INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Data Extraction From the Web Using XML

By
Hicham Ouahid

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

M.A.Sc.
in
Electrical Engineering
Ottawa-Carleton Institute of Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
June, 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-66096-6

Canada

Table of Contents

List of Figures

List of Tables

Abstract

This thesis presents a mechanism based on eXtensible Markup Language (XML) to extract data from HTML-based Web pages and populate relational databases. This task is performed by a system called the XML-based Web Agent (XWA). The data extraction is done in three phases. First, the Web pages are converted to well-formed XML documents to facilitate their processing. Second, the data is extracted from the well-formed XML documents and formatted into valid XML documents. Finally, the valid XML documents are mapped into tables to be stored in a relational database.

To extract specific data from the Web, the XWA requires information about the Web pages from which to extract the data, the location of the data within the Web pages, and how the extracted data should be formatted. This information is stored in Web Site Ontologies which are built using a language called the Web Ontology Description Language (WONDEL). WONDEL is based on XML and XML Pointer Language. It has been defined as a part of this work to allow users to specify the data they want, and let the XWA work offline to extract it and store it in a database. This has the advantage of saving users the time waiting for the Web pages to download, and taking benefit from the powerful query mechanism offered by database management systems.

Acknowledgements

I would like to thank Dr. Ahmed Karmouch, for his supervision and for giving me the opportunity to work in the GUD project.

I would like to express my gratitude to Dr. Thomas Kunz and Dr. Dwight J. Makaroff for reviewing this thesis, for their valuable feedback, and for being members of the jury.

Many thanks go to Dr. Martin Bouchard for his support as acting supervisor, and for being member of the jury.

Special thanks to Dr. Monique Frize for being the chairwoman of the jury and for here interest in this research.

I would like to acknowledge the cordial relationship with each member of the GUD project team, a group of dynamic and friendly colleagues.

I would like to thank my parents and my siblings for their support and their encouragement.

Chapter 1

Introduction

1.1 Motivation and Objectives

This thesis work is a part of the Global University Database (GUD) project whose goal is to provide a database that stores up-to-date information about several universities around the world. Such information covers faculties, departments, courses, faculty members, and research groups and patents for each university. Although, this information already exists in the Web site of those universities, the objective to collect it and store it in a database for fast searching and retrieval, and also to be able to make complex SQL queries to get refined results.

The role of this thesis in the GUD project is to provide a means to populate the database with accurate data and reasonably low human intervention. To achieve that, a system called the XML-based Web Agent has been developed. This system is responsible for extracting information from university Web sites, organizing the extracted data in a format that can be stored in a relational database, and keeping the database consistent with the changes that may occur on the Web.

Achieving this goal faces many issues due to the size of the Web and its disorganized nature. First, the target information in Web pages is often mixed with other non-desired data. Second, this information is frequently scattered throughout many locations in the Web. In addition, HTML-based Web pages content is not necessarily machine understandable because HTML was designed only for human consumption [10]. Another impediment that complicates the processing of Web pages is the frequent HTML syntax errors they contain. That is because existing Web browsers are able to display HTML documents with syntax errors which remain undetected by their authors [6].

To address those issues, this research work explores the use of a new language defined by the World Wide Web Consortium (W3C) and called eXtensible Markup Language (XML) [7]. XML was designed to be extensible, free from ambiguity and errors, and able to describe the content of Web pages[10]. Those features can significantly help to improve information processing on the Web. We will see in the remaining chapters of this thesis report how XML features are used to address the problem stated in this introduction.

Note that although the initial target of this research was to extract and store information from university web sites, the proposed approach has been designed to be generic enough in order to be applied to other areas of interest such as research, education, trading, etc.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 surveys related work and provides background information on the concepts used in this thesis. Chapter 3 describes the XML-based Web Agent system developed within this thesis. It starts by a high level description of the system and its architecture. Thereafter, the four main components of the system are examined in detail.

Chapter 4 is dedicated to describing Web ONtology DEscription Language (WONDEL), the XML-based language defined in this work to express the knowledge required to extract specific information from the Web. In this chapter, the notion of Web Site Ontologies is introduced, and the two categories of WONDEL documents, leaves and nodes, are both described in details. The last section proposes extensions to the XML Pointer language with examples to illustrate their use in WONDEL.

Chapter 5 describes the implementation of the XML-based Web Agent (XWA[1]) System. The class diagrams of the different packages of the system as well as some interaction diagrams to illustrate its behavior are provided. An application of the system to populate a relational database with information extracted from several university Web sites is also described in Chapter 5. The WONDEL Builder, an editor and execution environment for WONDEL is then developed. An evaluation of the system and a set of experimental results are provided in the fourth section. Finally, the limitations of the proposed approach are discussed. To conclude this thesis, the contributions are reviewed along with suggestions for further research in Chapter 6.

1.3 Main Contributions

This thesis contributes to the field of information extraction from the Web. It proposes a mechanism to populate relational databases with information extracted from the Web. In summary, the contributions of this thesis (discussed in subsequent paragraphs) are as follows:

1. A new XML-based language called Web ONtology DEscription Language (WONDEL[2]), to express the knowledge needed to extract information from the Web.
2. A tool that converts HTML-based Web pages into well-formed XML documents.

[1] For the remainder of this document, XWA will refer to the XML-base Web Agent System.
[2] Also, WONDEL will refer to Web Ontology DEscription Language for the remainder of this document.

3. An algorithm to map valid XML documents into relational tables.
4. New proposed extensions to XML Pointer Language [30].

We propose a Web ONtology DEscription Language (WONDEL), a language based on XML and XML Pointer language to build Web Site Ontologies. A Web Site Ontology is a tree of reusable WONDEL documents that help applications extract and format specific data from the Web. There are two kinds of WONDEL documents: leaves and nodes. A WONDEL leaf is designed to express the knowledge needed to extract information from a given Web page. A WONDEL leaf can be reused for other Web pages, provided they have similar formats. The purpose of WONDEL nodes is to describe the logical structure of information located in several Web pages that are usually unrelated. WONDEL nodes are used to marshal information after being extracted using WONDEL leaves.

In order to facilitate processing of Web page content, a filter that converts HTML-based Web pages into well-formed XML documents has been developed. The conversion involves the following operations: (i) syntactic mapping of HTML to XML, (ii) resolving ambiguity introduced by HTML tagging rules, and (iii) handling errors that may occur due to improper usage of HTML by the authors. After being converted to XML, Web pages become more convenient to process and make possible the use of XML-related technologies, such as the XML Pointer Language.

The information extracted from the Web pages is stored in valid XML documents, which in turn are mapped into tables stored in relational databases. Consequently, an algorithm to map valid XML documents to relational tables has been developed within this research. The mapping algorithm uses the Document Type Declarations (DTD) to represent database models. This keeps the mapping process independent of the database model, and therefore achieves one of the main features of the whole system which is the domain independence.

Finally, new extensions to XML Pointer Language have been proposed. Their purpose is to enhance the ability of the Language to make more powerful queries. These new extensions are designed to maintain conformity with the current language specification provided by the W3C in the sense that any XPointer that conforms to the W3C's specification conforms to our exended version of the language. They also meet the language design goals drawn by the W3C in [30].

Chapter 2

Background and Related Work

The first section of this chapter provides a description of the existing work that was found relevant to this work. It also explains why it was decided to develop a new system rather than using existing ones to achieve the goals of this thesis. Section 2.2 gives an overview of the concepts on which this work is based, notably software agents, XML and XML Pointer language.

2.1 Related Work

2.1.1 Related Web Agents

There are currently a large variety of Web agents providing several types of Web-related services such as resource discovery, shopping, Web page indexing, and information retrieval. The XWA system belongs to the category of Web agents dedicated to information retrieval. The existing Web agents belonging the same category that were investigated at the early stages of this thesis are ShopBot [9] and Exposé [27]. ShopBot is a Web mining agent [8] designed for extracting information about products from vendors' Web servers. Given a product specified by the user, ShopBot retrieves the available prices of the product by filling out the forms provided in vendors' Web servers. It then displays the list of found products ordered by their prices. ShopBot uses machine learning techniques to learn how to extract information from Web pages. Those techniques have two main features: first, they are domain independent, and second, learning-based systems become more intelligent as they are

carrying on their tasks. However, machine learning techniques require a certain level of regularity in Web pages to process them properly.

Another Web agent we looked at is an ontology-based Web agent called Exposé [27]. This agent deals with Web pages that contain SHOE instances. SHOE stands for Simple HTML Ontology Extensions, and will be discussed with further details in the next section. With SHOE, authors can write both ontologies and predicates which can be used as a framework to express implicit knowledge about Web page contents. Exposé maintains a knowledge base which is filled with ontology instances extracted from SHOE-enabled Web pages. The agent uses these ontology instances to interpret the predicates it finds in other SHOE-enabled Web pages and loads the interpretation results into its knowledge base. A typical application of this framework is to define ontologies for certain categories of Web pages and write predicates to express the relationships between different Web pages. This information can be stored in a knowledge base and used later to discover Web pages related to given domains and rate their contents based on predefined criteria. As Exposé operates, its knowledge base will grow and cover more Web resources.

Neither ShopBot nor Exposé could achieve the goals of this thesis because:

1. ShopBot is completely based on machine learning techniques which are efficient in regular environments. Vendor's Web sites are a typical example of regular Web sites because vendors try to make products easy to find in their Web pages which are in most cases generated from fixed templates. Unfortunately, university Web sites do not have the same regularity since they are divided in smaller web sites owned by different departments. Each department maintains its Web site independently. Moreover, even Web pages of the same department may not be uniform,

especially personal homepages. Therefore, applying a pure machine learning-based system such as ShopBot to university Web sites was not possible.

2. The main limitation of Exposé is that it deals only with Web pages that are annotated with SHOE instances. Obviously, university Web pages are not. We could have used SHOE to annotate Web pages instead of WONDEL, but we didn't find SHOE appropriate for this task as explained in the next section.

3. ShopBot and Exposé do not provide a mechanism to collect data in a format easy to store in a relational database. Having that mechanism is one of the goals of this thesis.

Although ShopBot and Exposé were not found appropriate to address the problem posed in this thesis, they provided two precious ideas that were used in the XWA system: taking benefit from existing regularities in Web sites and writing Ontologies to help the system in its task.

2.1.2 Related Web Ontology Description Languages

The most common definition of Ontologies was given by Gruber in 1994: "*An ontology is an explicit specification of conceptualization*" [17], where conceptualization is "*a way to view the world*" (Laresgoiti) [24]. In the context of agency, ontologies are intended to help an agent isolate and arrange the pieces of information that are relevant to the task for which it is designed [26]. Ontologies can provide a valuable help for Web agents to find and extract information on the Web by adding more machine-understandable semantics to the Web pages. This is exactly what WONDEL is designed for: building ontologies to help the XWA extract information from the Web pages. Besides WONDEL, there are other languages to write ontologies for the Web. Resource Description Framework (RDF)[25] [26], and Simple HTML Ontology Extensions (SHOE)[20][27] [34] are the most known ones.

RDF is a W3C standard to define machine-understandable information, called metadata, about Web content [25]. Applications of RDF metadata include resource discovery, content rating, electronic commerce, and security. RDF design has been influenced by many research communities:

- The digital library community with its metadata schema, called Dublin Core, used to build digital library catalogs. RDF's logic is mainly inspired from Dublin Core.
- The contribution of the structured document community consists of using XML to represent and store RDF metadata. Note however that Ora Lassila, the head of the W3C group working on RDF, mentioned that XML has been chosen for its relevance to the Web and not for its technical merit.
- The influence of the framework design community comes from using an object-oriented approach to build RDF metadata.
- Finally, the knowledge representation community influenced RDF design by semantic networks developed by Quillian [32] and Woods [41].

RDF does not rely on a particular vocabulary to write metadata, but lets each community use vocabularies, called schemata in RDF, appropriate to its domain.

An RDF metadata instance consists of a node associated with a list of property/value pairs. The node is typically a Web page, and properties are any relevant information about the Web page such as author, creation date, etc. Here is an example:

```
<RDF:RDF>
     <RDF:Description about="http://www.site.uottawa.ca/wondel.html">
          <DC:Creator>Hicham Ouahid</DC:Creator>
          <DC:Date>Sept. 15, 1998</DC:Date>
```

```
            <DC:Keywords>XML, Ontology, Web</DC:Keywords>
        </RDF:Description>
    </RDF:RDF>
```

This example represents an RDF metadata instance whose node is the Web page referenced by the URI "http://www.site.uottawa.ca/wondel.html", associated with three properties: Creator, whose value is "Hicham Ouahid", Date, whose value is "Sept. 15, 1998", and Keywords, whose value is "XML, Ontology, Web". This example looks very similar to what HTML's `META` and `LINK` tags provide. However, RDF statements can be more complex because property values can be RDF statements themselves. In addition, RDF has another important feature, incremental extensibility, which makes it much more powerful than HTML tags. This feature results from the inheritance capability in RDF. Indeed, schemata in RDF consist of a hierarchy of classes, each class inherits from one or many classes at the upper level in the hierarchy. Although RDF provides more power to add semantic to the Web pages than HTML, it is limited to binary relationships.

SHOE (Simple HTML Ontology Extensions) is another Web ontology description language very similar in concept to RDF but offering the possibility to represent any kind of first-order logic expressions as long as they do not contain negations. As its name indicates, SHOE is a set of tags intended to extend HTML for defining and handling ontology-based knowledge on the Web. Like RDF, SHOE allows incremental extensibility by multiple inheritance and reuse of existing knowledge and ontologies. There are two types of SHOE-enabled Web pages, those that define ontologies and those that define knowledge instances that include categories, relations, inference rules, constants, and arbitrary data types. To illustrate SHOE's use, consider the following inference, which can be expressed by a first-order logic expression:

$\forall x,y$ Person(x) $\wedge$ Age(x,y) $\wedge$ greaterThan(y,18) $\Rightarrow$ Adult(x)

This inference states that if x is a person, y is the age of x, and y is greater than 18, then x is adult. This is represented in SHOE as follows:

```
<DEF-INFERENCE>
      <INF-IF>
            <CATEGORY NAME="Person" VAR FOR="X">
            <RELATION Name ="Age">
                  <ARG POS=1 VAR VALUE="X">
                  <ARG POS=2 VAR VALUE="Y">
            </RELATION>
            <COMPARISON OP="greaterThan">
                  <ARG POS=1 VAR VALUE="Y">
                  <ARG POS=2 VAR VALUE=18>
            </COMPARISON>
      </INF-IF>
      <INF-THEN>
            <CATEGORY NAME="Adult" VAR FOR="X">
      <INF-THEN>
</DEF-INSTANCE>
```

Neither RDF nor SHOE meet the requirements of the environment being examined in this thesis for the reasons below:

1. Both RDF and SHOE were primarily designed to help discover information resources. What we needed is a means to extract information from those resources. Using SHOE and RDF for data extraction was not as convenient as we were hoping.

2. One of the main features of WONDEL is that it takes benefit from the existing regularity in Web pages. RDF and SHOE do not provide that feature.

3. Another important feature of WONDEL is collecting data in a format that can be easily stored in databases. Neither RDF nor SHOE provides such a capability.

4. One last reason which is not a showstopper but worth to mention is the influence of XML on WONDEL vs. RDF and SHOE. Because a lot has been written on XML as a promising language to solve many Web-related issues, we were interested to see how XML could improve data extraction from the Web. Although XML is also the notation adopted to write RDF instances, other notations can be used with no lack of convenience. Indeed, Ora Lassila says that XML has been chosen as notation for RDF for "*its perceived prevalence in Web software rather than its technical merit*"[25]. In contrast, XML is the raison d'être of WONDEL. XML is not only a notation for WONDEL documents, but WONDEL can be applied only to extract information from XML documents. This is because WONDEL relies on XML Pointer language, also called XPointer language, to locate information on Web pages, and XPointers work only with XML documents.

2.1.3 Virtual Database Technology

Another relevant work was presented in an article written by Rajaraman and Norvig [37] [3]. The concept they introduced is called Virtual Database (VDB) technology. They suggest having a database

[3] That article was published one month before the first paper based on this thesis work was submitted to the ICC'99

as interlocutor between the Internet and users. Instead of browsing the Web looking for information, users can query a database that stores this information after being extracted from the Web. However, the authors did not elaborate on how the challenges their approach face are addressed, i.e. how data is extracted from the Web and how their database is populated and kept up-to-date with the ever-changing Web content. Those are the questions that the XWA system and WONDEL try to address.

