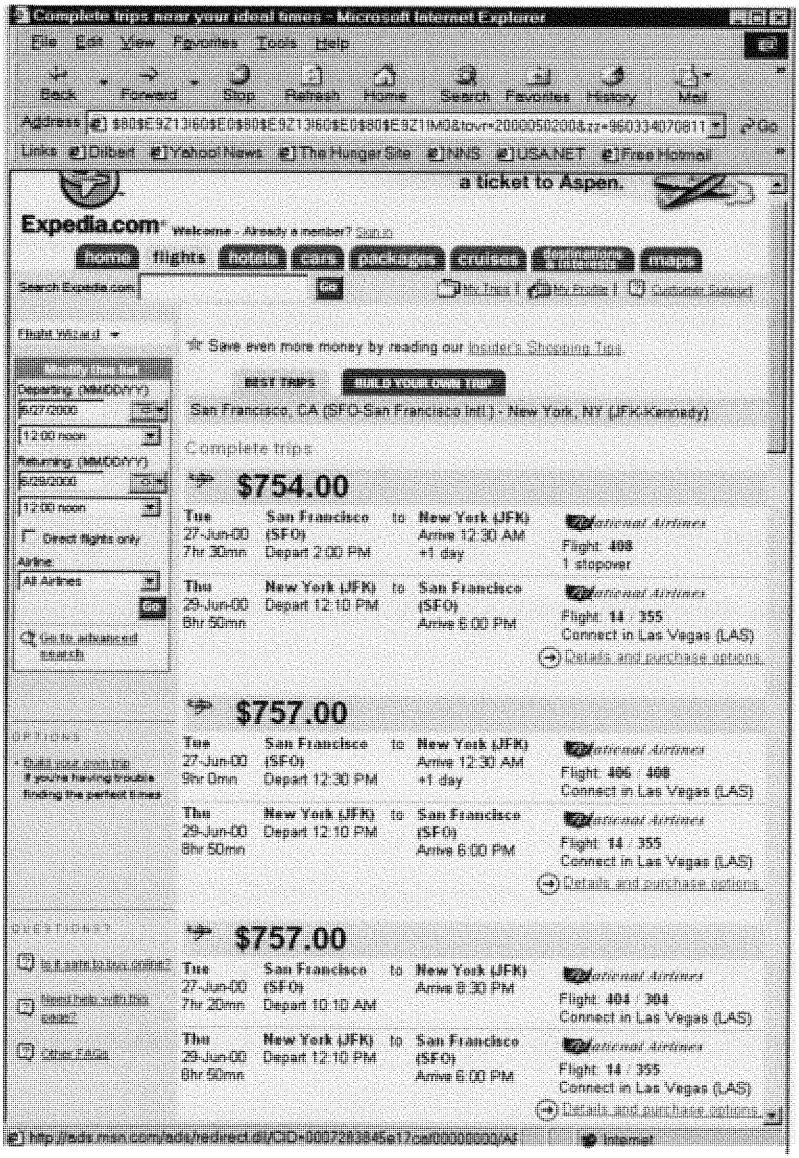

Figure 4-8 Flight information from Expedia.com

This schedule lists only brief details about each flight. The rest of the information user can get by clicking on the links next to flights (in our example—link called

"Details and purchase options"). We have already discussed the inefficiencies of trying to traverse all such links. However the analyst can craft wrapper in such a way that data is extracted efficiently.

This can be achieved by doing several data extraction tasks in parallel. Because all links to pages with details are known when the main page in Figure 4-8 is loaded, there is no reason why wrapper should visit them sequentially. Network access functionality becomes the bottleneck in sequential access when many processor cycles are wasted waiting for remote site to execute requests for pages and send pages back to client. Wrapper can be programmed to use multiple threads of execution, with each thread loading a page pointed to by a different link from the main page. When pages are retrieved and analyzed in this fashion, slowness of network access becomes less of an issue. If execution of one thread is blocking on network access others can continue to retrieve pages.

This approach significantly speeds up execution of wrappers. However, it is still far from being a "silver bullet". The speedup still will not bring overall speed close to cases when all information is available on a single page. The number of additional page retrievals, even if done in parallel, could run into hundreds and thousands, slowing down the wrapper. Additionally, parallel page retrieval is often hard to implement correctly and thus is more expensive in development and maintenance. To summarize, parallel page is a good technique that can increase wrapper performance, but it is better to select sites that do not require it.

Hidden data extraction

Data extraction is usually done on textual portions of the HTML page. To be effectively communicated to the user data has to be highly visible and occupy a prominent position inside the page. There are times, however, when useful data can be extracted from other, *hidden* parts of HTML. Good candidates for this are comments, tag attributes and their values, URLs and even scripts. Useful data might include IDs, prices, addresses, phone numbers, and other pieces of information. Extracting this information can save time on unnecessary page retrievals.

Let us return to an example of Zip2 (http://www.zip2.com). This company provides access to business address information of millions of companies in the US. User can query this information by cities and types of businesses. Figure 4-9 displays the results of a typical query for addresses of grocery stores in San Francisco, California.

Notice that the information presented here is incomplete—zip codes are missing from the list of addresses. In order to retrieve the zip code for a particular address user has to click on the icon with arrows displayed next to it. Figure 4-10 displays address details for "M & K Market". Notice that address now has a zip code, 94118.

This is of little help to us, however, because this means that for each item in a list that could be hundreds of items long we have to do an additional page download and parsing. This will slow the wrapper down significantly.

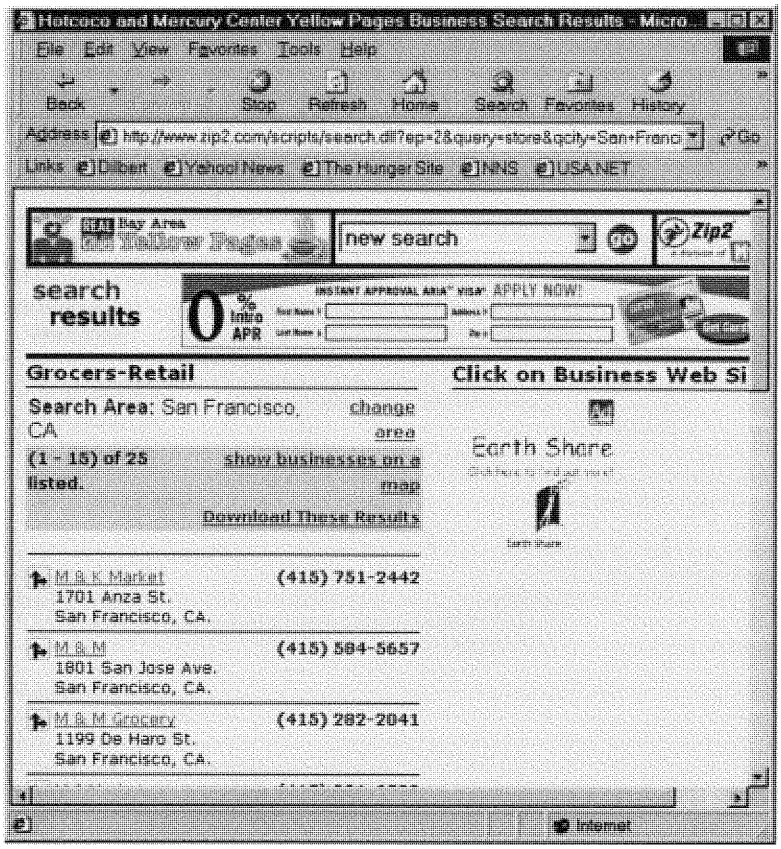

Figure 4-9 Zip2.com business address data

Hidden data retrieval rescues the situation. Let us take a close look at the HTML source of the page in Figure 4-9. A portion of this HTML is shown in Figure 4-11. After close examination of that figure one could clearly see that the highlighted portion of the URL, namely parameter "mzc2", contains the sought-after zip code, 94118. It can be easily extracted from that position. The information we need turns out to be present on the page despite the fact that it is not shown to the user. This means that we could extract the required data from the original HTML page without having to load additional pages, saving significant amounts of time.

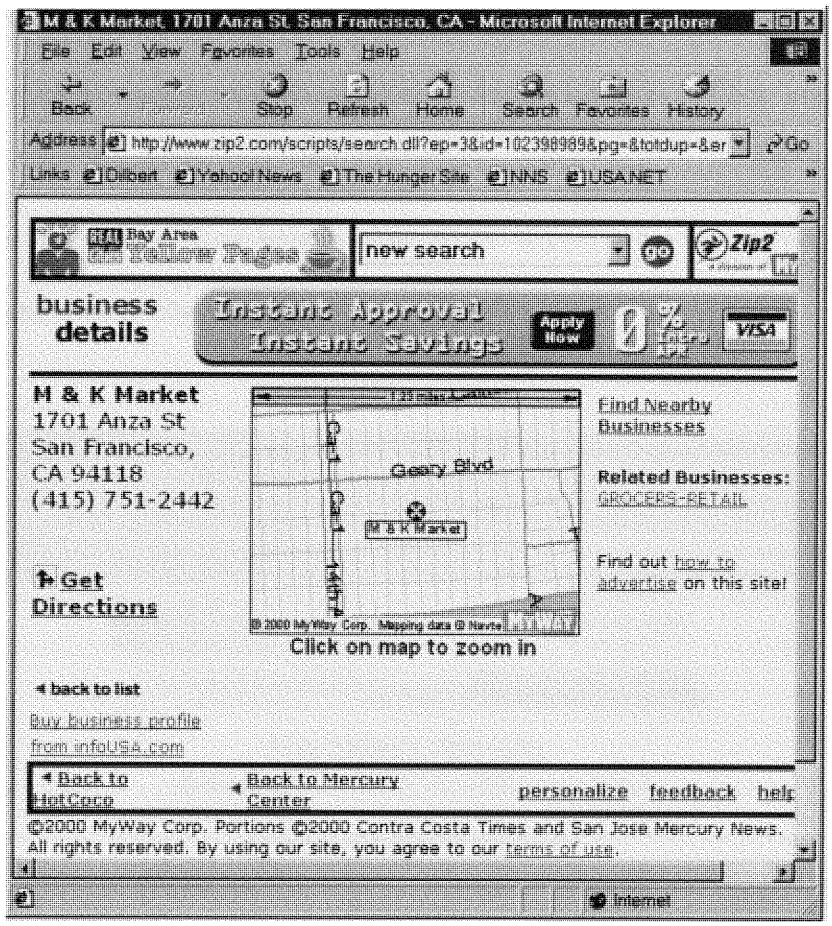

Figure 4-10 Details of address item

```
<TR BGCOLOR="white">
<TD VALIGN="TOP" ALIGN="RIGHT">
<A
HREF="http://www.zip2.com/scripts/doortodoor.dll?ep=8200&mid2=102398989&mal2=M+%26+K+M
arket&mad2=1701+Anza+St&mct2=San+Francisco&mst2=CA&mzc2=94118&mcn2=&mpn2=%28415%29+751
-
2442&query=grocery&qcity=San+Francisco&qstate=CA&narrowby=2&subquery=M&sic=541105&ck=&
userid=232228510&userpw=.&uh=232228510,0,&ccity=San+Francisco&cstate=CA&adrVer=-
1&ver=d3.0"><IMG ALT="Get driving directions" SRC="/images/directions.gif" BORDER="0"
WIDTH="18" HEIGHT="18"></A>
</TD>
<TD  ALIGN=LEFT VALIGN="TOP">
<FONT FACE="verdana,helvetica,arial" SIZE="1">
<A
HREF="http://www.zip2.com/scripts/search.dll?ep=3&id=102398989&pg=&totdup=&errp=&busco
b=&btl=on&query=grocery&qcity=San+Francisco&qstate=CA&narrowby=2&subquery=M&sic=541105
&ck=&userid=232228510&userpw=.&uh=232228510,0,&ccity=San+Francisco&cstate=CA&adrVer=-
1&ver=d3.0">M & K Market</A><BR>
1701 Anza St.<br>
San Francisco, CA.</FONT><BR>
```

Figure 4-11 HTML source of item "M & K Market"

Let us consider another example. Web site 1-800-96HOTEL (http://www.180096hotel.com) specializes in hotel bookings. When user selects a city, prices for hotels in that city are shown. After that, user can see detailed information about a hotel, or book a room there. Results of a sample request to this system can be seen in Figure 4-12.

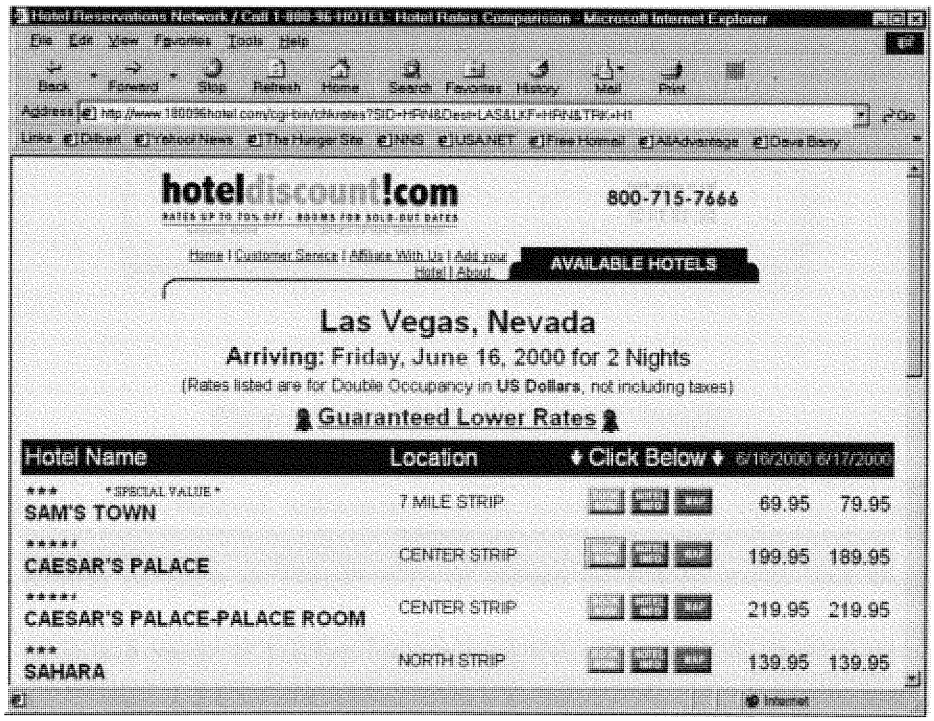

Figure 4-12 Hotel information at 1-800-96HOTEL

However, this list does not give complete details about rooms and prices available at each hotel. In order to retrieve all prices for a hotel we must click "BOOK NOW" button. Then all prices will be displayed, as shown in Figure 4-13. As in the previous example, this means that in order to retrieve complete information wrapper has to traverse at least one additional page for every hotel listed.

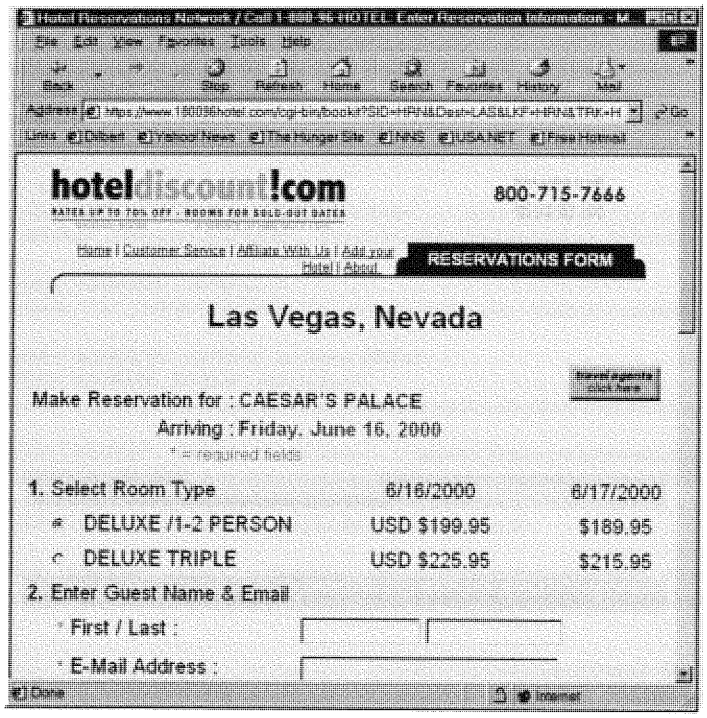

Figure 4-13 Hotel room price details

With hidden data retrieval unnecessary page downloads can also be avoided. The HTML source of page in Figure 4-12 is shown in Figure 4-14. The highlighted portion of it is a piece of hidden form associated with "BOOK NOW" button. The hidden fields of that form contain room name and prices for at least one room not shown to the user but available from the hotel. By extracting this information from the form we can avoid going into a room details page.

The examples above show that discovering and extracting hidden data from HTML is beneficial in speeding data extraction up.

```
<TR> <!-- ## BEGIN Hotel ROW ### -->
        <FORM ACTION="https://www.180096hotel.com/cgi-
bin/bookit?SID=HRN&Dest=LAS&LKF=HRN&TRK=H1" METHOD="POST" NAME="form2">
         <TD BGCOLOR="#EEEEEE" NOWRAP >
        <A HREF=/cgi-bin/starrate?SID=HRN&Dest=LAS&LKF=HRN&TRK=H1><IMG
SRC="/resources/images/ratings/4.5.stars.gif" WIDTH="56" HEIGHT="10" BORDER="0"></A>
<BR>
        <A STYLE="text-decoration: none"
HREF="javascript:document.form2.HotelInfo.value='1';document.form2.submit();"><FONT
FACE="HELVETICA, ARIAL" COLOR="#320065" WEIGHT=200><B>CAESAR'S PALACE
</FONT></A></TD>
        <TD BGCOLOR="#EEEEEE" NOWRAP >
          <FONT SIZE="-1" FACE="HELVETICA, ARIAL">  CENTER STRIP          </FONT>
</TD>
        <TD BGCOLOR="#EEEEEE" NOWRAP ALIGN=CENTER >
        <INPUT TYPE="HIDDEN" NAME="InputData"
VALUE="Year=00&Month=06&Day=16&Nights=02&Adults=02&Children=00&Beds=1&Smoking=N&Year4=
2000">
        <INPUT TYPE="HIDDEN" NAME="HotelData"  VALUE="Seq=010&HotelId=LAS
CAES&HotelName=CAESAR'S PALACE              &Rating=4.5&Location=CENTER STRIP
&Policy=4 DAYS    &NonRefundable=N&Special=N">
        <INPUT TYPE="HIDDEN" NAME="HotelInfo"  VALUE="0">

       <INPUT TYPE="HIDDEN" NAME="RoomData" VALUE="RoomType=DELUXE /1-2 PERSON
&InvType=A&InvCode=A1999999999999&RateType=036&TotalPrice=0426.00&Rates=199.95&Rates=1
89.95">

      <INPUT TYPE="HIDDEN" NAME="RoomData" VALUE="RoomType=DELUXE TRIPLE
&InvType=A&InvCode=A1999999999999&RateType=042&TotalPrice=0483.00&Rates=225.95&Rates=2
15.95">
```

Figure 4-14 HTML source of "Caesar's Palace" item

Scripting simulation

Modern Web sites are often *script-intensive*—in other words, they make wide use of JavaScript and VBScript for a variety of tasks. Some of these tasks, such as various visual effects, are not of particular interest to data extraction. Others, such as navigation, or form validation and submission, are important. The functionality that they provide lets user submit queries to the Web site and travel between pages—something that wrapper is supposed to do. Unfortunately, script execution is beyond the current capabilities of Data Extractor system. Therefore, careful analysis of what particular script does is required of the analyst. Furthermore, if particular script is

important for data retrieval (such as submitting the form or navigating to the pages that contain data) the actions that is performs have to be programmed in the wrapper. This way we will be *simulating* execution of the script without having to execute it.

Another issue with scripts is the *dynamic HTML generation*. Portions of HTML documents can be generated on the fly using JavaScript. Such generation may complicate data extraction because the page that will be parsed and processed is not the same page that the user will see. In such cases serious analysis is also necessary in order to determine a way to extract data of interest without executing the script routines.

