Programming the Web - Design and Implementation of a Multidatabase Browser

Marta Jakobisiak

CISL WP# 96-04
May 1996

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

Programming the Web - Design and Implementation of a Multidatabase Browser

by
Marta Jakóbisiak

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1996, in partial fulfillment of the
requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented the Multidatabase Browser and the Wrapper Generator. The Multidatabase Browser is a tool that allows a single query interface to heterogenous sources: relational databases and data published through the World Wide Web. The Browser is a front end application to the Context Interchange Network (COIN). Structured or semi-structured Web sources are incorporated to the COIN system through the Wrapper Generator. The Wrapper Generator is a Web agent that extracts data values from the WWW documents by following a specification. This allows COIN users to issue queries using Structured Query Langueage to both databases and WWW information sources.

Thesis Supervisor: Michael D. Siegel
Title: Principal Research Scientist, Sloan School of Management

Acknowledgments

I would like to thank my thesis advisor, Michael Siegel, and Professor Stuart Madnick for giving me the opportunity to work on the Context Interchange Project, and for their advice and support. I would like to thank the whole Context Interchange team for providing a great working atmosphere. In particular, Cheng Goh and Tom Lee for many excellent insights and suggestions.

Special thanks to Anne Hunter, for all the help to make this thesis a legal document and for reducing the Institute bureaucracy to the necessary minimum.

Finally, I would like to thank my parents, Elżbieta and Marek Jakóbisiak and my sister, Kasia and Gunnar for their love, encouragements and support. I would especially like to thank Gunnar for proof reading this thesis and correcting my grammar.

Contents

List of Figures

Chapter 1

Introduction

Since 1990, when it first came around, the World Wide Web (WWW) has experienced a tremendous growth, both in the academic and commercial worlds, and is now one of the primary means of information publishing on the Internet.

With a single mouse click you can reach everything from the CIA Factbook[1] to updates on Beavis and Butthead's latest episodes[2]. And it is not merely text files that are at your reach, multimedia is making its presence through pictures, sound and animation clips that you can download at your wish. The ever-increasing usage of the WWW, which has experienced exponential growth over the past five years, has established the need for, and usefulness of providing information and services over the Internet. Many institutions are trying to make their databases accessible via Internet. This is not an easy task, due to limitations of the Web technology for database support, but major database companies as well as independent software vendors are developing products aimed at bringing the database and networking worlds closer together.

As a result, the volume of data that one will be able to access via the Internet is growing, and so is the problem of trying to understand and interpret all the information. The chances of data providers and data receivers having different assumptions about the data exchanged are high. 2.01.95 might mean 1st of February in the US, but in Europe it means January the 2nd. Different context of the data lead to possible confusion and conflicts when the information from heterogeneous sources is brought together. The database integration community has been addressing those issues for a while.

The Context Interchange Network (COIN) project at the Sloan School of Management at MIT has proposed one of the possible approaches for achieving interoperability among heterogeneous data sources and receivers. The meaning and underlying assumptions about the data set are explicitly represented as data contexts. When the need to bring together data of different context arises, it is a job of Context Mediator to determine semantic conflicts between contexts and apply any transformations necessary to exchange the data in meaningful way.

The COIN systems uses the Structured Query Language (SQL) as a means of issuing queries to data sources. Each query issued by the receiver is translated to a source context if necessary by the Context Mediator and then passed on to the data source. The data returned is similarly translated to the receivers context and presented to the receiver. This process allows COIN to extract disparate data from different sources and produce consistent

[1] *http://www.odci.gov/cia/publications/95fact/index.html*
[2] *http://beavis.cba.uiuc.edu/*

results for data receivers.

The WWW has been chosen for the second prototype[3] implementation of the COIN project, as it provides an easy way of reaching the ever-growing volume of information. The heterogeneous nature of the data additionally establishes the need and justification for COIN's project.

The choice of WWW as an underlying infrastructure puts the Context Interchange Project at the very front of an emerging new technology, both in terms of database integration efforts and in building WWW applications. While knowledge representation and resolving semantic conflicts are extremely interesting research topics, development of interactive applications on the Web is a very interesting area as well, because of its relatively new nature and its inherent problems, that are due to the fact that WWW was originally designed to serve static documents, and not to interact with them. The goal of this thesis is to illustrate an example of designing and implementing two WWW applications for the COIN project, and to discuss the difficulties of making the Web an infrastructure for interactive applications.

In the remaining part of this Chapter we will state the purpose of the Multidatabase Browser and the Wrapper Generator - two WWW applications that this thesis is based on. In Chapter 2, background information on the Hypertext Transfer Protocol (HTTP), the backbone of WWW, will be discussed. In this Chapter, the COIN approach, as compared to standard indexing systems, will be examined in the light of the problem of resource discovery on the Web. Chapter 3 will present an overview of the COIN architecture, with a particular emphasis on the Multidatabase Browser[4] and the Wrapper Generator. The Multidatabase Browser's implementation will be examined in Chapter 4. This Chapter will also include a discussion of the algorithm behind the Wrapper Generator and address the issue of incorporating the data published on the World Wide Web to COIN. Finally, Chapter 5 will summarize lessons learned about the WWW as an application interface, and discuss the limitations of the current system.

1.1 Multidatabase Browser and Wrapper Generator

Client applications can interact with the data sources registered with COIN by issuing SQL queries. Those queries are then processed by the system. In some cases, when the data source does not comply to the relational database model, the SQL query is translated to the native query format of the source. This provides the abstraction barrier for the clients, who now have a homogeneous way (SQL) of querying sources which might not be relational databases, but Web pages, for example. The Web sources that are used in a sample application are those that provide financial information. Several such sources are now freely avalable. They provide a number of services ranging from up to date stock quotes information to conversion rates between different currencies. The Networth site, for example, serves stock quotes for companies, subject to a fifteen minutes time delay. Users can obtain quotes by entering a ticker symbol in the space provided and pressing a submit

[3]The first prototype implementation was based on a three-tiered client-server system [3]

[4]The name of this application is somewhat unfortunate, because the word *browser* in the Internet community usually refers to either Netscape Navigator or Mosaic. I decided however to keep the name as it is and apply the following convention: the Browser with the capital *"B"* will refer throughout this document to the application that is described in this Thesis. The browser with a lowercase *"b"* will refer to Netscape, Mosaic and others.

button. A screen dump of the Networth entry page is shown in Figure 1-1 on page 11. A typical result page is shown in Figure 1-2. Stock information, such as the price of last trade (*Last*), the number of shares traded so far a particular day (*Volume*), the ratio of price to per-share earnings (*P/E Ratio*), and more, is provided.

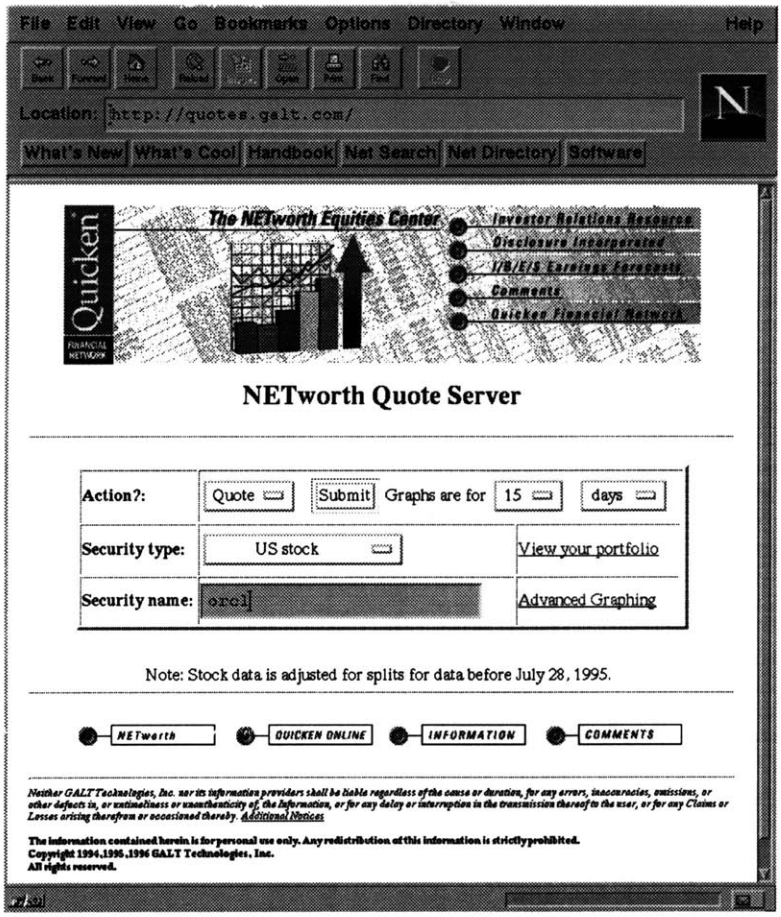

Figure 1-1: Networth quotes server: entry page.

The users of the COIN system are assumed to know the basics of SQL. It is one thing, however, to understand the SQL syntax, and yet another thing to be able to compose the query to a data source whose structure is not known to the user. The valid query must contain proper relation and attribute names. This information has to be available to the users without allowing them direct access to the data sources.