2.2 Background

2.2.1 Software Agents

A lot has been written about defining a software agent. Most of the definitions provided by researchers seem to reflect their use of the word "agent" [13]. It is therefore hard to find a clear and formal definition on which everybody agrees. This section is based on two agent definitions surveys provided respectively by Franklin and Graesser (1996) [13] and Bradshaw (1997) [5]. Some selected definitions from these two surveys will be presented, and thereafter the XWA is compared against agent properties listed in Franklin and Graesser's survey.

Maes defines autonomous agents as "*computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed*" [28, 1995, page 108]. The minimum requirement for XWA to comply with this definition is to have an environment in which it senses and acts. This environment is composed of two parts: the Web and a database. The XWA system acts on the database by populating it with data extracted from the Web. It can also sense the changes occurring on the Web and react by updating the information stored in the database.

A white paper on intelligent agents from IBM describes agents as "*software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires*" [13][4]. This definition considers that an agent should perform certain tasks on behalf of a user and should be supplied with knowledge of user's needs. This knowledge is called User Model by Gudivada et al. in [19]. They add that agents employ also some knowledge of the domain in which they act to perform their tasks [19]. In the case of XWA, both user and domain models are stored in WONDEL documents that are interpreted to extract information from the Web.

Another important aspect of agents is continuity as mentioned in Soham's definition: an agent is "*a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes*"[35]. Continuity is also present in XWA's behavior, since it can run in a continuous manner watching eventual changes on the Web to keep the database up-to-date.

While the three definitions discussed above approach the notion of agent from different angles, they all agree on one condition that every software agent should fulfill: *autonomy*. Although an agent needs to be provided with an agenda to act on behalf of users, this agenda is not supposed to be a set of instructions that should be followed to the letter. Instead, it should include directions from which the agent can infer the instructions to be executed [5]. The question that will definitely determine whether the XWA system can be considered an agent is how much autonomy it has when performing its task. The XWA system is not just a WONDEL Interpreter, but has a certain degree of autonomy that lets it go beyond what is indicated in WONDEL documents for the following reasons:

[4] This definition was taken from [13] as is. The URL of its original source is no longer available.

1. The XWA has the ability to discover Web pages other than those explicitly provided in its input. In fact, the large majority of the Web pages are discovered by the XWA itself as proven by the experimental results listed in Section 5.4.

2. The Web is a hostile environment for Web software. It is full of unexpected situations such as mal-formed Web pages, broken URIs, non-responding Web servers, etc. The XWA handles these situations in different manners: When it retrieves a corrupted or mal-formed Web page, it attempts to fix its format by "guessing" what it was intended to be, or by modifying its structure (see Section 3.2 for more details). The experiment that we conducted has proven that the XWA was successful in fixing almost all the mal-formed Web pages it retrieved. In the unlikely case when it fails to handle such situations, and also when a URI is broken or a Web server is not responding, the XWA is able to recover from any inconsistency this may cause and carry on its process.

The list of agent properties proposed by Franklin and Graesser [13] is shown in Table 1.1, in which we specify whether the XWA complies with each of the listed properties. Note that according to Franklin and Graesser's definition, any software agent must comply with the first four properties.

Property	Comply	Comments
Reactive	Yes	The XWA has an environment in which it senses and acts.
Autonomous	Yes	The XWA has a certain degree of autonomy as explained above.
Goal-oriented	Yes	The XWA is designed to extract information from the Web and populate databases
Temporally-continuous	Yes	The XWA runs continuously to extract information and watch changes on the Web
Communicative	No	The XWA doesn't communicate with other agents or with people
Learning	Possible	See the proposed extensions to this work in Section 6.2.
Mobile	No	The XWA is a static agent.
Flexible	Yes	The actions of XWA are not scripted. WONDEL documents contain directions or guidelines from which the XWA infer what to do.
Character	No	The XWA does not have believable personality or emotional state.

Table 1.1: Franklin and Graesser's Agent Properties

2.2.2 Extensible Markup Language (XML)

XML is a standard language, defined by The World Wide Web Consortium (W3C), for formatting documents on the Web. XML is a subset of an existing ISO standard language called Standard Generalized Markup Language (SGML). Because SGML was designed for all kinds of electronic documents, it is complex and hard to learn. Consequently, the Web community defined another language called HyperText Markup Language (HTML) [33], which is an application of SGML to format documents on the Web. Although HTML has had a tremendous success during the last ten years, it has shown some limitations that XML is designed to solve [10]:

- HTML is not extensible, that is, there is no way for developers to define new tags in HTML. Only the W3C has the authority to add new markup to HTML. However, with XML authors can define their own tags in a standard way by declaring them in a DTD (Document Type Definition) file.
- HTML is designed to specify the layout of Web pages but has limited capability to describe the content of these pages. XML is very efficient to describe both the format and the content of Web documents.
- Tagging rules in XML are more strict and precise than in HTML. This makes XML parsers thinner and faster.

The XML specification defines two types of XML documents: valid documents and well-formed documents. To understand the conditions that a document must fulfill to be well-formed, refer to [40] and [16]. For readers who have background knowledge on HTML, Section 3.2 of this thesis report discusses the differences between HTML and well-formed XML documents.

By definition, if a document is not well formed, it cannot be considered as an XML document [40]. Furthermore, the XML specification clearly says that any conformant XML software must fail and not try to handle eventual syntax errors while parsing XML documents [6]. This guarantees that no mistaken XML document will exist on the Web.

In well-formed documents, authors are free to include any element with any attribute and there is no constraint on how they should be structured. However, XML provides a mechanism, the Document Type Definition (DTD), to define constraints on the logical structure of an XML document. A valid document is a well-formed document associated with a DTD and obeys the constraints defined in it [16]. A DTD specifies all the elements that the associated documents may or must contain, their hierarchy, their types, and their attributes.

While the concept of a DTD exists even in HTML, the difference is that HTML has a fixed DTD whereas XML allows users to define their own DTDs. This is what makes XML extensible and enables building other languages on top of it. Currently, there are several languages based on XML such as Synchronized Multimedia Integration Language (SMIL) [39], Chemical Markup Language (CML) [31], and Mathematical Markup Language (MathML) [22]. To better understand the concept of the DTD, we provide in Figure 1.1 an example of a DTD and a conformant valid XML document.

```
<!------------ faq.xml ------------>
<?XML VERSION="1.0" ENCODING="UTF-8"
 RMD="EXTERNAL"?>
<!DOCTYPE FAQ SYSTEM "FAQ.DTD">
<FAQ>
  <INFO>
    <SUBJECT> WONDEL              </SUBJECT>
    <AUTHOR> HICHAM OUAHID         </AUTHOR>
    <EMAIL> ouahid@site.uottawa.ca </EMAIL>
    <VERSION> 1.0                 </VERSION>
    <DATE> April,20 1999             </DATE>
  </INFO>
  <PART TITLE="About WONDEL">
     <Q NO="1">
       <QTEXT> What is WONDEL? </QTEXT>
       <ATEXT>
          Web ONtology DEscription Language
       </ATEXT>
     </Q>
     <Q NO="2">
       <QTEXT>What is useful for?</QTEXT>
        <ATEXT>
          Describe Web site ontologies
        </ATEXT>
     </Q>
  </PART>
</FAQ>
```

```
<!-------  faq.dtd ------->
<!ELEMENT FAQ (INFO, PART+)>
<!ELEMENT INFO (SUBJECT, AUTHOR,
     EMAIL?, VERSION?, DATE?>
<!ELEMENT SUBJECT (#PCDATA)>
<!ELEMENT AUTHOR  (#PCDATA)>
<!ELEMENT EMAIL   (#PCDATA)>
<!ELEMENT VERSION (#PCDATA)>
<!ELEMENT DATE    (#PCDATA)>
<!ELEMENT PART    (Q+)>
<!ELEMENT Q       (QTEXT, ATEXT)>
<!ELEMENT QTEXT   (#PCDATA)>
<!ELEMENT ATEXT   (#PCDATA)>
<!ATTLIST  PART
   NO    CDATA  #IMPLIED
   TITLE CDATA  #REQUIRED>
<!ATTLIST  Q
   NO  CDATA  #IMPLIED>
```

Figure 1.1: Example of a Valid XML Document

The document presented on the left of Figure 1.1 is a valid XML document. It consists of two parts:

- The prolog which is the header of the document. This part contains two element declarations `?XML` and `!DOCTYPE`. The keyword `?XML` specifies the version of XML, the encoding type, and the type of the Required Markup Declaration (RMD). In the example above the version of XML is 1.0, the encoding type is UTF-8 and the RMD is external, which means that the document must obey a DTD that is located in another file. The other possible values of RMD are "INTERNAL" which means that the DTD is located in the same file and "ALL" if the XML document must obey the external and the internal DTD in the case where they are both provided. The keyword `!DOCTYPE` specifies the root of the document and the DTD. In the example of Figure 1.1 the root element is FAQ and the DTD is provided in a external file whose name is 'FAQ.DTD'.

- The body which contains the document itself.

The DTD given on the right of Figure 1.1 consists of a set of element declarations, identified by the keyword `!ELEMENT`, and attribute declarations which are identified by the keyword `!ATTLIST`. For example, the first element declaration declares the root element FAQ and specifies that it contains exactly one element INFO and one or many elements PART. Also, the first attribute declaration specifies that the element PART has two attributes, NO which is implied (optional) and TITLE which is required. The reader may go through the XML document on the left and check that it effectively obeys the constraints of the DTD. For the complete XML specification, see [7].

2.2.3 XML Pointer Language (XPointer)

XML Pointer Language is a language used for addressing the internal structures of XML documents. With this language, you can make reference to elements, character strings, or attributes inside XML documents [30]. XML Pointer language is based on Pointers called XPointers. An

XPointer consists of a series of location terms. Each location term specifies a location inside an XML document, relative to the location specified by the previous location term if it exists. A location term is composed of a keyword and a list of arguments. The XPointers have the format shown below:

```
XPointer: <LocationTerm_1>.<LocationTerm_2>.….<LocationTerm_N>
LocationTerm : Keyword(ListOfArguments)
```

The specification of XML Pointer language [30] specifies five types of location terms:

- **Absolute location terms**: these terms are not relative to any other location term. An XPointer may have at most one absolute location term, and if it has one, it must be the first location term. The keywords associated with these types of location terms are `ROOT`, `ORIGIN`, `HTML`, and `ID`.

- **Relative location terms**: these terms are computed relative to the element selected by the previous location term. Normally, they do not occur at the beginning of the XPointer, but if this happens, they are computed relative to the root element of the XML document. An XPointer may contain any combination of relative location terms. Relative location terms are identified by the following keywords: `CHILD`, `DESCENDANT`, `ANCESTOR`, `FOLLOWING`, `PRECEDING`, `PSIBLING`, and `FSIBLING`.

- **Spanning location terms**: they are identified by the keyword `SPAN` and have the following structure:

```
SPAN(XPointer1 , XPointer2)
```

A spanning location term selects a set of elements between two elements specified by two XPointers given as arguments. Spanning location terms are relative, which means that the two elements are specified according to the element selected by the previous location term.

- **Attribute location terms**: these location terms are used to extract attribute values. They are identified by the keyword ATTR that takes an attribute name as argument. This attribute is searched among the attributes of the element selected by the previous location term.

- **String location terms**: these are used to select a character string from the text contained in the element selected by the previous location term. They have two required arguments which are the string candidate to be found, and its occurrence number. String location terms are identified by the keyword STRING.

Figure 1.2 is an example of an XPointer that includes three types of location terms: absolute, relative, and spanning location terms.

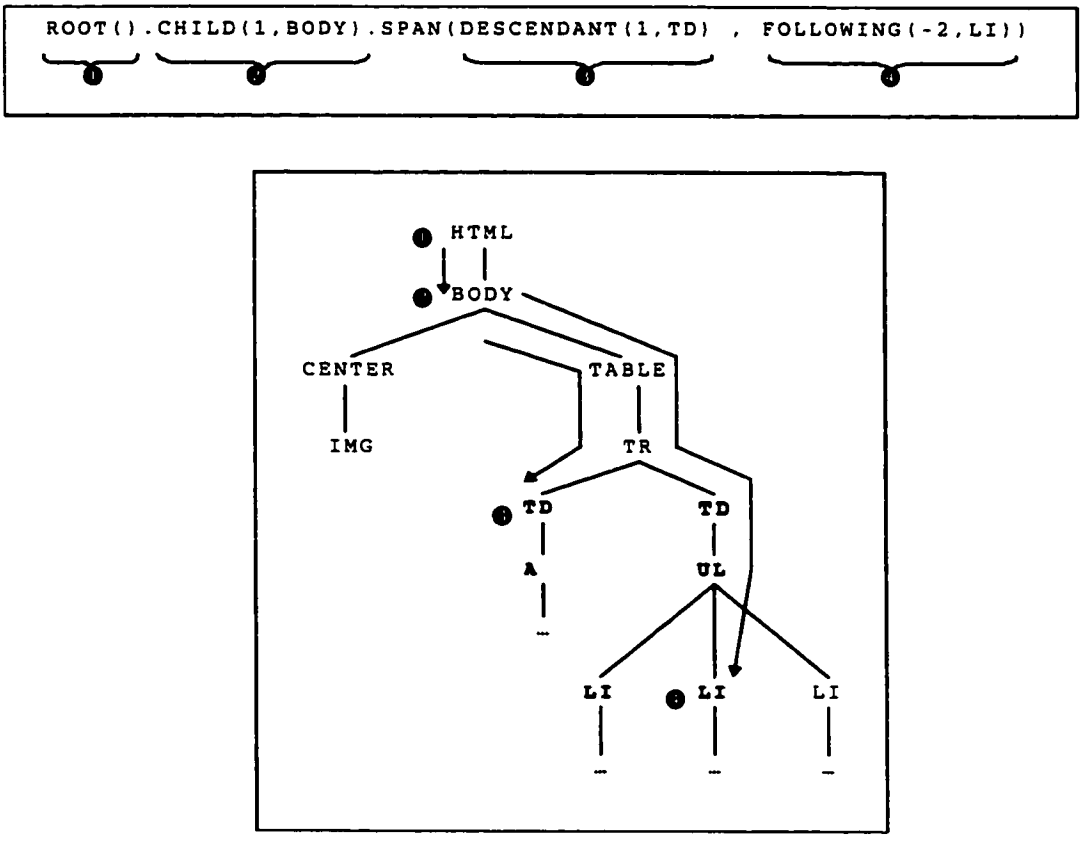


Figure 1.2: Example of an XPointer

In the tree of XML elements presented in Figure 1.2 we illustrate how XPointer Interpreter goes through the XML document hierarchy while interpreting the XPointer given above. In this figure, each element is labeled by the same number as the corresponding location term. For instance, the location term `ROOT()` specifies the element `HTML` and they are both labeled by ❶. This XPointer selects the elements specified by the last location term, which is a spanning location term. The elements selected are `TD`, `TD`, `A`, `UL`, `LI`, and `LI`, and they are written in bold text in Figure 1.2. Note that negative numbers allows starting by the last element and going in the reverse order. For example, the location term number ❹, identified by the keyword `FOLLOWING`, selects the penultimate element whose name is LI that is located after the element `BODY` in the hierarchy.

2.5 Summary

The large size of the Web, its disorganized nature, its fast growth, and the lack of semantics in the Web pages, are the major issues that complicate information searching and extraction on the Web. The efforts invested to facilitate information searching on the Web have taken three directions. The work deployed in the first direction consists of using indexing techniques to build directories which users can search and query to find the Web pages that most likely contain the information they desire. In the second direction, Information Retrieval researchers attempt to develop Web agents to filter the content of the Web pages and provide the users with the desired information instead of links. We have discussed two examples of Web agents. The first is based on machine learning techniques and has the advantage of being domain independent. However, the efficiency of learning-based systems relies on how much regularity exists in Web pages. Another approach presented in the second example is to give authors the ability to add semantic to HTML-based Web pages using a simple superset of HTML that adds knowledge markup syntax. The limitation of this approach is that it cannot be applied to existing

Web pages and it puts additional constraints on Web authors. The third direction taken by the W3C is to define new languages, such as XML, to facilitate automated data processing on the Web. One proposed approach is to write Web documents in XML and generate HTML code on the fly using XML Stylesheet Language (XSL) when display is necessary. This approach avoids loss of the existing HTML-based tools especially Web browsers. However, this approach can be applied only to the new added Web pages and therefore will not solve the problem for the large number of existing Web pages.