Image maps

Analysis similar to analysis of scripting has to be done for HTML *image maps*. Image maps associate parts of an image with URLs. When user clicks on a region inside of the image, browser follows the URL associated with that region. Because clicking on a map region is sometimes is the only way to get to the data of interest, analyst has to determine what URL has to be followed and simulate traveling to that URL in wrapper.

Weak binding

One of the most important techniques in successful wrapper design is *weak binding*. By *binding* we mean wrapper reliance on presence of particular features of the Web

site. In order to identify and extract data from HTML page wrapper has to look for certain features and markers inside the page. Such binding has to be *weak* so that wrapper could withstand minor Web site changes without having to be rewritten or corrected.

This issue is closely related to all site analysis and description techniques discussed so far and it has to be taken into account when applying them. Trying to find a balance between reliable data identification and weak binding has proved to be rather challenging. It is hard to come up with universal recipes on how to do this optimally, as these two tasks are inherently contradicting. The weaker the binding—the less reliable data identification within the site is. The stronger the binding—the better data identification is, and, unfortunately, the greater the chance that the wrapper will not withstand the next site change.

In absence of a clear-cut solution to this problem we suggest, that site analyst tries and identifies the smallest set of site features that will help pinpoint data location inside the site. When selecting these features analyst also has to make sure that they are *content-dependent* (could be searched for) rather than *structure-dependent* (specific positions inside markup trees), as the latter are more likely to change. When such set of features is identified it can be used to create a wrapper that is more tolerant to site changes.

4.3.4 Data output

The data that is extracted from Web sites either is returned to the user directly or fed into the calling application that analyzes and processes it. For wrapper to be integrated into heterogeneous and other database systems its interface has to act as a mini-database system that produces data in response to queries. Therefore, one of the major tasks of wrapper analysis and implementation is the definition of the structure of wrapper output, or *schema.* Schema description and registration is done through a data repository, called *knowledgebase,* that will be described later.

We define schema by specifying names and types for pieces of data, or *fields*, that the Web site provides. Wrapper is then programmed to output data using the field information defined in the schema.

In Data Extractor system wrappers return data row-by-row rather than in bigger chunks. The output of a field is done as soon as all the data that is contained in it is extracted from the Web site. This approach simplifies concurrent execution of wrapper and applications that process and consume data. As soon as the wrapper has extracted and returned the first record to the calling application, data can immediately be filtered, modified or otherwise processed by that application.

There are problems that are associated with schema definition. Data available on the Web site is usually taken from a data source or database internal to that Web site. Only a small portion of that database is displayed to the user. Knowledge about the

database schema is not exposed: no information is given on how data is decomposed into tables internally or what relations exist between tables. Some fields (e.g. internal codes or product IDs) are rarely shown to the user. Finally, the size of the data set displayed is often less than the one that is stored in the database. All these factors make schema definition complicated.

Wrappers in Data Extractor project return data in simple two-dimensional tables similar to ones used in relational databases. There is a single table defined for each wrapper. In the future we plan to use more complex data structures and generate multiple tables from a single wrapper.

4.3.5 Wrapper parameters

Some of the advantages of a wrapper lie in its ability to shield user from Web site complexity, and to generate data in response to requests made through a simple interface. In order to be truly useful, wrapper has to be flexible as well. It has to change its behavior when requested to do so by the user, and be able to execute a class of queries, not only a single query. For example, a wrapper that extracts weather information for major cities within a country, or even for most cities in the world is valuable. On the other hand, wrapper that can only give weather for Tampa, Florida is significantly less valuable.

In order to tell wrapper what kind of information we would like to generate from a particular source we introduced *wrapper parameters*. Wrapper parameters are named values that are passed to the wrapper when it is started.

Parameters contain important pieces of information that wrapper needs to query a Web site. For example, to find all books on Amazon.com that are related to a particular topic, that topic must be specified as a parameter. After wrapper receives and analyzes its parameters it can use them to correctly fill out forms, filter out unnecessary records and so on.

The values are passed to wrappers in two forms—name and single value:

$$\{name_i, value_i\}$$

and name and a set of values:

$$\{name_i, \{value_i1, value_i2, ..., value_in\}\}$$

For the majority of cases single value parameters are sufficient. Multivalue parameters are useful in situations when wrapper has to simulate selection of multiple values in HTML list box, or when the query has to be repeated for a variety of values of the same parameter (e.g. lookup of ticket information to multiple alternative destinations).

4.3.6 Coding and debugging

When the Web site analysis is completed the next step is to create a wrapper application that extracts data from the site. In Data Extractor project wrapper code is implemented in Java [JAV] using functionality of Data Extraction Library. The results of site analysis that describe steps to traverse the site and acquire data are implemented in Java and debugged using *wrapper executor*. Wrapper executor is a Java application that through a simple interface allows programmer to specify wrapper parameters, execute it and step through the wrapper code in any Java development environment. Debug information about network communications and returned data is given as feedback to programmer during wrapper execution.

When the wrapper is debugged and tested, it can be integrated in the Data Extractor system and used in MSemODB framework.

4.3.7 Knowledgebase

To make wrapper known to the Data Extractor system, it has to be registered in a wrapper repository that is called *knowledgebase*. Knowldegebase is a database that contains information about available wrappers, their parameters, output fields, locations and so on. The database administrator maintains knowledgebase and is responsible for keeping it up-to-date.

Knowledgebase stores information necessary to describe all wrappers currently available in the system. Wrapper status is given for each wrapper in order to clearly mark wrappers that are inactive or outdated. Information about the site that wrapper processes is stored. A list of output fields along with their names, types and other information is associated with each wrapper record. Wrapper input parameters are listed by names and types. A wrapper type is listed (at this point, wrappers can be either "Java" or "script" (script wrappers are discussed later in this work)). Finally, for each wrapper its storage location is given, which can be either in the knowledgebase, or in the file system.

The knowledgebase is stored in a Semantic Object-oriented Database System (SemODB). The detailed description of Semantic Modeling Approach to database design can be found in [Ris92]. The simplified schema of the knowledgebase is given in Figure 4-15.

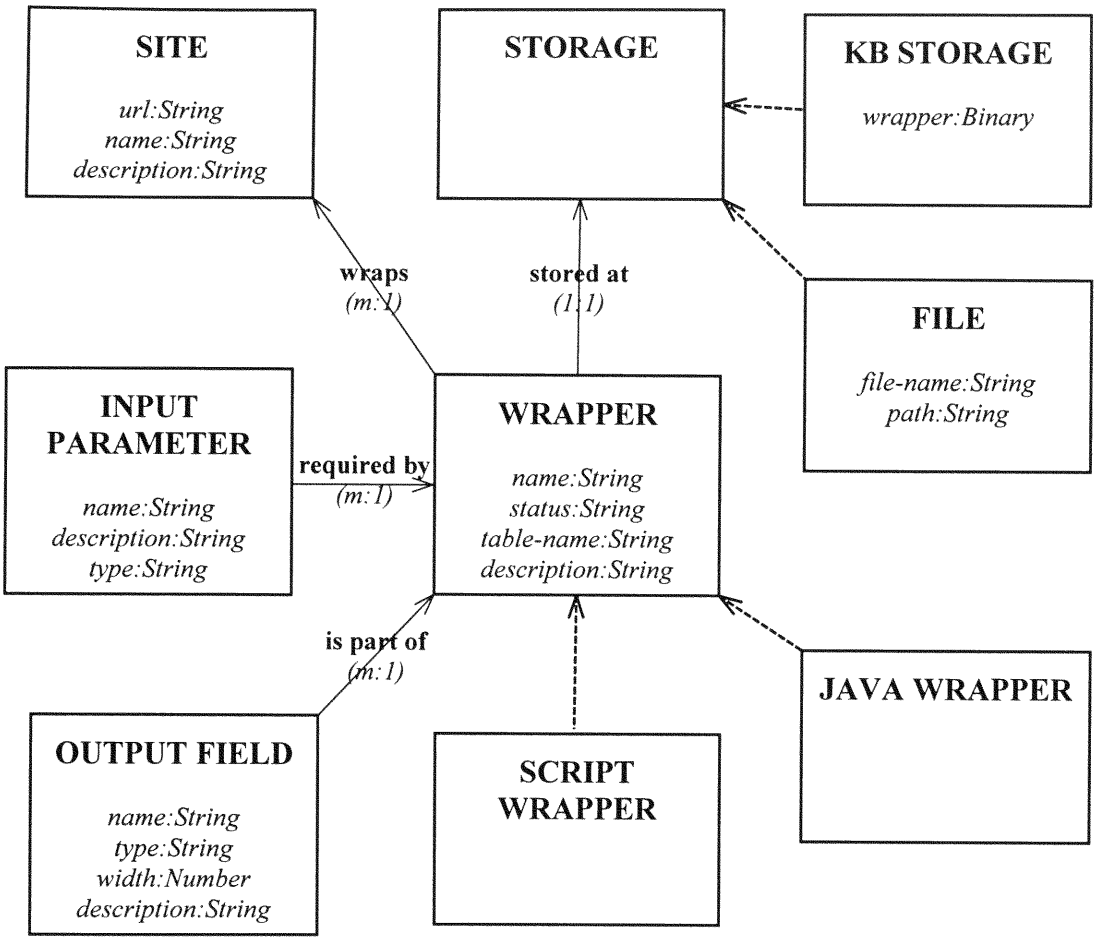

Figure 4-15 Simplified schema of the knowledgebase

Let us list categories, attributes and relations in the schema:

OUTPUT-FIELD — category (This category defines output fields generated by wrapper)

name — attribute of *OUTPUT-FIELD*, range: *String* (*m:1*) (Output field name)

type — attribute of *OUTPUT-FIELD*, range: *String* (*m:1*) (Output field type)

width — attribute of *OUTPUT-FIELD*, range: *Number* (*m:1*) (Output field width in characters or decimal precision)

description — attribute of *OUTPUT-FIELD*, range: *String* (*m:1*) (Output filed description)

INPUT-PARAMETER — category (This category defines wrapper input parameters)

name — attribute of *INPUT-PARAMETER*, range: *String* (*m:1*) (Parameter name)

description — attribute of *INPUT-PARAMETER*, range: *String* (*m:1*) (Parameter description)

type — attribute of *INPUT-PARAMETER*, range: *String* (*m:1*) (Parameter type)

SITE — category (This category describes site that is processed by the associated wrapper)

url — attribute of *SITE*, range: *String* (*m:1*) (Site URL address)

name — attribute of *SITE*, range: *String* (*m:1*) (Short descriptive site name)

description — attribute of *SITE*, range: *String* (*m:1*) (Site description)

WRAPPER — category (This category defines facts about the wrapper)

name — attribute of *WRAPPER*, range: *String* (*m:1*) (Wrapper name)

status — attribute of *WRAPPER*, range: *String* (*m:1*) (Wrapper status)

table-name — attribute of *WRAPPER*, range: *String* (*m:1*) (Name of the table generated by this wrapper)

description — attribute of *WRAPPER*, range: *String* (*m:1*) (Wrapper description)

STORAGE — category (This category defines wrapper storage)

FILE — subcategory of *STORAGE* (This category defines wrappers that are stored in the file system)

file-name — attribute of *FILE*, range: *String* (*m:1*) (Name of the file in which wrapper is stored)

path — attribute of *FILE*, range: *String* (*m:1*) (Path to wrapper file)

KB-STORAGE — subcategory of *STORAGE* (This category facilitates storage of wrappers in the knowledgebase)

wrapper — attribute of *KB-STORAGE*, range: *Binary* (*m:1*) (Stored wrapper)

JAVA-WRAPPER — subcategory of *WRAPPER* (This category defines Java-based wrappers)

SCRIPT-WRAPPER — subcategory of *WRAPPER* (This category defines wrappers created using a scripting language)

wraps — relation from *WRAPPER* to *SITE* (*m:1*) (This relation associates wrappers with sites they process)

is-part-of — relation from *OUTPUT-FIELD* to *WRAPPER* (*m:1*) (This relation connects output fields to the wrapper)

required-by — relation from *INPUT-PARAMETER* to *WRAPPER* (*m:1*) (This relation connects input fields to the wrapper)

stored-at — relation from *WRAPPER* to *STORAGE* (*1:1*) (This relation associates wrapper with the place it's stored at)

4.4 Data Extraction Library

4.4.1 Overview

One of the most important parts of the Data Extractor project is the Data Extraction Library. Its purpose is to provide Web document retrieval, parsing and data extraction functionality for wrappers.

The wrappers in Data Extractor project are built in Java in order to ensure their multi-platform availability, compactness and portability. Java is often being named as a language of choice for building Web-based applications, owing, in part, to its versatile text processing and network communication functionality that comes standard in Java function libraries. Java programs are often more compact, secure and well organized even in comparison to their C++ counterparts.

Why build an additional library of functionality and not just use the core libraries of functions? Unfortunately Java portability comes at a price of decreased performance. This often discourages authors of network tools that require high-performance, such as browsers, Web servers and Web crawlers, from using Java, because it performs worse than C++ on such tasks. However, as it was shown in [HN99] and done in Data Extraction Library, performance problems can be remedied. Another reason for implementing a custom library is the fact that Data Extractor project required a large number of specialized functionality, such as Web page retrieval and parsing, HTML

tree traversing, and so on. These functions are not provided in the standard set of core library routines.

Yet another concern when developing the library was "reinvention of the bicycle." Why build HTML parsing functionality or and HTTP client functionality when such functions are available in dozens of commercial packages sold on the market today? We have, in fact, evaluated several such libraries, both commercial and free, and determined that none of them satisfied all the library requirements:

- *Compactness.* Most libraries had much more functionality than was necessary and as a result the code size was often quite big. In comparison, our HTTP and HTML parsing functionality is only around 50 Kbytes and, therefore, is easy to transmit even over slow connections. Compactness is one of the key features for distributing data extraction functionality to the client side (discussed further).

- *Modularity.* Some of the libraries that we reviewed could not be decomposed in separate compact packages to be used independently, but rather were a monolithic inseparable whole. Often such libraries, along with core functionality contain utility functions that are rarely, if ever, used, but cannot be removed easily. Our library was designed with little interdependence between packages and contains only the essential functionality.

- *Extensibility.* Some of the libraries we have seen could not be easily extended to accommodate additional protocols or support of new technologies. By having control over the source code of the library we can easily add new features.

- *Browser simulation.* The library had to support a full set of functionality to simulate browser interaction with the Web sites. It had to be able to fill out and submit forms, follow links, accept cookies, and have other functionality that is commonly expected in a browser. Not all libraries that we saw supported this.

- *HTML parsing, traversing and searching functionality.* Requirements for such functionality in our case were dissimilar to most approaches taken by HTML libraries. As a result we had to implement our own routines for HTML processing.

The Data Extraction Library provides a set of Java interfaces that facilitate building of wrappers. They can be separated into four main groups of functionality: *page retrieval, HTML processing, data representation* and *wrapper interface*. Let us review these groups.

4.4.2 Page retrieval functionality

Fast and reliable page retrieval from the Internet is crucial to wrapper operation. In Data Extraction Library page requesting and retrieval is done through *sessions*. A session is a conversation with a Web server, the result of which is a stream of data.

This data can be read as a stream or (when data is actually an HTML page) can be converted into a *document*. Sessions are more than simple HTTP request/response pairs. They provide rich functionality for building of requests and modification of parameters (similar to filling out forms in browsers). The *HTTPSession* class is responsible for session creation and manipulation.

```
class HTTPSession
      HTTPSession(URL address);
      void open();
      void close();
      HTTPParameterList getParameterList();
      void setParameterList(HTTPParameterList list);
      void useSubmit(String name);
      InputStream getInputStream();
      HTTPConnection getHTTPConnection();
      int getConnectionMethod();
      void setConnectionMethod(int method);
      URL getURL();
      URL getConnectionURL();
```

Figure 4-16 Class HTTPSession

When we create a session, we specify the URL of the page to be retrieved. Session has a parameter list associated with it that can be fully modified and manipulated. Even parameters embedded in the URL can be accessed and changed.

Session also has a *connection* and a *connection type* associated with it. Connection is a module that encapsulated an "HTTP conversation" between the Web site and the wrapper. The standard Java library connection functionality was not sufficiently flexible, powerful and fast for the purposes of creating efficient wrappers. Because of this we implemented custom connection functionality and encapsulated it in *HTTPConnection* class.

```
class HTTPConnection
      HTTPConnection(URL address);
      void connect();
      void close();
      void setRequestProperty(String name, String value);
      String getRequestProperty(String name);
      int getResponseCode();
      InputStream getInputStream();
      OutputStream getOutputStream();
      int getHeaderFieldNum();
      String getHeaderField(int field);
      String getHeaderField(String field);
      URL getURL();
      URL getOriginalURL();
      void setConnectionMethod(int method);
      int getConnectionMethod();
      String getHeaderFieldKey(int key);
```

Figure 4-17 Class HTTPConnection

Connection functionality allows wrapper to take full control of composition and submission of HTTP requests, as well as specification of request and response parameters. The most widely used connection methods—GET and PUT—are supported. Connection functionality supports transparent redirection, and keeps track of both initial and redirected addresses. Cookies are also fully implemented.

```
class HTTPParameter
      void setValue(String value);
      String getValue();
      void setValues(Vector values);
      void setParameterType(int type);
      int getParameterType();
      Vector getValues();
      void setValidValues(Vector values);
      Vector getValidValues();
      void setDefaultValues(Vector values);
      Vector getDefaultValues();
      void addValue(String value);
      void removeValue(String value);
      void removeAllValues();
      void setName(String name);
      String getName();
```

Figure 4-18 Class HTTPParameter

HTTP parameters that are commonly specified as part of URL are implemented in a separate class, called *HTTPParameter*. HTML parameters can have single values or multiple (in case of checkboxes and multiple selection lists) values. Parameters can also have sets of valid values (e.g. the values that are available in the HTML drop-down list) and default values (e.g. the initial value of the text input filed). Full support for all HTML parameter types and their functionality is implemented in HTTPParameter.

```
class HTTPParameterList
      void removeParameter(String name);
      String getValue(String name);
      void addParameter(HTTPParameter param);
      void addParameter(String name, String value);
      void addParameter(String name, Vector values);
      Vector getValues(String name);
      HTTPParameter findByName(String name, int startIndex, boolean
caseInsensitive);
      int setParameter(String name, String value);
      int setParameter(String name, Vector value);
      void removeAllParameters();
```

Figure 4-19 Class HTTPParameterList

HTTPParameterList handles lists of HTTPParameters. Its functionality includes adding, removing, modifying and searching for parameters. This class is also responsible for automatic encoding and decoding of parameters and their values for transmission to and from the Web site.

4.4.3 HTML processing functionality

The contents of the retrieved HTML pages are stored internally in a form of a tree that consists of markup and text elements. Each HTML page that is retrieved from the

Web is automatically converted into an HTML tree. The parsing and document storage functionality was built to parse HTML and any other markup language that is based on SGML. It can be easily modified to store and process XML documents.