Enter the Multidatabase Browser. As the name suggests, this application is a single query interface that allows for structured acceess to the information available in the sources registered with the COIN project. The Browser guides clients through the formulation of SQL queries in a Query By Example (QBE) [4] fashion, similar to Microsoft's Access, by going through a set of HTML forms. The resulting query is then shipped by the Browser to the Mediation Engine, and processed results are presented back to the clients in a tabular form. The purpose of the Multidatabase Browser is clearly to provide the front end access to the COIN system via the WWW. In doing so, one of the Browser's functions is to communicate with the underlying data sources. Accessing databases through the Web is an old problem [12], and many interfaces have been developed for that purpose by software

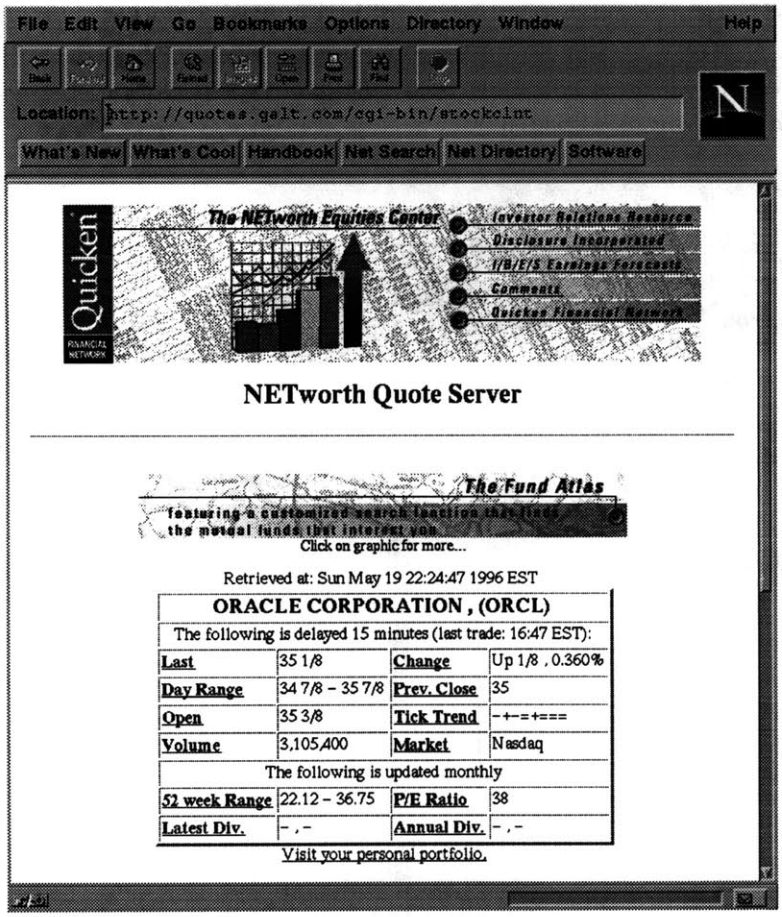

Figure 1-2: Networth quotes server: results for *"orcl"*.

vendors. Since the database and WWW server often run on the same machine, it is a simple matter of providing a gateway interface that can communicate both with the HTTP protocol and understand the database commands. A comprehensive summary of currently available solutions for Unix and PC platforms can be found at:

http://cscsun1.larc.nasa.gov/~beowulf/db/existing_products.html.

It is an entirely different issue, however, to communicate with a database that one can only access via a gateway. Among many financial services available on the Web today, each one has a unique interface, and a unique pattern of interaction, and it might be confusing for the information gatherers to learn all the details necessary to collect meaningful data distributed over a number of sites. The COIN project has developed a strategy to incorporate the data freely available on the Web into the system, and allow clients to formulate SQL queries to all that data in a uniform fashion - through the Multidatabase Browser interface. Each such site is represented to the system through its specification file. The Wrapper Generator then takes in the SQL query, translates it to the native format for the source, and opens an HTTP connection to the document server. A reformulated query, constructed by the Wrapper Generator, is then sent to the server, which returns the requested HTML pages. All the needed data is then extracted from those pages, arranged in a table, and, finally, presented back to the Multidatabase Browser. The commands necessary to perform all the steps involved in this interaction are generated automatically from the specification file.

The strategy behind this process of "borrowing" the WWW data will be discussed in detail in Chapter 4.

Chapter 2

Background

2.1 The Web

In recent years, the Internet has received considerable attention from academic communities, industry and the government. The ever growing number of users and resources available justifies the name "Information Superhighway". There is no question about the importance of the Internet as the primary means of information exchange in the future. As "surfing" the Net becomes more and more popular, it is important to understand the underlying mechanism of the Web and the promises and constraints it imposes on the WWW developers.

2.2 The HTTP protocol

On the surface, the Web is just a friendly interface, provided by Mosaic or Netscape browsers. Underneath, there are messages transferred through the net that speak the Hypertext Transfer Protocol (HTTP).

A typical interaction works as follows. A Web browser, or independent Web agent (such as a Web robot, for example) connects to a known in advance telnet port on a machine that stores the document the browser needs to access [1].

Each document has an identity: a Uniform Resource Locator (URL), which tells the browser the protocol of the document (besides HTTP, it can also be FTP, Gopher, WAIS, and others), the server and a direct path to the document on that server (Figure 2-1). Once the connection is established, the browser issues a command to the server, captures the document, and displays it. The command, also called a request method, can be one of the following: Get, Head, Post, Put, Delete, and, possibly, others. The most popular request method is HTTP Get, which tells the server to return all the contents of the document.

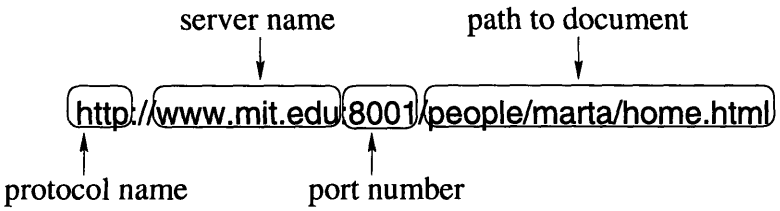

Figure 2-1: A Typical URL.

HTTP Post, originally designed for posting articles to the newsgroups, is currently mostly used for processing fill-out forms.

A browser usually makes several requests to retrieve a typical document: text and graphics are retrieved separately.

2.3 HTML and CGI

The actual documents transferred over the Web are written in Hypertext Markup Language (HTML). HTML specifies a set of tags that tell the browser how to layout the ASCI string retrieved from the server. There are tags that denote document title (<TITLE>), headers (<H1...H4>), paragraphs (<PR>), different fonts or styles, graphic images, links to other documents and even tables[1]. The browser can choose to interpret all the tags or ignore some of them[2]. Document writers must keep this in mind: certain layout features supported by their favorite browser may not be supported by others.

The Web was originally created to transfer static documents, but soon the need for interactive applications became apparent. Mainly driven by the need to query the databases residing on a server [12], the Common Gateway Interface (CGI) was developed. CGI describes a standard way for executing programs that run on the server in response to the client's request, and can usually do much more then merely accessing a database. CGI programs accept arguments and create HTML content on the fly, based on user input, and then return it to the browser for display. Interaction with CGI programs is achieved through the standard set of widgets, such as text entry fields, toggle buttons, push-buttons or check-boxes. The most direct way to send data to a CGI script is to append it to the URL. The query string begins with a question mark, arguments are separated by & signs and spaces (and other reserved characters) are escaped to + signs.

http://quotes.galt.com/stockclnt?stock=Oracle+Systems&action=search
translates to: search the index for the string "Oracle Systems".

2.4 Advantages and limitations of programming the Web

The explosion of the Web's popularity created an exciting opportunity for the application programmers and system designers. By making an application Web-accessible, one can avoid the headaches of porting to a variety of platforms and costs of distributing new releases, upgrades, and bug fixes, all of which are so painfully present in the traditional software development cycle [6]. Those administrative advantages are of special importance for small research institutions and organizations who simply cannot afford providing system support for multiple platforms.

The Web-based application also has a better chance of attracting a larger user community, because it places just one requirement on the users - having access to a Web browser. This has been especially attractive for commercial vendors. As a downside, a large user community translates to a large number of requests per day that have to be serviced by the Web server, and can be a mixed blessing, in some cases.

The true problems with using the Web as a development environment come in two flavors. First, it is hard to implement certain features in the Web application due to the

[1] in HTML version 3.0.

[2] The Lynx browser ignores graphics, for example.

restrictions of the HTTP protocol. Second, in many cases, it is hard to use such applications efficiently.

2.4.1 Security Concerns

Ever since CGI emerged as the standard for programming the Web, and made it possible to provide database access over the Internet, it became clear that the Web lacks the facilities to secure user privacy by authenticating incoming requests.

More importantly, poorly written scripts can be abused by users to access local system information and resources when not intended, and potentially damage them. Such problems can easily be avoided by carefully scanning the user input for abusive commands[3] before passing it on to the server for execution. Another concern for the script writers is the Web's inherent lack of identification of its users. When knowledge of the user's identity is crucial for the application program, such as in the case of credit card transactions, additional work has to be done [2].

2.4.2 Restrictions of HTTP

Another problem is caused by the stateless nature of the HTTP protocol - no data is stored on a server unless the application itself does so explicitly. Applications that require persistent storage can achieve this by storing the data in a form of hidden fields so that it persists on a client (the hidden fields are explained in more detail is Section 4.2.2).

It is also currently impossible to implement many features of sophisticated user interfaces over the Web [11]. Since there are only two ways a browser can transmit information to the server: either by a user selecting a hypertext link or by pressing a submit button; it is impossible for the server to determine anything about the intermediate actions of the user and act upon it. It is impossible, therefore, to provide direct feedback to the users, before they commit to submitting their requests. Applications also have very limited control over their appearance. It is up to each individual browser to determine how to render HTML anchors and how to layout the pages[4]. The limited widget set forces complex interactions to be broken up into many simple steps. In order to help application developers to provide a consistent look to their interfaces, and hence eliminate the user's confusion, many Web style guides have been written, for example *"Style for on-line hypertext"* written by Tim Berners-Lee and available at:

http://www.pku.edu.cn/on_line/w3html/Provider/Style/Overview.html.

Some of the more interesting remarks include paying attention to the size of the document to avoid scrolling, always signing the work with authors email address to allow for users feedback, and not formatting the work for a specific browser so that even a user with only a text terminal can make use of the application, and avoiding blinking text[5].

---

[3]Such as: `eval 'rm -r *'`, which would recursively remove all the Unix system files.

[4]Netscape recognizes HTML 3.0 tables, the Mosaic browser simply ignores them.

[5]Netscape 1.1 and higher versions provide this option, which when used results in HTML pages that are irritating to the eye and hard to read.

2.5 COIN versus standard indexing systems

The explosive growth of the Web is making it difficult for the users to locate the information relevant to their interests. There are just too many sites one has to browse through to find answers to simple questions. Let us consider for a moment a family planning a vacation in Greece. They want to avoid the tourist rush and go off season, in June. What sort of temperatures should they expect? The answer to their question is published on the Internet. The problem is that this answer is hard to find.