This thesis presents an approach of using XML to add semantic information to the existing Web pages, without having to change their content. The approach proposes to describe the meaning of the Web pages in separate documents that computer programs can refer to whenever they need to manipulate or extract data from these Web pages. These documents are written in Web ONtology DEscription Language (WONDEL), an XML and XPointer-based language to express the basic knowledge required to automatically understand and extract specific information from the Web. Like ShopBot, WONDEL takes advantage of the existing regularity in the Web pages, which significantly reduces the amount of WONDEL code needed to manipulate them. WONDEL has been successfully used by an XML-based Web Agent developed within this thesis to extract information about several universities around the world.

Chapter 3

An XML-based Web Agent

The objective of the XWA system is threefold: (i) extracting information from the Web, (ii) structuring of this information so that it can be stored and manipulated in a relational database, and (iii) keeping the database consistent with the changes.

The XWA system is composed of four main components: the HTML2XML Converter, the WONDEL Interpreter, the Database Adapter, and the Web Change Detector. Note that only the first three components have been implemented in this thesis work.

This chapter is dedicated to describing the XWA system and its main components. The next section provides an overview of the system and draws its architecture. The remaining four sections describe the components of the XWA system.

3.1 Overview and architecture

The XML-based Web Agent (XWA) is a system that uses XML and its related technologies to extract specific information from the Web and populate relational databases. This system is designed to be domain independent with limited human intervention. It starts by converting Web pages into well-

formed XML documents and then extracts data from them. The extracted data is formatted into valid XML documents which are then mapped to relational tables before being stored into relational databases.

The XWA system is composed of four main components as shown in the architecture depicted in Figure 3.1.

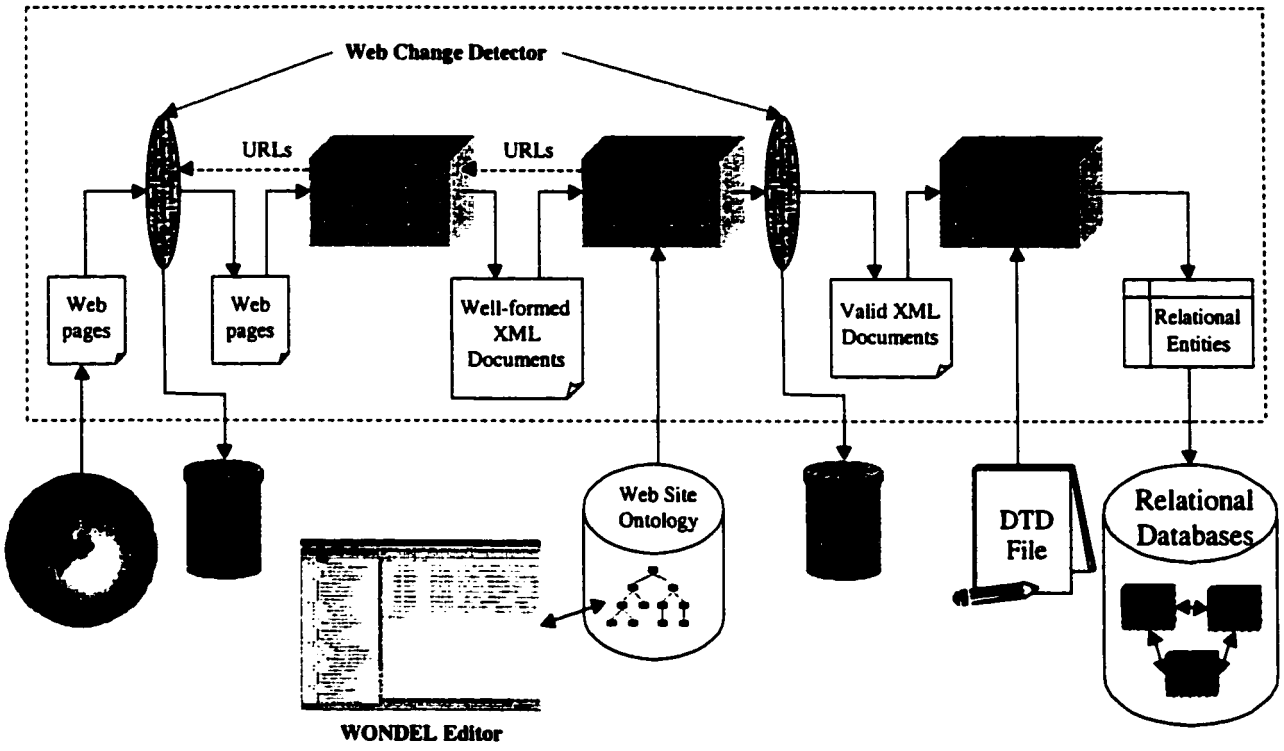


Figure 2.1: The XML-based Web Agent Architecture

To process HTML-based Web pages, the system should first handle the frequent HTML syntax errors and remove the ambiguity introduced by HTML tagging rules. This task is performed by the *HTML2XML Converter*. This component is responsible for converting all the Web pages that will be processed by the system into well-formed XML documents [2]. It guarantees to the other components that these converted Web pages are well formed, free from any ambiguity or error, and respect XML syntax. The conversion process is explained in detail in section 3.2.

The WONDEL Interpreter is the heart of the XWA system. It is the component that extracts and marshals data from Web pages. It generates valid XML documents that obey the declarations provided by the DTD (Document Type Declarations) file. Achieving this task requires information about the content of Web sites. This information is stored in a Web Site Ontology, a knowledge base that consists of a tree of WONDEL documents.

The documents generated by the WONDEL Interpreter have a tree structure, and because relational databases do not handle well such structures, a Database Adapter has been included into the XWA system to map valid XML documents into relational tables. The mapping process is discussed in detail in Section 3.4.

The last component of the WMA system is the Web Change Detector. It is responsible for keeping the databases consistent with the changes that may occur on the Web. This component is discussed in Section 3.5. Note that this component has not been implemented within this thesis.

3.2 HTML2XML Converter

This section discusses the HTML2XML Converter, the component that is used to convert Web pages into well-formed XML documents. This conversion involves the following operations, (i) syntactic mapping of HTML to XML, (ii) resolving ambiguity introduced by HTML tagging rules, and (iii) handling errors that may occur due to improper usage of HTML by Web authors. The need for converting Web documents to XML is due to the inadequacies of HTML that complicate Web page processing in contrast with the features of XML discussed in Section 2.3.

To better understand the conversion process, a brief description of HTML-based Web documents is provided before describing the conversion process.

3.2.1 Structure and Format of HTML-based Web Documents

HTML-based Web documents are structured as trees of elements that are generally characterized by three parts: a start tag, content, and an end tag. HTML specification defines two types of elements: non-empty elements that have a content delimited between two tags, and empty elements that do not have a content and have only one tag [1, §3.2.1]. Figure 2.2 shows an example of an HTML document and the associated tree.

```
<HTML>
  <HEAD>
    <TITLE> Example </TITLE>
  </HEAD>
  <BODY>
    <P>
      This is an example
      <IMG SRC="image.gif">
    </P>
  <BODY>
</HTML>
```

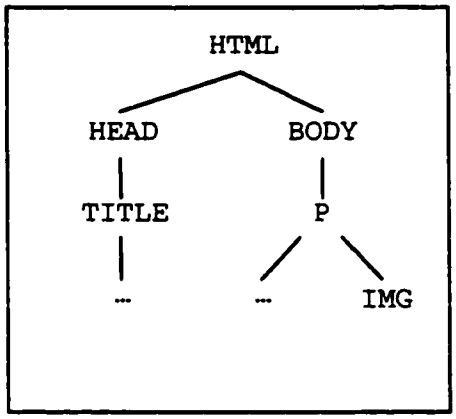


Figure 2.2: Example of an HTML Document and the Associated Tree

HTML is very popular because it's easy to use and its tagging rules are flexible. However, this flexibility makes HTML documents hard to process, as explained in the following examples:

- HTML does not provide rules to detect empty elements. The parser has to explicitly know each empty element. In addition, end tags are optional for some non-empty elements such as P and BODY [1, §3.2.1]. This adds a lot of special-case code to the parser.

- The way of declaring attributes in begin tags is ambiguous. Normally, attribute values are delimited using either double quotes or single quotes. However, in certain cases, authors may specify the value of an attribute without quotes [1, §3.2.2]. This also complicates the parsing. Moreover, a

single attribute may be declared several times in the same begin tag. Thus, HTML parsers have to decide which value should be assigned to the attribute declared more than once in the same begin tag.

- Current browsers allow certain freedom in structuring HTML documents. This implies that even the documents that do not strictly conform to the HTML specifications are correctly displayed and thus, errors in those documents remain undetected by authors [7].

3.2.2 Converting Web Pages into Well-formed XML Documents

This sub-section is dedicated to describing all the transformations made during the conversion process.

1. **Changing tag names to uppercase**: Unlike HTML, the XML parser is case sensitive when processing elements' names. This requires that characters in begin and end tags of the same element must have the same case. Therefore, all characters in begin and end tags are converted to uppercase. Figure 2.3 shows an example of this conversion.

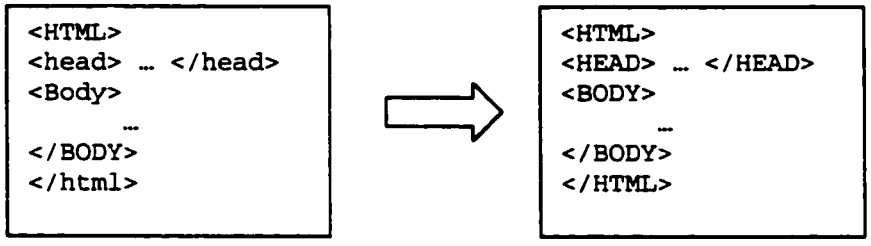


Figure 2.3: Change all Tag Names to Uppercase

2. **Changing empty-element tag syntax**: HTML 4.0 declares the following elements as empty elements: IMG, BR, META, HR, AREA, BASE, COL, FRAME, ISINDEX, LINK, and PARAM [1, pp

329-332]. For these elements we need to insert the character '/' before closing the tag. An example of this transformation is illustrated in Figure 2.4.

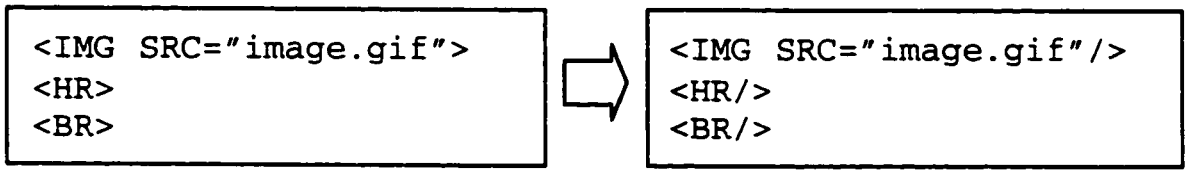


Figure 2.4: Empty Elements

3. **Removing ambiguity from attribute declarations**: In XML, attribute values must be enclosed by two similar quotation marks (either ' or "). However, in HTML documents, attribute values may appear without any quotation mark. Therefore, we need to check each attribute declaration and add quotes if they do not appear. Moreover, HTML documents may contain attributes without any value assigned to them which constitutes an error in XML and, thus, need to be removed. Furthermore, in Web documents, an attribute may be declared more than once in the same start tag. In this case, repeated occurrences of the attribute are removed. Figure 2.5 illustrates these transformations.

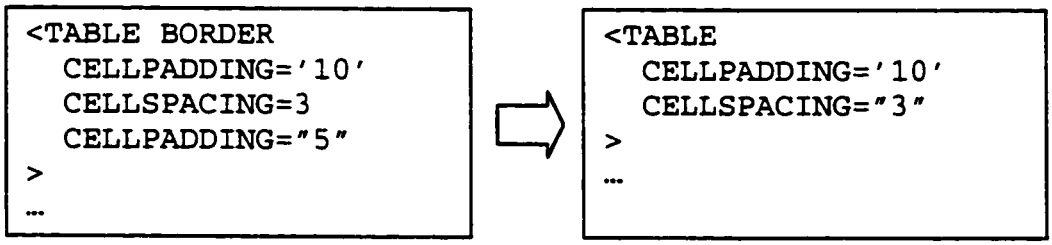


Figure 2.5: Changes Made on Attribute Declarations

4. **Fixing element-nesting errors**: In HTML, there are some non-empty elements for which end tags are optional. In addition, authors of HTML documents may accidentally omit end tags even when they are required. In XML, all non-empty elements must have an end tag. Thus, while converting

Web documents into HTML documents we need to insert missing end tags in the correct location. This operation is illustrated in Figure 2.6.

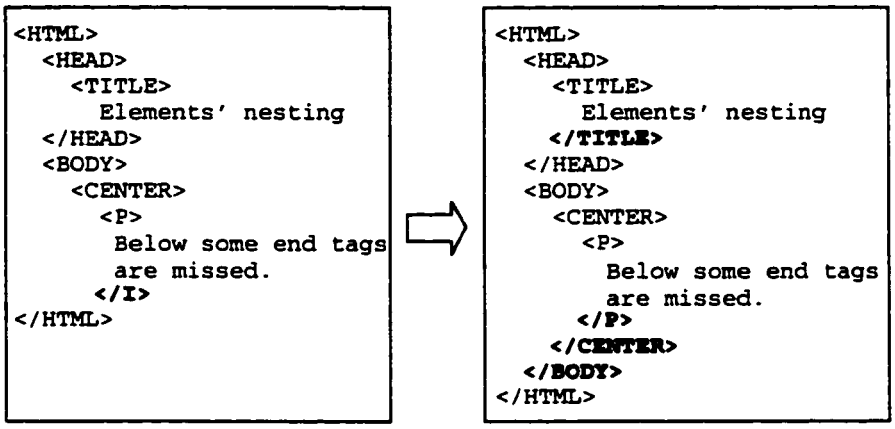


Figure 2.6: Add End-Tags to Fix Elements' Nesting

In Figure 2.6 end tags for the elements TITLE, P, CENTER and BODY have been added, and the end tag </I> has been removed.

5. **Resolving relative URIs**: One of the most important features of HTML is the ability to make reference to other resources from a given document. This feature is commonly known as Hypertext. Resources can be HTML documents, images, videos, or any other text or binary file. A reference to any kind of resource is called URI (Universal Resource Identifier). There are two types of URIs: absolute URIs that uniquely defines a resource location, and relative URIs that must be resolved according to a base URI in order to find the exact location of the corresponding resource. During the conversion process, the HTML2XML Converter resolves all relative URIs and converts them to absolute ones. The complete algorithm of resolving a relative URI is provided below. See [1] for more details.

   1. If the element BASE exists, then the Base URI is the value of the attribute named HREF. Otherwise the base URI is the URI of the given document.

2. For each URI in the document:

   a. Check if it is an absolute URI. If it is, no change is made.

   b. If not, check if the relative URI starts with the character '/'.

      i. If it starts with '/', this means that it's the absolute path name of the resource. In this case the new resolved URI is the concatenation of: the protocol name given in the base URI + '://' + the host name given in the base URI + the relative URI.

      ii. If it does not start with '/', this means that it is a relative file path. In this case the new resolved URI is the concatenation of: the protocol name given in the base URI + '://' + the host name given in the Base URI + the absolute path given in the base URI + the relative URI.

   An example of this conversion is presented in Figure 2.7.