The HTML parser that we implemented cannot, however, be called *general-purpose parser* because it is lacking the *validation* mechanism. The validation functionality is not necessary in Data Extraction Library, because we only need to store documents in memory, traverse them and search them for information. Little attention is paid to meanings of tags. This makes parser more robust, because it does not refuse to process the document that is syntactically incorrect and is able to build a tree (even incorrect one) for any document. Correctness of the tree does not concern us because we only need a structure that can be traversed and that is a representation of the original document. It is not our job to try to interpret this structure because we do not render HTML on the screen - we only need to traverse it to extract data from it.

```
class TaggedStream
      TaggedStream(HTTPSession session);
      TaggedStream(InputStream stream);
      void close();
      long getBytesRead();
      MarkupElement read();
      int getPageLength();
      URL getURL();
      void setURL(URL address);
```

Figure 4-20 Class TaggedStream

When the document is received from the network or from the file it is automatically presented as a stream of *markup elements*. Markup elements (described in greater detail further in this document) are the syntactic pieces that, when combined, create an HTML or other SGML-like document. This stream is processed by class

TaggedStream. It is automatically converted into a tree document structure. For the purposes of some applications it can be read as it is, without converting it into a document. This may be necessary for some simple searches or other operations on very large documents.

```
class TaggedDocument
      TaggedDocument(URL address);
      TaggedDocument(HTTPSession session);
      TaggedDocument(TaggedStream stream);
      Enumeration findAttribute(String text, int parameters);
      Enumeration findAttribute(String text, String tagName, int
parameters);
      Tag getRoot();
      URL getBaseURL();
      void setBaseURL(URL address);
      URL getURL();
      void setURL(URL address);
      Enumeration find(String value, int parameters);
```

Figure 4-21 Class TaggedDocument

When the document is retrieved, a *TaggedDocument* object is constructed. It contains the tree representation of the document, like the one shown in Figure 4-3. The document can be created from a document on the Internet or from a local file. The object is initialized by specifying the document's URL, or, in more complex cases, by specifying the session through which the document is supposed to come. Documents can be queried for their source URLs, and in some cases, for base URLs that are used as references to relative objects inside documents, such as pictures. Powerful search functions are implemented in the TaggedDocument. They are similar to tree search functions described later in this work.

As it was already mentioned, the basic element into which every document is split is a *markup element*. Markup elements are text, comments and tags, to name the most

common. In Data Extraction Library markup elements are represented by class *MarkupElement*. It implements behavior common to all types of elements.

```
class MarkupElement
      boolean isDescendantOf(Tag parent);
      Tag getParent();
      MarkupElement nextElementInSubtree(Tag tag);
      MarkupElement nextElementFlat();
      MarkupElement getLeftSibling();
      MarkupElement getRightSibling();
      Enumeration findInSubtree(String value, int parameters);
      Enumeration findFlat(String value, int parameters);
      Enumeration findAttributeFlat(String value, int parameters);
      Enumeration findAttributeInSubtree(String value, int
parameters);
      Enumeration findAttributeInSubtree(String value, String tag,
int parameters);
      TaggedDocument getParentDocument();
```

Figure 4-22 Class MarkupElement

Each markup element is a node in a document tree. *MarkupElement* class implements functionality for traversing this tree. Use it to find parents, children and siblings of the element, as well as its parent document. Class provides functions for stepping through the tree levels as well as traversing the document linearly.

Search functionality is also implemented in this class. We can search for pieces of text in tags; tag attribute names and values; text and comments. Searching can be done linearly and in subtrees. Search can be case sensitive and insensitive, and can be done for pieces of text that "begin", "end", "contain" or are "exactly like" the search string.

When the search is initiated the search function returns a *cursor* (called an *enumeration)* to the calling application. Using this enumeration the application can step through the found instances.

Functionality specific to each type of element is implemented in the descendants of *MarkupElement*. The hierarchy of these classes can be seen in Figure 4-23.

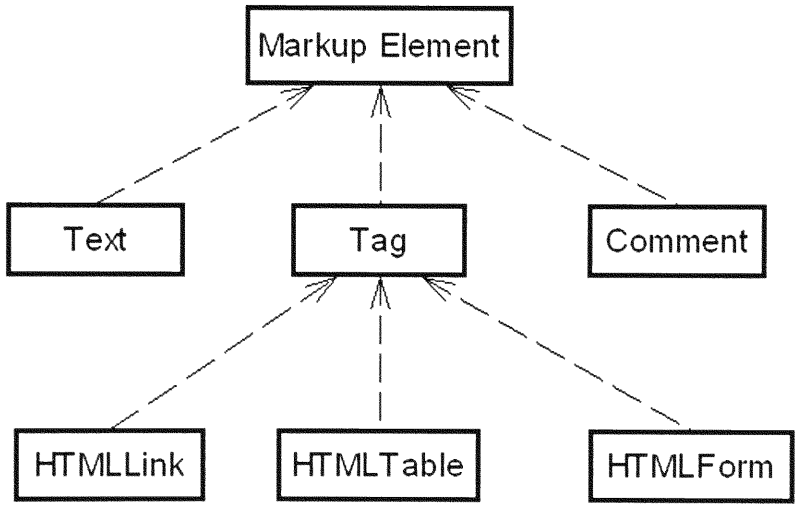

Figure 4-23 Class hierarchy for markup elements

```
class Text extends MarkupElement
      String getText();
      void setText(String value);
```

Figure 4-24 Class Text

Text elements are implemented in class *Text*. This class inherits behavior from the *MarkupElement* class and adds two simple assignment and query routines. Text elements (as well as comments) can only be leaves in the document tree and their behavior is very simple.

```
class Comment extends MarkupElement
      String getText();
      void setText(String value);
```

Figure 4-25 Class Comment

Comment class implements HTML comments and its behavior is very similar to that of the *Text* class.

```
class Tag extends MarkupElement
      Vector extractText();
      String getAttributeValue(String name);
      Vector getAttributes();
      void setAttributes(Vector values);
      Vector getChildren();
      void setChildren(Vector children);
      void addChild(MarkupElement child);
      void setClosingTagExists();
      boolean closingTagExists();
      String getName();
      void setName(String name);
```

Figure 4-26 Class Tag

Tag class implements HTML tags. In the development of parsing algorithms no differentiation (with few minor exceptions) was made between different HTML tags—they are all treated equally and stored in objects of the same class. This simplifies the parsing algorithm and provides opportunity to use the Data Extraction Library for XML, SGML and other SGML-based markup languages.

Use this class to modifies tag name and attributes, and traverse its children. It simplifies extraction of all text elements that are children of a particular tag, making data extraction easier.

There are several types of tags that require implementation of special functionality:

```
class HTMLLink extends Tag
      HTTPSession getHTTPSession();
```

Figure 4-27 Class HTMLLink

HTMLLink class implements HTML link tags, namely A and LINK. They provide links to other documents. A special function is implemented that lets programmer construct an *HTTPSession* out of a link, which then can be used to retrieve the document to which the link points. This function significantly simplifies traveling between documents on a Web site.

```
class HTMLForm extends Tag
      HTTPSession getHTTPSession();
```

Figure 4-28 Class HTMLForm

HTMLForm implements functionality of HTML forms. Use this class to construct a session from a form. This session will contain all of the form's parameters, including hidden ones, along with their valid and default values. Such session can then be used to simulate the form submission. Parameters of the session can be modified to simulate modification of form fields.

```
class HTMLTable extends Tag
      DataTable getTable();
      DataTable getTable(boolean firstRowHeaders);
```

Figure 4-29 Class HTMLTable

HTMLTable class simplifies data extraction from the HTML tables. Data on Web sites is often stored in HTML tables. In such cases we can extract *DataTable* object (to be described later) using HTMLTable, and return it to the calling application.

```
class TagAttribute
      String getValue();
      void setValue(String value);
      Tag getParent();
      void setParent(Tag parent);
      String getName();
      void setName(String name);
```

Figure 4-30 Class TagAttribute

Tag attribute functionality is implemented in class *TagAttribute*. It provides functionality for working with attribute names and values.

```
class MarkupEnumeration
      Object nextElement();
      boolean hasMoreElements();
```

Figure 4-31 Class MarkupEnumeration

The two classes that facilitate stepping through the search results are *MarkupEnumeration* and *TagAttributeEnumeration*. *MarkupEnumeration* steps through found elements of the document tree.

```
class TagAttributeEnumeration
      Object nextElement();
      boolean hasMoreElements();
```

Figure 4-32 Class TagAttributeEnumeration

TagAttributeEnumeration searches through attribute names and values.

4.4.4 Data representation functionality

When data is extracted from the Web it is stored in a standard way and shipped to consumer. There are two ways to store data in Data Extraction Library—in rows and in tables. Tables are collections of rows.

```
class DataVector
      DataVector();
      DataVector(Object[] values);
      DataVector(Vector values);
      int indexOfStringValue(String value);
      void insertElementAt(Object value, int position);
      void removeAllElements();
      int size();
      Object elementAt(int position);
      void setElementAt(Object value, int position);
      void addElement(Object value);
      void removeElementAt(int position);
      boolean isEmpty();
```

Figure 4-33 Class DataVector

Rows are implemented by instances of class *DataVector*. DataVector stores text strings, although it potentially can store objects of any kind. Rows can grow and shrink without limitations, elements can be inserted anywhere in the row or replace other elements. DataVector also provides functionality for searching for substrings inside of elements.

```
class DataTable
      void setColumnNames(DataVector names);
      DataVector getColumnNames();
      Object getValue(int row, int column);
      void setValue(Object value, int row, int column);
      void removeRowAt(int row);
      DataVector rowAt(int row);
      DataVector columnAt(int column);
      void removeColumnAt(int column);
      void setColumnName(int column, String name);
      String getColumnName(int column);
      void removeAllElements();
      void addColumn(DataVector column);
      void addColumn(DataVector column, String name);
      void addRow(DataVector row);
      void insertRowAt(DataVector row, int rowPos);
      int getHeight();
      void insertColumnAt(DataVector column, int columnPos);
      void insertColumnAt(DataVector column, String name, int
columnPos);
      int getWidth();
      int getColumnIndex(String name);
      DataTable subTable(int rowBegin, int columnBegin, int rowEnd,
int columnEnd);
```

Figure 4-34 Class DataTable

DataTable implements functionality for creating and modifying tables of data. Rows and columns can be added, removed and modified anywhere in the table. The table dimensions are automatically adjusted to maintain its rectangular shape. Columns can be labeled with descriptive names. Applications can extract "subtables" of any size from tables to form new tables.

4.4.5 Wrapper interface functionality

Wrapper interface functionality provides a simple communication and control mechanism that simplifies implementation of wrappers and transmission of data.

```
class DataExtractor
      void returnRow(DataVector vector);
      void returnTable(DataTable table);
      void getData(Parameters parameters);
```

Figure 4-35 Class DataExtractor

Every wrapper implements a *getData* function through which it is automatically integrated into the wrapper framework. Wrapper execution starts when the controlling environment executes this function and ends when the execution of this function ends, either naturally, or because of an error. While working, wrapper returns data to the calling application using *returnRow* and *returnTable* functions.

```
class Parameters
      void set(String name, Object value);
      Object get(String name);
```

Figure 4-36 Class Parameters

Every time the wrapper is executed its behavior is modified through parameters. Parameters are transmitted to wrapper inside the *Parameters* class. This class assigns names to values (both single and multiple) and provides access to these values.

Exceptions provide convenient mechanism for notifying the calling application about the exceptional situations and errors that occur inside of wrapper.

```
class SiteFormatException
```

Figure 4-37 Class SiteFormatException

The most common class of errors occurs when the Web site structure changes. In the dynamic environment of the Web such changes are inevitable and usually it is not a question of "if" but "when" the site will change. When the wrapper detects that the site format is different from what it expects it to be it is often impossible to continue. In such cases wrapper should throw a *SiteFormatException* exception to notifying the wrapper controller of the site format error. A common automated response to such error on the system side is termination of wrapper execution. After such wrapper is stopped it is marked as "outdated" in the knowledgebase and systems administrator is notified.

```
class SiteErrorException
```

Figure 4-38 Class SiteErrorException

SiteErrorException error is used to tell the calling application that the fatal error has occurred and that further data extraction is impossible. Such errors are not generated frequently and should be used to indicate a genuinely serious problem. Most errors

can be safely ignored and wrapper can simply generate an empty result set to indicate unavailability of data.

```
class BadInputException
```

Figure 4-39 Class BadInputException

BadInputException is thrown when wrapper detects an error in parameters that were supplied to it or when the required parameters or values are omitted.

4.4.6 Challenges

Some difficulties were encountered while implementing the HTML parser functionality and page retrieval.

Syntax errors

In the course of building wrappers for a variety of sites we found a large portion of HTML pages to be syntactically incorrect. A shocking estimated 90% of all Web pages on commercial Web sites we have analyzed so far had syntax errors. The high quality of modern Web browsers is partially to blame for such a high percentage of incorrect HTML documents. In order to accommodate the widest possible variety of Web sites and make an effort to display any Web page, no matter how badly structured, the browsers were made extremely forgiving. HTML page author can miss or mismatch closing tags, put end tags in the wrong order, not close comments or make other mistakes—and browser will not alert her to the problem.

Faulty HTML is often created by inexperienced or careless designers. Sometimes errors are introduced by buggy or poorly written HTML editor tools that generate incorrect HTML or do not enforce correct syntax. Browser forgiveness, while being a convenience to a Web surfer, condones bad HTML design habits and makes the job of HTML parsing much harder.

Syntax errors, fortunately, had little impact on the work of our parsing functionality. Only for one site out of over a hundred analyzed a wrapper could not be built due to hopelessly incorrect HTML pages. For all other sites the parser did a satisfactory job of building markup trees out of pages. Such trees weren't always "correct" in the sense that it is impossible to build a correct tree for an incorrect document. However, they provided a representation of the document that is adequate for traversing and information extraction. The job of HTML parser in our case is merely to attempt a split the text of an HTML document into a markup tree; it is not required to assure that this tree is valid syntactically or that it represents a correct interpretation of the HTML source. With the exception of link and form tags semantic meanings of tags are ignored.

Slowness of core network functionality

Java is an interpreted language and this takes its toll on performance of time-critical routines in the standard Java libraries. Slowness of the network functionality in Java contributed the most to the overall slowdown of the wrapper operations. In our tests between 40% and 70% of the overall wrapper execution time was spent connecting to

Web servers, sending requests for pages and receiving pages. In comparison only about 5-10% of the time was spent on parsing and processing, and the rest—on operations associated with data extraction and wrapper execution control.

Network operations were somewhat sped up when standard Java HTTP protocol implementation was re-written using sockets. This way a lot of unnecessary operations were eliminated, making the implementation leaner, faster and flexible enough to accommodate redirects, cookies and other protocols.

In the future we expect the slowness of the network functionality to remain one of the stumbling blocks for successful wrapper implementation. Slowness of such operations is not inherent to Java—the same operations implemented in C++ for comparison purposes performed only insignificantly faster. This decreased performance significantly reduces usefulness of the applications written using wrapper technology because the speed of data set generation is slow and sometimes insufficient for satisfactory user experience.

Scripted Web sites

The majority of Web sites today use some kind of scripting. The two commonly used scripting languages are JavaScript (or JScript) and VBScript. Most of the Web page scripts are used for decoration purposes only and carry no semantics. However, in some cases scripts assist navigation or validate and fill forms.

In our opinion attempting to parse and execute such scripts in the course of data extraction is an interesting technological task, but it does not seem to facilitate such extraction in a significant way. All scripted functionality for page navigation and processing of forms can be duplicated in wrapper code. This, to our mind, is less resource and time consuming task than building a script interpreter. At this point in time it was decided against implementing script interpretation functionality in the Data Extraction Library.

Applets, Plug-ins, ActiveX controls

A significant number of Web sites use Java Applets, ActiveX controls and Netscape Plug-ins to provide users with functionality that cannot be provided through scripting or to display data that cannot be displayed in HTML or images. It is clear that extracting data and navigation information from these mini-applications is significantly harder that doing so with embedded Web page scripts. Attempting to do this is a very large project in itself. Also, very few Web sites provide data of interest to data extraction application using these technologies due to their low portability and demand on computer resource. With the increasing number of Web users accessing the Web through low-performance, low capability devices such as Personal Digital Assistants (PDAs) and cellular phones we do not expect popularity of these technologies to increase. We decided not to pursue the path of analyzing and attempting to extract data from these browser mini-applications.

Secure Sockets Layer (SSL)

Implementation of the SSL support for the library is an important issue. Many Web sites today support this protocol for secure client-server communication. For Data Extraction Library to be able to access these Web sites, SSL protocol support has to be implemented inside the library. The SSL protocol was studied and the functionality that establishes a secure connection was added to the library. Unfortunately problems of different nature precluded us from using it or distributing it. Currently the encryption technology used in SSL and on a majority of sites supporting SSL is patented by RSA Inc. (http://www.rsa.com), and licensing fees have to be paid for its use and distribution. This hampered any further work on the issue. The issue of SSL support might be solved soon, because relevant patents are set to expire in Fall 2000.

During Web site analysis we also determined that the vast majority of the information providers either offer two ways of accessing information (both with SSL and without it) or offer portions of information that are free without SSL. Because Data Extractor technology is primarily geared towards the free access to information this allowed us to avoid having to incorporate SSL into our software. In the future, however, SSL has to be supported, so that the wrappers would be able to cover the same set of Web sites the browsers usually cover.

4.5 Sample wrapper

So far we have described the process of Web site selection and analysis, issues involved in wrapper construction, and, finally, the data extraction functionality available through the Data Extraction Library. Let us now apply this knowledge and step through the process of wrapper analysis and construction for a simple Web site.

4.5.1 Sample Web site

The Web site that we will be analyzing is a fictitious online bookstore called "Makebelieve Bookstore". We decided not to analyze the real world Web site because the details of such site and some of the problems with its analysis could have been distracting. Instead of concentrating on the complete outline of process and the functionality it could have been easy to get lost in discussion of details that, while important, could have prevented us from seeing the big picture.

The functionality of the bookstore is very simple. The user is first greeted with a homepage that contains book database query form (see Figure 4-40). Using this form bookstore visitor can search for Computer Science books. Search can be performed by a partial book title, publisher and price range. To find a book at least one publisher has to be selected. Available price ranges are "less than $20", "less than $50" and "less than $100".

When user selects the search criteria and presses "Find books" button search request is submitted to the bookstore Web server. The server application finds books that have specified features in its internal database and displays results to the user (see Figure 4-42). If more than 10 books are found the results are displayed on multiple linked pages that user can go through by selecting "Next" and "Previous" links. When no books match the specified criteria bookstore responds with a message "No books found."

As we can see this is a very simple site that is easy to analyze. At the same time it bears all the characteristics of the sites we usually select for data extraction:

- *Search form*. We know that most Web sites today use HTML forms that let user search their internal databases for information.

- *Error and success messages*. Search results are clearly identified by messages, whether it is the number of records found, or indication that the query did not generate any results.

- *Multiple records of the same structure, containing different data types*. The kinds of data that are generated are diverse, including text, currency and date information. Data is organized in sets of records that (we assume) closely resemble the records of the database where the Web site is getting its information. Multiple records are available, all having the same structure.

- *Multiple result pages*. Data is shown on multiple pages and can to be accessed by navigating the links or submitting forms.