Several solutions have been proposed to address this so called Internet resource discovery problem. Most of them involve an indexing system of some sort. Behind each system, there is a tool, usually called a Web robot, that automatically navigates the Web, and indexes the document titles and the document content. The users can then issue queries directly to a precomputed index or a search program. AltaVista is one of the better known publically available search engines[6]. In our example, the family finds the AltaVista homepage and submits a search for the keyword *"travel"*. 1000000 responses are found, including tips for business travelers, a guide to African safari, and many more documents of to value for our family. This illustrates the main problem with AltaVista and similar Internet indexing systems: the effectiveness of answering user queries in low. No matter how good the search and ranking algorithm is, it does not guarantee that the documents with high relevance scores are actually relevant to the users information needs [8],[10]. Also, the sheer volume of information retrieved when searching on popular keywords might be overwhelming to the user. Scanning through the megabytes of pages to find out an answer to a specific query is not very practical and can be quite agitating.

The COIN system addresses the Internet discovery problem in a different way. First of all, all user queries to COIN are issued using SQL. This gives users more control over the number of answers they get back: the more restrictions they put in the query, the more specific answers they get back. The second very important difference is that COIN is not an indexing system. The answers to users queries are data elements arrangement in a table rather then links to HTML documents. The Wrapper Generator does all the work to extract relevant data values from HTML documents. COIN also helps the users formulate queries through the Multidatabase Browser interface. If a source *weather* that carries international weather forecast information was registered with COIN, our family could find out all they wanted about the weather in Greece by means of a single query:

select *weather.temperature, weather.humidity*
from *weather*
where *weather.location = 'Greece' and weather.month = 'June'*

The *weather* source would have to be registered with COIN: certain description files would have to be created that tell the system what data elements this source can export (i.e. *temerature, humidity, location, etc.*) and where they are located in the original HTML document. This means that someone would have to write description files in advance. This is yet another difference between COIN and WWW indexing systems. The Web robots discover and index new sites automatically. New sites can be added to the system without any human interaction. Moreover, those sites are not required to have any structure. COIN's interest, on the other hand is in structured and semi-structured sites that can

[6]AltaVista was developed by Digital's Research Laboratories in Palo Alto. It indexes over 30 million HTML pages. It can be found at
http://www.altavista.digital.com

be respresented using a relational model. Such sites have to be carefully analized and description files have to be written for them to be of any use to COIN. Naturally, the number of sites registered with COIN will always be several orders of magnitude smaller than the number of sites indexed by a Web robot. One could say, therefore, that COIN's strength is in the quality, rather than the quantity of the information it provides.

Chapter 3

Design

3.1 COIN's architecture

The Context Interchange Network was designed using a three tier architecture paradigm. The World Wide Web provided an application interface to the system. This Chapter discusses COIN's main components. The overall structure of the network prototype implementation is shown in Figure 3-1.

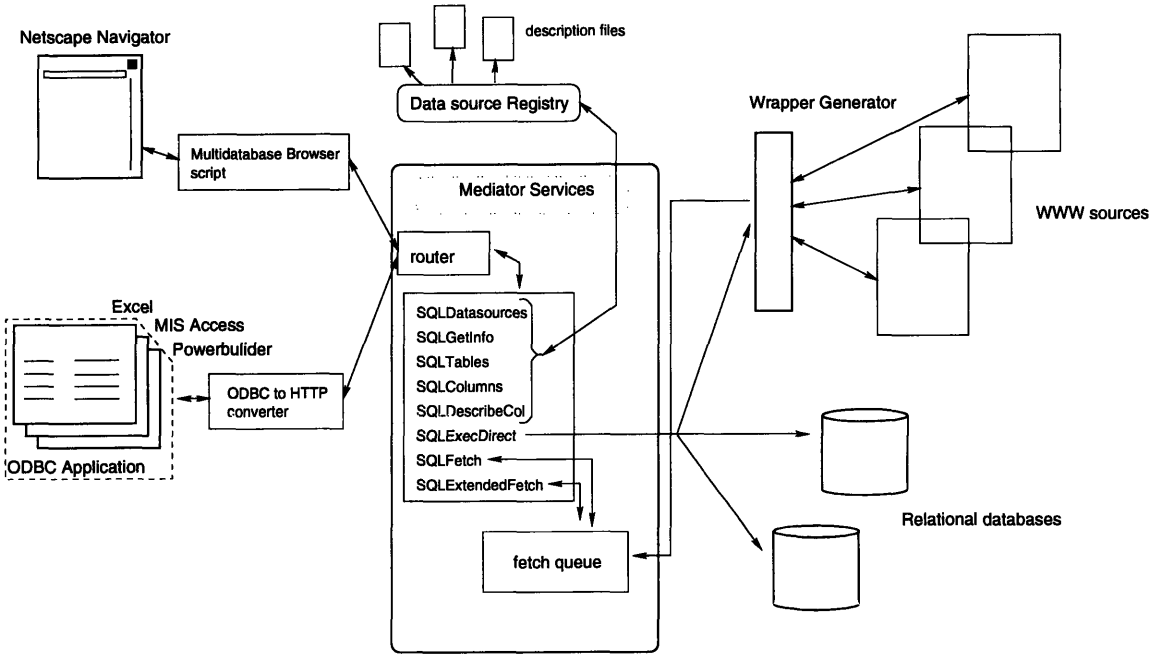

Figure 3-1: COIN Architecture.

At the heart of the system lie the mediator middleware services. The middle layer also contains data source directory services. The backend consists of gateways to the actual sources of information, which are currently both Web servers and full fledged relational databases.

3.1.1 Information Receivers

The Context Interchange Network defines a simple interface for all its clients. Eight basic commands are currently supported. They mimic some basic ODBC (Open Database Connectivity) [9] calls:

- *SQLDataSources*: lists all the sources in the Datasource Registry.
- *SQLGetInfo*: returns information about the source. At this point, this is simply a URL for the actual Web server, but additional information can be added in the future.
- *SQLTables*: returns a list of table names stored in a specific data source.
- *SQLColumns*: returns a list of column names in a specified table.
- *SQLDescribeCol*: provides the datatype of the data in a particular column.
- *SQLExecDirect*: executes an SQL statement and stores results in a Fetch Queue.
- *SQLFetch*: returns a result row.
- *SQLExtendedFetch*: returns all results rows at the same time.

Client applications can issue these commands to help users formulate SQL queries and display results in a custom form. Since the API commands resemble ODBC functions, it will be possible for the ODBC complaint applications, such as Microsoft Access or Excel spreadsheet, to communicate with the mediator services through the ODBC - Mediator API converter. So far the only front end application implemented is the Multidatabase Browser.

3.1.2 Datasource Registry

In order for a data source to become part of the Context Interchange Network, its description has to appear in the Datasource Registry. Each source has a descriptor file, which is an HTML document. Its location is not, therefore, limited to the Context server. This guarantees system scalability once the number of registered sources grows beyond storage capabilities of our server and allows the source maintainers to update the information about the source if applicable. Each descriptor file currently contains three pointers (HTML links): to an export schema, to the specification file, and a URL for the actual source of the data, whether it is a Web site or a full fledged relational dababase system. An example of a descriptor file for the Networth source is shown in Figure 3-2.

An export schema defines what data elements are available from the source, and it is organized in the form of attributes and relations. The Networth source can be modeled, for example, as having one table, called *networth*, and several attributes (*Company, Last, P/E Ratio* and others). An export schema also contains the datatypes[1] asssociated with each column. In the above example, *Company* would be represented by a VARCHAR: a character string of a variable length, and *P/E Ratio* would be a NUMBER.

The specification file describes the actions that need to be performed by the Wrapper Generator in order to retrieve the actual data values from the site. Depending on the nature of the source, those can be as simple as passing an SQL query to the WWW - database gateway (in the case of RDBMS's) or performing a sequence of interactions with

[1]Datatypes are required for the ODBC applications that will be connected to COIN in the future.

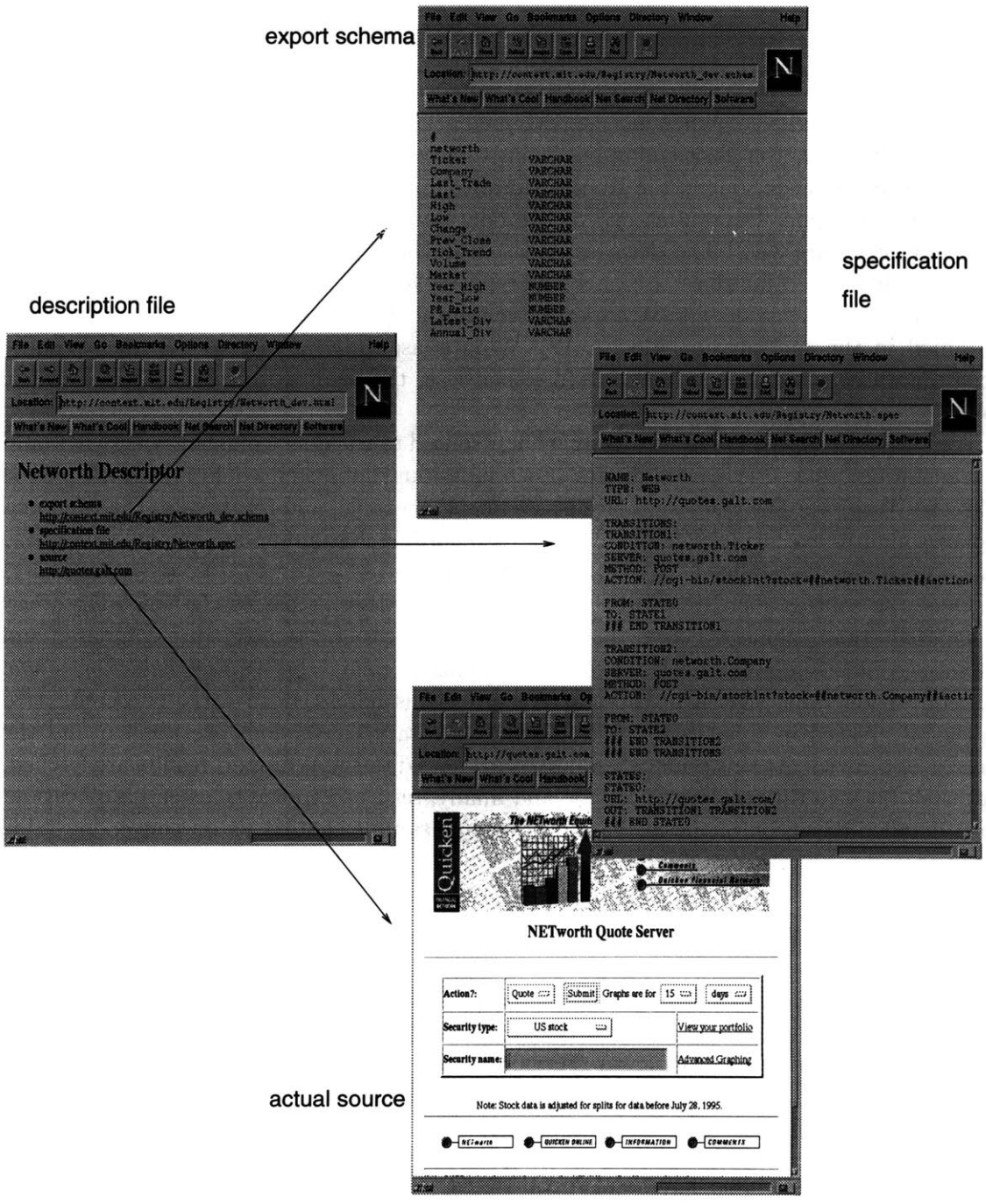

Figure 3-2: Description file for Networth. Entries for an export schema, specification file and the actual quote server shown.