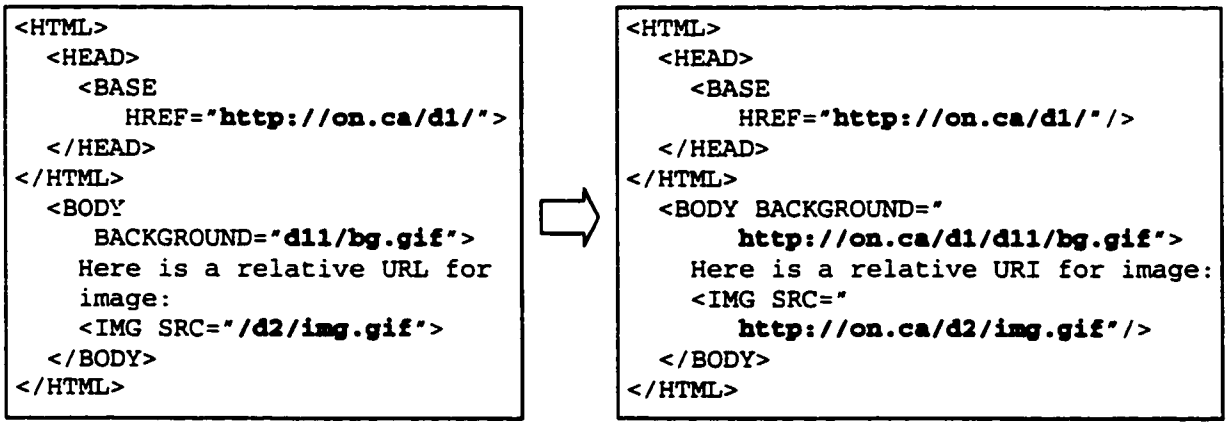


Figure 2.7: Resolving Relative URIs

Note that resolving relative URIs is not required to conform to XML rules but it is necessary to avoid loss of information.

6. **Adding tags to avoid mixed content**: In XML as well as in HTML, non-empty elements may contain text, other elements, or a mixture of text and elements. However, mixed content in XML or

HTML elements hinders the detection of the desired information within the Web pages. For example, consider the document shown at the left of Figure 2.8. Suppose that we are interested in "text 1" in this document. Using XML Pointer language, we can only detect the smallest element that contains that text. However, this element is BODY, which actually contains all the text of the document. By enclosing all the pieces of text existing in this document by the tags TEXTFLOW, as shown on the right of Figure 2.8, we are able to do a more refined search. Indeed, after this conversion, the smallest element that contains "text 1" contains just the desired text.

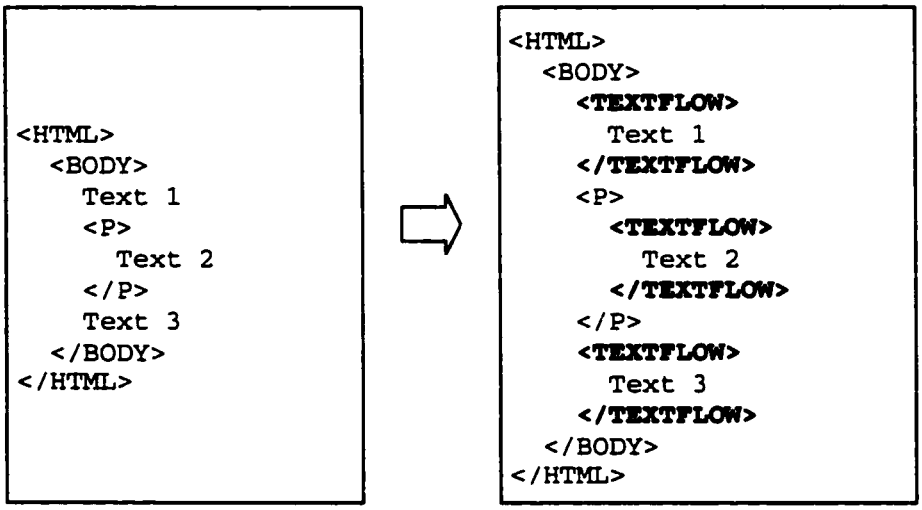


Figure 2.8: Adding TEXTFLOW Tags to Avoid Mixed Content

Note also that this transformation is not required to conform to XML rules.

3.3 WONDEL Interpreter

The WONDEL Interpreter is the main component of the XWA system. It is the component that extracts and marshals data from Web pages. It formats the extracted data into valid XML documents that obey the constraints specified by a given DTD. WONDEL is an XML and XPointer-based

language. Consequently, WONDEL Interpreter uses XML Parser and XPointer Interpreter to interpret the WONDEL documents stored in the Web Site Ontology, as illustrated in Figure 2.9.

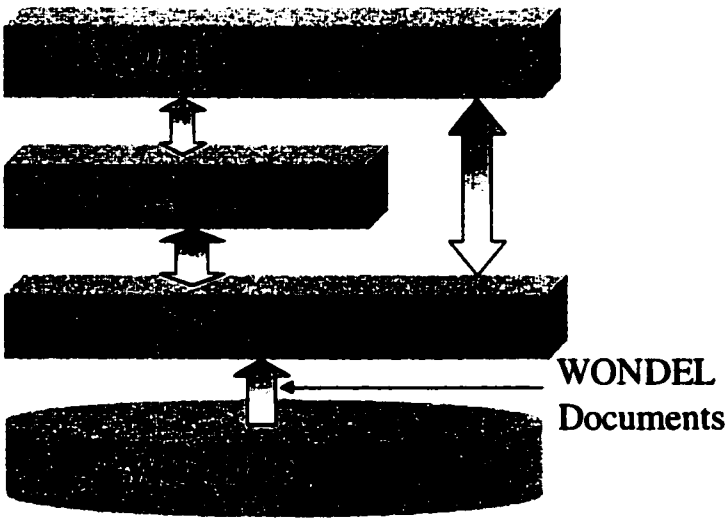


Figure 2.9: The WONDEL Interpreter

A Web Site Ontology is a tree of WONDEL documents. Therefore, we distinguish two kinds of WONDEL documents: leaf documents and node documents. Figure 2.10 is a snapshot of WONDEL Builder that provides a presentation of a Web Site Ontology. In this figure, leaf documents are labeled by sheet icons whereas node documents are labeled by folder icons.

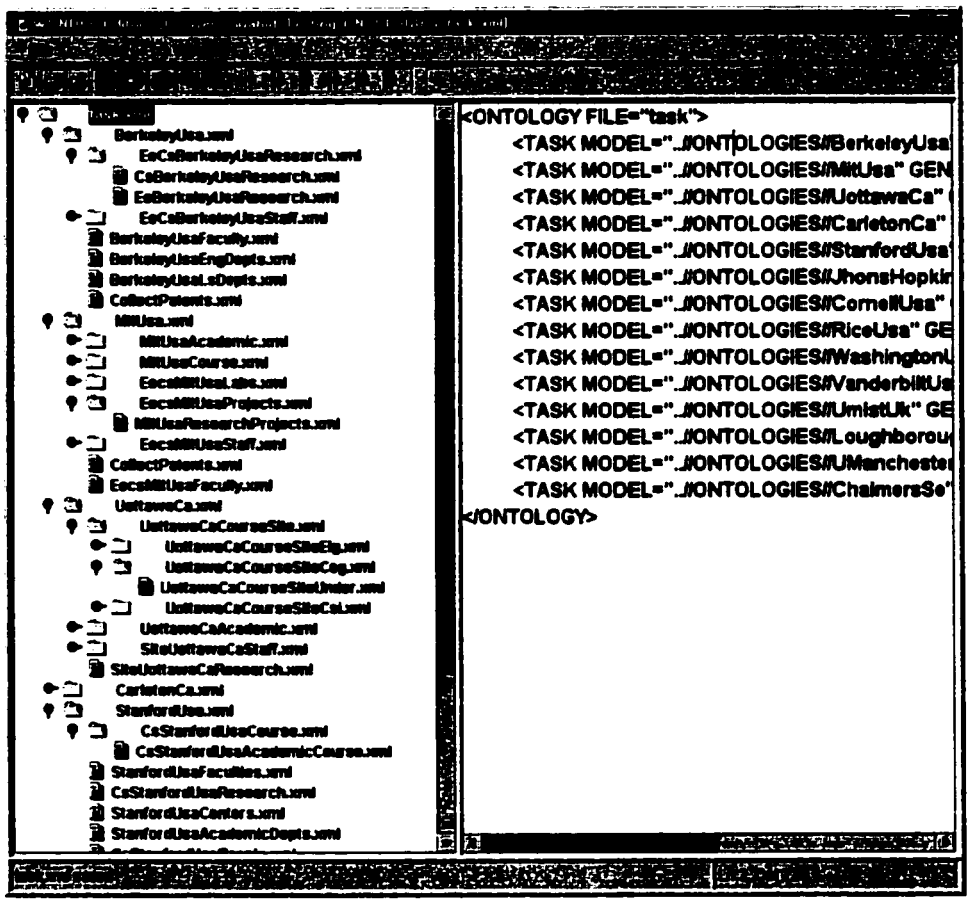


Figure 2.10: A Web Site Ontology

A WONDEL leaf document is the user's view of one or many Web pages. It specifies the location of the desired data within a given page, and describes its semantics. A WONDEL leaf document is the only input required by WONDEL Interpreter to extract information from a given Web page. Nevertheless, the desired data does not always exist in one page, instead it is usually scattered throughout many Web pages. Therefore, we need a mechanism that marshals the extracted data without online human intervention. This is the main role of the WONDEL node documents. Another important role of the WONDEL nodes is to call WONDEL leaf documents and specify the locations of the Web pages to which they will be applied. This way, WONDEL leaves can be reused for other Web pages. On the other hand, a WONDEL node can include calls to other nodes as well. Therefore, we can

organize a set of WONDEL documents in a hierarchical structure that should make them easier to manage and maintain.

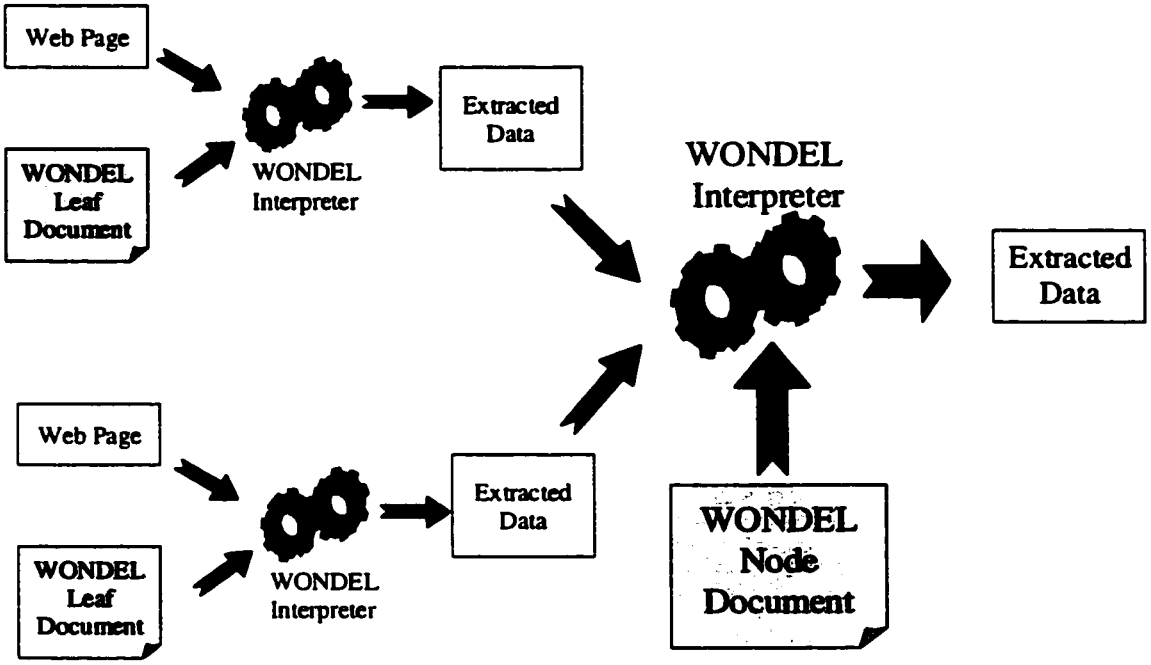


Figure 2.11: The Use of WONDEL Node Documents

The WONDEL Interpreter interprets each document in a Web Site Ontology, starting from its root and traversing the Ontology in depth-first search. Each time a leaf is reached, the Web pages to which it will be applied are downloaded and converted into well-formed XML documents by the HTML2XML Converter before extracting the desired data from them. By interpreting the nodes, the WONDEL Interpreter marshals the extracted data and recursively builds a valid XML document that contains all the data extracted from a given Web site. This mechanism is partially illustrated in Figure 2.11.

Chapter 4 provides the whole specification of WONDEL as well as examples in which full scenarios of interpreting WONDEL leaves and nodes are presented.

3.4 Database Adapter

The Database Adapter has been introduced into the XML-based Web Agent to map the data extracted from the Web into relational tables. The Database Adapter is domain independent, in the sense that it can deal with any relational database model. This is due to the use of the DTD that stores the information related to the database model.

Figure 2.12 shows the input and the output of the Database Adapter. Given a DTD that represents a database model, and a valid XML document that conforms to the DTD and contains the information extracted for Web pages, the Database Adapter produces relational tables that can be stored into the database.

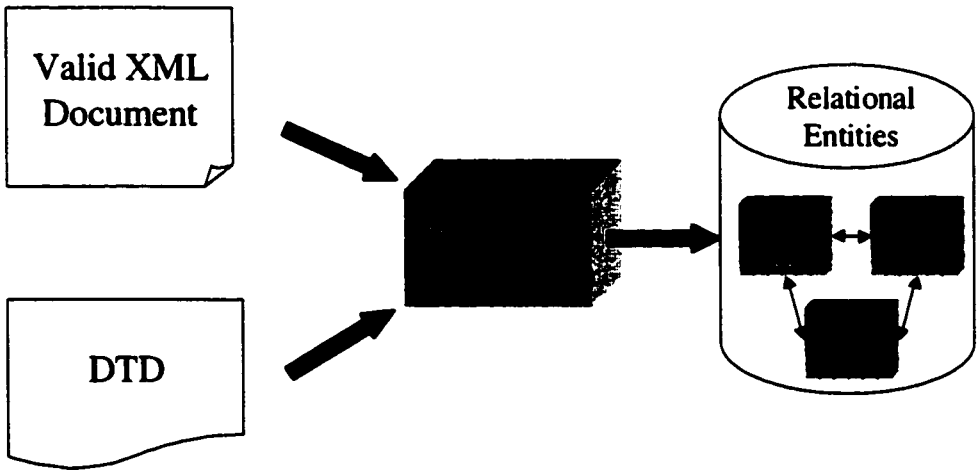


Figure 2.12: Using the DTD by the Database Adapter

The following sub-section explains how to represent a database model using a DTD.

3.4.1 Representing an Entity-Relationship Diagram with a DTD

Consider a simple database designed to store information about universities, departments and courses. The database model is represented by an Entity/Relationship diagram given at the left of Figure 2.13. Based on this example, we will explain the procedure to write the DTD that corresponds to an E/R diagram. Note that the following procedure involves a few technical words related to database technologies. See [11] for the explanation of those words. In the remainder of this section, we adopt the following conventions to alleviate the text explaining the procedure:

- The word 'element' refers to an XML element.
- The word 'entity' refers to an entity type.
- The word 'relationship' refers to a relationship type.