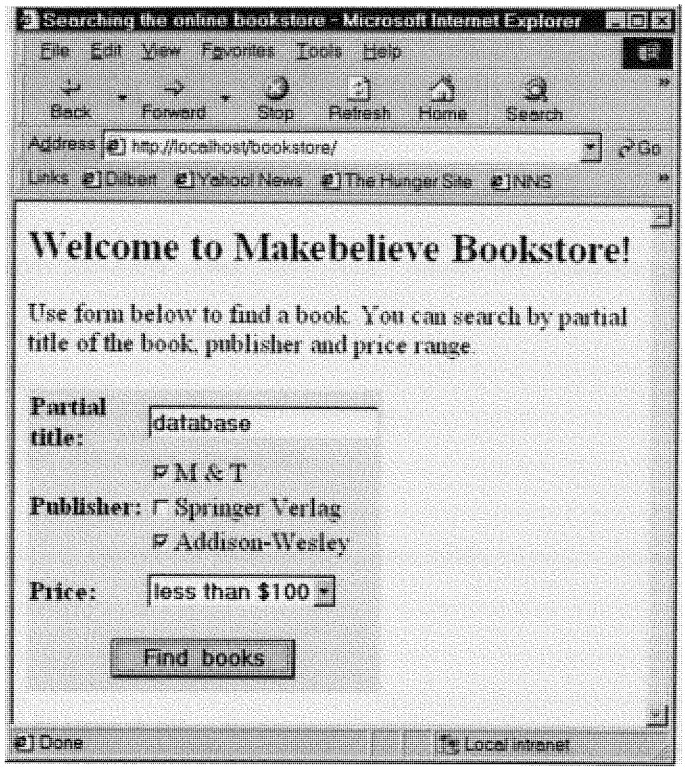

Figure 4-40 Bookstore query form

4.5.2 Site analysis

Before implementing the wrapper it is important to do a preliminary analysis of the site's features. As we know from previous sections such analysis can significantly speed up development and improve wrapper performance.

```
<HTML>
<HEAD><TITLE>Searching the online bookstore</TITLE></HEAD>
<BODY>
<H2>Welcome to Makebelieve Bookstore!</H2>
<P>Use form below to find a book. You can search by partial <BR>
title of the book, publisher and price range. </P>

<!-- Book search form -->
<FORM ACTION="books.jsp" METHOD="GET" NAME="search_form">
<TABLE WIDTH="35%" BORDER=0 BGCOLOR="#D4E3F1" CELLSPACING=0 CELLPADDING=2>
<TR>
  <TD HEIGHT=50><B>Partial title:</B></TD>
  <TD HEIGHT=50><INPUT TYPE="TEXT" NAME="title"></TD>
</TR>
<TR>
  <TD ROWSPAN=3><B>Publisher:</B></TD>
  <TD><INPUT TYPE="CHECKBOX" NAME="publisher" VALUE="MT">M & T</TD>
</TR>
<TR><TD><INPUT TYPE="CHECKBOX" NAME="publisher" VALUE="SV">Springer Verlag</TD></TR>
<TR><TD><INPUT TYPE="CHECKBOX" NAME="publisher" VALUE="AW" CHECKED>Addison-Wesley</TD>
</TR>
<TR>
  <TD HEIGHT=50><B>Price:</B></TD>
  <TD HEIGHT=50>
    <SELECT NAME="price">
      <OPTION VALUE="20" SELECTED>less than $20
      <OPTION VALUE="50">less than $50
      <OPTION VALUE="100">less than $100
    </SELECT>
  </TD>
</TR>
<TR><TD ALIGN="CENTER" HEIGHT=50 COLSPAN=2><INPUT TYPE="SUBMIT" VALUE="Find
books"></TD> </TR>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Figure 4-41 Source of bookstore query form

We begin by analyzing the data entry form. We see that it is the only form in the page and can be found by simply looking for the *FORM* tag (see Figure 4-41). We analyze form fields and decide which of them we will modify and which will be left unchanged. We try various combinations of input parameters to determine how Web site responds to a variety of entered values. In particular, we are interested in how Web site responds to queries that bring no results (e.g. when the partial book title cannot be matched in the database). We determine that when we search bookstore for a title that is guaranteed not to exists (e.g. "@#$%") Web site responds with a page containing string "No books found" (similar approach to recognizing empty result pages is described in [DEW97]).

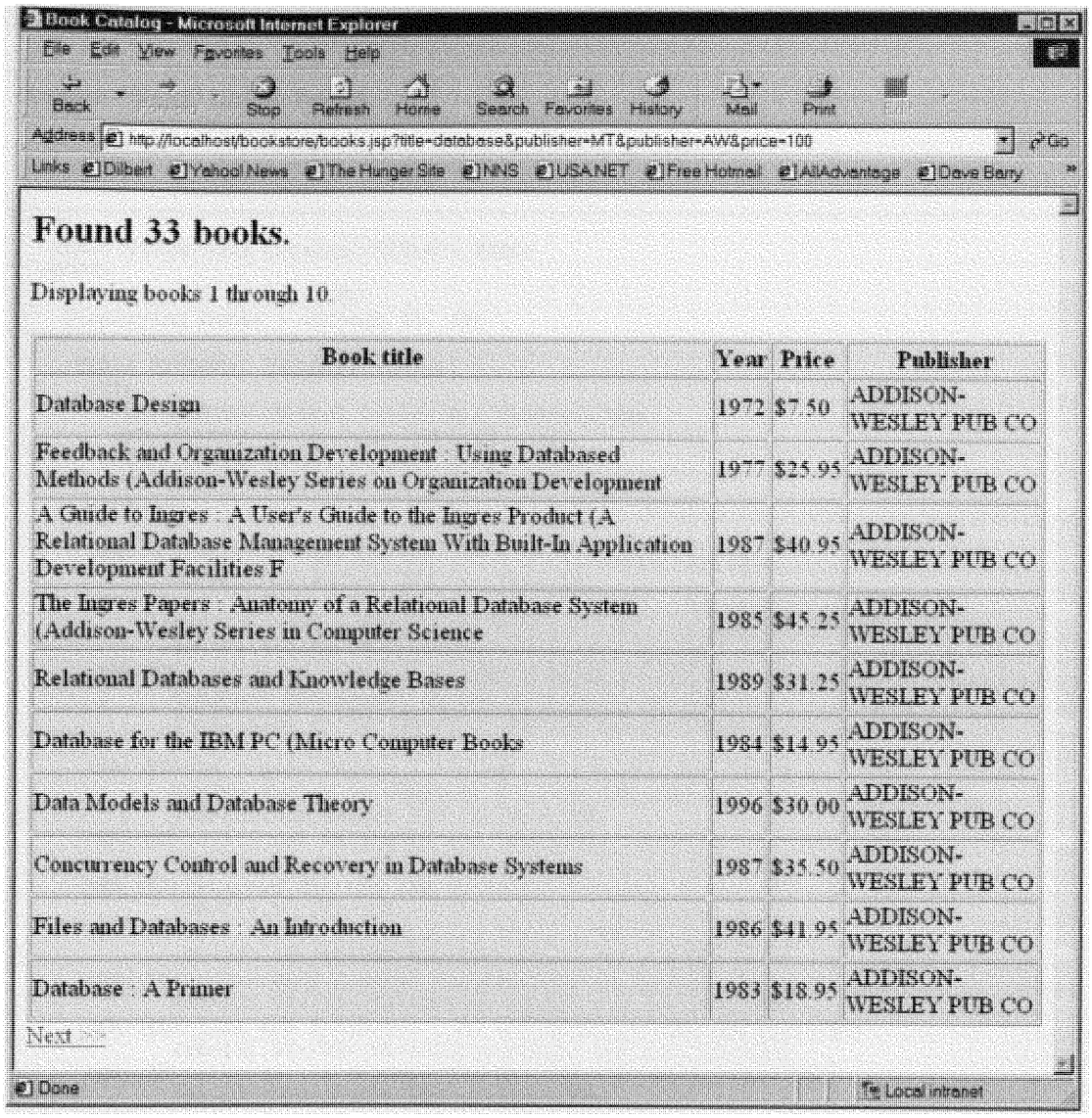

Figure 4-42 Bookstore search results page

We also study cases when queries bring one page of results and when results do not fit on a single page and are scattered across multiple pages. It turns out that the difference between such cases is minimal. In Figure 4-42 we see a common results page. At the bottom of the page we see a link called "Next". When all results fit on one page no such link is present. When results occupy several pages such link is

present and in order to see all results we must follow it until a page that doesn't contain "Next" link is found.

```
<HTML>
  <HEAD><TITLE>Book Catalog</TITLE></HEAD>
  <BODY>
    <H2>Found 33 books.</H2>
    Displaying books 1 through 10. <P>

    <TABLE BORDER=1 BGCOLOR="#D4E3F1">
      <TR>
        <TD><CENTER><B>Book title</B></CENTER></TD>
        <TD><CENTER><B>Year</B></CENTER></TD>
        <TD><CENTER><B>Price</B></CENTER></TD>
        <TD><CENTER><B>Publisher</B></CENTER></TD>
      </TR>

      <TR><TD>Database Design</TD><TD>1972</TD><TD>$7.50</TD>
            <TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Feedback and Organization Development : Using Databased Methods
            (Addison-Wesley Series on Organization Development</TD><TD>1977</TD>
            <TD>$25.95</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>A Guide to Ingres : A User's Guide to the Ingres Product (A Relational
             Database Management System With Built-In Application Development
             Facilities F</TD><TD>1987</TD><TD>$40.95</TD><TD>ADDISON-WESLEY PUB
             CO</TD></TR>
      <TR><TD>The Ingres Papers : Anatomy of a Relational Database System (Addison-
             Wesley Series in Computer Science</TD><TD>1985</TD><TD>$45.25</TD>
             <TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Relational Databases and Knowledge Bases</TD><TD>1989</TD>
             <TD>$31.25</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Database for the IBM PC (Micro Computer Books</TD><TD>1984</TD>
             <TD>$14.95</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Data Models and Database Theory</TD><TD>1996</TD>
             <TD>$30.00</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Concurrency Control and Recovery in Database Systems</TD><TD>1987</TD>
             <TD>$35.50</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Files and Databases : An Introduction</TD><TD>1986</TD>
             <TD>$41.95</TD><TD>ADDISON-WESLEY PUB CO</TD></TR>
      <TR><TD>Database : A Primer</TD><TD>1983</TD><TD>$18.95</TD>
             <TD>ADDISON-WESLEY PUB CO</TD></TR>
    </TABLE>

  <A HREF="/bookstore/books.jsp?publisher=MT&publisher=AW&price=100&title=database
&start=11">Next &gt;&gt;</A>

  </BODY>
</HTML>
```

Figure 4-43 Bookstore search results page source

Finally, we determine how we can extract data from the pages. Let us study the source of the page with results (see Figure 4-43). The table that contains the book information is the first and only one in the page. It has 4 columns and a title row, which has to be skipped during data extraction. The format of the data table is the

same on every result page. We also determine that the first column contains book title, second contains year, third contains book price and fourth contains the name of the publisher. Price always starts with a dollar sign.

At this point we are done with analysis and are ready to implement the wrapper code. Of course, this Web site is a "toy" site, and for real cases the complexity of the analysis and implementation will be bigger.

4.5.3 Implementation

We will implement our wrapper in Java using Data Extraction Library. The resulting portion of the source code that performs data extraction is shown in Figure 4-44. The source code presented here is not a complete application but rather a stripped down piece of source code, simplified for better understanding. In the complete application a variety of additions can be made in order to strengthen wrapper's robustness. Multiple additional safety checks and data purification code can strengthen wrapper's error detection and make data output cleaner. The choice to do that depends on the analyst and is usually dictated by the level of perfection that has to be achieved and amount of time available for implementation. We argue that the functionality presented here is adequate given the simplicity of the Web site and non-crucial nature of the wrapper.

```
// loading and filling out Bookstore search form ------------------------------------
TaggedDocument td = new TaggedDocument(new URL("http://localhost/bookstore/"));

Enumeration enum = td.find("form",MarkupElement.TAG); // find form inside the document
if (!enum.hasMoreElements())
  throw new SiteFormatException("Form not found",td.getURL());

HTMLForm hf = (HTMLForm)enum.nextElement();    // access form
HTTPSession hs = hf.getHTTPSession();          // convert it to HTTP session
HTTPParameterList pl = hs.getParameterList(); // get form parameters

Vector v = new Vector();              // select 2 publishers ...
v.addElement("MT");                   // ... M & T and ...
v.addElement("AW");                   // ... Addison-Wesley
pl.setParameter("publisher",v);
pl.setParameter("price","100");       // select price to be < 100
pl.setParameter("title","database"); // select partial book title to be "database"

// submitting form and retrieving results page --------------------------------------
hs.open(); td = new TaggedDocument(hs); hs.close();

// processing results pages ---------------------------------------------------------
while (true)
{
// looking for the book table -------------------------------------------------------
  enum = td.find("table",MarkupElement.TAG);
  if (!enum.hasMoreElements())
  {
    // if the search found no books then just return empty table
    if (td.find("No books found.",MarkupElement.TEXT).hasMoreElements())
      return;
    else throw new SiteFormatException("Book table not found",td.getURL());
  }
  Tag ht = (Tag)enum.nextElement(); // go to the table

// looking for table rows -----------------------------------------------------------
  enum = ht.findInSubtree("tr",MarkupElement.TAG);
  if (!enum.hasMoreElements())
    throw new SiteFormatException("No rows found",td.getURL(),ht);

  enum.nextElement(); // skip first row (header)
  while (enum.hasMoreElements())
  {
// extracting text from every row and returning it ----------------------------------
    Tag tr = (Tag)enum.nextElement();  // extract text from every row
    Vector strings = tr.extractText();
    if (strings.size() != 4)           // check for correct table width
      throw new SiteFormatException("Wrong table width",td.getURL(),tr);

    DataVector dv = new DataVector();  // compose return data table row
    dv.addElement(((String)strings.elementAt(0)).trim());
    dv.addElement(((String)strings.elementAt(1)).trim());
    String price = ((String)strings.elementAt(2)).trim();
    if (price.charAt(0) == '$')        // remove dollar sign from the price
      dv.addElement(price.substring(1));
    dv.addElement(((String)strings.elementAt(3)).trim());
    returnRow(dv);                     // return next data row
  }
// processing Next link, if one exists ----------------------------------------------
  enum = td.find("Next >>",MarkupElement.TEXT);
  if (!enum.hasMoreElements()) break;  // not found - we are done

  // link found - follow it
  hs = ((HTMLLink)((Text)enum.nextElement()).getParent()).getHTTPSession();
  hs.open(); td = new TaggedDocument(hs); hs.close();
}
```

Figure 4-44 Bookstore data extraction wrapper

We begin by downloading the home page of the bookstore from the address http://localhost/bookstore/. This page is converted into a searchable *TaggedDocument* upon download. We know from the analysis that the form that we need is the only form on that page. By means of a simple search for a form tag we determine if the page that we have received is the page we are looking for. If the form is not found it might be an indication of the site change or unavailability, and we notify system of this by throwing appropriate Java exception.

Once the form is found it is converted to an *HTTPSession* and is ready to be filled out with our search criteria. For the purposes of a sample application we will implement a predefined query for this bookstore Web site. We will request books published by M&T and Addison-Wesley that are priced less than $100 and contain word "database" in their titles. In a "real world" wrapper the developer would probably add wrapper parameters that for title, publisher and price. This way the bookstore can be searched for a wide variety of books without having to modify the wrapper code.

When the form is filled out, it is submitted, the page with results is received, and a new *TaggedDocument* is created from it. The first thing we do in the new page is check whether the document that we received contains an HTML table. If it does not, this means that the document that we received does not contain any data—we know from analysis that an HTML table containing data has to be present. If this is the case then we check for the presence of message "No books found" inside the page text. If it exists that means that our search merely did not yield any results. This is a normal situation and we just return no results to the calling application—an equivalent to the

empty result set. If, however, such message was not found then this is an error and we have to throw Java exception to declare that the site format has been changed.

Once we have found the table that we are looking for, we can process its rows. First we skip the title row that contains column headings. After the title row all the rows are contain actual data. We extract textual information from every row and check that there are exactly 4 columns in each row. If there aren't, then the table format has been changed. In this case we have to notify system of the changed site structure, so that the wrapper could be updated quickly.

When the data is extracted, it is cleaned from the extra spaces that might be present inside and the dollar sign is stripped from the price. After the data is cleaned it is put into a *DataVector* and returned to a calling application. Data cleaning and extraction is performed until no rows are left in the table.

After the table is fully processed we can go to the next page to extract data from there. In order to do that we search the page for the text containing string "Next >>". If such text is found we go to its parent element, which must be a link. If it is a link then we convert this link to an *HTTPSession* and use it to retrieve the next page of data. When that page is retrieved and parsed into a document we repeat the process of data extraction.

The process of extracting data from the page and moving to the next page is done while the "Next" link is present in pages. When the last page is reached the data is extracted from it and the wrapper execution is finished.

4.6 Data Extractor Scripting Language

4.6.1 Introduction

As the Data Extractor project was progressing two things soon became apparent. First, the majority of the Web sites which were analyzed and for which wrappers were implemented had simple structure and did not need the full power of Java for data extraction. Second, maintenance of Java wrappers became cumbersome in some cases, where Web sites would change their structure once in three months or even more often, which in turn required changing the Java source of the appropriate wrapper. The need was felt for a simpler way to define wrappers which could give wrapper designer access to the wealth of data extraction functionality already created, but at the same time would allow her to formulate wrapper in a simple way. As a result a simple language that facilitates fast definition of wrappers for the majority of Web sites was proposed. The language is called *Data Extractor Scripting Language* or *DESL* (pronounced as "dazzle").

We followed several requirements when we designed DESL. First, it had to be simple, expressive, and cover only the functionality necessary to extract data from HTML. It was not necessary to create another programming language similar in power to Java. This would have defeated the purpose of such language, because the combination of Java and Data Extraction Library are already well suited for building wrappers. Simplicity and expressiveness of the language improve understandability reduce code size, reducing overall maintenance time. Java can still be used in cases when site is too complex for a scripting language.

Second, we had to be able to generate scripts in this language using a user-friendly GUI. One of the plans for future development of Data Extractor is to build a GUI environment, where a designer could create and update wrappers quickly, using a WYSIWYG ("What You See Is What You Get") interface. A wrapper would be a result of a "macro recording" of the steps the designer takes through the Web site and the data extraction instructions generated in response to the data fields that designer highlights inside the site. Because of the demands of the GUI, DESL must support "round trip engineering". This means that we should not only be able to generate the script based on designer actions, but also import it into the GUI afterwards and modify it if the need arises.

4.6.2 Overview

DESL is a platform-independent, interpreted programming language. It is not a general-purpose programming language, however, because it is lacking the language control structures and data operations that are commonly expected in a programming language. These features were omitted to simplify the language and make it suitable for computer-assisted script generation and modification. DESL navigate Web pages, extract data from them, purify it, and send it to the calling process.

Syntax of DESL in Extended Backus-Naur Form (EBNF) is given in Appendix. Scripts in DESL are composed and stored as text files, one per script. Script file has `".desl"` extension.

Scripts are compiled before being interpreted. Compilation may be performed into some external representation, like "bytecode" that can be executed, or Java source code, that can be compiled and then executed. Alternatively, scripts may be compiled into memory and executed from there without being saved externally. Particular compilation and execution methods can be chosen depending on the situation.

Scripts behave the same way Java-based wrappers do in Data Extractor system. They acquire execution parameters from user, traverse and analyze Web pages, extract and purify data, and return data to the calling process.

4.6.3 Document blocks

In scripts, execution and decision-making is structured around HTML documents. Script travels from document to document, extracting data along the way. Depending on the contents of the document decisions are made about whether to travel further, extract data, abort execution, or take some other action.

DESL script consists of a number of *document blocks*. Each block has a unique name followed by braces ("{}") that contain sequences of commands for matching and extracting data. Every document name must begin with a letter and contain letters, digits or an underscore ("_"). It is case-sensitive. An empty document block looks like this:

```
simple_block
{
}
```

Each block describes an HTML document and contains instructions that find and extract data. Instructions can be of two types: *statements* and *pattern expressions*.

Statements perform data manipulation and flow control, and are represented by *commands* and *assignments*. Every statement ends with a semicolon (";"). A lone semicolon represents an empty statement. Statements can be grouped in a *statement block*, or a sequence of statements surrounded by braces:

```
{
  statement1;
  statement2;
}
```

Pattern expressions are instructions for traveling through the document tree, matching groups of elements, and manipulating data in the tree. Pattern expression consists of a *pattern* and an *action*. Pattern matches element sequences in the tree. Action is a statement or sequence of statements that execute when the pattern is matched.

More details of pattern expressions and statements are given further in this work.

Document blocks are linked to each other with special commands—*@follow* and *@spawn*. These commands load new documents and execute blocks associated with them. Consider, for example the following short script:

```
title_page
{
  //… load title page and process it
  //… now go to query page:
  @follow(query_page);
}
query_page
{
  //… specify query
  //… get results in the new document and process them:
  @follow(results);
}
results
{
  //… process results
}
```

In this example we first load the title page, obtain a link to the page that contains query form and follow that link. In the query page we specify query parameters and submit the form by calling follow. Finally we get to the results page where we extract data.

Execution of the script starts with processing the first document block. If that block contains references to other HTML documents they are also processed. The execution of the script ends when all the referenced document blocks have been processed.

The first document block in the script must have parameters. These parameters define the URL and other information related to the first document that the script would retrieve and process. For example:

```
yahoo_search("http://search.yahoo.com/bin/search",
             @method("get"),@par("p","computers"))
{
}
```

The only parameter that always has to be specified is the document URL—all other information is optional.

Documents can make recursive calls to themselves. This becomes necessary, for example, when a linked set of HTML pages contains search results. In this case structure of every document in a set is the same and the same processing instructions

can be used. One of the ways to process such chain of documents is to reference document from itself:

```
process_results
{
  //… do required processing of results on the page
  //… check if we need to go to the next page of results
  //… if no - stop execution
  @follow($url,process_results);
}
```

4.6.4 Data types

The only data type that DESL scripts manipulate is *string*. Text strings consist of zero or more characters and are enclosed in double quotes. Long strings can be composed from several pieces using a plus ("+") sign. String constants can contain the following escape sequences to denote control or nongraphic characters:

\b	Backspace	\t	Tabulation
\n	Linefeed	\f	form feed
\r	Carriage return	\"	Double quote
\'	Single quote	\\	Backslash
\ddd	Character with decimal code *ddd*	\xhh	Character with hexadecimal code *hh*

The following are examples of strings:

```
""    "St"+"ring"    "\tThis is an ampersand: \x26"
```

Most commands and functions in DESL operate on *string expressions*. String expression is a combination of string constants, variables and functions that return string results combined using a plus sign. For example:

```
"The price for " + $product + " is " + @trim(@alltext)
```

Aside from strings, DESL also includes rudimentary integer type support. In functions that require integer arguments (e.g. a *@sub* function that is described later) integer constants can be used. Neither integer arithmetic, nor type conversion between strings and integers is supported yet.

4.6.5 Variables

DESL scripts manipulate text strings. Strings can be temporarily stored in *variables*. Variables in DESL are global and are accessible throughout the script. Variables do not require explicit definition—they are allocated and become accessible as soon as they are referenced in the script.

New variables are initialized with an empty string (""). Once placed inside the variable, a value will remain there until the end of execution or until it is replaced by another value.

Each variable name must be preceded with a symbol that shows its type. All variable names used in scripts must begin with a letter and contain letters, digits or an underscore ("_"). Variable names are case-sensitive.

There are three major types of variables: *temporary variables, input parameters*, and *output fields*.

Temporary variables

Temporary variables are used to temporarily store values that are extracted from the document. Their names begin with a dollar sign:

```
$current_temperature
```

Input parameters

Values are passed into the script via input parameters. Input parameters start with a percent sign:

```
%airport_code
```

As we know from discussion of wrappers parameters can have single and multiple values. When the variable has multiple values a special iterator—*@foreach*—can be used to cycle through all of them. For example:

```
@foreach(%stock)
  @spawn("http://finance.yahoo.com/q",process,@par("s",%stock));
```

In this example *@foreach* is used to iterate through a list of stock symbols, spawning a separate thread to query information for each of the stocks. At each iteration of *@foreach* reference to *%stock* in the second line of the example above is substituted with the next value from the value list of input parameter *%stock*.

Output fields

Output fields define the fields that the script returns to the calling process. The output field names start with a pound ("#") sign:

```
#publisher
```

The script fills output fields row-by-row. Once the set of fields is filled, script can invoke a special command (*@endrow*) and submit the row to the calling process. After that the new set of values can be filled in.

4.6.6 Assignments

Assignment operator ("=") is used to assign values to variables:

```
$stock_price = "7 1/2";
```

Assignment is allowed only for output fields and temporary variables. Input parameters cannot be modified and no values can be assigned to them.

A special case of assignment is the *append* operator ("+="). It was introduced in order to support experimental multi-valued output fields:

```
#fare += $next_fare;
```

Append *adds* a value to the output field instead of overwriting it. After this operator is applied, field is converted to a multiple-valued field and all subsequent append operators will continue adding values to the value list. Note that this operator works only with output fields.

4.6.7 Comments

Comments in DESL are similar to Java. They can be both single line (*// comment*) and multi-line (*/* comment */*). Comments can be used wherever whitespace is allowed.

4.6.8 Pattern expressions

As we already know, every pattern expression consists of a *pattern* and an *action*:

```
pattern action;
```

Action is a single statement or a statement block. Action can contain new patterns that can have new actions, and so on, for example:

```
pattern1
{
  pattern2 statement1;
  statement2;
  pattern3
  {
    pattern4 statement3;
  }
}

pattern5
{
  statement5;
  statement6;
}
```

Pattern is a special sequence of characters that allows us to match parts of document tree and move through it. Pattern cannot contain spaces unless quotes or double quotes surround them.

When the pattern is matched the action associated with it is executed. However, when the pattern is not matched, the script *fails and reports to the system that site format*

has been changed. Why? Patterns were designed to match document elements that are expected to be present in the document. Failure to find necessary elements indicates that the document could not be retrieved or the document structure has been changed. In both cases we cannot continue execution and must report an error.

There is an exception to this rule. When the pattern contains operators "?" or "*" (described below) failure to match pattern does not terminate the script and execution continues.

Matching of a pattern in a new document block always starts from the beginning of the document tree. A *tree pointer*, or the current position in a tree, moves through the tree during matching. When the match is found, the tree pointer is positioned at the end of the match. This is done to simplify data extraction from that point in the tree. In the following example, the pointer will be positioned at text element "Price" before `action` is executed:

```
td."Price" action;
```

When the action finishes execution the pointer is returned to the position at which it was before matching was started. In the previous example, if the tree pointer was at the beginning of the document before the matching was initiated, then after `action` is executed it will return back to the beginning of the document. This default pointer behavior can be modified through movement operators described below.

Tree element matching

Tags in a tree are matched by name. For example, to match tag *table* write the following pattern:

```
table action;
```

To match tag *table* that contains attribute *border* with value "0" and contains attribute *width* use the following expression:

```
table(border="0",width) action;
```

Inside the attribute values Perl5 regular expressions can be used. (See [PERL] or http://www.cpan.org/doc/manual/html/pod/perlre.html for details)

Text elements are matched using string constants surrounded by double quotes, for example:

```
"Stock value" action;
```

When matching text elements, combinations of variables and string constants can be used. Unlike in string expressions, however, variables and strings have to be concatenated without a plus sign. For example:

```
%a$b"c" action;
```

Here concatenated values of input parameter *a,* temporary variable *b* and a string constant "c" form the body of the text element.

Matching of comments is done the same way as matching of text elements, except instead of double quotes single quotes are used, for example:

```
'Comment' action;
```

Variables can also be used, with only difference that the variable name cannot be the first element in the expression. If it has to be first it has to be preceded with an empty comment element - '' (two single quotes).

Perl5 regular expressions can be used both in text element and comment matches.

Tree levels

Level operator (".") separates tree elements in a pattern and shows that one element is a child of another. For example, to show that tag *td* is a child of tag *tr* use the following expression:

```
tr.td action;
```

When used as a first element in a pattern, level operator causes matching to start from the top of the tree. For example, to look for tag *html* at the top of the document use the following:

```
.html action;
```

To match *any* number of levels between elements use operator "-". For example, to look for text "Feedback" *anywhere* inside of a form use the following:

```
form-"Feedback" action;
```

The matching that we have described so far matching has been hierarchical—it was done in subtrees. Occasionally there is a need to match patterns *linearly*. In linear matching searching is not done inside one of the subtrees of the document tree. Rather, document is treated as a flat file and searching is done from a point in that file until its end.

For linear matches operator "," is used. For instance, to match text "Prices" anywhere in the document after text "Search results" use the following:

```
"Search results","Prices" action;
```

Tree pointer movement

The default pointer behavior during and after matching was described above. There are, however, ways to alter it.

In order to make pointer stay at a location other than default after matching, operator "!" is used. The following example will cause tree pointer to stay at tag *a* before `action` is executed (instead of staying at text element "Price"):

```
table-!a-"Price" action;
```

Sometimes there is a need to control where pointer stays *after* `action` associated with the pattern has finished executing (by default, pointer returns to position at which it was before matching has started). In such cases operator "`" (backtick) is used. In the following example pointer will stay at text element instead of returning to the place at which it was before pattern match was started:

```
`"Search results" action;
```

A set of operators is used to explicitly specify movements through the tree:

^	Move pointer one level up
>	Move pointer right
<	Move pointer left

Both "<" and ">" will go to the next element in a tree linearly, if relevant left or right sibling does not exist.

Square brackets ("[*x*]"), when used at the beginning of the pattern or after a ".", "," or a "-" refer to (*x*+1)'st child of the current tag (*x* is zero-based). For example:

```
^.[3]-"Time"> action;
```

Here we first move to a tag that is one level above our current position and then go to its 4th child. After that we look for text "Time" in the subtree and when such element is found we go to its right sibling. Notice the period that separates the "^" and the brackets—it is necessary to define meaning of square brackets as a "selector of the child". As we see from the next example other interpretations are possible.

Square brackets used after "^", ">" or "<" specify the number of times that operation has to be performed. For example to get to a 3rd left sibling of a text element the following can be used:

```
"Current price"<[3] action;
```

Nonmatching

Occasionally there is a need to match *absence* of an element from the tree. It could be error messages or other information that should not be present. A tilde ("~") operator is used to match absence of tree elements. Tilde can be used in front of the tag, text or comment element. For example:

```
~"Page not found" action;
```

Recurrence

When data is extracted, patterns often have to be matched repetitively, especially when data is arranged in recurring sets inside the document. Such recurrence can be applied to patterns through the use of recurrence operators:

*	Match any number of occurrences of pattern
+	Match at least once
?	Match 0 or 1 times
{x,y}	Match between *x* and *y* times; *y* can be omitted to show that the upper limit is infinity.

Recurrence works like a loop and matches the pattern specified number of times. Action assigned to pattern is executed every time the pattern is matched. The following expression will match every table cell in a document and execute `action` for every match:

```
td* action;
```

Some recurrence operators—"*" and "?"—are used to relax the rule that the pattern must be matched or the script will fail with an error. Patterns in which these operators are used are allowed not to match, because by definition they allow zero matches.

Recurrence operators can be specified multiple times in a single pattern. This effectively creates multiple nested loops. The following pattern will execute `action` for all table row tags and for every cell inside every row:

```
tr*-td+ action;
```

If the square brackets ("[*x*]") are specified before a tag, text or comment element they refer to the (*x*+1)'st occurrence of a pattern. The following will match 4th occurrence of tag *a*:

```
a[3] action;
```

4.6.9 Commands and functions

Reserved words in DESL identify predefined commands, functions and other special functionality. All reserved words start with an "at" sign ("@"). This makes them more visible and makes programs more understandable.

Both commands and functions perform predefined actions. No user-defined functions can be defined. Commands do not return values and typically are used alone. Functions return values and are used in expressions and assignments. A special kind of command—an *iterator*—works as a loop statement and is used to go through multi-value variables.

Let us review details of commands and functions. For each of them we will outline its syntax, parameters and return values, and give an example. When describing parameters and return values we will use the following notation for parameter types:

string	string variable or expression
integer	constant integer value
ipref	reference to an input parameter
block	reference to a document block name
cgipar	special data type that defines CGI parameters for page retrieval and form submission

`@endrow`

Description Closes the current row of data and returns it to the calling process.
Parameters None
Return value None
Example `@endrow;`

`@end`

Description Stops execution of the script or the current script branch (if launched by *@spawn*)
Parameters None
Return value None
Example `@end;`

`@break`

Description Finishes the execution in the current block surrounded by braces. When executed in the outermost block (document block) this command ends script execution.
Parameters None
Return value None
Example `@break;`

`@continue`

Description Continues execution at the next iteration of the loop. Can be used in recurring pattern matches to skip to the next match, or in *@foreach* to go to the next value of the variable.
Parameters None
Return value None
Example `@continue;`

```
@error(message : string [,parameter : ipref])
```

Description Reports a fatal error to the system. If the input parameter is specified considers error to be caused by that parameter and associates error with it.
Errors that are caused by changes in site structure do not have to be reported—every pattern match failure and its place will be automatically recorded by the system.

Parameters *message* - briefly describes the nature of the error
parameter - input parameter that is the cause of the error

Return value None

Example
```
@error("Invalid book ID", %book_id);
```

```
@follow([url : string,] doc : block
        [, parameter : cgipar, ...])
```

Description This function stops execution of the current document block and transfers control to another block whose name is specified in `doc` parameter. Its behavior is equivalent to submitting a form or following a hyperlink, when user browser leaves the current Web page and loads a different one.
@follow exists in two modifications, depending on whether the first parameter, `url`, is present. If it is present, it is used as an address of the page that is loaded and processed by `doc` document block.
If it is not present, the URL in the nearest *form* or *a* tag is used for the new document and its parameters. The nearest tag is found by taking the current tree position and traveling up the tree until such tag is met. For correct execution of *@follow* it is desirable to set tree position in the pattern at or below such tag.
Optional `parameter`(s) specify CGI parameters that accompany form or link and are described in "CGI parameter functions" section below.

Parameters *url* - address of the page to retrieve
doc - document block that will process the page
parameter, ... - request parameters

Return value None

Example
```
a-"Next"
  @follow(nextpage);
```

```
@spawn([url : string,] doc : block
       [, parameter : cgipar, ...])
```

Description This command behaves exactly like *@follow*, with a single difference: instead of transferring control to the new document block, a separate thread of execution is spawned to process that block. The execution of the original block continues concurrently with the execution on the spawned thread. This command allows us to process many documents simultaneously, thus speeding up the wrapper execution.
Script execution finishes when the last of the spawned document blocks finishes its execution.

Parameters *url* - address of the page to retrieve
doc - document block that will process the page
parameter, ... - request parameters

Return value None

Example
```
@spawn("http://search.yahoo.com/bin/search",
          results, @par("s",$search_text));
```

Iterators

```
@foreach(parameter : ipref) commands
```

Description Iterates through specified multi-value input parameter. For each value on the list executes the commands that are specified as a single command or a group of commands in braces. Every reference to the parameter inside the command block is replaced by the value of the parameter at that iteration.

Parameters *parameter* - input parameter to iterate through

Return value None

Example
```
form-input(value="Get Quotes")
  @foreach(%symbol)
    @spawn(results,@par("stock",%symbol));
```

`@sub(text : string, from : integer [, len : integer]) : string`

Description Extracts substring from `text` starting at a given position `from`. Extracts `len` characters, or until the end of the string (if `len` is omitted).

Parameters *text* – text to extract substring from
from – position from which to start extraction
len – number of characters to extract

Return value Extracted string.

Example
```
#price = @sub("$12.34",1);
```

`@match(pattern : string, text : string [,position : integer]) : string`

Description This function is used to match text using Perl5 regular expressions. The `pattern` specifies syntactic pattern for the string inside `text`. The part that is enclosed in parentheses (*grouping*) specifies the portion of the matched pattern that has to be returned. If many groupings exist, then the zero-based `position` parameter must specify which grouping contains data of interest, otherwise the contents of the first grouping will be returned. If no grouping is specified the entire matched pattern will be returned. If nothing was matched an empty string is returned.
Perl5 and extended Perl5 expressions are supported. (See [PERL] or http://www.cpan.org/doc/manual/html/pod/perlre.html for details) Currently only matching is supported. Replacement is not implemented.

Parameters *pattern* - pattern that is used for matching text
text - data on which matching is done
position - the position of the grouping to be returned

Return value Matched sequence of characters inside the specified grouping.

Example
```
// The following will match a sequence of
// words separated by whitespace, followed
// by a "$" and a price with possible
// decimal dot inside. Only the price will
// be extracted.
#price = @match("(\w+\s+)*\$(\d+(\.\d+)?)",
                "The price is $56.78",1);
```

`@trim(text : string) : string`

Description Removes leading and trailing whitespace and control characters from the string (behaves similarly to Java trim() function).
Parameters *text* – string to be processed
Return value String without leading and trailing whitespace.
Example
```
@trim("    Only text will remain     ");
```

Informational functions

`@url : string`

Description URL of the document that is currently processed by the script.
Parameters None
Return value The URL of the document that is being processed.
Example
```
$site = @match("http://((\w+\.)+\w+)",@url);
```

`@text : string`

Description Contents of the current text or comment element. For comments, the text is given without the outer "<!-- -->".
Parameters None
Return value Value of the current text or comment element.
Example
```
$temperature = @text;
```

`@alltext : string`

Description Collects and concatenates all text elements that are descendants of the current element of the tree.
Parameters None
Return value Concatenated text elements.
Example
```
#description = @trim(@alltext);
```

`@attrval(name : string) : string`

Description Retrieves value of the attribute from the current tag in a tree.
Parameters *name* – tag attribute name.
Return value Value of the tag attribute.
Example
```
#id = @attrval("id");
```

CGI parameter functions

CGI parameter functions can be used only as parameters for *@spawn*, *@follow* and the first document block.

`@method(method : string) : cgipar`

Description Selects which HTTP method to use to retrieve a new document. Currently only "get" and "post" are supported. "get" is the default method and does not have to be specified.
Parameters *method* – submission method.
Return value Selected method.
Example
```
a-"Next"
  @follow(newdoc,@method("post"));
```

`@par(name : string, value1 : string, ...) : formpar`

Description Specifies CGI parameter name and value. CGI parameters appear in URLs after "?" and are commonly assigned through fields in HTML forms. Multiple values can be specified.
Parameters *name* – parameter name
value1, ... - parameter value(s).
Return value Parameter name and value combination.
Example
```
form[3]
  @spawn(doc,@par("state","FL","CA","NY"));
```

`@submit(name : string [, value : string]) : cgipar`

Description Forces the form to use the Submit button with the specified name, or name and value.
Parameters *name* – button name
value – button value
Return value Reference to the submit button to be used.
Example

```
form-"Please fill in your preferences"
  @spawn(doc,@submit("Prices","airprice"));
```

`@selectbyval(name : string, substr : string) : cgipar`

Description Selects an item from a drop-down list or a list of radio buttons in a form. The item whose *value* attribute matches substring `substr` is selected.
Parameters *name* – name of the parameter
substr – substring of the parameter value
Return value Selected name and value.
Example

```
form
  @spawn(doc,@selectbyval("city","Lauder"));
```

`@selectbytext(name : string, substr : string) : cgipar`

Description Selects an item from a drop-down list or a list of radio buttons in a form. The item whose *text* matches substring `substr` is selected.
Parameters *name* – name of the parameter
substr – substring of the parameter text
Return value Selected name and value.
Example

```
form
  @spawn(doc,@selectbytext("state","Flor"));
```

4.6.10 Example

Now let us review a complete example of a DESL script applied to a real Web site—Yahoo! Finance (http://finance.yahoo.com). DESL script that is extracting information from this site is given in Figure 4-45.