the site's server, in the case of Web sources[2]. An export schema for the Networth site and a corresponding specification file are shown in Appendix A.

To publish data with COIN, the source merely needs to define an export schema and write a specification file.

3.1.3 Mediator Services

In the simplest scenario, when no context mediation is performed on the user queries, COIN simply allows the front end applications to access registered data sources. When the query involves a single source, the Router simply ships the query off to either, for the Web sources, the Wrapper Generator, or, for the relational databases, to the WWW-database gateway.(see Figure 3-1). The decision is based on the type information contained in the specifications file (so far there are only two types: "WEB", for sources such as Networth, and "DB" for databases). The Wrapper processes the query, according to the information contained in the specification file. A WWW-database gateway passes the query to the actual database system for execution. In either case, the results are returned back to the Router.

A multidatabase query would ideally be first routed to the query planner, where it would be transformed to a series of subqueries. Each individual subquery would in turn be passed on to the Wrapper. The results would be assembled in a table, and returned to the front end application after executing *SQLFetch* or *SQLExtendedFetch* commands. The multidatabase query processing capability is currently not implemented.

Once the context mediation process is enabled, the Router passes the incoming query to the Mediation Engine. At first, the contexts[3] of sources and receivers are compared and semantic conflicts among them are detected. This is called the ***conflict detection*** stage. The conversion table that lists all the transformations necessary to carry one context to another is created in this stage. Next, all the necessary conversions are applied and the query is optimized. At this point, the query is presented to the Wrapper, results are fetched, converted back to the receiver's context, and finally, returned to the front end application. For a detailed discusion of the mediation process, a sample context and a conversion table refer to [3].

3.2 Design of Multidatabase Browser

The Multidatabase Browser is an example of a COIN client application. The main goal of the Browser is to guide the user through the process of formulating an SQL query to be posted to the mediator middleware services. The design of the Browser was focused on making the interface as clear and intuitive for the users as possible, subject to the constraints imposed by the Web environment.

The SQL query itself is composed in a Query By Example (QBE) fashion, similar to Microsoft's Access. Studies have shown [15] that QBE takes less training time and results in a higher query writer confidence when compared to bare SQL. In a typical interaction, the user goes through a set of screens and narrows down the choices for his query step by step. First, the user has to specify which data source, or data sources he wants to query

[2]This interaction is described in detail in Section 3.3

[3]all sources and receivers must describe their contexts beforehand, with respect to a collection of shared ontologies. For a good discussion on contexts, ontologies and conflict detection see [7]

from the list of all sources registered with COIN. Then, he needs to select tables of interest, attributes from those tables, and finally place some constraints on the requested data - write the "WHERE" clause. At this point, the Browser supports simple conditions and conjunctions of conditions. Future plans include adding disjunction capabilities. A detailed example of the query formulation process is described in Section 4.2.

Even though the users are expected to have some basic understanding of the SQL syntax, the formulation of a query might be a confusing and frustrating process, especially when the user does not have prior experience with the source he is querying. The Multidatabase Browser addresses this issue by providing an extensive on-line help utility. At each point throughout the interaction, the user can view the query that he has created. A help file is available to answer stage specific questions about the meaning of checkboxes and pushbuttons on the Browser screens. This file also includes a guided tour, which takes the user through composition of a simple query. A *"mailto"* link is also provided on every page for more complicated questions, comments and bug reports.

Before sending the query to the router for the execution, the user can select his local context and switch on the mediation process. Query results, mediated or not, are presented back to the user in the form of the table.

In the early implementations of the Browser, it was realized that the users often want to perform the same query repeatedly over a prolonged period of time. Checking a company stock price on a daily basis is an example of a such query. Rather than having to go through all the steps of composing the appropriate SQL query every time the user wants to execute it, a feature was added to the Browser that allows users to save their favorite queries, along with a short description in a file. An *"open query"* button allows for the execution of such a query directly from the Browser's entry page.

3.2.1 Interface Design Considerations

In designing the Multidatabase Browser, special emphasis has been placed on making the interface as user friendly as possible. The growing number of Web browsers provide varying support for the HTTP protocol and various features of CGI. Some browsers, such as Lynx, are text-based and do not support graphics. For this reason, it was stressed that the Multidatabase Browser should use only the features that are most commonly available across browsers. The interface has very little graphics and no background picture, which not only provides consistent appearance but also preserves the network bandwidth and speeds up the interaction process.

Many Web applications use custom bitmapped graphics or image maps for purposes of navigation between pages. The Multidatabase Browser uses standard submit buttons with names describing where each one of them leads to. The users never have to wonder where to click on the page to get to the next stage of the interaction, which eliminates possible confusion.

Each page of the Browser corresponds to a different stage in the process of formulating the SQL query, but all of them have the same layout and uniform look. The length of each page depends on the characteristics of the particular source that is queried[4], but the page layout is compact to achieve short pages for most typical data sources, and thus eliminate unnecessary scrolling and mouse travel. This might seem like a subtle point, but studies have shown that users are less likely to pay attention to the material if it appears at the

[4]i.e. how many relations/attributes does it have.

bottom of the page, or they might not look at it at all [11].

3.3 The Wrapper Generator

The Wrapper Generator is an engine that allows for the incorporation of the data originating from Web sites (and possibly other legacy systems) in the Context Interchange Network. The Wrapper automatically performs all the steps necessary to obtain data values from the Web site. It issues HTTP commands directly to the data server, thus mimicking the interaction that would normally take place between a human user and a WWW site.

By providing an automated way of retrieving information from Internet sources, the Wrapper Generator greatly expands COIN's knowledge pool. The Wrapper establishes a uniform way of communicating with any Web server, which helps COIN's users avoid the headaches of learning the individual interaction patterns for a number of sources. The query mechanism chosen for the legacy sources is SQL. It is the Wrapper's job to translate the user's SQL queries to native queries for each source. This requires the overhead of maintaining a recipe of the interaction on a per source basis. All this information is kept in a specification file. The payoff is having a uniform way of querying both legacy Web sources and relational databases.

3.3.1 Web accessible sources

Currently, the Wrapper Generator is capable of interacting with sources that run under the HTTP protocol[5]. In order to be incorporated in the Context Interchange Network, the source has to be semi-structured, i.e. it has to preserve persistent layout between interactions. This is necessary, because the description file which tells the Wrapper Generator where to look for the data items to be extracted from each HTML page relies heavily on the presence of certain keywords in the page[6]. The Wrapper Generator also assumes that the names of CGI input variable do not change over time. Some documents randomly generate field names for the widgets and for that reason they cannot be published through COIN.

3.3.2 Specification File

The Wrapper Generator can be thought of as a very high level intepreter: it executes steps necessary to answer SQL queries to the Web sites, based on the recipe contained in the specification file. Because the specification file contains all the details necessary to perform an automated document retrieval from the Web site, along with patterns that match the data values to be extracted from those documents, writing a specification file is by no means an easy process, and requires that the users are familiar with the basics of HTTP and HTML. This task should be performed by the site maintainers.

The specification file is a template for the interaction with the Web server that results in retrieval of information that is interesting for the COIN user. Typical interactions are filling out an HTML form, making a selection from a popup menu (followed by, in both cases, pressing a submit button) or simply clicking on any of the links in the page. Not all the possible interactions with the server have to be modeled in a specification file. Sending

[5]Future implementations might include other protocols, such as FTP, for example.

[6]For example, in the name of a particular page is probably located between <TITLE> and </TITLE> HTML anchors

an electronic mail comment to the site webmasters, for instance, should probably not be represented, because it has no relevance to the information that COIN can obtain from that particular site. Thus far, specification files have been written for sources of financial information, such as the Olsen & Associates currency converter[7], and Security APL Quote Server[8].

The model that best represents the information that is contained in a spec file is a directed, acyclic graph, where nodes correspond to the HTML documents, and edges correspond to the HTTP actions that need to take place in order to get to those documents. Currently, the Wrapper supports two types of actions, namely the HTTP Post and Get methods. Following a URL link corresponds to HTTP Get, and pressing a submit button usually corresponds to HTTP Post.

The specification file for the Networth source is listed in Appendix A. The file can be divided into three main sections: general information, a list of edges (transitions), and a list of nodes (states). Throughout the spec files, all the variables are surrounded by double hash marks (##). We call the variables that must be provided in a query (e.g. ticker symbol when querying a stock quotes server) bound variables, and those that can be retrieved at a particular step (from an HTML document), and do not have to be specified beforehand, free variables. Each transition contains the name of the HTTP server that publishes the document for the upcoming state, a retrieval method (either Post or Get), and a path to the actual document on that server (including CGI query variables). Should the transition require a bound variable to be specified a priori (such as a ticker symbol in the scenario described above), it would be listed under *"Condition:"* section. Each state enumerates all the outgoing transitions. The transition actually taken when leaving the state is determined by the input query in a deterministic way. For each free variable to be retrieved in a given state the state description contains a "pattern" to be matched against the entire HTML document. For example:

"networth.Ticker ,\s\((.?)\)\</FONT\>#"*

means that the networth.Ticker free variable should be found between a "," and a </FONT> anchor[9].