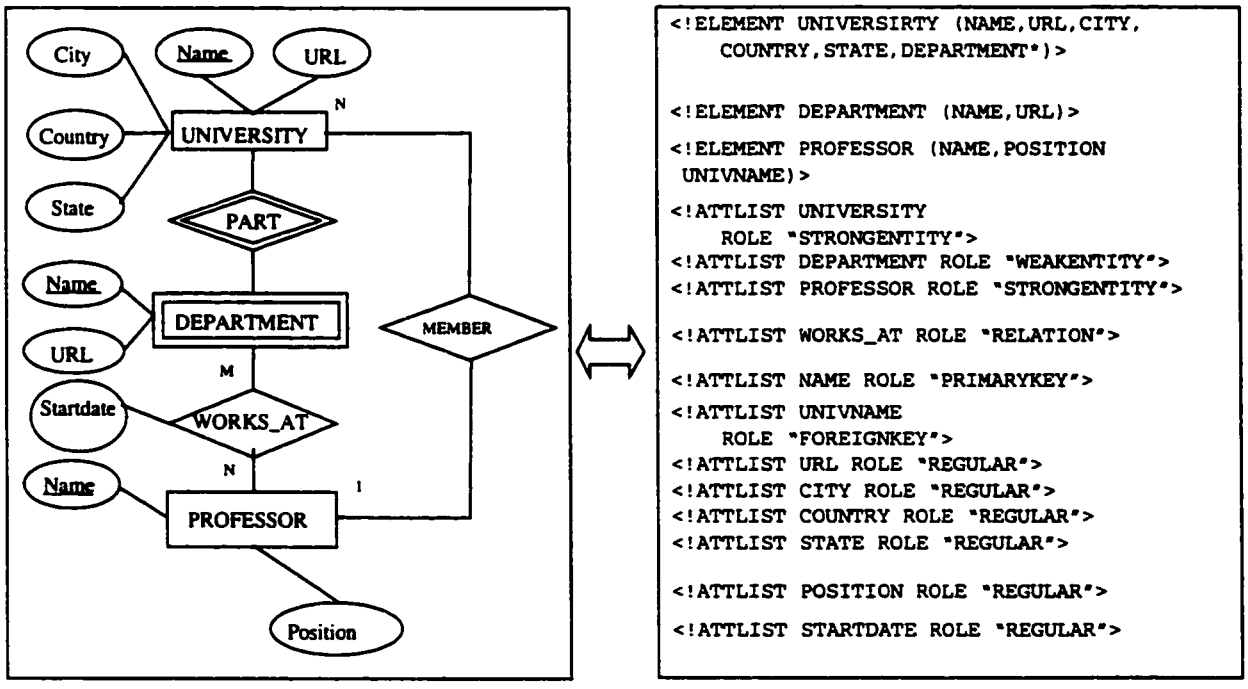


Figure 2.13: Example of an E/R Diagram and the Corresponding DTD

Following is the procedure to build the DTD corresponding to a given E/R diagram:

1 Create a root element whose name can be chosen by the user.

Example: We choose the name UDB that stands for University DataBase in the example of Figure 2.13.

2 For each strong entity, create an element with the same name and add it as a child to the root element.

Example: We have in Figure 2.13 two strong entities, UNIVERSITY and PROFESSOR, whose corresponding elements are children of the root UDB.

3 For each weak entity E1:

3.1 Detect the entity E2 on which it depends.

3.2 Create an element with the same name as E1.

3.3 Add it as a child to the element corresponding to E2.

Example: In our example we have one weak entity, DEPARTMENT, whose corresponding element is a child of the element UNIVERSITY.

4 For each N:M relationship R:

4.1 Detect the two entities, E1 and E2, associated by R.

4.2 Create an element with the same name as R.

4.3 Add it as child to either E1 or E2. The best entity is the one that has the less instances involved in the relationship.

Example: In the example of Figure 2.13, we have one N:M relationship, WORKS_AT, for which we have created an element. This element is added to the element DEPARTMENT because the number of professors working at a department is usually bigger than the number of departments in which a professor may work.

5 For each attribute A, except those belonging to a 1:N relationship:

5.1 Detect the entity or the relationship T to which it belongs.

5.2 Create an element with the same name as A and add it as child to the element corresponding to T.

Example: For example, in Figure 2.13, the element URL is added as child to the entity UNIVERSITY.

6 For each entity E at the N-side of a 1:N relationship T:

6.1 Create an element corresponding to each one of the attributes forming the foreign keys of E and the attributes belonging to T.

6.2 Add the new created elements as children to the element corresponding to E.

Note that the attributes forming the foreign key of an entity at the N-side of a 1:N relationship are those that form the primary key of the entity at the 1-side of the same relationship.

Example: In the example of Figure 2.13, MEMBER is a 1:N relationship between the entities PROFESSOR, at the N-side, and UNIVERSITY at the 1-side. Therefore, we have added to the element DEPARTMENT a child element, UNIVNAME, that corresponds to the attribute Name which is the primary key of the entity UNIVERSITY. In addition, we have added another child element, POSITION, which belongs to the relationship MEMBER.

7 For all the elements, except the root element, assign an attribute ROLE whose value is as follows:

- If the element represents an entity, then the value is STRONGENTITY or WEAKENTITY depending on the type of the entity.
- If it represents an N:M relation, then the value is RELATION.
- If it represents an attribute the value is:
  - PRIMARYKEY if the attribute is a part of a primary key.
  - FOREIGNKEY if the attribute is a part of a foreign key.

➢ REGULAR otherwise.

Example: According to Figure 2.13:

- The elements UNIVERSITY and PROFESSOR have the ROLE "STRONGENTITY",
- DEPARTMENT has the ROLE "WEAKENTITY",
- WORKS_AT has the ROLE "RELATION",
- NAME has the ROLE "PRIMARYKEY",
- UNIVNAME has the ROLE "FOREIGNKEY",
- and the other elements, such as URL, have the ROLE "REGULAR".

Any XML document that obeys such a DTD can be mapped into relational tables by the algorithm presented in the next section.

3.4.2 Mapping Valid XML Document into Relational Tables

The mapping algorithm is implemented by a recursive function, *ProcessElement*. This function has two parameters: *Element* and *Level*. Initially, *Element* refers to the root element of the valid XML document to be mapped. *Level* is an integer that is initially set to 0.

Mapping algorithm

Data structures:

- Entity: represents the relational tables that form the output of the Database Adapter.
- Attribute: represents the attributes of the relational tables.
- Stack: a stack where the relational tables are stored temporarily.
- List: list of relational tables. This is the output of the Database Adapter.

Parameters:

- Element: an XML element in the tree of the XML elements.

◆ Level: the level of Element in the tree. The root element has the level 0.

Initial state: Element = Root element, Level = 0.

Function *ProcessElement*(Element , Level)

If (Level=0) **then**

Foreach Element2 child of Element **Do**

ProcessElement(Element2 , Level+1);

If (Level=1) **Then**

Generate a new Entity;

Entity.Name ← Element.Name;

Stack.push(Entity);

Foreach Element2 child of Element **Do**

ProcessElement(Element2 , Level+1);

Add Stack.top() To List;

Stack.pop();

If (Level>1) **then**

If (Element.Role = PRIMARYKEY or FOREIGNKEY or REGULAR) **then**

Generate a new Attribute;

Attribute.Name ← Element.Name;

Attribute.Value ← Element.Text;

Attribute.Type ← Element.Role;

Add Attribute To Stack.top();

If (Element.Role = RELATION or STRONGENTITY or WEAKENTITY) **Then**

Generate a new Entity;

```
Entity.Name ← Element.Name;
Foreach Attribute In Stack.top() Do
    If (Attribute.Type = PRIMARYKEY) Then
        If (Element.Role = STRONGENTITY) Then
            Attribute.Type ← FOREIGNKEY;
        If (Element.Role = WEAKENTITY or RELATION) Then
            Attribute.Type ← PRIMARYKEY;
        Add Attribute To Entity;
Stack.push(Entity);
Foreach Element2 Child of Element Do
    ProcessElement(Element2 , Level+1);
Add Stack.top() To List;
Stack.pop();
```

End of Function *ProcessElement.*

An illustrative example

To better understand how a valid XML document is mapped to relational tables with the algorithm described above, consider the valid XML document presented in Figure 2.14 that obeys the constraints provided by the DTD of Figure 2.13.

```
<UDB>
    <UNIVERSITY>
        <NAME> University of Ottawa </NAME>
        <URL> http://www.uottawa.ca </URL>
        <COUNTRY> Canada </COUNTRY>
        <DEPARTMENT>
            <NAME> Electrical engineering </NAME>
            <URL> http://www.site.uottawa.ca </URL>
            <WORKS_AT>
                <PROFNAME>
                    John Smith
                </PROFNAME>
                <STARTDATE> 1991 </STARTDATE>
            </WORKS_AT>
        </DEPARTMENT>
    </UNIVERSITY>
    <PROFESSOR>
        <NAME> John Smith </NAME>
        <POSITION> Professor </POSITION>
        <UNIVNAME> University of Ottawa </UNIVNAME>
    </PROFESSOR>
</UDB>
```

Figure 2.14: A Valid XML Document

By applying the algorithm above, the Database Adapter generates the XML document given in Figure 2.15. This document contains a set of XML elements that represent the generated relational tables. As seen in Figure 2.15, four relational tables have been extracted from the above document that are of the following types: UNIVERSITY, DEPARTMENT, PROFESSOR, and WORKS_AT. Each attribute is given with its type, such as PRIMARYKEY. Notice how the XML elements have been disassembled in order to fit them into a tabular form.

```
<UDB>
   <UNIVERSITY>
      <NAME TYPE="PRIMARYKEY">
         University of Ottawa
      </NAME>
      <URL TYPE="REGULAR">
         http://www.uottawa.ca
      </URL>
      <COUNTRY TYPE="REGULAR">
         Canada
      </COUNTRY>
   </UNIVERSITY>
   <DEPARTMENT>
      <UNIVNAME TYPE="PRIMARYKEY">
         University of Ottawa
      </UNIVNAME>
      <NAME TYPE="PRIMARYKEY">
         Electrical engineering
      </NAME>
      <URL TYPE="REGULAR">
         http://www.site.uottawa.ca/
      </URL>
   </DEPARTMENT>
```

```
   <PROFESSOR>
      <NAME TYPE="PRIMARYKEY">
         John Smith
      </NAME>
      <POSITION TYPE="REGULAR">
         Professor
      </POSITION>
      <UNIVNAME TYPE="FOREIGNKEY">
         University of Ottawa
      </UNIVNAME>
   </PROFESSOR>
   <WORKS_AT>
      <UNIVNAME TYPE="PRIMARYKEY">
         University of Ottawa
      </UNIVNAME>
      <PROFNAME TYPE="PRIMARYKEY">
         John Smith
      </PRONAME>
      <STARTDATE TYPE="REGULAR">
         1991
      </STARTDATE>
   </WORKS_AT>
</UDB>
```

Figure 2.15: Generated Relational Entities

3.5 Web Change Detector

After loading information from the Web to the database, the XWA system needs to keep the database consistent when changes occur on the Web. This is performed by periodically interpreting each Web site ontology and updating the database with just the information that has changed. The simplest way to do that is to extract at each iteration all the desired information and check which entity has to be updated or added to the database. This approach is unacceptable because it requires a large storage capacity. The Web Change Detector, the fourth main component of the XWA system, uses other techniques to detect changes without consuming much of system resources. It consists of three filters distributed out of three stages as shown in Figure 2.16.

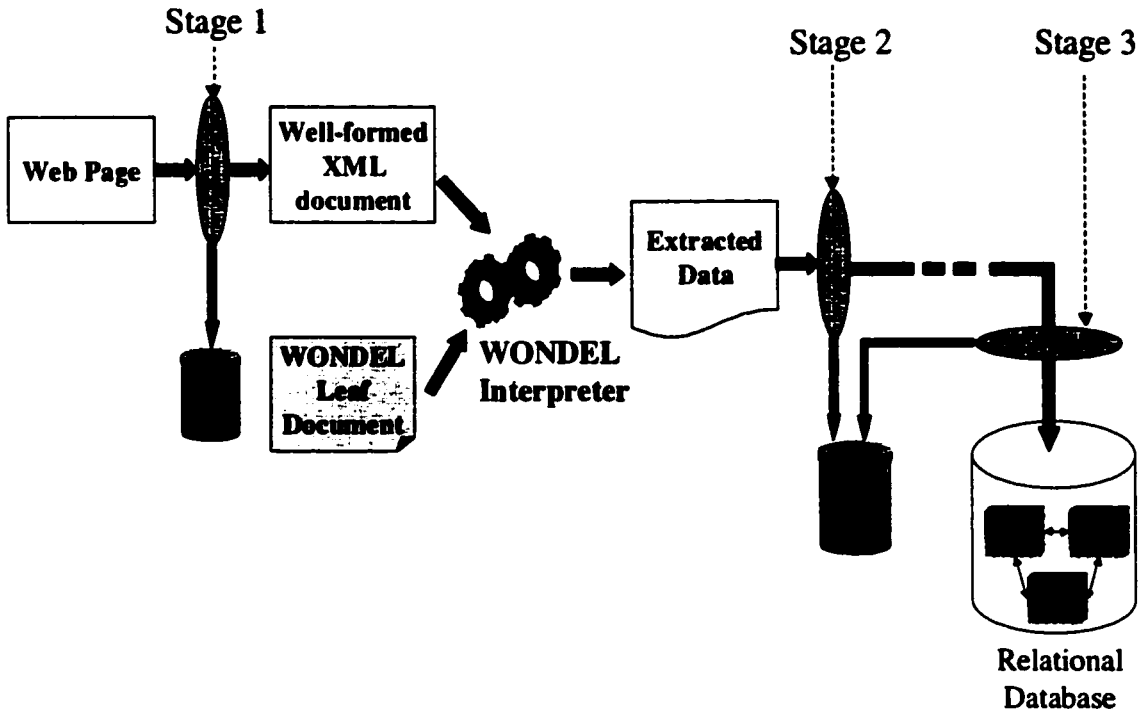


Figure 2.16: Three Stages of Web Change Detection

The first filter maintains a log file that includes the URIs of all the Web pages that have been visited by the XWA system and their last modification dates. This filter rejects any Web page that was previously visited and whose last modification date has not changed.

Because changes in a Web page do not necessarily affect the data of interest, a second filter is placed immediately after extracting data from the Web page. This filter holds a table that maps the file names of the generated documents to the checksums calculated by the Cyclic Redundancy Check (CRC) algorithm as shown in Figure 2.17.

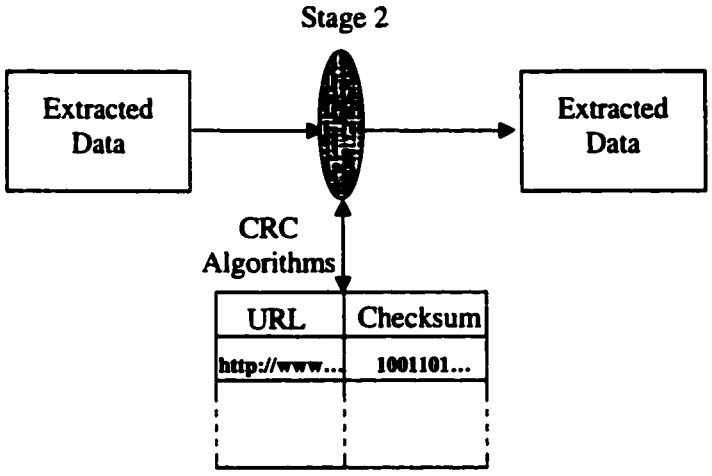


Figure 2.17: The Filter of Stage 2

The checksums help detect changes between two versions of data extracted form a Web page, in the same way errors are detected in data packets in computer networks. This is illustrated by the flowchart of Figure 2.18.

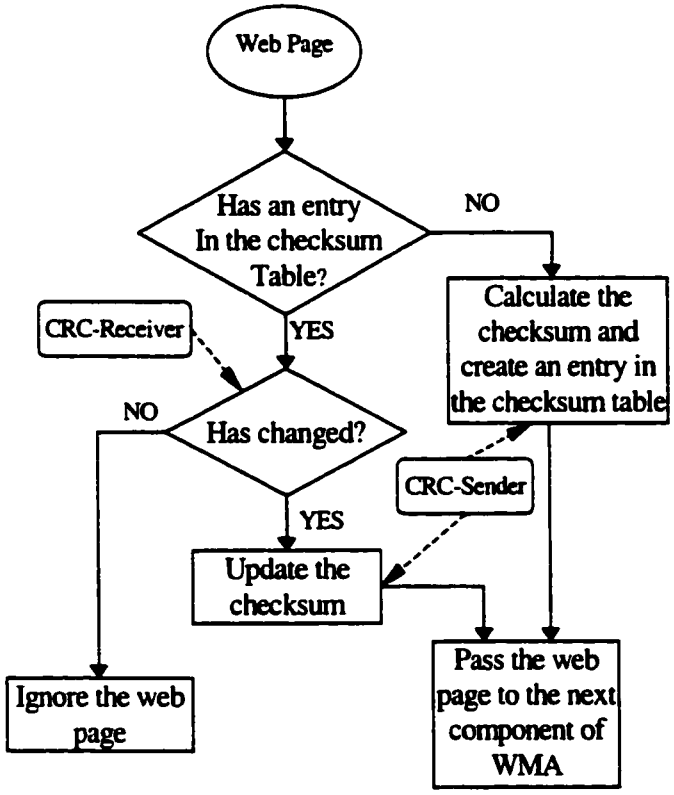


Figure 2.18: The Algorithm of Change Detection at Stage 2

The filters of the first and the second stages eliminate most of the duplicated data. The last stage occurs when populating the database. At this stage the agent adds or updates only the new entities or those that have changed.