```
1. yahoo("http://www.yahoo.com/",@method("get"))
2. {
3.   a-"Stock Quotes" @follow(quotes);
4. }
5.
6. quotes
7. {
8.   form-input(value="Get Quotes")
9.   {
10.    @foreach(%symbol)
11.      @spawn(results,@par("s",%symbol));
12.  }
13.}
14.
15.results
16.{
17.  b."^Views",'table<
18.    $date = @match("\w+ (\w+ \d+)",@text);
19.
20.  table-table-tr*
21.  {
22.    "No such ticker symbol."? @continue;
23.    #date = $date;
24.    td[0] #symbol          = @alltext;
25.    td[1] #last_trade_time = @alltext;
26.    td[2] #last_trade_val  = @alltext;
27.    td[3] #change_val      = @alltext;
28.    td[4] #change_percent  = @alltext;
29.    td[5] #volume          = @alltext;
30.    @endrow;
31.  }
32.}
```

Figure 4-45 Yahoo! stock quotes script

In this script we will demonstrate the following major features of DESL: multi-valued input parameter processing, multi-threaded execution, flexible form submission, document feature finding, tabular data extraction and regular expression matching.

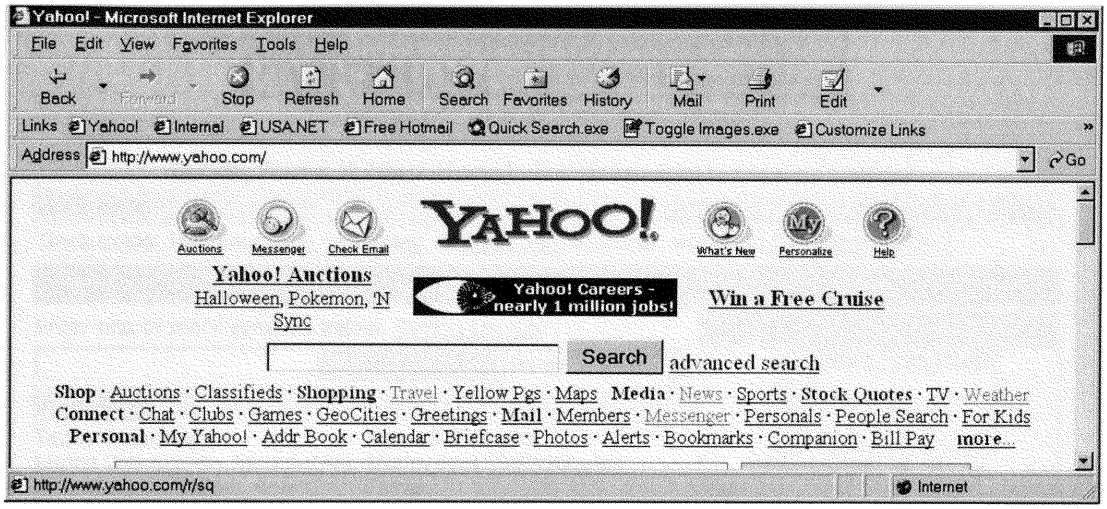

Figure 4-46 Yahoo! home page

Script starts with the document block *yahoo* and its load parameters on line 1. This document is Yahoo! home page (available at http://www.yahoo.com and shown in Figure 4-46) The request must be made using HTTP method GET (*@method* call in this case is really unnecessary, because GET is used by default). Inside the block we try to find a link (tag *a*) that contains text "Stock Quotes" (line 3). If such link is found we follow it. If it is not found, script terminates with an error, because the pattern that we tried to match was not optional. This makes sense, because if Yahoo! has removed a link to Stock Quotes from its Web site, then chances are that stock quote functionality itself was changed significantly or was disabled altogether. In this case script appropriately generates a "site changed" error.

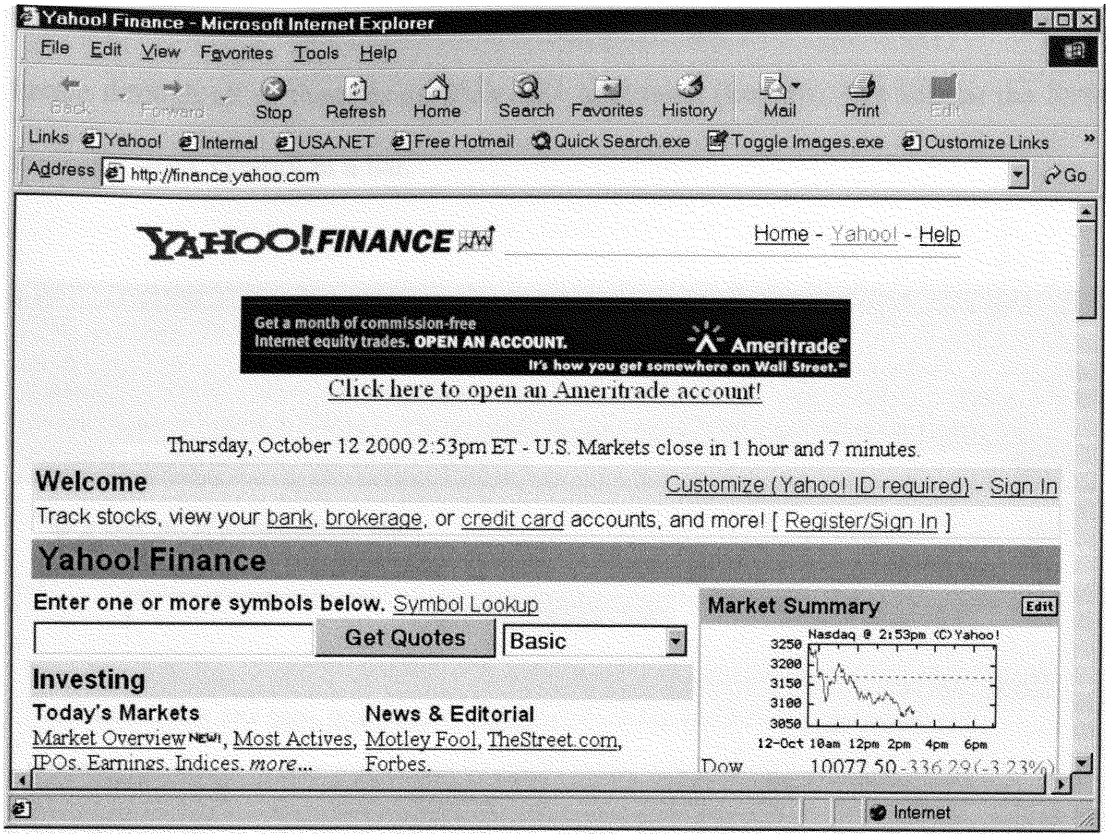

Figure 4-47 Yahoo! Finance home page

If the link was found, we follow it. The document that is retrieved (shown in Figure 4-47) is handled by document block *quotes* on line 6. Here we try to find a form through which we can specify a stock symbol (line 8). From preliminary analysis of the site we know that such form should contain button called "Get Stocks". Our form-finding pattern will match only if this button exists. If it does not, the script will fail because site has most likely been changed.

When the form is found we can use it to retrieve information about the stocks that interest us. It is quite possible that we will need information on several stocks. Here,

if stock symbols are specified in a multi-valued input parameter *%symbol* we can iterate through all of them using *@foreach* command (line 10). We submit the form for every stock symbol in a list.

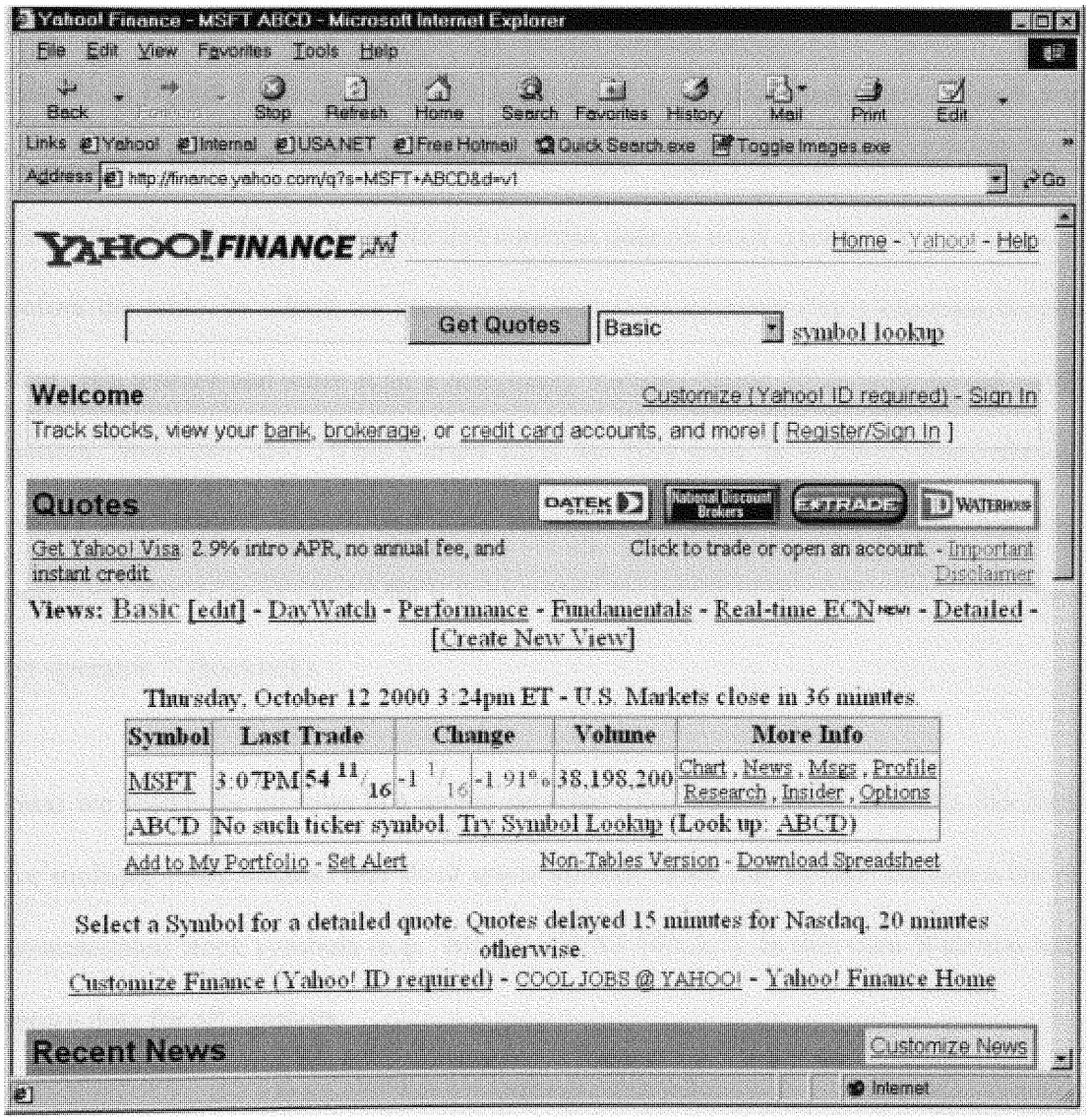

Figure 4-48 Yahoo! Finance sample stock quote report

In order to make things more efficient we submit all forms in parallel spawning a separate processing thread for each submission (line 11). Script will finish its execution once all parallel requests have been finished.

Results document block on line 15 extracts stock information from the results page (shown in Figure 4-48). In this block we first check if there exists a text element that is written in bold font and begins with text "View" (line 17). If it does, we look for a first table after it. This is the table that contains data on stock quotes. The text element before the table, to which we move using *left move operator* ("<"), contains current date. We extract and store it in a temporary variable *$date*. Date is extracted using pattern "\w+ (\w+ \d+)". This pattern skips the first word in a text element (day of week) and extracts month and date combination. After pattern is matched and command that is associated with it is executed we return to the *table* tag as specified by operator "`" (backtick).

Next we search for *all* rows in a nested subtable using operator "*" (line 20). We look for multiple rows because one stock quote symbol value can, in fact, contain a list of concatenated stock symbols (e.g. "MSFT YHOO ORCL"). In this case Web site will return data for all symbols.

When a row is found we first check if it contains an error message (line 22). If it does, we continue to the next search iteration. Notice that if the error is not found, script is not terminated (this is possible because "?" operator is used—we match 0 or 1 occurrence of the message). If the error is not found we assume that the row contains

valid data and start to fill output fields. First we store our saved date—it has to be stored with every row that goes out (line 23). Next we search for table cells (*td* tags). Each table cell is associated with particular output field, as shown in lines 24 through 29. From each cell we extract all text elements and represent them as a single string. We do this because there can be many such elements, separated by tags that specify fonts and colors, and we just want to have pure text. At the end of every iteration we close and send out the row of data (line 30).

Note that the script given in Figure 4-45 is not the most compact implementation for the given task—it was filled with extra code to demonstrate features of the language. The same result could have been achieved by using a slightly modified version of *results* document block.

4.6.11 Strong features

DESL possesses several strong characteristics that make it useful and attractive in data extraction tasks that were previously only implemented in Java.

- *Support for round-trip engineering and manual coding.* Programs can be coded in DESL manually. However, it can also be generated and modified easily using a GUI design tool.

- *Parallel execution.* Simplicity with which branches of the script can run in parallel allows analysts to develop efficient and fast scripts quickly, without worrying about process synchronization and data safety.

- *Brevity.* DESL is expressive and powerful, allowing developer to fit a lot of functionality in a short piece of code.

- *Powerful tree pattern matching and tree navigation.* Tree navigation is done through a set of short pattern matching expressions that replace complex combinations of Java searches and loops.

- *Powerful text matching and extraction.* Text matching through regular expressions simplifies extraction of pieces of text from pages. Resulting code size is small in comparison to code required to perform similar operations in Java.

4.6.12 Future development

The development of DESL is an ongoing process. As the prototype of the interpreter is being developed and tested we expect many additions and corrections to be made to the language features and tools that use it. Implementation of the wrapper designer's GUI will have a significant impact on DESL features.

Some of the changes may include:

- *Expanded data type support.* DESL primarily handles text strings. It might not be convenient in cases where good support for integer, date/time and other data types is required.

- *Data output capabilities.* In some existing Web data extraction solutions data is retrieved in a form of trees rather than a two-dimensional tables. We have already been experimenting with tree-like structures for storing data, and might incorporate this functionality in our solution. Additionally, we are considering adding script functionality that will generate multiple connected tables instead of just one.

- *New control structures.* When we designed DESL we assumed that sophisticated control structures such as, for example, "if", "while" and "switch" in C++, were too heavy for a scripting language that has to support round-trip engineering. Creating scripts in such language using GUI would be cumbersome and would not give us ease of use and simplicity (in comparison to Java) that we were striving for. This assumption might have to be re-evaluated as we gain more experience using DESL and use it in a variety of projects. The need for more sophisticated control structures might eventually come up.

- *Rich text-processing functionality.* Text-processing features of DESL are sufficient for the majority of data extraction tasks. New text functionality will be added as the need arises.

- *Extended tree-processing functionality.* Matching of tree paths and tree navigation functionality are some of the strongest features of DESL. However we foresee that changes and extensions to these features, particularly in tree navigation, will become necessary as we experiment with DESL.

- *Strict type checking.* Currently, information about the input parameters that script takes as well as the output fields that it generates is very simplistic. We plan to extend it by adding data types and sizes. We might also add lists of valid values and validation rules.

- *Script information.* We are evaluating the idea for making the scripts *self-describing.* This means putting some of the information related to the wrapper that is commonly recorded in the knowledgebase into the script itself. Also, information about author, revisions, versions and other data may be stored in the script.

4.7 Summary

In this chapter we have described the Data Extractor system and its components. Web site analysis issues were reviewed. A number of wrapper development techniques were demonstrated. The Data Extraction Library, as well as the functionality for accessing Web sites and extracting data from HTML documents was described. A sample wrapper that uses these techniques was presented and discussed. Finally a scripting language DESL that simplifies wrapper construction was introduced.

5 Mobile Data Retrieval Agents

5.1 Introduction

In the previous chapters we described a mechanism for dynamic extraction of data from the Web. This mechanism allows us to treat virtually any Web site as a read-only data source. The data collected on such Web site can be used in a myriad of academic and business applications—anywhere where data collected in real time can be applied.

We also know of difficulties associated with using Data Extractor wrapper technology. Here are some of them:

- *Performance in multi-client conditions.* The Data Extractor system is designed to be a client-server system with the data extraction functionality executed on the server. This creates a potentially serious bottleneck. As the number of users (especially remote users) grows, the system will become overloaded with requests. This problem was observed in trial runs of Data Extractor prototype—when the number of clients grew the response time for each individual request became longer. This problem is associated with increased number of simultaneous network connections that the server makes on behalf of the wrappers. The root of this problem is in non-distributed nature of the Data Extractor system and might be solved by distributing the software across multiple cooperating servers or moving the data extraction functionality to the client.

- *Potentially long data delivery route.* When the client is located remotely, the server-based data extraction can potentially have prohibitively long data delivery routes. Suppose the client is located in San Francisco and it uses remote services of Data Extractor server in Berlin. If the data that is being extracted on behalf of client is located on the site in Los Angeles and its volume is high, we will end up shipping high volumes of data in a round trip half way across the world. This process might have potentially been done locally and data would have been transmitted faster and over a much shorter distance.

- *Server in potentially slow network segment.* Network communications are the slowest operations in data extractions. In case when network links between data extraction server and data source, or data extraction server and data consumer are slow the resulting data extraction will be slow. If, however, the direct link between data provider and consumer is fast, doing extraction at consumer site could speed up the data extraction and delivery.

- *Legal issues.* In rare cases server maintainers might be prevented by law from accessing data on certain sites on behalf of the client, as it might constitute copyright violations. In these same cases giving user ability to extract data directly, without the services of a middleman, might be legal.

One solution for these problems might be installing local server for an exclusive use of a small number of clients, but costs and complexity associated with such an operation could be high.

In an attempt to resolve these and other problems we introduce a mechanism for performing data extraction on the client side through *Mobile Data Retrieval Agents (MDRA).*

5.2 Idea

The idea of MDRA is in distributing the data extraction functionality to the client computer, close to consumer of extracted data. We expect this approach to help resolve performance problems associated with extracting data on remote server on behalf of the user. The issue of distributing data extraction to remote sites is not new - an example of a similar system is described in [DF96]. In [MMM96] users interact with Java applets (that always execute on the client computer) to pose queries to the Web.

MDRA approach is different from shipping the complete server functionality to the client side, as the knowledgebase of wrappers as well as all the mechanisms for their maintenance will still be stored on the server and maintained centrally. Users, who subscribe to the MDRA service, will be able to connect to *wrapper portal,* a service that publishes available wrappers and applications associated with them, and request a particular wrapper or application to be executed on the client computer. In response to that request a package containing functionality necessary to perform data extraction for a particular Web source will be constructed and shipped to the client computer. It will then be executed there and produce data for the user.

This approach, although in a seemingly different fashion, is already used in some business applications, such as downloadable personal shopping bots from GoTo Shopping (http://shop.goto.com/) and R U Sure (http://www.rusure.com/). User downloads and installs an application associated with one of these services on her computer and then can use it to perform price comparison for a variety of products by querying Web sites in real time. From time to time comparison shopping application automatically downloads from the central location up-to-date instructions on how to get prices from a variety of online stores. Technologies used in these services are proprietary and cannot be evaluated to discuss their merits and disadvantages. Our technology is planned as an open platform that will not be geared only towards comparison but will support any application that requires data extraction from the Web.

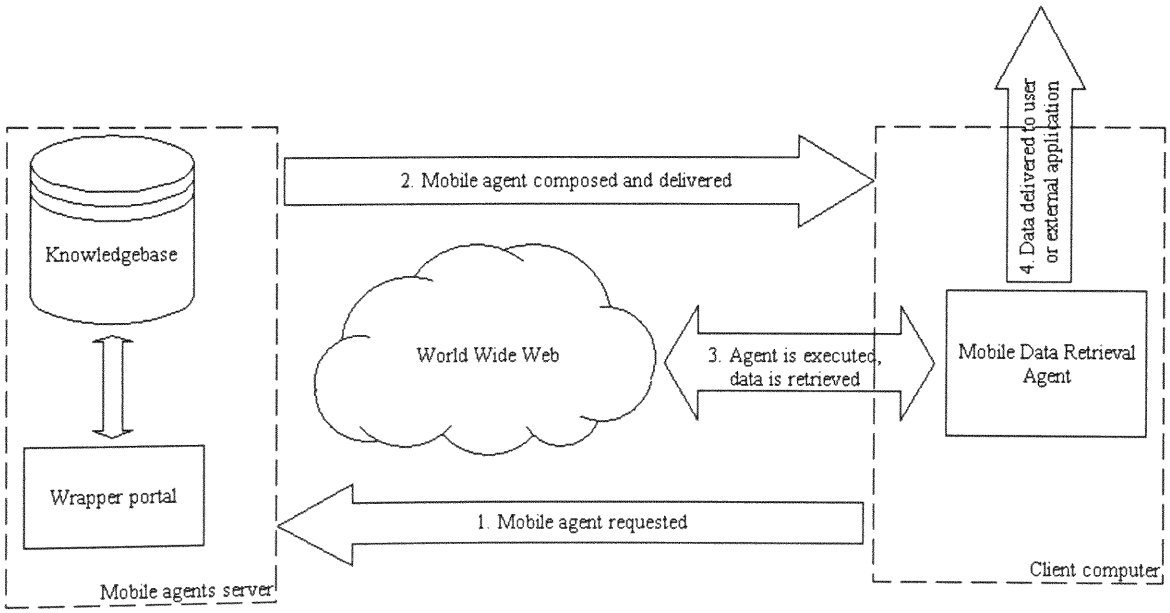

Figure 5-1 MDRA composition, delivery and execution sequence

5.3 *Architecture*

MDRA architecture is based on the architecture of Data Extractor system. In fact, many of the Data Extractor's components can be used in both systems. The internal structure of the MDRA is shown in Figure 5-2. The sequence of events for composition, delivery and execution of MDRA can be seen in Figure 5-1.

Let us describe components of MDRA framework in greater detail.

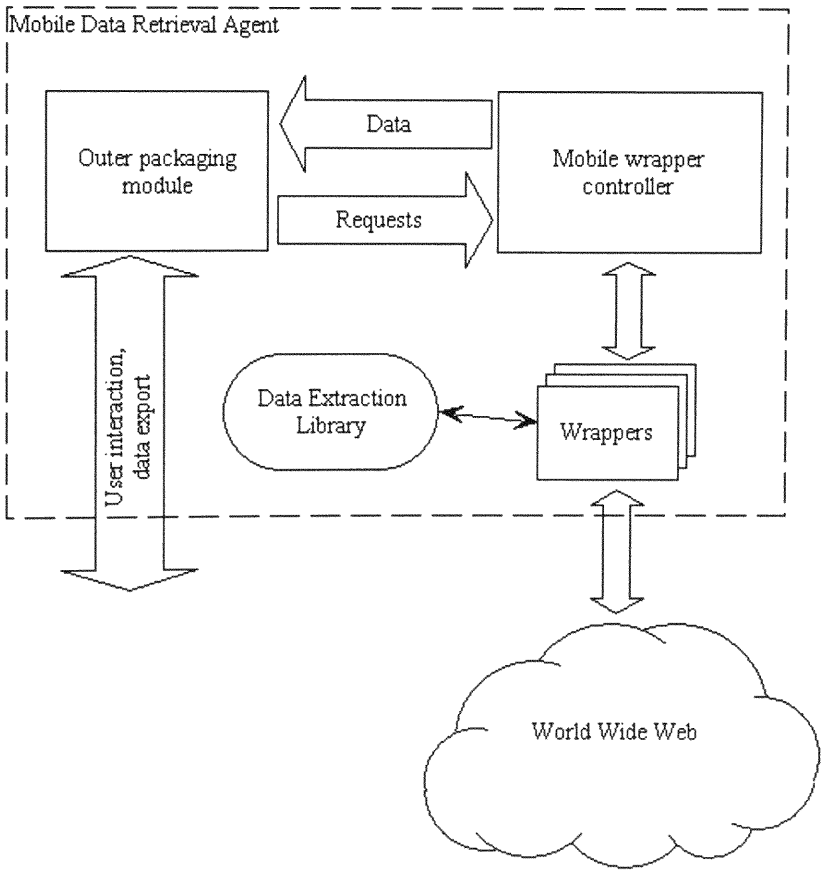

Figure 5-2 Mobile Data Retrieval Agents architecture

- *Wrapper portal.* At the heart of MDRA server is a wrapper portal system. This module is a Web-based catalog that allows users to select and execute wrappers from the knowledgebase. Upon user request it lists available wrappers and wrapper-based applications. User then can select the one that she is interested in. Once the wrapper or application is selected, it is packaged and delivered to the client computer and executed there, producing data for the user. Aside from listing and packaging wrappers, portal authenticates users, allows them to change and save their preferences, and save and retrieve previously created *queries* (references to wrappers combined with wrapper parameters).

- *Agent knowledgebase.* The knowledgebase used in MDRA server system is similar to the original Data Extractor knowledgebase. It also contains information about available wrappers, their parameters and status. In addition, however, it may contain information required by the wrapper portal. For example, it could store user account information, such as access privileges and preferences. Names and execution parameters of wrappers that users have run so far can also be stored in this database. Using this information, wrappers can be executed with the same parameters on a regular basis without having to specify them every time. Also, lightweight applications that use wrapper output or act as intermediaries between wrapper and applications on client computers can be stored in the knowledgebase, together with necessary composition and parameter information.

- *Mobile wrapper controller.* MDRA package is a collection of modules that are needed to perform data extraction on the client computer and delivery of data to

user. Wrapper controller is responsible for controlling the behavior of wrappers, passing parameters to them and directing the flow of data from them. In this sense it is very similar to wrapper controller used in Data Extractor system, but, perhaps, optimized for shipment to client computer and execution there. For wrappers that are written in DESL the interpreter for that scripting language will also be shipped as part of wrapper controller.

- *Wrappers*. The same wrappers can be used in the Data Extractor and MDRA implementations. They will be created and stored on the server and managed centrally for all users of MDRA service. This significantly simplifies service maintenance and ensures correct operation and timely updates that will be available to all users of the system.

- *Data Extraction Library*. Data Extraction Library contains functionality that is essential for performing data extraction and networking operations and has to be shipped with every MDRA. Our implementation of it is very compact and will be transmitted to the client computer quickly even on slow links.

- *Outer packaging*. Outer packaging component is a module that unites all other modules in the MDRA. In different implementations it can be implemented as a Java applet, an application, a browser plugin or take some other form. The job of this component is to communicate user commands to the wrapper controller and receive results generated by the wrapper. Packaging component can be designed to work in an interactive mode, where it would request parameters for wrapper

execution from the user via the user interface. Alternatively it can be delivered packaged with parameters selected by the user at the wrapper portal. This way it can work without user involvement.

Packaging component can act as a browser of data, data exporter, wrapper based application or connector to outside applications:

- *Browser.* Browser works as a flexible display tool. It displays data that it receives in a tabular format, like a mini-spreadsheet. Columns can be adjusted, collapsed and sorted. Data can be edited, searched, copied, printed and exported. This mode is useful for browsing and modifying data generated by the wrapper.

- *Exporter.* This type of packaging is useful for a non-interactive mode of operation. Component can be configured to automatically generate a data file on user's computer that will contain data output by the wrapper. Data files can be in a variety of formats – plain ASCII, Comma Separated Values, Microsoft Excel, XML and others.

- *Wrapper-based application.* Lightweight applications can be developed to perform simple operations on data generated by wrappers. Such applications can work interactively with user, executing wrappers based on user input and performing complex operations on the data received from wrappers.

- *Connector*. This type of packaging is useful in cases when data received from the Web has to be exported into applications running on user's computer. We can write connectors that populate tables in DBMSs or import live data into analytical or financial packages.

5.4 Agents composition and execution

5.4.1 Query formulation

User interaction with the system (see Figure 5-1) begins with connecting to the wrapper portal. Wrapper portal lists available wrappers and packaging configurations that user can run on her computer. This information along with wrappers and applications is stored in the system knowledgebase. Server maintainers continuously update knowledgebase. A variety of tools such as GUI editors and wrapper integrity checkers that are designed for Data Extractor system can be used here as well.

When the necessary configuration and packaging is selected, user optionally can specify execution parameters and save this configuration for future reference. In some cases additional information may be required from user. This information may incluse usernames and passwords for wrappers that access pay-per-use sites, or credit card information for queries that are not free. After all necessary information has been collected from the user, she may ask the system to build, deliver and execute the agent.

5.4.2 Agent construction and delivery

Once the wrapper portal receives the request for an agent it begins packaging it. Several components, including outer packaging module, wrapper parameter information, wrapper controller, wrapper, optional DESL interpreter and Data Extraction Library can be combined in a single package for delivery to client computer. Optionally, components that change frequently (such as wrappers and their parameters) can be packaged separately from the ones that do not change often. With separate packaging the part that does not change might be cached on the client computer. Depending on particular implementation, the package can be compressed and/or digitally signed. When packaging components, special attention must be paid to keeping agent as compact and platform-independent as possible.

Once the package is ready for delivery it is sent to the user computer. In different implementations such delivery can be performed in a variety of ways—from automatic Java applet delivery to manual download and installation.

5.4.3 Agent execution

When the agent is delivered to the client computer it is executed based on parameters supplied to it. Parameters can be specified at the portal or through dialogue with the user. Outer packaging component handles the dialogue with the user and controls wrapper execution through commands to wrapper controller. Wrappers interact with the Web sites, extract data and pass it to the outer packaging component module

through the controller. Overall agent execution, including stopping and restarting, is controlled through its user interface.

5.4.4 Data delivery

When the data is retrieved from the Web it can be returned in several forms: it can be fed into other applications, displayed to the user or exported to the file system.

5.5 Implementation

5.5.1 Language

The prototype of the MDRA system is currently being implemented in Java programming language. Java was chosen for a variety of reasons. Because the mobile agents are based on existing implementation of Data Extractor and Data Extraction Library, which are implemented in Java, compatibility was important, as was reuse of the existing modules. Portability was also an important consideration because MDRA code is shipped to the client side and executed there, and thus has to be supported with no or little modifications on a variety of platforms.