The state diagram for the Networth site is shown in Figure 3-3. The two transitions out of State0 correspond to the two possible ways in which the Networth server can be queried: the user can either enter the ticker symbol, or a company name (the third possible way to query Networth server would be to look for graphs of ticker trends for a given period of time and get a GIF image as a result. COIN does not support non character data at this point, so this type of interaction would not be represented in a specification file). There might be more then one answer to a string search on a company name[10], so that the states that those two transitions lead to are actually different: State1 yields a specific quotes for a given ticker symbol, State2 lists all the company names matching the query string.

---

[7] *http://www.olsen.ch/cgi-bin/exmenu*

[8] *http://www.secapl.com/cgi-bin/qso*

[9] For more information on Perls regular expressions consult Perls on-line documentation: *http://www.mit.edu/perl/perlop.html*

[10] entering *"shell"* yields several responses, including: "Shell Trans & Trading Plc", "Shell Canada Ltd", and "Shells Seafood Restaurants Inc".

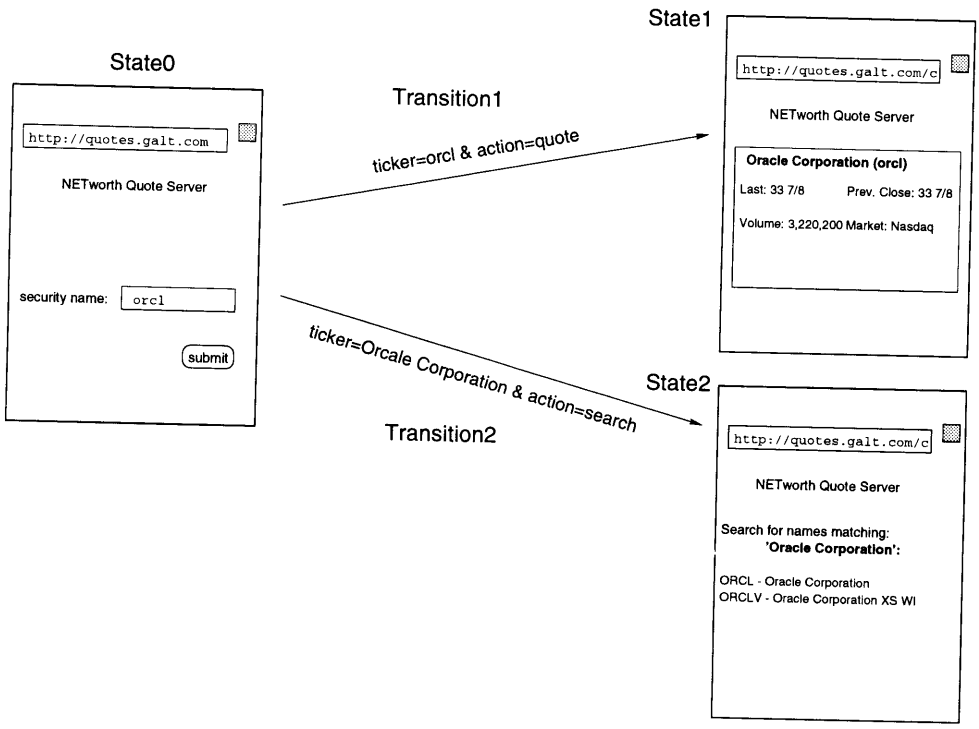

Figure 3-3: State transition diagram for Networth quotes server.

Chapter 4

Implementation

This chapter describes the implementation details of the Multidatabase Browser and the Wrapper Generator, as two examples of Web based applications.

4.1 CGI and Perl

The Multidatabase Browser is a Common Gateway Interface (CGI) version 1.1 script. The choice of using CGI for the purpose of implementation of the Multidatabase Browser was well justified when the Browser was first written, in the summer of 1995, due to the lack of other paradigms for Web programming. Nowadays, with the availability of such powerful programming languages as Java and JavaScript, this choice could be questioned. In defense of CGI comes the simplicity of the interface, good on-line documentation, and also some inherent problems with Java, which will be mentioned in Chapter 5, section 5.0.3 on page 49.

The second important implementation choice was a choice of a programming language to write the code itself. Since it first came around, CGI has been implemented in many languages, such as C, C++, Tcl and others. For this project, the language chosen was Perl5, written by Larry Wall. There were two main reasons for this decision. First of all, Perl is an excellent language for a variety of tasks, especially those which require text management and data parsing. It has built-in memory management and other security features which can save implementors a lot of headaches and result in more secure programs that are also easier to maintain. Finally, it has several convenient high-level constructs (list and hash tables as datatypes, regular expression operators) which account for fewer lines of code to do the same task as programs written in C, for example. Moreover, fewer lines of code usually translate to fewer potential bugs[1]. The second important reason for choosing Perl was that several Perl libraries that implement CGI are publicly available. In this project, it was decided to use the CGI.pm package[2].

CGI.pm is a Perl5 library written and distributed by Lincoln D. Stein[3]. CGI.pm uses Perl5 objects to create fill-out forms on the fly and parse their contents [13]. It provides a simple interface for parsing and interpreting query strings passed to CGI scripts. CGI.pm provides a set of functions that create HTML elements each time the script is invoked, so

[1]Of course there are some trade offs, when using Perl, such as performance, as it is always an issue with interpreted languages.

[2]The most recent version of the library and a copyright notice can be found at *http://www-genome.wi.mit.edu/ftp/pub/software/WWW/*

[3]Whitehead Institute, MIT Center for Genome Research. Email: *lstein@genome.wi.mit.edu*

there is no need for remembering exact HTML syntax when using this library. CGI objects also handle correctly HTTP Post and Get methods. Provided you have created a new CGI object called **query**, you can create a text field named *'foo'* initialized to the string *'hello'* as follows:

```
print $query->textfiled('foo', 'hello');
```

Other form elements are handled in a similar fashion.

4.2 Multidatabase Browser Interface

The Multidatabase Browser can be found at:
http://context.mit.edu/Development/browser_mediated.cgi.
The best way to understand how interaction with the Browser works is to go through an example. We will therefore query the Networth site for the latest stock price of Oracle Corporation stocks[4].

4.2.1 A typical SQL query

The following sections list the most important steps involved in a typical user interaction with the Browser.

- **Data source Selection**

  The first step in formulating a query is selecting a data source to be queried (or several data sources, in the case of a multidatabase query) by pressing an appropriate button. The data source selection requires, unfortunately, that the user knows which data source contains information valuable to him. To some extent, this problem can be overcome by browsing through the context of each source. The link to the Ontology Editor, which allows one to view and modify contexts and underlying ontologies, is located on the right hand side of the data source name (see Figure 4-1). Only users with prior exposure to the concept of ontology are recommended to use the link. Next to the link to the Ontology Editor, there is a link for the original source of data. In case of RDBMS's this is a link to the database - WWW gateway, in case of Web based sources - it is a URL for the actual HTML pages.

  Once the data source has been selected, the user then clicks on the *"choose a database"* button to proceed. An *"open query"* button is also provided as a way to short cut to execute pre-saved query.

- **Table Selection**

  When the data source has been specified, the next step is to chose relations of interest to the user. This corresponds to building the "FROM" clause of the SQL statement. For example, one can select table *networth* from source Networth (Figure 4-2). Other sources can have more than one table. Before proceeding to the next stage, the user has to press the *"submit table selection"* button.

[4]the actual SQL query would be:
select *networth.Ticker, networth.Company, networth.Last*
from *networth*
where *networth.Ticker = 'orcl'*

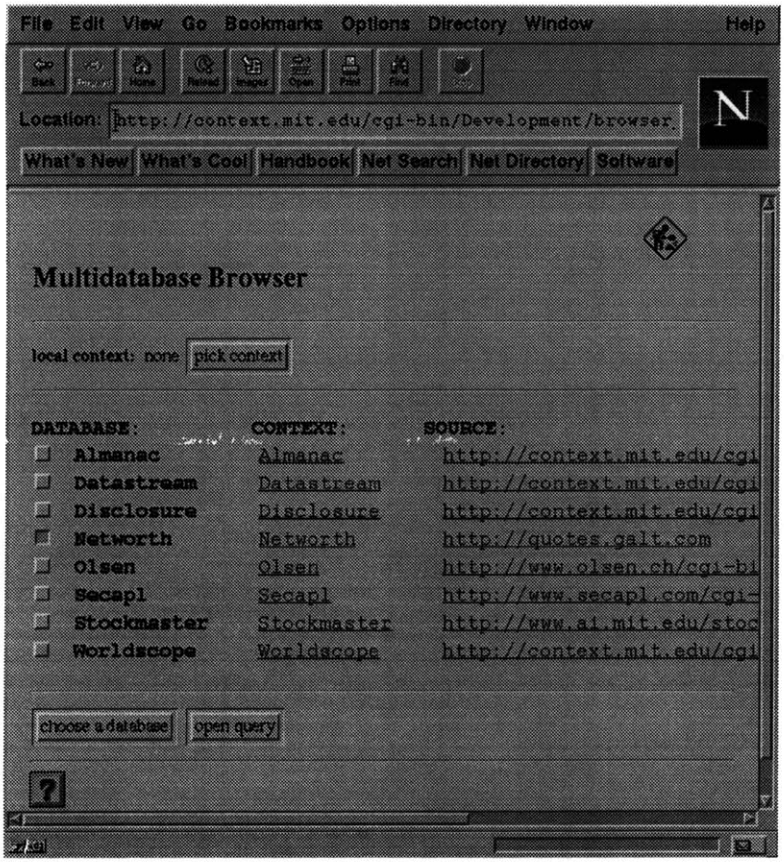

Figure 4-1: User selects Networth source to be queried.

- **Attribute Selection**

  In this step, the user selects attributes (columns) of the relations (tables) that were chosen in the previous steps. Those are both attributes that end up in the "SELECT" and "WHERE" clause of the final query. The user is not committed to the final "SELECT" clause here - instead he should choose all attributes of interest to him (Figure 4-3).