3.6 Summary

The purpose of the XWA system is to extract information from the Web and store it into relational databases. This is achieved in four steps. First the Web pages are converted into well-formed XML documents to simplify their processing and to be able to use XML techniques in the extraction process. Then, data is extracted from the converted Web pages. Thereafter, the extracted data is collected to form valid XML documents. Finally, the valid XML documents are mapped to relational tables that are stored in relational databases. These four steps are introduced to enable the system to work offline, i.e. without continuous human assistance.

The XWA system is composed of four components. The first component, called the HTML2XML Converter, transforms HTML-based Web pages into well-formed XML documents. The second component, the WONDEL Interpreter, extracts and collects data to generate valid XML documents. The third component, which is called the Database Adapter, maps valid XML documents into relational tables. To keep the database consistent with the changes, a fourth component called the Web Change Detector is introduced. This component checks periodically whether the source of the extracted data has changed.

To extract data from the Web, the system is provided with information about the content of the Web pages as well as the data of interest to users. This information is stored in Web Site Ontologies

which consist of sets of documents organized hierarchically. These documents are written in WONDEL which is fully described in the next chapter.

Chapter 4

A Web ONtology DEscription Language (WONDEL)

The most important component of the XML-based Web Agent is the WONDEL Interpreter. This component is responsible for extracting the data wanted by users from Web sites. As seen in Section 3.3, the knowledge required by WONDEL Interpreter is stored in Web Site Ontologies which consist of trees of WONDEL leaf and node documents. This chapter describes WONDEL leaves and nodes and provides the specification of WONDEL. WONDEL leaves are described in the first section whereas WONDEL nodes are discussed in Section 4.2. Finally, the last section presents the new extensions that have been added to XML Pointer Language.

4.1 WONDEL Leaf Documents

WONDEL Leaf documents provide the information needed to extract data from individual Web pages. To understand their use, an example of WONDEL leaves is provided to introduce the basic features of WONDEL leaves. Thereafter, additional features and the specification of WONDEL leaves are discussed.

4.1.1 An Introductory Example

Consider the Web page presented in Figure 3.1 which contains a list of faculty members in the computer science department at a given university.

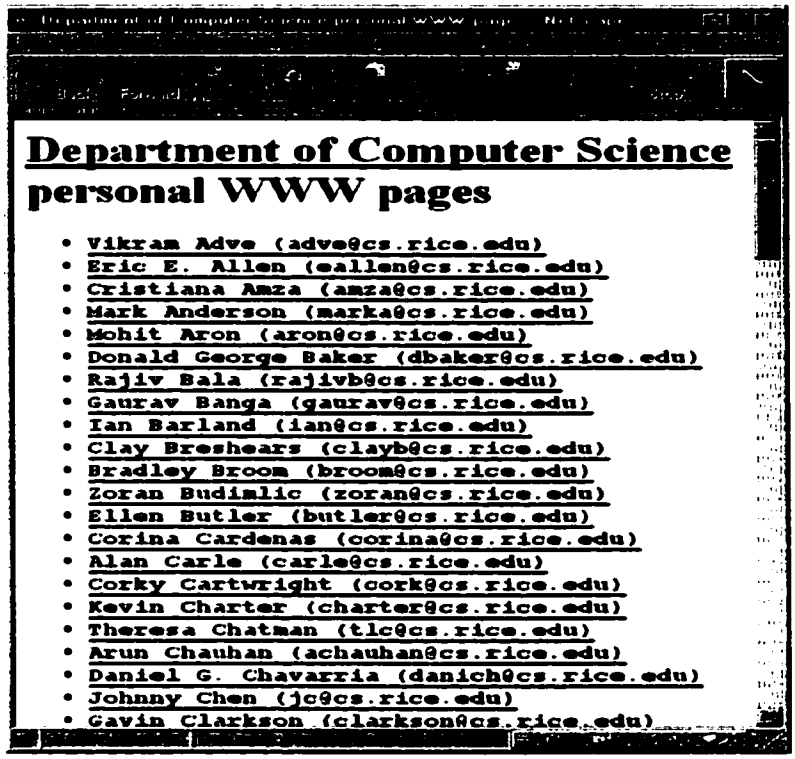


Figure 3.1: Example of a Web Page

Assume that we want to extract the names, e-mail addresses, and the homepages of the people listed in the Web page above. We also want to provide the extracted data with XML tags that describe its meaning as shown partially in Figure 3.2.

```
<STAFF>
  <HOMEPAGE>
    http://www.cs.rice.edu:80/~adve/
  </HOMEPAGE>
  <STAFFNAME>
    Vikram Adve
  </STAFFNAME>
  <EMAIL>
    adve@cs.rice.edu
  </EMAIL>
  <DEPARTMENTNAME>
    Computer Science
  </DEPARTMENTNAME>
</STAFF>
<STAFF>
  <HOMEPAGE>
    http://www.cs.rice.edu:80/~eallen/
  </HOMEPAGE>
  <STAFFNAME>
    Eric E. Allen
  </STAFFNAME>
  <EMAIL>
    eallen@cs.rice.edu
  </EMAIL>
  <DEPARTMENTNAME>
```

Figure 3.2: Data Extracted Using a WONDEL Leaf Document

The WONDEL leaf document that fulfills these requirements is given in Figure 3.3.

```
<LEAF FILE="CsRiceUsaPeople">
  <ENTITY NAME="STAFF" ISLOOP="YES" ROOT="ROOT().DESCENDANT(ALL,LI)">
    <ATTRIBUTE NAME="HOMEPAGE" SWITCH="NO">
      CHILD(1,A).ATTR(HREF)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="STAFFNAME" SWITCH="NO">
      CHILD(1,A).CHILD(1,TEXTFLOW).STRING(1,"(",0,-50)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="EMAIL" SWITCH="NO">
      CHILD(1,A). STRING(1,"(",1,50).STRING(1,")",0,-50)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="DEPARTMENTNAME" SWITCH="NO">
      Computer Science
    </ATTRIBUTE>
  </ENTITY>
</LEAF>
```

Figure 3.3: Example of WONDEL Leaf Document

As with any XML document, a WONDEL document consists of a tree of XML elements. In particular, a WONDEL leaf document is identified by the root element LEAF. The values of some attributes are XPointers used to select data within the processed Web page. Table 3.1 describes the scenario for interpreting the WONDEL leaf document of Figure 3.3.

ACTION	COMMENT
Select all the elements `LI` in the processed Web page	Those are the elements selected by the XPointer specified by the attribute `ROOT` of the element `ENTITY`: "`ROOT().DESCENDANT(ALL,LI)`"
For each selected element `LI` **Do**:	We have a loop because the attribute `ISLOOP` is set to `YES`.
• Generate an element whose name is `STAFF`	`STAFF` is specified by the attribute `NAME` of the element `ENTITY`.
• **For each** element `ATTRIBUTE` child of the element `ENTITY` **Do**:	We have four elements `ATTRIBUTE` inside the element `ENTITY` in Figure 4.3
– Generate an element whose name is specified by the attribute `NAME` of the element `ATTRIBUTE`.	For example, the first element `ATTRIBUTE` will generate the element `HOMEPAGE`.
– Select the information using the Xpointer contained in the element `ATTRIBUTE` and put it as content of the generated element.	For example, the element `HOMEPAGE` will contain the value of the attribute `HREF` of the first element `A` child of the current element `LI`. This is the result of the interpretation of the XPointer "`CHILD(1,A).ATTR(HREF)`"

Table 3.1: Interpreting the WONDEL leaf document of Figure 4.3

The example of Figure 3.3 shows the basic features of WONDEL leaves. More features are covered in the following sub-section.

4.1.2 The WONDEL Leaf Specification

The WONDEL specification includes two different DTDs that define the structure of the WONDEL leaves and nodes respectively. The DTD of WONDEL leaves is given in Figure 3.4.

```
<!ELEMENT LEAF       (ENTITY)+>
<!ELEMENT ENTITY     (ATTRIBUTE|ENTITY)+>
<!ELEMENT ATTRIBUTE (SWITCH+|CONCAT+|#PCDATA)>
<!ELEMENT SWITCH     (CONCAT+|#PCDATA)>
<!ELEMENT CONCAT     (#PCDATA)>

<!ATTLIST LEAF
          FILE   CDATA     #IMPLIED>
<!ATTLIST ENTITY
          NAME   CDATA     #REQUIRED
          ISLOOP (YES|NO)  #REQUIRED
          ROOT   CDATA     #REQUIRED>
<!ATTLIST ATTRIBUTE
          NAME   CDATA     #REQUIRED
          SWITCH (YES|NO)  #REQUIRED>
```

Figure 3.4: The Document Type Declarations of WONDEL Leaf Documents

The following is a description of each element and attribute declared by the DTD of WONDEL leaf documents.

1 **The element LEAF**: This is the root element of any WONDEL leaf document. It identifies a WONDEL leaf document and contains one or more elements `ENTITY`. The element `LEAF` has the following attribute:

1.1 *The attribute FILE*: An optional attribute that specifies the filename of the given WONDEL leaf document.

2 **The element ENTITY**: This element defines the XML elements that form the output document generated by WONDEL Interpreter when applying the given WONDEL leaf to a Web page. The

element ENTITY contains one or many elements of type ATTRIBUTE or ENTITY. It has three mandatory attributes:

2.1 *The attribute NAME*: Specifies the name of the XML element defined by ENTITY.

2.2 *The attribute ROOT*: Provides an XPointer that selects one or many XML elements form the Web page to which the current WONDEL leaf is applied.

2.3 *The attribute ISLOOP*: Indicates to the WONDEL Interpreter whether it has to generate one or many instances. If the value of ISLOOP is 'YES', the interpreter generates as many instances as there are elements selected by the XPointer of the attribute ROOT.

3 **The element ATTRIBUTE**: Like ENTITY, the purpose of the element ATTRIBUTE is to define an XML element in the output document. The element ATTRIBUTE can contain an XPointer, normal text, or one or many elements SWITCH or CONCAT. The content of the XML element defined by ATTRIBUTE is as described in Table 3.2.

The content of the element ATTRIBUTE	**The content of the XML element defined by ATTRIBUTE**
Simple text	The same text.
XPointer	The content of the element selected by the XPointer from the Web page to which the WONDEL leaf is applied.
Element(s) SWITCH	See the description of SWITCH elements later in this section.
Element(s) CONCAT	See the description of CONCAT elements later in this section.

Table 3.2: The Content of an XML Element Generated by an Element ATTRIBUTE

The element ATTRIBUTE has two mandatory attributes:

3.1 *The attribute NAME*: Specifies the name of the XML element defined by ATTRIBUTE.

3.2 *The attribute SWITCH*: Its value can be either "Yes" or "No". If set to "Yes", it means the element contains elements of type SWITCH.

4. **The element SWITCH**: The role of this element is very similar to the logical operator "OR" in programming languages. The example given in Figure 3.5 illustrates how it works.

```
...
<ATTRIBUTE NAME=" HOMEPAGE" SWITCH="YES">
  <SWITCH>
    CHILD(1,A).ATTR(HREF)
  </SWITCH>
  <SWITCH>
    Homepage not provided
  </SWITCH>
</ATTRIBUTE>
...
```

Figure 3.5: The Use of the Elements SWITCH

In this example, the WONDEL Interpreter attempts to find the data referenced by the XPointer CHILD(1,A).ATTR(HREF). If the data is not found, the interpreter will then interpret the content of the second element, and so on. The elements SWITCH have been introduced to give the WONDEL Interpreter more ability to search data in Web pages whose format is not very regular.

5. **The element CONCAT**: This element can be a child of an element of type ATTRIBUTE or SWITCH. It acts like the logical operator "AND". Its purpose is to allow the XML elements defined by an element ATTRIBUTE to contain a concatenation of pieces of text. An example of the use of CONCAT elements is given in Figure 3.6.

```
<ATTRIBUTE NAME=" STAFFNAME" SWITCH="NO">
  <CONCAT>
    Hello Mr.
  </CONCAT>
  <CONCAT>
    CHILD(1,A).CHILD(1,TEXTFLOW)
  </CONCAT>
</ATTRIBUTE>
```

Figure 3.6: The Use of the Elements CONCAT

The element ATTRIBUTE in this example defines an XML element whose name is STAFFNAME and content is the concatenation of the character "Hello Mr." and the text selected by the XPointer "CHILD(1,A).CHILD(1,TEXTFLOW)".

4.2 WONDEL Node Documents

The role of WONDEL nodes is to specify how to collect and structure the data extracted from Web pages using WONDEL leaves. This section provides a detailed description of the WONDEL nodes. Similar to the previous section, an introductory example is given followed by the specification of WONDEL nodes.

4.2.1 An Introductory Example

Consider the example shown in Figure 3.7.

```
<NODE FILE="UottawaCa">
  <CALLNODE MODEL="Department"  GENERATED="DepartmentGen"/>
  <CALLLEAF MODEL="Faculty"  GENERATED="FacultyGen">
    <WEBPAGE TYPE="URL" INLINE="YES">
      http://www.uottawa.ca/academic/depts.html
    </WEBPAGE>
  </CALLLEAF>
  <COLLECT NAME="FACULTY" ISLOOP="NO">
    <ELEMENT FILE="FacultyGen">
      ROOT().CHILD(2,FACULTY)
    </ELEMENT>
    <COLLECT NAME="DEPARTMENT" ISLOOP="YES"
        FOREACH="DepartmentGen#ROOT().CHILD(1,DEPTS).CHILD(ALL,DEPARTMENT)">
      <ELEMENT FILE="DepartmentGen">
        $FOREACH;
      </ELEMENT>
    </COLLECT>
  </COLLECT >
</NODE>
```

Figure 3.7: An Example of a WONDEL Node

As shown in this example, a WONDEL node document is composed of two parts:

- A call part which consists of a set of elements CALLLEAF and CALLNODE. Each specifies an invocation of another WONDEL document.
- A collect part in which the information generated by the called documents is collected.

A node invocation is performed by the element CALLNODE which has two mandatory attributes: MODEL that specifies the name of the invoked document, and GENERATED which indicates the filename of the document that will contain the generated data. In the example of Figure 3.7, the element CALLNODE invokes a WONDEL node whose filename is 'Department.xml' and the information generated by interpreting this node is stored in a file whose name is 'DepartmentGen.xml'.

WONDEL leaf documents are invoked by the element CALLLEAF, which has the same two attributes as the element CALLNODE. Unlike nodes, invoking a leaf document requires specifying the location(s) of the Web page(s) to which it will be applied. These locations are provided by the element

WEBPAGE that has two mandatory attributes: TYPE and INLINE. Their purpose will be discussed later in this section. By interpreting the element CALLEAF in the document of Figure 3.7, the WONDEL Interpreter applies the WONDEL leaf whose filename is 'Faculty.xml' to the Web page whose URI is 'http://www.uottawa.ca/academic/depts.html' and stores the interpretation result in a file whose name is 'FacultyGen.xml'.