There are, of course, concerns about Java performance, as it is, by definition of an interpreted language, slower than its compiled counterparts, such as C++. Some of the performance problems did indeed manifest themselves at the early stages of

implementation of Data Extractor system and Data Extraction Library functionality. These problems primarily appeared when the load on the Data Extractor system increases dramatically because of multiple connected clients. Most of these problems were identified and resolved. In MDRA framework execution will be dedicated to a single client and as a result we do not expect any noticeable performance degradation.

5.5.2 Framework

For MDRA technology to be easy to use it has to be user-friendly. This starts with installation procedures. For the majority of computer users downloading software from the Web site and installing it on their computer is unattractive. There are many reasons for that: the process of software downloading and installation is often inconvenient and confusing; user might be afraid of viruses or have insufficient permissions to install software on her computer.

One of the easiest ways to ship MDRA functionality to the user's computer and execute it there is through a Java applet. Because it starts automatically and integrates with browsers well, even inexperienced users will be able to use it. Most popular Web browsers provide good support for Java applets, so chances are that whichever browser or computing platform user chooses applet-based agent framework is likely to be supported there.

When user requests the agent the browser will receive a Java applet packaged with all the necessary system components and libraries. The applet will then execute in browser context, querying the Web and supplying data to the user.

Although we expect the package size to be minimal we might decide split it into the part that does not change often, such as framework code and libraries, and parts that vary, such as wrappers and parameters. This way the browser that caches Java applets will keep the part that does not change in the cache and user will have to wait less next time she wishes to execute the agent.

Other options for packaging MDRA technology might include ActiveX controls, Netscape plugins, and, in absence of alternatives, downloadable applications. These options, however, are associated with a set of problems, such as limited portability, size, access restrictions and others.

5.5.3 Security

One of the strengths of Java applets—security—becomes a challenge in the context of mobile agents. Browsers prohibit Java applets from accessing system on user computer. This is done to prevent attacks from maliciously written applets that could sabotage systems or steal information.

MDRA, however, need to access system resources and perform other actions that applets are normally prohibited from doing. These actions include accessing network

resources (for data extraction) and file system and system resources (to save data on the system or feed it into other applications on the system).

A partial solution to this problem is to create a proxy application on the server where the applet came from (applets are allowed to communicate with the home server). Through such proxy (whose role could even played by a standard HTTP proxy server) the applet would be able to download pages from the third party Web sites. This approach, however, does not solve the problem of data export—applet will still be prohibited from exporting the data that it extracts to anywhere on the user's computer, becoming, essentially, just a viewer for such data. Also, the proxy application will become a bottleneck that will affect system performance in high-volume data extraction applications. Disadvantages in this case will outweigh the positive features.

Another solution—*applet signing*—appears to be more feasible. Applet signing is a technology that allows developer to "sign" the applet with a digital certificate purchased from a certification authority. Browser automatically detects a signed applet and by checking the certificate can verify the applet's origin and make sure that it was not maliciously modified during transmission. Signed applets are allowed certain degree of freedom inside the browser. They can, for example, request additional permissions from the user, such as permission to access network resources or the file system. When applet requests such permission browser asks user if she wants to grant or deny the request. If the permission is granted the applet will have

the same degree of freedom as regular applications installed on user's computer. This freedom, however, applies only to functionality that was requested.

To use this functionality the applet that contains MDRA will be signed before delivery to the client computer, and the code that requests permissions from browser will be inserted in applet initialization routines.

5.6 Conclusions

In this chapter we have introduced an approach to distributing data extraction functionality to the client side in a form of Mobile Data Retrieval Agents. This technology is based on Web site analysis and data extraction techniques described earlier in this work. We have reviewed the agent support architecture, and agent composition and execution specifics. Possible implementation problems were reviewed and solutions to them suggested.

6 Conclusions and future work

6.1 Contributions of this study

In this work we have described a number of new ideas and techniques, and emphasized some that are not new but are commonly overlooked in data retrieval solutions.

- *Heterogeneous databases.* We have reviewed HPDRC research efforts in the area of heterogeneous database integration. The Data Extractor system is an integral part of MSemODB heterogeneous database system. The ability to access data not only from relational and semantic databases, but also from the unstructured Web data sources significantly increases the power of MSemODB and extends the range of applications it could be applied to.

- *Wrapper construction.* We introduced a novel two-fold approach for wrapper design—coverage of simple Web sites by data extraction scripts that are easy to create and maintain, and data extraction from complex sites using compact Java wrappers.

  A number of techniques and guidelines for site analysis and wrapper design for Web data sources were reviewed in this work:

- *Site analysis and data extraction techniques*. Many researchers concentrate on schema discovery and definition, and do not give enough exposure to methods for site analysis. We attempt to rectify this by describing a set of methods and techniques for site analysis and data discovery.

- *Session and document concepts*. We feel that the ability to view the Web site data extraction process as a sequence of communication sessions and document processing steps is important. This approach introduces clarity and well-defined steps into data retrieval, and makes wrappers easier to understand and maintain.

- *Document structure trees*. Researchers commonly suggest doing simple searches on HTML document text, thus ignoring its structure. We feel, however, that the document structure can be instrumental in data identification and detecting data boundaries. Our parsing of documents into tree structures facilitates this approach.

- *No validation necessary*. Relaxed handling of documents, sometimes even at risk of creating incorrect tree representation, simplifies and speeds up parsing algorithms, and makes them tolerant to syntax errors in documents.

- *Processing of linked pages*. Not all research prototypes of wrapper technologies that we have reviewed can traverse multiple linked documents and extract data from them. Such limitation may severely affect quality of

data extraction, because the majority of data providers today publish data in sequences of linked pages.

- *Form filling.* We feel that being able to fill out forms is one of the most important features of a wrapper system, because without it data from the majority of providers would have been unavailable for extraction. Many of the research projects that we have reviewed lack this functionality.

- *Embeddability.* The ability to embed Data Extractor server as part of another application opens up many potential uses for this technology. It can act as a data provider for virtually any application without having to deploy a separate server.

- *Portability.* The Data Extractor is being implemented in Java and contains no ties to any particular platform. This makes it highly portable to any platform on which Java is supported.

- *Mobile Data Retrieval Agents (MDRA).* A novel approach of developing and deploying data retrieval agents was introduced. Such agents significantly increase performance of the Web data extraction mechanism by moving it off the centralized server and to client site. This distributed approach allows us to free-up server resources and increase overall system performance.

Some of the methods used in this study were weaker than the ones suggested by other researchers. The relative complexity of wrapper construction and maintenance is high (which is noted in virtually every research paper on the subject). The scripting language DESL that we have introduced is not as simple as SQL-like wrapper definition languages that some of the research projects use. The GUI that would allow interactive generation and correction of wrappers has not yet been implemented and we do not know how effective it will be. Another issue is related to tree representation of data - although it simplifies document processing, it also uses precious processor cycles and is not 100% reliable on poorly written HTML. We intend to work on these issues to simplify and speed up wrapper construction.

6.2 Future improvements

The Data Extractor and MDRA systems are being implemented and we anticipate some improvements to be introduced:

- *Wrapper design and maintenance time.* As we already stated before, wrapper design time is one of the major problems that might slow down projects where hundreds of wrappers have to be maintained up to date. Current design time for a Java-based wrapper for an experienced developer is around 2-3 days for a Web site of average complexity. Because the majority of that time is taken by coding and debugging, and not by Web site analysis, we introduced a simple scripting language DESL which does not require strong programming skills from the analyst. DESL reduces wrapper size and makes them more maintainable. We

anticipate the script design and maintenance time to be cut down to several hours or one day for a skilled analyst. Additionally, we have touched on the importance of creation of a specialized GUI tool which, through the simplicity of point-and-click interface, will facilitate semi-automatic wrapper construction. Such tool will in most cases eliminate the need for hand coding of wrappers. The benefit of this tool will be twofold: it will be used by analysts that do not possess programming skills and will cause the majority of analyst's time will be spent on analysis and data labeling, rather than coding and debugging.

Further research has to be done on the issue of wrapper expiration. We feel that machine-assisted monitoring of wrapper quality and notification of analysts when wrapper becomes outdated will cause additional cuts in maintenance costs. Good error diagnostics and maintenance mechanisms will help analysts with speedy error resolution.

Finally, the issue of weaker binding of wrappers to site features has to be investigated in greater detail. Currently, we are lacking comprehensive methodology on minimization of the document feature set that is used for reliable data identification and extraction.

- *Performance issues.* Java has its advantages and disadvantages as an implementation platform. A major disadvantage is the comparatively low speed of interpreted Java bytecode applications in comparison to equivalent programs written in languages that are compiled into a machine code for a given platform.

While abandoning Java as an implementation platform might not be a valid option for a variety of reasons, serious consideration has to be given to optimizing the critical parts of the system functionality, especially the one that deals with expensive memory allocation operations, text processing and networking. As it was shown in [HN99] there is hope for speeding up some common Java operations to the point where their speed rivals that of a C++ application. What particularly encourages us is the fact that optimizations described there were applied to the development of a high-speed Java Web crawler, which is certainly an application similar in spirit to the Data Extractor project. While some of the recommendations given in [HN99] were successfully implemented by us there is still room for further improvement.

- *Agents.* MDRA technology is currently being developed and is still at an early stage of implementation. The exact algorithms for dynamic applet construction are yet to be finalized. MDRA implementation will be one of the primary topics of our research in the near future.

- *Protocols and standards.* The Data Extraction Library currently supports a number of Internet protocols and standards, but this support can certainly be enhanced, because some of them are not implemented to the fullest extent. This was done intentionally in order to save development time on features that are almost never used and to reduce the resulting code size. However, in the future the implementation can be revised in order to ensure stricter compliance with standards. For example, while the Data Extraction Library supports virtually any

SGML-based markup language (of which the two most prominent examples are HTML and XML) such support might require extending the library functionality to support import of Document Type Definition (DTD) for correct tree parsing. Current support for these languages is based on a built-in DTD.

Another example is Secure Sockets Layer (SSL) support. Most of the e-commerce sites today support SSL for secure transactions and data transfer. We too have experimented with SSL and were successful in integrating it with our library and were able to effectively perform data extraction from secure Web pages. However SSL is not currently a part of our functionality, owing primarily to various patent and royalty issues associated with SSL. When these issues are resolved SSL will be incorporated into the library.

Finally, the implementation of popular Web scripting languages such as JavaScript should be seriously considered as it will allow wrappers to much better simulate user actions when accessing the Web site. This will also simplify wrapper construction and extend the number of sites supported by the Data Extraction Library.

6.3 Future research directions

- *XML*. Today, XML is taking the business world by storm. It is quickly becoming the data interchange format of choice for many areas of business communications as well as a favorite data storage format. The main goal of the Web wrapper

technology is to restore semantics and database schema for the data that is "hidden" in the HTML pages. XML keeps the semantics of the data that it presents intact. It might seem that wide use of XML on the Web sites will render wrapper technology useless, because no data extraction is necessary in it. We think, however, that there will still be a need for wrapper technology for quite some time to come. There are multiple reasons for this:

- *XML rendering to HTML.* Many of today's XML-based applications use XML on the back end while rendering the data into HTML using XSL for display in client browsers. Data extraction will be needed for such sites.

- *Restricted access to data.* Some vendors will be hesitant to open their data to the world and will prefer to use XML to communicate to their partners while showing HTML-only versions on the Web sites.

- *Proliferation and simplicity of HTML browsers.* The HTML-only browsers still constitute the majority of the browsers in the world and it might take a while for the browsers with XML support to be distributed widely. Also, a lot of the browsers with limited sets of functionality, like browsers in PDAs and cell phones might consider XML support unnecessary in order to maintain small code footprint.

- *Legacy HTML Web sites.* Finally, even as more and more XML-only Web sites appear there will still be a sizeable number of sites that will continue

publishing information in HTML. For example, publishing documents that are not generated from a data source but rather composed by a human operator will probably be easier in HTML. Such sites might find it difficult or unnecessary to make switch to XML.

There might be a strong case for applying wrapper technology to XML. We intend to research this, as our parsing algorithms already support it and the existing software framework should be able to accommodate it without a major overhaul. Of course, the efficiency of the algorithms and the variety of functionality currently used will have to be reviewed with XML characteristics in mind.

- *Semistructured data.* A significant number of Web sites today contain data that is *semistructured*, that is, its structure is irregular and does not fit well in the two-dimensional table concept of relational databases. Often this is caused by the loss of original data source at the time of HTML generation. Whatever information was originally stored in a database in a set of linked tables, it is now a single HTML page full of data.

There can be several approaches to representing semistructured data once it has been extracted. One of the ways to do this is for wrapper to present data in multiple linked tables, instead of one, thus attempting to recreate the original relational schema. Another approach is to change the data presentation format

from a two-dimensional table to XML or other. Whatever the solution, further research is required to tackle this issue.

- *Other document types.* Data on the Internet is stored not only in HTML or XML, but also in flat files, word processor documents, raw data files and other formats. We see a benefit in extending the Data Extractor project to handle these data sources. This, however, will involve making significant changes to system design and researching data representations and processing techniques suitable for such sources.

- *Wrapper construction simplification.* Wrapper programming and maintenance are the most time-consuming tasks in data collection on the Internet. Of the time spent on these tasks only a fraction is spent on site analysis—the rest is used programming and debugging code. The dynamic nature of the Web and the frequency with which the Web sites change only exacerbate this problem.

  From the conception of the Data Extractor project we were searching for ways to simplify wrapper definition. Scripting language DESL is a good first step in that direction. The future research, however, should concentrate on simplifying the process even further. Automatic, "round-trip," development of scripts using the GUI of the site analyst is one of the directions that we would like to explore. Further simplifications and increase in power of the language syntax are also being considered.

- *Automated wrapper construction.* While we feel that automated wrapper construction for an arbitrary Web data source is unlikely to be successful in the near future, some researchers ([DEW97]) have been successful in this area. They produced automated wrapper generators, capable of extracting data from selected types of sites, such as online stores. This approach is certainly interesting and deserves further attention. Being able to produce wrappers for large classes of sites automatically will definitely simplify the process of data extraction tremendously and make it less costly in terms of development time and resources.

- *Data purification.* An issue that requires close attention is data purification. To successfully integrate Web site sources in a heterogeneous database system we have to make sure that the data that comes from those sources is pure and uniform. By *purity* we mean the validity of data supplied by the wrapper to the calling process from the semantic point of view. Such data has to be free from incorrect entries or misplaced records. In reality, impure data is occasionally extracted due to imperfections in wrappers and fluid nature of Web site structure.

  Uniformity of data across different data sources is a more complicated issue. By uniformity we mean the semantic similarity of data coming from different Web sources. For example, to be successfully used in a heterogeneous system geographical data coming from several air ticket reservation sites has to be uniform. In practice, however, cities, states and airport names are often spelled and abbreviated differently in different source, which makes it hard to unite records coming from different sites. One can imaging how hard it is for the

heterogeneous system to extract all flights coming to Fort Lauderdale from several Web sources if one of them encodes it as "Fort Lauderdale", the other—as "Ft. Lauderdale", and yet another—as "FLL". Introduction of special translation tables or other purification techniques for such mixed data might be necessary and has to be thoroughly investigated.

Issues of both purity and uniformity of data can be addressed on many different levels, from wrapper to the heterogeneous database system modules. While research into the ways to purify data has been started with some promising preliminary results, these issues have to be investigated further.

- *Complete data extraction solution.* The site wrappers play an important role in integrating Web data sources with other types of data sources in a heterogeneous database system. However one can certainly imagine a variety of additional uses for the wrapper technology. There are many applications, both academic and commercial, for the wrapper system being used as a standalone tool or embedded into software systems. This is particularly important for the applications for which the heterogeneous approach is "too heavy", that is when there is no need to unite data sources of different types. In such applications the primary need might be to have dynamic access to data on the Internet as if it was stored in a local database table. There are many types of applications that may fit this description: portal integration services, stock quote analysis, comparison shopping systems, travel agent integration, intelligent search engines and many others. We are currently

working on designing a standalone solution that will incorporate the wrapper technology and make it available to a wide variety of applications.

6.4 Conclusions

In this work we have described a heterogeneous database model that is currently being developed at HPDRC, with particular attention paid to the mechanism of integrating Web data sources into this model. Challenges of Web site analysis and wrapper construction were addressed. We defined custom functionality for implementing wrappers using a high-level programming language and data extraction library, as well as a powerful special-purpose scripting language. We presented a description of a wrapper construction process and its end result. Finally a new scheme for distribution of data extraction functionality was introduced.

7 Appendix

7.1 DESL syntax in EBNF