- **Building "WHERE" clause**

  This is perhaps the most confusing step in formulating the SQL query. In the left column, all the attributes that were selected in the previous stages are listed. In order to output attributes of one's choice to the final target list, the corresponding checkboxes should be clicked. The right hand side column contains conditions which form the "WHERE" clause. Initially, no conditions are specified and an "edit condition" message appears next to each button. In order to specify a restriction on a given attribute, the user should select the appropriate checkbox and depress the *"edit condition"* button. The "WHERE" clause becomes a conjunction of all the conditions specified by the user (Figure 4-4).

- **Displaying the Query**

  Should the user want to see the SQL query composed so far, the *"display query"* button is provided at each screen (Figure 4-5).

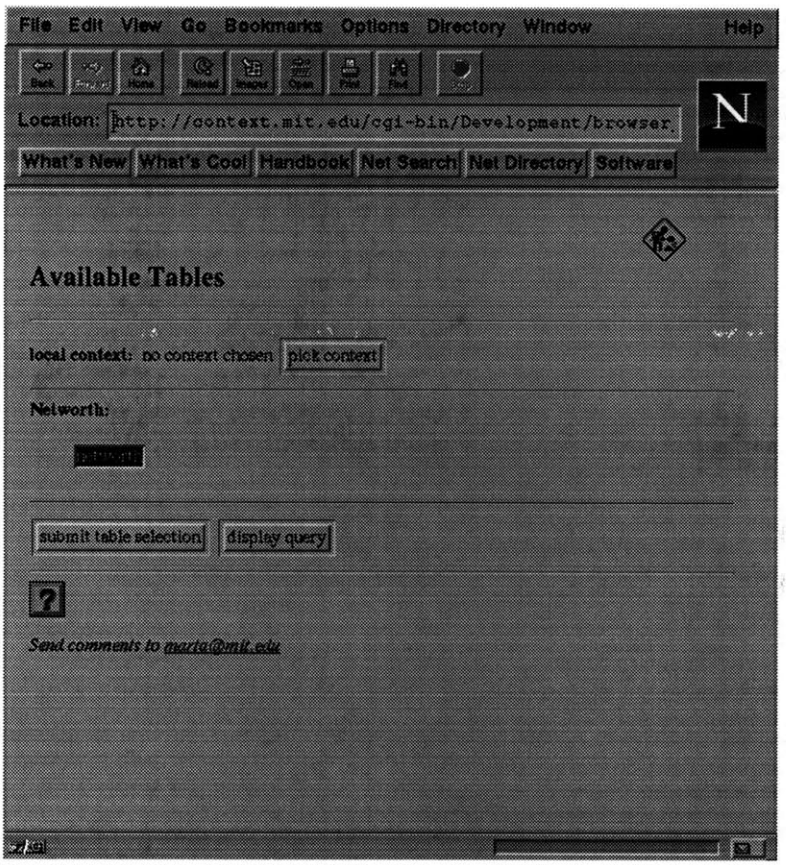

Figure 4-2: Networth source has only one relation - "networth".

- **Query Results**

  Finally, the user presses the *"execute button"*, and query results are displayed in a tabular form (Figure 4-6). At this point, the user may choose to diplay the final query, save the results or proceed with a new query.
  Any questions and comments about the Browser should be directed to *webmaster@context.mit.edu.*

4.2.2 Testing

The Multidatabase Browser script has been tested on a Netscape 2.1 browser (as well as earlier versions) on a Unix platform. Since the script contains only standard CGI elements, it should also run under any other browser that supports the following features:

- **hidden data** - this is necessary in order to maintain the state information between script invocations. For example, the knowledge of what data sources, relations and attributes have been selected thus far is kept in the form of hidden fields. Hidden fields are similar to textarea or checkbox fields, except they are not displayed when HTML source code is rendered by the browser. One can see hidden data when using "view source" option of the browser[5]

[5]Here is an example of hidden variable that keeps the table selection information:
<**INPUT TYPE**="hidden" **NAME**="Networth:tables" **VALUE**="networth">

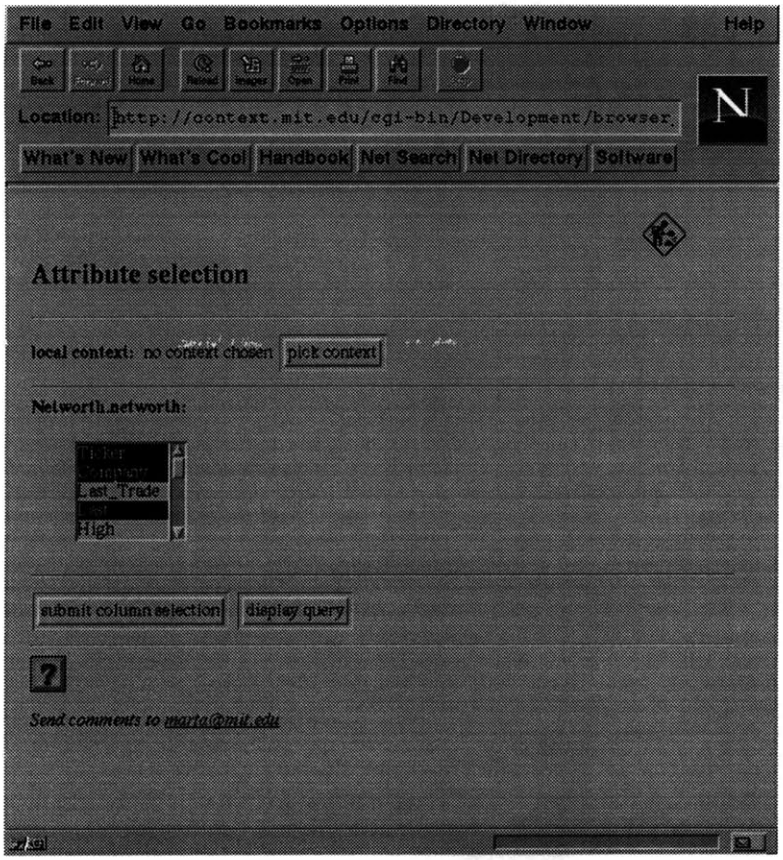

Figure 4-3: "Ticker", "Company" and "Last" attributes are selected.

- **named submit buttons** - those are required to provide several options when exiting a form. In the case of the Multidatabase Browser, one can choose, for example, to either proceed with table selection or to view the SQL source for the query that was created up to that point.
- **multiple selections from a list** - this is the only way for users to pick more then one item from a list, such as picking several data sources to queried.

In order to find out whether a particular browser supports features listed above, a simple test can be run using Digital's HTML Form-Testing Home Page, located at:
http://www.research.digital.com/nsl/formtest/home.html.

4.3 Wrapper Generator Algorithm

As was mentioned before, the purpose of the Wrapper Generator is to provide COIN with on-the-fly access to data originating from publicly accessible Web sites. The Wrapper Generator is an anonymous WWW agent which directly communicates with the WWW servers using the HTTP protocol. It visits sites specified by the specification file, retrieves relevant pages, and extracts pre-defined sets of values.

The Wrapper takes in two inputs: a specification file and an SQL query to be performed against the site. The SQL query is assumed to be a simple conjunction of conditions. It is

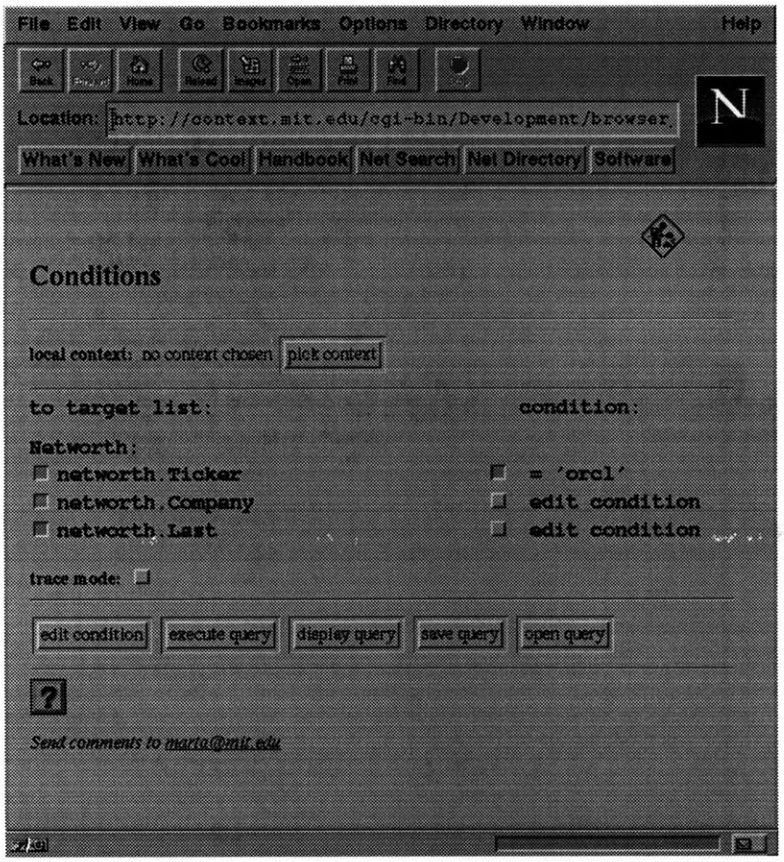

Figure 4-4: The user wishes to see the latest stock price and company name for Oracle Corporation.

beyond the current Wrapper Generator's capability to deal with more complicated queries. Provided that the source is capable of answering the input query, the Wrapper Generator returns answers in the form of an HTML table. It is the specification file that defines the interaction with a WWW server or a set of servers which yields to document retrieval. The specification file consists of states and transitions. It is essentially a directed acyclic graph, where states correspond to nodes, and transitions to edges between those nodes. Each state corresponds to a single HTML document. Each state's description contains a list of Perl5 patterns, or regular expressions, corresponding to keywords that are extracted in that stage. The patterns use HTML anchors to specify the relative location of values of interest in the document. To avoid repeated retrieval of the same document from the server, local variables should be used. Such variables are surrounded by double hash marks (##). Local variables can only appear in the body of the pattern if their corresponding values were retrieved in the states preceding the current state in the execution path, or were retrieved by earlier patterns in that state. The order of patterns in the state specification is, therefore, essential. For example, let us picture a document that contains a table of company stock information keyed by date, and current date is shown on the top of the page. In order to obtain the latest stock quotes from that document, the Wrapper Generator would first have to retrieve the current date, store it in a bound variable ##DATE##, and then use this variable to find out the location of the latest quotes in the table. The corresponding regular expressions would look as follows:

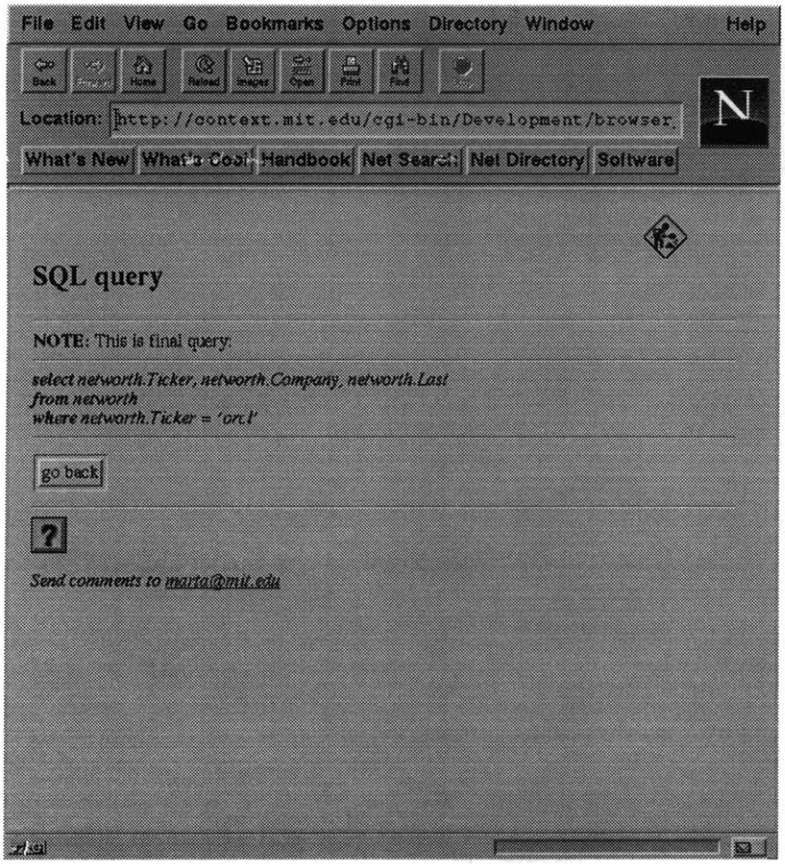

Figure 4-5: The SQL syntax.

Variable name:	Pattern:
DATE	Current date (.*?).#
STOCK_PRICE	##DATE##...(.*?)...#

The transitions describe HTTP methods that should be used for the retrieval of the document corresponding to the next state in the query execution. Two methods are supported: HTTP Post and Get. Each transition's action URL can be parameterized much like patterns in state specification. Each transition may contain a condition which puts a constraint on when the transition should be taken. Conditions are either local variables, assumed to be extracted from the earlier documents by the time the transition is taken, or values supplied by the constraints of the input SQL query. For the Networth source, the ticker symbol has to be specified before obtaining the stock information, so it is a condition for one of the transitions leaving State0 (See Figure 3-3 on page 33).

The process of answering the SQL query can be divided into two steps. First, the Wrapper Generator builds up a plan for the query execution. The plan always starts with State0. Transitions originating from that state are considered in the order they appear in the state description. For each transition, the conditions are matched against the set of attributes and local values known thus far. At the beginning, the set of known attributes is initialized to values supplied by the constraints of the query and whatever keywords were extracted in State0. The first transition for which conditions are satisfied is added to the plan along with its destination state. The plan is build incrementally until the state is reached for which there are no outgoing transition, i.e., a final state. At this point the plan

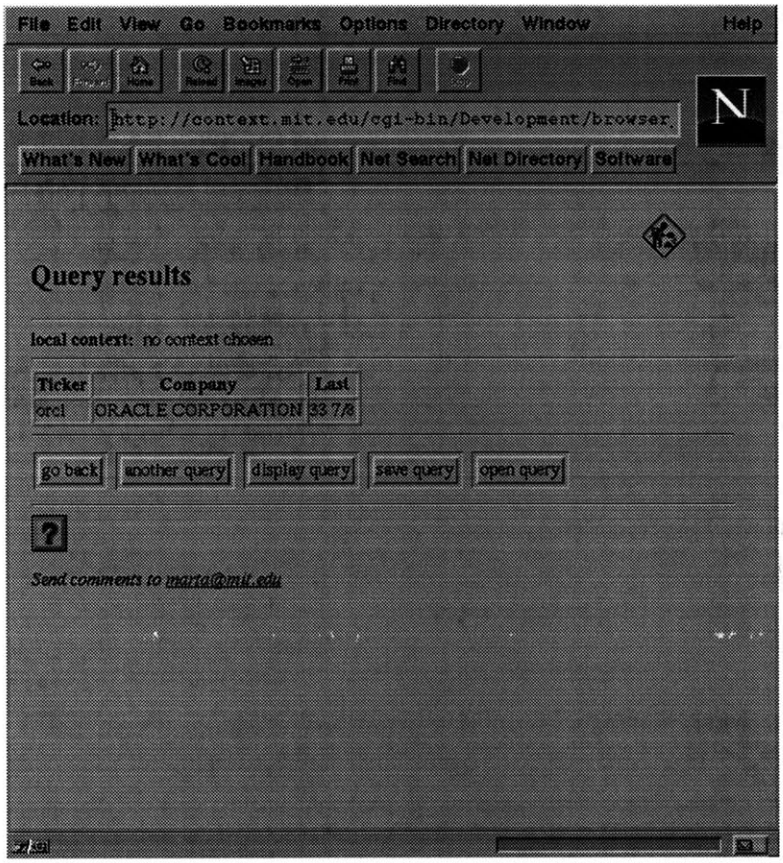

Figure 4-6: And finally, the results.

is passed to the Wrapper's execution engine. For each state, the engine retrieves an HTML document via HTTP Post or Get and extracts all the keywords, replacing all free variables with values known so far. For each transition, the engine looks up the method and action for the document to be retrieved next. When a final state is reached, the engine generates an HTML table based on the attributes requested in the SQL query and the values retrieved by the plan execution. The entries in the table appear in the same order as the query targets.

4.3.1 Problems and Limitations

The Wrapper Generator provides a solution to the problem of incorporating data published on the Web in the COIN system. It is important to realize, however, that the methods employed by the Wrapper are by no means a general solution to this problem. To build an efficient tool that extracts information from a variety of WWW sources, each having a unique interface and a unique interaction pattern, and present all this information in a uniform way, is not an easy task. The main trade-off is between having a single query interface for all the sources and requring to write a specification file for each Web source. The complexity of the interaction with a WWW site is buried at the level of specification file.

It is not a suprise, that each time the location of any of the sites described by the specification file changes, the file has to be updated accordingly. The same holds true for any changes made to the layout of the documents, because regular expressions rely heavily

on the location of certain HTML tags and keywords in the document. This problem can, to some extent, be overcome by making regular expressions as general as possible and hoping that, should the document change slightly, the patters will still hold. In case of either changes to the document location[6] or updates to the page layout, no changes are needed to the Wrapper Generator script.

Another problematic issue is that of the specification file. Writing a new specification file requires a good understanding of how the underlying WWW sites operates. The specification file writer should have a good understanding of HTTP and CGI. He should be able to model the site in the form of a state transition diagram and abstract the human user interaction that the site is built for to the level of the inner workings of CGI. For example, the action of clicking on a submit button is nothing else than setting the CGI variable "submit_button" to its "on" value. Once the writer has the general picture of how to represent the site in a form of states and transitions, he has to specify how to extract data from the site: write regular expressions. This step requires both understanding of a relational data model and knowledge of Perl5 regural expresions. So far all the specification files for the sources registered with COIN have been writtern by project team members, and it has not been tested how a person from outside our research group would perform a similar task.

[6]It should be noted that the URL changes are perhaps one the most annoying problems with the Web today. Each time the document's location and thus URL changes, all the references to that document have to be updated manually.

Chapter 5

Lessons Learned and Future Directions

The Multidatabase Browser has proved to be a very useful tool that provides a single query interface to both relational databases and World Wide Web applications. The Browser's position in the Context Interchange system is very strategic: it is currently the only front end interface to COIN. In the future, other applications such as Excel and Access will also be able to hook up to COIN using the standard interface of ODBC-like calls (*SQLDataSources, SQLExecDirect*, etc.). Until that happens, the Browser will be responsible for carrying out all the user interaction.

5.0.2 Future Work - Multidatabase Browser

Overall, the Multidatabase Browser has proved to be a very successful interface. This chapter discusses some additions to the Browser that could improve its usefulness and practicality from the user's point of view.

One of the most important improvements would be to extend the Browsers capabilities in terms of the types of queries it can produce. Right now, the Browser can only formulate simple SQL queries, that is, conjunctions of conditions and simple joins on attributes. It would be desirable for the Browser to handle "or"'s as well as "and"'s and possibly other SQL operators. In order to support more operators, changes would have to be made to the Browser's interface. A scrolling list with a greater selection of operators could be added, so that the users could specify how they wish to combine simple conditions in their queries.

The next issue which was not addressed very well in the current implementation is the issue of user's privacy. The Browser was primarily used within the COIN research group, and there was no immediate need to worry about the security problems that arise when the number of users increases. Should the Browser be presented to a larger audience, certain measures would have to be taken to ensure individual privacy. First of all, a username/password login mechanism should be implemented[1]. The main goal of a such mechanism would be to give each user a private working space, where he could save his favorite queries, define his context and perform other customizations. User identification is also necessary for developing access control in COIN. It is very likely that, in the future, it will be desired to vary users privileges to access certain data sources or change the

[1] A prototype login screen can be found at:
http://context.mit.edu/cgi-bin/mint.cgi

definitions of some contexts.