The data extracted in the call part is marshaled in the collect part. To illustrate the marshaling, assume that the two documents 'DepartmentGen.xml' and 'FacultyGen.xml' generated in the call phase are as shown in Figure 3.8.

```
<UDB>
 <DEPTS>
  <DEPARTMENT>
   <NAME>Electrical Engineering</NAME>
   <URL>
    http://www.elg.uottawa.ca/
   </URL>
  </DEPARTMENT>
  <DEPARTMENT>
   <NAME>Civil Engineering</NAME>
   <URL>
    http://by.genie.uottawa.ca/cvg/
   </URL>
  </DEPARTMENT>
 </DEPTS>
 <DEPTS>
  <DEPARTMENT>
   <NAME>Biology</NAME>
   <URL>
    http://www.science.uottawa.ca/biology/
   </URL>
  </DEPARTMENT>
  <DEPARTMENT>
   <NAME>Physics</NAME>
   <URL>
    http://www.physics.uottawa.ca/
   </URL>
  </DEPARTMENT>
 </DEPTS>
</UDB>
```

DepartmentGen.xml

```
<UDB>
 <FACULTY>
   <NAME>
    Science
   </NAME>
   <URL>
    http://www.science.uottawa.ca/
   </URL>
 </FACULTY>
 <FACULTY>
   <NAME>
    Engineering
   </NAME>
   <URL>
    http://www.genie.uottawa.ca/
   </URL>
 </FACULTY>
</UDB>
```

FacultyGen.xml

Figure 3.8: The Documents Generated by Interpreting the Call Part in The document of Figure 3.7

In the following, we explain step by step how the WONDEL Interpreter interprets the collect part of the document given in Figure 3.7 to marshal some of the data stored in the two documents of Figure 3.8. We will proceed as follows: at each step we indicate the piece of WONDEL code being interpreted, the result of the interpretation and then explain the behavior of WONDEL Interpreter.

- **First Step:**
  - **The WONDEL Code:**
```
<COLLECT NAME="FACULTY" ISLOOP="NO">
```
  - **The Interpretation Result:** Add the tag <FACULTY> to the generated document.
  - **Explanation:** The interpreter generates a begin tag whose name is given by the attribute NAME. Since the attribute ISLOOP is set to 'NO', the element COLLECT will be interpreted only once.

- **Second Step:**
  - **The WONDEL Code:**
```
<ELEMENT FILE="FacultyGen">
     ROOT().CHILD(2,FACULTY)
</ELEMENT>
```
  - **The Interpretation Result**: Add the following text to the output document.
```
<NAME>
   Engineering
</NAME>
<URL>
   http://www.genie.uottawa.ca/
</URL>
```
  - **Explanation**: The element ELEMENT which specifies the content to be inserted into the output document, has one mandatory attribute FILE, which specifies the name of the file where the data is located. The content of ELEMENT is an XPointer which specifies the location of the data within the given file. Therefore, the WONDEL Interpreter will insert into the output document

the content of the element selected by the XPointer 'ROOT().CHILD(2,FACULTY)' from the file whose name is 'Faculty.xml'.

- **Third Step**:

  - **The WONDEL Code:**

```
<COLLECT NAME="DEPARTMENT" ISLOOP="YES"
  FOREACH="DepartmentGen#ROOT().CHILD(1,DEPTS).CHILD(ALL,DEPARTMENT)">
        <ELEMENT FILE="DepartmentGen">
            $FOREACH;
        </ELEMENT>
</COLLECT>
```

  - **The Interpretation Result:** Add the following text to the output document.

```
<DEPARTMENT>
     <NAME>Electrical Engineering</NAME>
     <URL>
         http://www.elg.uottawa.ca/
     </URL>
 </DEPARTMENT>
 <DEPARTMENT>
     <NAME>Civil Engineering</NAME>
     <URL>
        http://by.genie.uottawa.ca/cvg/
     </URL>
</DEPARTMENT>
```

  - **Explanation:** Whenever ISLOOP is set to 'YES', the attribute FOREACH must be specified. The value of this attribute has the following format: <FileName>#<XPointer>, where <FileName> is the name of the file that contains the data to be inserted into the output document, and <XPointer> is the XPointer that selects that data. Because the attribute ISLOOP is set to "YES", the WONDEL Interpreter will repeat an action similar to the one explained in the first and the second steps as many times as the number of the elements selected by <XPointer>. Therefore, since the following XPointer:

```
'ROOT().CHILD(1,DEPTS).CHILD(ALL,DEPARTMENT)'.
```

selects two elements DEPARTMENT, the WONDEL Interpreter iterates twice. The content of ELEMENT is the special word '$FOREACH;' that refers to the element selected by the XPointer and corresponds to the current iteration. At each iteration, the interpreter inserts the begin tag '<DEPARTMENT>' specified by the attribute NAME of the element COLLECT, adds the content of the selected element corresponding to the current iteration, and closes that content by the tag '</DEPARTMENT>'.

- **Last Step**:
  - **The WONDEL Code:**
    ```
    </COLLECT>
    ```
  - **The Interpretation Result:** Add the closing tag </DEPARTMENT> to the output document.
  - **Explanation:** Whenever WONDEL Interpreter meets the closing tag of the element COLLECT, it adds a closing tag whose name is specified by the attribute NAME of that element.

The document generated by interpreting the WONDEL node of Figure 3.7 is shown in Figure 3.9. As shown in this Figure, the WONDEL Interpreter was able to collect the information concerning the faculty of engineering and its departments and organize it in a format that is logically consistent. Indeed, the elements DEPARTMENT are now children of the element FACULTY, which reflects the fact that the departments belong to their corresponding faculty.

```
<UDB>
  <FACULTY>
    <NAME>
      Engineering
    </NAME>
    <URL>
      http://www.genie.uottawa.ca/
    </URL>
    <DEPARTMENT>
      <NAME>Electrical Engineering</NAME>
      <URL>
        http://www.elg.uottawa.ca/
      </URL>
    </DEPARTMENT>
    <DEPARTMENT>
      <NAME>Civil Engineering</NAME>
      <URL>
        http://by.genie.uottawa.ca/cvg/
      </URL>
    </DEPARTMENT>
  </FACULTY>
</UDB>
```

Figure 3.9: The Document Generated at the Collect Phase

The WONDEL node document given in Figure 3.7 links one faculty to its departments. Figure 3.7 is a modified version of this document which links all faculties to their corresponding departments.

```
<NODE FILE="UottawaCa">
  <CALLNODE MODEL="Department" GENERATED="DepartmentGen"/>
  <CALLLEAF MODEL="Faculty" GENERATED="FacultyGen">
    <WEBPAGE TYPE="URL" INLINE="YES">
      http://www.uottawa.ca/academic/depts.html
    </WEBPAGE>
  </CALLLEAF>

  <COLLECT NAME=" FACULTY" ISLOOP="YES"
     FOREACH="FacultyGen#ROOT().CHILD(ALL,FACULTY)">
    <ELEMENT FILE="FacultyGen">
      $FOREACH;
    </ELEMENT>
    <ELEMENT FILE="DepartmentGen" MANY="YES">
      ROOT().DESCENDANT(ALL,DEPTS)
    </ELEMENT>
  </COLLECT>
</NODE>
```

Figure 3.10: Another Example of WONDEL Node Documents

In the collect part of the example of Figure 3.10, there is one element COLLECT whose attribute ISLOOP is set to 'YES'. This means that the WONDEL Interpreter iterates as many times as the number of the elements selected by the XPointer specified by the attribute FOREACH. Now consider the first iteration. During this iteration, the interpreter will act as follows:

1. It first inserts the tag <FACULTY> into the output document.

2. Then it adds the content of the first element FACULTY from the document 'FacultyGen.xml' to the output document

3. Now, the interpreter reaches the second ELEMENT. This element has an attribute MANY that is set to 'YES'. In this case, the interpreter expects that the content of ELEMENT is an XPointer that selects many XML elements from the document whose name is specified by the attribute FILE. In our example, the XPointer selects two elements DEPTS from the document 'DepartmentGen.xml'. At the first iteration, and because the attribute MANY is set to 'YES', the interpreter inserts the content of the first element selected by the XPointer into the output document.

4. At the end of the first iteration, the interpreter inserts the closing tag </FACULTY>.

This scenario will be repeated for the second iteration. The result of this process is the document shown in Figure 3.11.

```
<UDB>
  <FACULTY>
    <NAME>
      Engineering
    </NAME>
    <URL>
      http://www.genie.uottawa.ca/
    </URL>
   <DEPARTMENT>
    <NAME>
      Electrical Engineering
    </NAME>
    <URL>
      http://www.elg.uottawa.ca/
    </URL>
   </DEPARTMENT>
   <DEPARTMENT>
    <NAME>
     Civil Engineering
    </NAME>
    <URL>
      http://by.genie.uottawa.ca/cvg/
    </URL>
   </DEPARTMENT>
  </FACULTY>
```

```
<FACULTY>
    <NAME>
      Science
    </NAME>
    <URL>
      http://www.science.uottawa.ca/
    </URL>
   <DEPARTMENT>
    <NAME>
      Biology
    </NAME>
    <URL>
      http://www.science.uottawa.ca/biology/
    </URL>
   </DEPARTMENT>
   <DEPARTMENT>
    <NAME>
      Physics
    </NAME>
    <URL>
      http://www.physics.uottawa.ca/
    </URL>
   </DEPARTMENT>
  </FACULTY>
</UDB>
```

Figure 3.11: The Interpretation result of the WONDEL Node given at Figure 3.10

4.2.2 The WONDEL Node Specification

This section presents the Document Type Declarations (DTD) of WONDEL nodes and then explains the purpose of each element and each attribute declared in the DTD.

```
<!ELEMENT NODE       (CALLNODE|CALLLEAF|COLLECT)+>
<!ELEMENT CALLNODE    EMPTY>
<!ELEMENT CALLLEAF   (WEBPAGE)+>
<!ELEMENT WEBPAGE    (#PCDATA)>
<!ELEMENT COLLECT    (ELEMENT|COLLECT)+>
<!ELEMENT ELEMENT    (#PCDATA)>

<!ATTLIST NODE
          FILE                    CDATA     #IMPLIED>
<!ATTLIST CALLNODE
          MODEL                   CDATA     #REQUIRED
          GENERATED               CDATA     #REQUIRED
          RECURSIVECONDITION      CDATA     #IMPLIED
          PREVIOUSGENERATED       CDATA     #IMPLIRED
          MAXIMUMRECURSIVEDEPTH CDATA       #IMPLIRED>
<!ATTLIST CALLLEAF
          MODEL                   CDATA     #REQUIRED
          GENERATED               CDATA     #REQUIRED>
<!ATTLIST WEBPAGE
          TYPE                    CDATA     #REQUIRED
          INLINE                  (YES|NO)  #REQUIRED>
<!ATTLIST COLLECT
          NAME                    CDATA     #REQUIRED
          ISLOOP                  (YES|NO)  #REQUIRED
          FOREACH                 CDATA     #IMPLIED>
<!ATTLIST ELEMENT
          FILE                    CDATA     #REQUIRED
          MANY                    (YES|NO)  #IMPLIED>
```

Figure 3.12: The Document Type Declarations of WONDEL Node Documents

1 **The element NODE**: It is the root element and the identifier of any WONDEL node document. It can contain any combination of the elements CALLNODE, CALLLEAF, and COLLECT. The element NODE has the following attribute:

1.1 *The attribute FILE*: An optional attribute that specifies the file name of the WONDEL node.

2 **The element CALLNODE**: It is used to call another WONDEL node. It has two mandatory attributes:

2.1 *The attribute MODEL*: Specifies the location of the WONDEL node to be invoked.

2.2 *The attribute GENERATED*: Indicates the filename of the document which will contain the result of the interpretation of the invoked node.

Recursive invocations are permitted within elements CALLNODE. A recursive call is a sequence of invocations of a WONDEL node by itself until a given condition is no longer satisfied. An example of an element CALLNODE that performs a recursive call is given in Figure 3.13.

```
<CALLNODE MODEL="RecursiveCall"
          GENERATED="RecursiveCallGen"
          RECURSIVECONDITION="LeafFileGen#ROOT().DESCENDANT(ALL,URL)"
          PREVIOUSGENERATED="PreRecursiveCallGen"
          MAXIMUMRECURSIVEDEPTH="6"
/>
```

Figure 3.13: A Recursive Call

When a CALLNODE performs a recursive call, it must have three additional attributes:

2.3 *The attribute RECURSIVECONDITION*: Specifies the condition that has to be satisfied to perform the recursive call. The value of this attribute consists of a filename and an XPointer that selects elements from the XML document whose filename is given. The condition is satisfied only if the XPointer selects a non-empty set of elements that were selected in the previous invocations of the node within the same recursive call.

2.4 *The attribute PREVIOUSGENERATED*: Specifies the filename of a document that is used as a temporal backup of the document generated at the previous invocation. This backup is necessary because the information it contains could be lost before it is collected by an element COLLECT.

2.5 *The attribute MAXIMUMRECURSIVEDEPTH*: Specifies the maximum depth of a recursive call. In other words, it specifies the maximum number of invocations within a recursive call.

Although this attribute is optional, it can be used to control recursive calls when there is a risk that the recursive condition is perpetually satisfied whereby the call would never end.

3 **The element CALLLEAF**: It is used to call WONDEL leaf documents. It has the same two mandatory attributes as the element CALLNODE:

3.1 *The attribute MODEL:* Specifies the location of the WONDEL leaf to be called.

3.2 *The attribute GENERATED:* Provides the filename of the document to be generated by interpreting the called leaf.

4 **The element WEBPAGE**: Unlike nodes, calling a leaf document requires specifying the locations of the Web pages to which it will be applied. These locations are specified by the element WEBPAGE. An element CALLLEAF may contain many elements WEBPAGE to allow the invocation of the same WONDEL document on many Web pages. The element WEBPAGE has two mandatory attributes:

4.1 *The attribute TYPE*: Specifies whether the Web page location is a URI or a local filename.

4.2 *The attribute INLINE*: Indicates the type of the call: inline or offline call.

The notion of offline calls is borrowed from the XLink specification [4]. It allows the invocation of a WONDEL Leaf document on a set of Web pages whose locations are listed in another page. An example of an offline call is given in Figure 3.14.

```
<CALLLEAF MODEL="HomePages" GENERATED="HomePagesGen">
   <WEBPAGE TYPE="URL" INLINE="NO">
      UottawaCaPeopleGen#ROOT().DESCENDANT(ALL,HOMEPAGE)
   </WEBPAGE>
</CALLLEAF>
```

Figure 3.14: Example of an Offline Call

In an offline call, the content of the element WEBPAGE has the following format: <FileLocation>#<XPointer>, where <FileLocation> is the location of a document that contains the locations of the Web pages to which the WONDEL leaf will be applied, and <XPointer> is an XPointer that specifies where to find those locations within that document.

5 **The element COLLECT**: It is used to collect information generated by the WONDEL documents called by the current node or any other WONDEL node. While interpreting an element COLLECT, the WONDEL Interpreter generates one or many XML elements that contain the collected information. An element COLLECT has two mandatory attributes:

5.1 *The attribute NAME*: It provides the name of the XML element(s) to be generated.

5.2 *The attribute ISLOOP*: It specifies whether one or many XML elements will be generated.

When ISLOOP is set to YES, COLLECT has to have the following attribute:

5.3 *The attribute FOREACH*: The value of this attribute has the following format: <FileName>#<XPointer>. <FileName> is the filename of the document that contains the information to be collected, and <XPointer> is an XPointer that provides the location of that information within the given document. In this case, WONDEL Interpreter will generate as many XML elements as those selected by the XPointer.

An element COLLECT contains one or many elements ELEMENT or COLLECT.

6 **The element ELEMENT**: It can be a child of an element COLLECT only. It specifies the information to be inserted into the output document. It has one mandatory attribute:

6.1 *The attribute FILE*: It provides the filename of a document from which information will be collected.

The element ELEMENT can have one of the following contents:

- An XPointer that selects, from a document whose filename is specified by the attribute FILE, an XML element whose content is to be inserted into the output document.
- The special word '$FOREACH;' that is permitted only if the element COLLECT, parent of ELEMENT, has the attribute FOREACH. The purpose of this special word is explained in the introductory example of Section 4.2.1.
- Any other normal text that will be inserted as is into the output document.

This section as well as the previous one shows clearly that XML Pointer Language (XPointer) is extensively used in WONDEL documents. XPointer language provides a simple and efficient way to locate information in Web pages. This is why the XPointer specification provided by the W3C has been fully implemented within this thesis. In addition, this thesis proposes new extensions to the XPointer language that are discussed in the next section.

4.3 Extensions to XML Pointer Language

To make the XML Pointer language able to express more powerful queries in order to produce more accurate results, new keywords are added to the language and the arguments of some location terms have been slightly modified. These additions are listed in the following Sub-section. Two examples that illustrate their use and their advantages are provided in Sub-section 4.3.2. Note that understanding the remainder of this section requires background knowledge on the XPointer language. The reader can find a survey of the language in Section 2.4 of this thesis report.