```
  desl-script ::= first-doc doc*

    first-doc ::= id doc-params brace-block

          doc ::= id brace-block

   doc-params ::= '(' string-expr (',' doc-req-param)?  ')'

  brace-block ::= '{' statement* '}'

        block ::= statement
                | brace-block

    statement ::= match-pattern block
                | command? ';'
                | '@foreach' '(' parameter ')' block

      command ::= assign-expr
                | append-expr
                | doc-request
                | '@endrow'
                | '@end'
                | '@break'
                | '@continue'
                | '@error' '(' string-expr (',' parameter)? ')'

  doc-request ::= ('@follow' | '@spawn') '(' (string-expr ',')? id
                      (',' doc-req-param)? ')'

doc-req-param ::= '@method' '(' ('get' | 'post') ')'
                | '@par' '(' string-expr (',' string-expr)+ ')'
                | ('@selectbytext' | '@selectbyval')
                      '(' string-expr ',' string-expr ')'
                | '@submit' '(' string-expr (',' string-expr)? ')'

match-pattern ::= '.'? ((','? | movement*)
                          (node-match (separator | movement+))*
                              node-match movement*)
                      | movement+)

   node-match ::= '~'? '!'? '`'?
                      ((id attr? | match-text | match-comment)
                          (index-expr | repeat-num)?
                      | index-expr)

   match-text ::= (string-const | var)+
```

```
match-comment ::= comment-const (comment-const | var)*

    separator ::= '.'
                | ','
                | '-'

     movement ::= '!'? '`'? ('<' | '>' | '^') index-expr? '!'? '`'?

   index-expr ::= '[' number-expr ']'

         attr ::= '(' attr-expr (',' attr-expr)* ')'

    attr-expr ::= id ('=' string-expr)?

   repeat-num ::= '*'
                | '?'
                | '+'
                | '{' number-expr ',' number-expr? '}'

          var ::= parameter
                | output-field
                | temp-var

    parameter ::= '%' id

 output-field ::= '#' id

     temp-var ::= '$' id

  assign-expr ::= (output-field | temp-var) '=' string-expr

  append-expr ::= output-field '+=' string-expr

  string-expr ::= str-element ('+' str-element)*

  str-element ::= string-const
                | var
                | '@text'
                | '@alltext'
                | '@url'
                | '@trim' '(' string-expr ')'
                | '@sub' '(' string-expr ',' number-expr
                      (',' number-expr)? ')'
                | '@attrval' '(' string-expr ')'
                | '@match' '(' string-expr ',' string-expr
                      (',' number-expr)? ')'

 string-const ::= '"' string '"'

comment-const ::= ''' string '''

       string ::= any-char*

  number-expr ::= [0-9]+

           id ::= [A-Za-z] [A-Za-z0-9_]*
```

```
comment ::= '//' any-char* '\n'
          | '/*' any-char* '*/'

any-char ::= [.\n]
```

Note: White space is allowed between terminals and non-terminals of this grammar except inside the following rules and their descendants: "string", "id", "number-expr", "match-pattern", "parameter", "output-field", "temp-var".

References

[AD99] Adelberg, B. and Denny, M. Nodose version 2.0. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, Pages 559 - 561

[AM97] Atzeni, P. and Mecca, G. Cut and paste. *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997, Pages 144 - 153

[AM98] Arocena, G. and Mendelzon, A. WebOQL: Restructuring Documents, Databases, and Webs. *Proceedings of ICDE'98*, Orlando, February 1998.

[AMM97] Atzeni, P., Mecca, G., Merialdo, P. To Weave the Web. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB'97)*, 1997

[Ath00] Athauda, R., Heterogeneity Resolution in MSemODB, *Technical Report 2000-02*, School of Computer Science, Florida International University, 2000.

[BD99] Bauer, M. and Dengler, D. InfoBeams--configuration of personalized information assistants. *Proceedings of the 1999 International Conference on Intelligent User Interfaces*, 1999, Pages 153 - 156

[BD99a] Bauer, M. and Dengler, D. TrIAs: trainable information assistants for cooperative problem solving. *Proceedings of the 3rd Annual Conference on Autonomous Agents*, 1999, Pages 260 - 267

[BDHS96] Buneman, P., Davidson, S., Hillebrand, G., Suciu, D. A query language and optimization techniques for unstructured data *University of Pennsylvania, Computer and Information Science Department Technical Report*, Number 96-09, 1996

[BDKM99] Barish, G., DiPasquo, D., Knoblock, C. and Minton, S. An efficient plan execution system for information management agents.

Proceedings of the Second International Workshop on Web Information and Data Management, 1999, Pages 1 - 5

[BDP00] Bauer, M., Dengler, D. and Paul, G. Instructible information agents for Web mining. *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, 2000, Pages 21 - 28

[BGL+99] Baru, C., Gupta, A., Ludascher, B., Marciano, R., Papakonstantinou, Y., Velikhov, P. and Chu, V. XML-based information mediation with MIX. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, Pages 597 - 599

[CCS00] Christophides, V., Cluet, S., and Simeon, J. On wrapping query languages and efficient XML integration. *Proceedings of the 2000 ACM SIGMOD on Management of Data*, 2000, Pages 141 - 152

[CDS+98] Cluet, S., Delobel, C., Simeon, J. and Smaga, K. Your mediators need data conversion! *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998, Pages 177 - 188

[CDTW00] Chen, J., DeWitt, D., Tian, F., and Wang, Y. NiagaraCQ: a scalable continuous query system for Internet databases. *Proceedings of the 2000 ACM SIGMOD on Management of Data*, 2000, Pages 379 - 390

[CM98] Crescenzi, V., Mecca, G. Grammars Have Exceptions. *Information Systems, Special Issue on Semistructured Data*, 1998

[Coh98] Cohen, W. A Web-based information system that reasons with structured collections of text. *Proceedings of the 2nd International Conference on Autonomous Agents*, 1998, Pages 400 - 407

[DEW97] Doorenbos, R., Etzioni, O., Weld, D. A Scalable Comparison-Shopping Agent for the World-Wide Web. *Autonomous Agents '97*

[DF96] Dharap, C. and Freeman, M. Information agents for automated browsing. *Proceedings of the 5th International Conference on Information and Knowledge Management*, 1996, Pages 296 - 305

[DFF98] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D. XML-QL: A Query Language for XML http://www.w3.org/TR/NOTE-xml-ql

[DFKR99] Davulcu, H., Freire, J., Kifer, M. and Ramakrishnan, I. V. A layered architecture for querying dynamic Web content. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, Pages 491 - 502

[DOM] World Wide Web Consortium. Document Object Model (DOM) http://www.w3.org/DOM

[DYKR00] Davulcu, H., Yang, G., Kifer, M., and Ramakrishnan, I. Computational aspects of resilient data extraction from semistructured sources (extended abstract). *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000, Pages 136 - 144

[ECJ99] Embley, D., Campbell, D., Jiang, Y., Liddle, S., Lonsdale, D., Ng, Y.-K., Smith, R. Conceptual-Model-Based Data Extraction from Multiple-Record Web Pages. *Data and Knowledge Engineering*, 11/99

[EJN99] Embley, D. W., Jiang, Y. and Ng, Y.-K. Record-boundary discovery in Web documents. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, Pages 467 - 478

[FS99] Frank, M., Szekely, P. Collapsible user interfaces for information retrieval agents. *Proceedings of the 1999 International Conference on Intelligent User Interfaces*, 1999, Pages 15 - 22

[FSW+99] Fernandez, M., Simeon, J., Wadler, P., Cluet, S., Deutsch, A., Florescu, D., Levy, A., Maier, D., McHugh, J., Robie, J., Suciu, D., Widom, J. XML Query Languages: Experiences and Exemplars http://www.w3.org/1999/ 09/ql/docs/xquery.html

[FWM97] Fiebig, T., Weiss, J., Moerkotte, G. RAW: A Relational Algebra for the Web. *Workshop on Management of Semistructured Data*, 1997, Tucson, Arizona

[GM99] Grumbach, S., Mecca, G. In Search of the Lost Schema. In *Proceedings of International Conference on Database Theory* (ICDT'99), 1999

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

[HGI+95] Hammer, J., Garcia-Molina, H., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom, J. Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System. In *Exhibits Program of the Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 483, San Jose, California, June 1995.

[HLM+97] Harie, S., Le Maitre, J., Murisasco, E., Veronis, J., Bruno, E. SgmlQL Language Reference http://www.univ-tln.fr/~gect/simm/SgmlQL/Doc/MQL2.html

[HN99] Heydon, A., and Najork, M. Performance Limitations of the Java Core Libraries. In *Proceedings of the 1999 ACM Java Grande Conference*, pages 35-41, June, 1999.

[HTML] World Wide Web Consortium. HyperText Markup Language (HTML) http://www.w3.org/MarkUp

[HTTP] World Wide Web Consortium. HTTP - Hypertext Transfer Protocol http://www.w3.org/Protocols

[JAV] Java Programming Language http://www.javasoft.com

[KS98] Konopnicki, D., Shmueli, O. Information Gathering in the WWW: The W3QL Query Language and the W3QS system. *ACM TODS* 23(4), Dec. 1998.

[LC99] Lee, D. and Chu, W. Semantic caching via query matching for web sources. *Proceedings of the 8th International Conference on Information Knowledge Management*, 1999, Pages 77 - 85

[LD96] Loke, S. W. and Davison, A. Logic programming with the World-Wide Web. *Proceedings of the 7th ACM Conference on Hypertext*, 1996, Pages 235 - 245

[LHB+99] Liu, L., Han, W., Buttler, D., Pu, C. and Tang, W. An XJML-based wrapper generator for Web information extraction. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999, Pages 540 - 543

[LN99] Lim, S.-J. and Ng, Y.-K. An automated approach for retrieving hierarchical data from HTML tables. *Proceedings of the 8th International Conference on Information Knowledge Management*, 1999, Pages 466 - 474

[LSS96] Lakshmanan, L., Sadri, F., and Subramanian, I. A Declarative Language for Querying and Restructuring the World-Wide-Web. *Post-ICDE IEEE Workshop on Research Issues in Data Engineering* (RIDE-NDS'96), New Orleans, February 1996.

[MAM+98] Mecca, G., Atzeni, P., Masci, A., Sindoni, G. and Merialdo, P. The Araneus Web-based management system. *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1998, Pages 544 - 546

[MMK99] Muslea, M., Minton, S. and Knoblock, C. A hierarchical approach to wrapper induction. *Proceedings of the 3rd Annual Conference on Autonomous Agents*, 1999, Pages 190 - 197

[MMM96] Mendelzon, A., Mihaila, G., Milo, T. Querying the World Wide Web. *PDIS 1996*, December 1996, pages 80-91

[MS99] Milo, T., and Suciu, D. Type inference for queries on semistructured data. *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1999, Pages 215 - 226

[MSS99] Mattox, D., Seligman, L. and Smith, K. Rapper: a wrapper generator with linguistic knowledge. *Proceedings of the 2nd International Workshop on Web Information and Data Management*, 1999, Pages 6 - 11

[NQL] Network Query Language http://www.networkquerylanguage.com

[NS00] Neven, F., and Schwentick, T. Expressive and efficient pattern languages for tree-structured data (extended abstract). *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2000, Pages 145 - 156

[OMNI] OmniMark Documentation http://www.omnimark.com/doc

[PERL] Perl Programming Language http://www.perl.com/pub/v/documentation/

[RAYC00] Rishe, N., Athauda, R., Yuan, J. and Chen, S.C. Knowledge Management for Database Interoperability. Submitted to the *International Conference on Information Reuse and Integration*, Honolulu, Hawaii, November 1 - 3, 2000.

[RAYC00a] Rishe, N., Athauda, R.I., Yuan, J. and Chen, S.C. Semantic relations: The key to integrating and query processing in heterogeneous databases. To appear in *The 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, Florida, July 23 – 26, 2000.

[Ris92] Rishe, N. Database Design: the semantic modeling approach. *McGraw-Hill*, 1992, 528 pp.

[RLS98] Robie, J., Lapp, J., Schach, D. XML Query Language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html

[RYA+00] Rishe, N., Yuan, J., Athauda, R., Lu, X., Ma, X., Vaschillo, A., Shaposhnikov, A., Vasilevsky, D. and Chen, S.C. SemanticAccess: Semantic Interface for Querying Databases. To appear in *The International Conference on Very Large Data Bases (VLDB 2000)*, September 10-14, 2000.

[RYA+00a] Rishe, N., Yuan, J., Athauda, R., Lu, X. and Ma, X. SemWrap: A semantic wrapper over relational databases, with substantial size reduction of user's SQL queries. In the *Proceedings of the 7th Extending Database Technology (EDBT 2000) - Software Demonstrations Track*, 2000, pp. 13-14.

[SK98] Sugiura, A. and Koseki, Y. Internet scrapbook: automating Web browsing tasks by demonstration. *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, 1998, Pages 9 - 18

[W4F] W4F toolkit http://www.tropea-inc.com/technology/W4F

[WEBL] WebL: A programming language for the Web http://research.compaq.com/SRC/WebL

[XML] World Wide Web Consortium. Extensible Markup Language (XML) http://www.w3.org/XML

[XPATH] World Wide Web Consortium. XML Path Language (XPath). *W3C Recommendation.* http://www.w3. org/TR/xpath

[XPTR] World Wide Web Consortium. XML Pointer Language (XPointer). *W3C Candidate Recommendation.* http://www.w3.org/TR/xptr

[XSLT] World Wide Web Consortium. XSL Transformations (XSLT). *W3C Recommendation.* http://www.w3.org/TR/xslt

VITA

DMITRIY BERYOZA

1973	Born Moscow, Russia
1994	B.S. with honors in Computational Mathematics Rochester Institute of Technology Rochester, NY
1996	M.S. in Computer Science Rochester Institute of Technology Rochester, NY
1995-1996	Software Engineer Lenel Systems International Inc. Rochester, NY
1996-1997	Senior Systems Analyst Powernet International Inc. Miami, FL
1997-2000	Ph.D. student in Computer Science Florida International University Miami, FL
1997-2000	Researcher, Technical Team Lead High Performance Database Research Center Florida International University Miami, FL
1998-2000	Presidential Fellowship Award Recipient Florida International University Miami, FL
1999	Winner of ACM/Microsoft "Quest for Windows CE" application contest

PUBLICATIONS AND PRESENTATIONS

[RSC+99] Rishe, N., Sun, W., Chekmasov, M., Prabhakaran, N., Beryoza, D., and Chekmasova, M. Infrastructure for Research and Training on High-Performance Heterogeneous Distributed Database Management. *Infrastructure '99: NSF, CISE, EIA, RI and MII PI's Workshop.* New Mexico State University, Las Cruces, New Mexico, August 7-9, 1999, pp.42-46.

[BUPR98] Beryoza, D., Uppal, J., Pardo, P., and Rishe, N. Interfacing Java to Semantic DBMS. *Proceedings of Workshop on Next Generation Database Design and Applications*, Miami, May 1998.

[Ber92] Beryoza, D. Graphic fonts by Borland Inc., *PC World/Russian edition*, No. 8, 1992.

[Ber99] Beryoza, D. Microsoft Data Access Technologies and Sem-ODB, *Internal HPDRC Presentation*, April 14th, 1999.

[Ber99a] Beryoza, D. Common Gateway Interface (CGI), *FIU ACM Student Chapter Presentation*, September 22nd, 1999.