After the prototype version of the Multidatabase Browser had been developed, an important question was raised regarding the language chosen for implementation purposes. As mentioned before, the HTML/CGI style of programing does not support certain features that one would like to see in an fully interactive application. There are only two ways to transmit information to the WWW application; selecting an HTML link transmits a request for chosen document, and pressing a submit button transmits the state information about the widgets. It is therefore not possible for a server to determine anything about intermediate activities performed by the user, such as typing text, toggling radio buttons or moving the mouse. This is a serious limitation to some of the advanced features of user interfaces, such as providing immediate feedback to the user. In the case of the Multidatabase Browser, it would be nice to have the syntax of the SQL query be displayed to the user on the fly, instead of having to go through a separate page. It would also be nice to see a visual display of what happens after the user query is submitted for execution. One could imagine, for example, a picture that shows a COIN architecture diagram (such as Figure 3-1 on page 23) with arrows moving across to indicate the flow of data. It would be much preferable for the user to have this kind of visual feedback as opposed to just clicking a *"submit query"* button, waiting, and getting the results back after a while. In the HTML/CGI world, this type of immediate visual feedback (synchronized with the background processing of the query) is impossible to implement. What is needed to make it happen is a programming environment that supports asynchronous communication and animation. For those reasons, it was considered to have the Multidatabase Browser completely rewritten in Java.

5.0.3 Java and Javascript

Java is an object oriented and platform independent programming language developed at Sun Microsystems. Java's popularity is due to its simple yet robust code and cross platform portability[2]. From the perspective of the Multidatabase Browser, however, its most crucial feature is its support for multithreading. Java allows building applications with many concurrent threads of activity. This results in a high degree of interactivity for the end user. One can imagine, for example, an application that executes users query and concurrently displays an animation clip that explains what happens. Java applications can easily be incorporated in HTML pages by placing an applet tag inside HTML source (for example, the tag <**APPLET CODE**="Applet.class" **WIDTH**=150 **HEIGHT**=150></APPLET>, tells a Java capable browser to run the java code Applet.class in a box with width 150 pixels and height 150 pixels). Currently, the only Browser capable of interpreting applet tags is Netscape 2.0 or higher versions.

The problems with using Java to implement the Browser is one of the built in security mechanisms. It is very hard to write an applet that both communicates with the network and a local file system. Both features are required for the would-be Multidatabase Browser applet: the file system access is required to open/save user queries and update context information, the network access is required to communicate with the Wrapper Generator script and Datasource Registry. In Netscape, any attempts by an applet to read or write the local file system results in a dialog box for the user to grant approval, which would only complicate user interaction. Another serious problem is the number of demonstrated flaws,

[2]For more information of Java see Sun's Java homepage at:
http://java.sun.com/

which compromise security of Java applets [5].

Another language that supports high level graphical user interface features is Javascript, developed by Netscape Communications Corporation. It resembles Java in most features, but it is a scripting language, rather then a programming language. It was designed for Web developers rather then object oriented programmers, and it is easier to use then Java[3]

5.1 Conclusions

The Multidatabase Browser and the Wrapper Generator are two tools developed for the Context Interchange project. Both tools demonstrate some very important ideas to the future of information exchange over the Internet: the Multidatabase Browser provides a single query SQL access to heterogenous data sources, and the Wrapper Generator allows for the incorporatation of new WWW sources in the COIN system.

[3]Javascript documentation can be found at:
http://home.netscape.com/

Appendix A

Networth's description files

A.1 An Export Schema

```
#
networth
Company        VARCHAR
Ticker         VARCHAR
Last_Trade      VARCHAR
Last           VARCHAR
High           VARCHAR
Low            VARCHAR
Change         VARCHAR
Prev_Close      VARCHAR
Tick_Trend      VARCHAR
Volume         VARCHAR
Market         VARCHAR
Year_High      NUMBER
Year_Low       NUMBER
PE_Ratio       NUMBER
Latest_Div     VARCHAR
Annual_Div     VARCHAR
```

A.2 A Specification File

```
NAME: Networth
TYPE: WEB
URL: http://quotes.galt.com

TRANSITIONS:
TRANSITION1:
CONDITION: networth.Ticker
SERVER: quotes.galt.com
METHOD: POST
ACTION: //cgi-bin/stocklnt?stock=##networth.Ticker###&action=0&period=
15&periodunit=0&sectype=0&submit=Submit
```

```
FROM: STATE0
TO: STATE1
### END TRANSITION1

TRANSITION2:
CONDITION: networth.Company
SERVER: quotes.galt.com
METHOD: POST
ACTION: //cgi-bin/stocklnt?stock=##networth.Company##&action=2&period=
15&periodunit=0&sectype=0&submit=Submit
FROM: STATE0
TO: STATE2
### END TRANSITION2
### END TRANSITIONS

STATES:
STATE0:
URL: http://quotes.galt.com/
OUT: TRANSITION1 TRANSITION2
### END STATE0

STATE1:
URL: http://quotes.galt.com/cgi-bin/stockclnt
OUT: NONE

REGULAR EXPRESSIONS:

networth.Company        \<FONT\sSIZE=\d\>(.*?),\s+\W\w+\W#
networth.Ticker        \s\((.*?)\)\</FONT\>#
networth.Last           Last\</A\>\s+(.*?)\s+\<A#
networth.Last_Trade     \(last\strade:\s(.*?)\sEST\W#
networth.Low            Day\sRange\</A\>\s\<TD\>\s*(.*?)\s*-\s*\d#
networth.High           Day\sRange\</A\>\s\<TD\>.*?\s*-\s*(.*?)\s\s+#
networth.Change         Change\</A\>\s+(.*?)\n#
networth.Prev_Close     Prev.\sClose\</A\>\s(.*?)\n#
networth.Tick_Trend     Tick\sTrend\</A\>\s+\W+TD\>(.*?)\</TD\>#
networth.Volume         Volume\</A\>\s+(.*?)\s+\<A#
networth.Market         Market\</A\>\s+(.*?)\s+\n#
networth.Open           Open:\</A\>\s+(.*?)\s+<A#
networth.Year_Low       52\sweeks\Range\</A\>\s+(.*?)\s-#
networth.Year_High      52\sweek\sRange\</A\>\s+.*-\s(.*?)\n#
networth.PE_Ratio       P/E\sRatio\</A\>\s+(.*?)\s+\<A#
networth.Latest_Div     Latest\sDiv.\</A\>\s+(.*?)\s*\<A#
networth.Annual_Div     Annual\sDiv.\</A\>\s+(.*?)\s*\</PRE\>#

### END STATE1

STATE2
```

```
URL: http://quotes.galt.com/cgi-bin/stockclnt
OUT: NONE

REGULAR EXPRESSIONS:

networth.Company        &stock=.*"\>.*\s-\s(.*?)\</A\>#
networth.Ticker         &stock=.*"\>(.*?)\s-#

### END STATE2
### END STATES
```

A.2.1 Legend

The specification file uses Perl5 regular expressions. A good discussion of regular expressions can be found on-line at:
http://www.mit.edu/perl/perlop.html[1].
The meaning of Perl operators used in this specification file is expplained below:

\s	a single whitespace character (e.g. space)
\s+	1 or more spaces
\s*	0 or more spaces
\W	A non-word character
\n	newline
\d	a single digit
\<,\>	'<' and '>' are reserved characters for Perl5 regural expresions and have to be preceeded by a backslash
#	# is COIN convention for specifying the end of a pattern.

[1]For an overview of Perl5 patterns one can also see [14]

Bibliography

[1] Tim Bernes-Lee. Hypertext Transfer Protocol.
URL ftp://info.cern.ch/pub/www/doc/http-specs.ps, 1993.

[2] Eric Bina, Rob McCool, Vicky Jones, and Marianne Winslett. Secure Access to Data over the Internet.
URL http://bunny.cs.uiuc.edu/CADR/WinslettGroup/pubs/secureDBAccess.ps, 1994.

[3] Adil Daruwala, Cheng Goh, Scott Hofmeister, Karim Hussein, Stuart Madnick, and Michael Siegel. The context interchange network prototype. Working Paper 3797, MIT Sloan School of Management, MIT Sloan School of Management, Cambridge, Massachusetts, February 1995.

[4] C. J. Date. *An Introduction to Database Systems*, volume 1 of *Addison-Wesley System Programming Series.* Addison-Wesley Publishing Company, fourth edition, 1986.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach.
Java security: From hotjava to netscape and beyond. In *1996 IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

[6] Brian R. Gaines. Porting Interactive Applications to the Web.
URL http://ksi.cpsc.ucalgary.ca/KSI, 1995.

[7] Cheng H. Goh, Stuart E. Madnick, and Michael D. Siegel. Ontologies, Contexts, and Mediation: Representing and Reasoning about Semantics Conflicts in Heterogenous and Autonomous Systems. Working Paper 3848, MIT Sloan School of Management, MIT Sloan School of Management, Cambridge, Massachusetts, August 1995.

[8] Martijn Koster. Robots in the web: threat or treat?
URL http://info.webcrawler.com/mak/projects/robots/threat-or-treat.html,
1994.

[9] Microsoft Corporation. *ODBC 2.0 Programmer's Reference and SDK Guide*, 1992.

[10] B. Pinkerton. Finding what people want: Experiences with the webcrawler. In *Proceedings of the Second International World Wide Web Conference*, Chicago, IL, October 1994.

[11] J. Rice, A. Farquhar, F. Piernot, and T. Gruber. Lessons Learned Using the Web as an Application Interface.
URL http://www-ksl.standford.edu/KSL_Abstracts/KSL-95-69.html, September 1995.

[12] Lincoln D. Stein. *How to Set Up and Maintain a World Wide Web Site. The Guide for Information Providers.* Addison-Wesley Publishing Company, 1995.

[13] Lincoln D. Stein. Cgi.pm - a Perl5 CGI Library.
URL http://www-genome.wi.mit.edu/ftp/pub/software/WWW/, 1996.

[14] Dave Till. *Teach Yourself Perl in 21 Days.* UNIX Librabry. SAMS Publishing, first edition, 1995.

[15] Sylvia Willie. Graphically Supporting Information Search. In *Proceedings QCHI94 Symposium,* Gold Coast, Australia, July 1994.