4.3.1 The List of the proposed Extensions to the XPointer Language

1. **The keyword ATOM**: This keyword selects the smallest XML element that contains a given character string. The relative location term associated with this keyword has the following syntax:

   ```
   ATOM(Occurrence , Text)
   ```

   For example, the XPointer:

   ```
   ROOT().ATOM(3,"Multimedia")
   ```

   selects the smallest element that contains the third occurrence of the string "Multimedia".

2. **The keyword WEBPAGE**: This keyword allows moving from the XML document to which the XPointer is applied, to another one. The relative location term associated with this keyword has one argument that may have one of the following values:

   - The name of an attribute whose value is a URI or a file identifier of a Web page or an XML document. In this case, the previous location term must specify an element that has that attribute.

- The URI or the file identifier of a Web page or an XML document. This value has to be enclosed between two quotation marks. In such a case, there is no constraint on the previous location term, which is simply ignored.

The location term associated with the keyword WEBPAGE selects the root element of the document whose location is given as argument. For instance, the XPointer:

```
ROOT().DESCENDANT(1,A).WEBPAGE(HREF)
```

selects the root element of the Web page whose URI is specified by the attribute HREF of the first element A in the current XML document.

3. **The search type switches: BFS and DFS**: Some keywords of the XPointer language require traversing the tree of XML elements to find results. These keywords are DESCENDANT, FOLLOWING, PRECEDING, and SPAN, in addition to the new keyword ATOM. The current specification prescribes that for the first three keywords the search is depth-first. For the keyword SPAN, the type of search is not specified. The purpose of the two new keywords BFS and DFS is to enable users to choose the type of search: breadth-first or depth-first. Following is an example that shows how the search type switches BFS and DFS can be used:

```
ROOT().DESCENDANT(1,UL).BFS().FOLLOWING(1,LI).DFS().ATOM(2,"XML")
```

By default, the search is depth-first. Once one of the two keywords is met, it affects the way of traversing the tree for all the following location terms until another search type switch is met.

4. **The keyword SUBSET**: The keyword SUBSET is proposed to allow the selection of a specific subset of elements among those selected by a spanning location term. Therefore, the location term associated with the keyword SUBSET should only appear after a spanning location term as illustrated in the following example.

```
ROOT().SPAN(DESCENDANT(1,LI),DESCENDANT(-1,LI)).SUBSET(ALL,A)
```

The XPointer above selects all the elements A located between the first and the last element LI in the current XML document. Note that the location term associated with SUBSET has the same arguments as the seven relative location terms defined in the W3C's specification.

5. **Selecting sub-sets of elements in relative location terms**: The current specification of the XPointer language defines seven keywords for relative location terms. All of them have the following syntax (the arguments between two square brackets are optional):

```
KEYWORD(Occurrence,ElementName[,AttributeName,AttributeValue])
```

The first argument Occurrence may be either a number, in which case the location term selects one element, or the character string "ALL", which means that all the elements that satisfy the conditions given by the other arguments are selected. Nevertheless, there is no way to select a sub-set of elements that satisfy the given conditions. This may be possible if the argument Occurrence can also be written as "N-M" where N and M are two numbers. For example, the XPointer:

```
ROOT().DESCENDANT(2-4,A)
```

selects the second, the third and the fourth element A. Note that both N and M can be positive or negative numbers. For example, the following XPointer:

```
ROOT().DESCENDANT(2--1,A)
```

selects from the second element A till the last one.

6. **Using regular expressions for text search**: The XPointer interpreter that has been implemented as a part of this thesis work uses regular expressions for text search. Although regular expressions are included in an ancestor of the XPointer language called the TEI Extended Pointers [5], they are not

included in the XPointer specification itself. However, the use of regular expressions in the XWA system has significant impact because it helps filter the information extracted from Web pages and produce more accurate results. Regular expressions can now be used in the second argument of the locations terms identified by the keywords STRING and ATOM, and the fourth argument of all the relative location terms, which are identified by the keywords CHILD, DESCENDANT, FOLLOWING, ANCESTOR, FSIBLING, PSIBLING, and SUBSET. Their use is illustrated in one of the examples given below.

4.3.2 Illustrative Examples

Below, two examples that illustrate the use of the new features added to the XPointer language are provided. The new keywords involved in the first example are DFS, BFS, and SUBSET. The second example illustrates the use of other new features in a WONDEL leaf document.

- **Example 1**: The Web page shown in Figure 3.15 contains information about faculties, departments, and programs at a given university. This Web page is composed of a set of elements LI organized hierarchically in three levels: level 1 for the faculties, 2 for the departments, and level 3 for the programs offered by each department. We will see how to use some of the new keyword to write a XPointer that selects the elements LI of level 2, i.e. those who contain information about the departments.

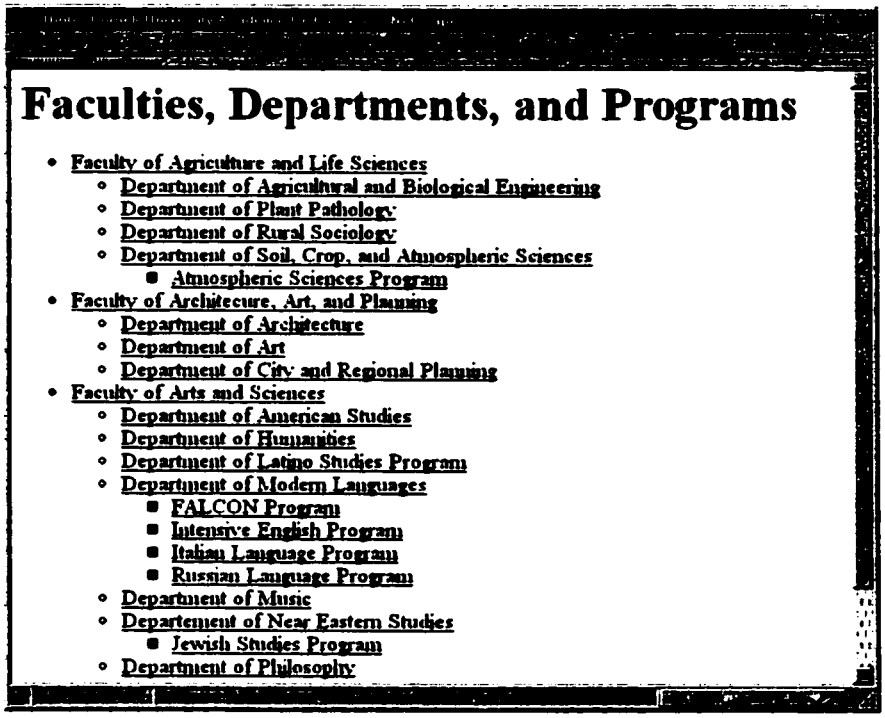


Figure 3.15: Faculties, Departments, and Programs

The XPointer is given in Figure 3.16. The new keywords that are included in this XPointer are `BFS`, `DFS` and `SUBSET`. Below is explained how this XPointer works.

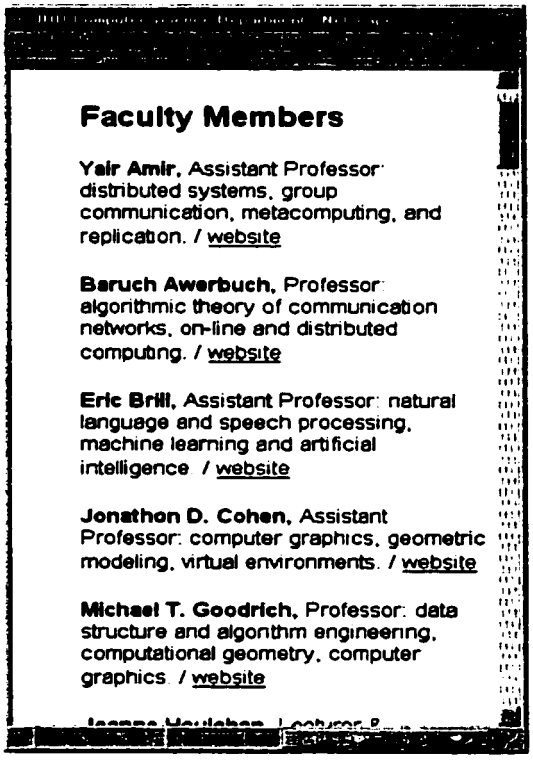
```
ROOT().BFS().SPAN(DFS().DESCENDANT(2,LI),DFS().DESCENDANT(-1,LI)).SUBSET(ALL,LI)
                  ❶                ❷                ❸                ❹
```

Figure 3.16: An XPointer with the elements BFS, DFS, and SUBSET

The objective is to select all and only the elements LI in level 2. Therefore, since the breadth-first search traverses a tree level by level, the keyword `BFS` is used in ❶. A spanning location term is used to restrain the traverse between the first and the last elements `LI` in level 2. These two elements are respectively selected by the XPointers ❷ and ❸. Notice that the first and the last element `LI` in level 2 are respectively the second and the last `LI` in the document, if the tree is traversed in depth-first search. This is why the keyword `DFS` is included in ❷ and ❸ to switch the search locally to depth-first. Finally, from the elements selected by the spanning location term, the

keyword SUBSET included in ❶ selects just the elements LI. With the use of the three new keywords DFS, BFS and SUBSET, we have been able to select a set of elements that could not be selected using the old keywords of the XPointer language.

- **Example 2**: The new keywords used in this example are the keywords ATOM and WEBPAGE as well as regular expressions. The objective here is to write a WONDEL leaf document to extract names, e-mail addresses, positions, and homepages of the people listed in the Web page shown on the left of Figure 3.17, and generate the document shown partially on the right of the same figure below.




```
<STAFF>
  <STAFFNAME>
    Yair Amir,
  </STAFFNAME>
  <HOMEPAGE>
    http://cs.jhu.edu/~yairamir
  </HOMEPAGE>
  <POSITION>
    Assistant Professor
  </POSITION>
  <EMAIL>
    yairamir@cs.jhu.edu
  </EMAIL>
</STAFF>
<STAFF>
  <STAFFNAME>
    Baruch Awerbuch,
  </STAFFNAME>
  <HOMEPAGE>
    http://cs.jhu.edu/~baruch
  </HOMEPAGE>
  <POSITION>
```

Figure 3.17: A Web Page and the Extracted Information

The WONDEL leaf that fulfills these requirements is given in Figure 3.18.

```
<LEAF FILE="CsJhuUsaPeople">
  <ENTITY NAME="STAFF" ISLOOP="YES"
        ROOT="ROOT().DESCENDANT(ALL,STRONG)">
    <ATTRIBUTE NAME="STAFFNAME" SWITCH="NO">
      DESCENDANT(1,NOELEMENT)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="HOMEPAGE" SWITCH="NO">
      FSIBLING(1,A).ATTR(HREF)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="POSITION" SWITCH="NO">
      FSIBLING(1,.).STRING(1,":",0,-120)
    </ATTRIBUTE>
    <ATTRIBUTE NAME="EMAIL" SWITCH="NO">
      FSIBLING(1,A).WEBPAGE(HREF).ATOM(1,"@").STRING(1,"\w+@(\w+\.)+\w+")
    </ATTRIBUTE>
  </ENTITY>
</LEAF>
```

Figure 3.18: A WONDEL Leaf with New XPointer Extensions

Let us focus on the XPointer contained in the fourth element `ATTRIBUTE` in the document of Figure 3.18. This XPointer extracts the email addresses of the faculty members listed in the Web page of Figure 3.17. We are particularly interested in this XPointer because it involves three of the new features of the XPointer language. Because the email address of each faculty member is located in his or her homepage, the keyword `WEBPAGE` is used to move to the root element of that homepage. We have assumed, and we were right according to our experiments, that the first character "@" in the homepage is in fact the famous "at" in the email address. This is why the location term `ATOM(1,"@")` is used to select the smallest element that contains the character "@". Nevertheless, the selected element may contain text other than the email address and needs to be filtered. For this reason, the location term `STRING(1,"\w+@(\w+\.)+\w+")` includes a regular expression that matches exactly the email address.

Note that what was achieved by the keyword `WEBPAGE`, could have been done by writing additional WONDEL leaves and nodes: we could have defined one more leaf to extract email addresses from

professors' homepages, and another node to gather the information extracted by both leaves. However, the keyword WEBPAGE provides a much simpler way to achieve the same task: we wrote just three lines to our existing leaf instead of adding two additional WONDEL documents. In addition, the readability of our WONDEL code is now better: it is easier for readers to understand what kind of information this WONDEL leaf meant to extract. With the other approach, readers have to read three files and make the effort to understand how they relate to each other in order to see the task they are trying to achieve.

4.5 Summary

This chapter provided an exhaustive description of the Web ONology DEscription Language (WONDEL) as well as new extensions to the XML Pointer language. WONDEL is used to build Web Site Ontologies that store the information needed to extract data from given Web sites. A Web Site Ontology consists of a set of WONDEL documents organized hierarchically. We distinguish two kinds of these documents: leaves and nodes.

The WONDEL leaves are the information units within an ontology. The information stored in a WONDEL leaf is used to extract data from individual Web pages. This information includes the location and the meaning of particular data within a Web page. Although the WONDEL leaves seem at first to be specific to the Web pages for which they have been designed, our experience shows that they can be reused for other Web pages as well. A WONDEL leaf may be applicable to many Web pages even if their formats are not exactly the same. An example that we have seen quite often is when the data in two different Web pages is located in LI elements that are differently distributed within the two pages. In this case, the same XPointer 'ROOT().DESCENDANT(ALL,LI)' can select these elements. This implies that the same WONDEL leaf can be applicable to both of them. Theoretically, it is not

guaranteed that WONDEL leaves are always reusable but the experience shows that they are so in many practical cases. This was one of the initial motivations of this work.

The WONDEL nodes concern the organizational aspect of the Web Site Ontologies. They are used to marshal the data extracted from the Web pages using the WONDEL leaves. A WONDEL node is composed of two parts: a call part and a collect part. The purpose of the call part is to invoke WONDEL leaves and provide them with the URIs of the Web pages to which they are applied. Other WONDEL nodes can also be invoked in the call part. By interpreting the collect part, the WONDEL Interpreter marshals the information generated by the WONDEL documents that were invoked in the call part. The collect part provides the interpreter with the filenames of the documents from which the data is to be collected, the location of that data within those documents, and how it should be organized.

XML Pointer language (XPointer) is used in WONDEL to locate information in Web pages, as well as in the XML documents generated by the WONDEL Interpreter. As part of this work, new extensions have been added to the XPointer language to enhance the capability of WONDEL. Five new keywords have been added. The keyword `WEBPAGE` allows searching XML documents other than the one to which the current XPointer is applied. The location term identified by the keyword `ATOM` selects the smallest XML element that contains a given text. `BFS` and `DFS` are two keywords added to change the way of traversing a tree of XML elements. `BFS` corresponds to breadth-first search while `DFS` corresponds to depth-first search. A location term identified by the keyword `SUBSET` selects a subset of elements from those that have been selected by a spanning location term. In addition, we have introduced regular expressions as arguments of many location terms. Finally, all relative location terms have been enabled to select subsets of elements that match a given condition.

Chapter 5

Implementation

The XML-based Web Agent has been implemented on a networked Windows NT 4.0 platform using Java Development Kit (JDK) 1.2. Microsoft XML Parser for Java version 1.8 is used to parse both well-formed and valid XML documents. An HTML Parser able to handle HTML syntax errors in the Web pages has been implemented within this work. The HTML Parser is used as a part of the HTML2XML Converter which converts Web pages into well-formed XML documents. Also, the XML Pointer Language (XPointer) specification has been implemented as defined by the W3C in [3] along with the new extensions described in Section 4.3 of this thesis report. In addition, the WONDEL Builder, an editor for WONDEL documents and a debugger for Web site ontologies has been implemented. The graphical user interface of the WONDEL Builder is based on Sun's Swing package.

The first section of this chapter provides the class diagrams of some components of the system. The second section reports on the application of the system to extract information from university Web sites. The third section describes the WONDEL Builder. The fourth section provides experimental results obtained from applying the XML-based Web Agent to 13 university Web sites. Finally, the last section discusses the limitations of the proposed mechanism.

5.1 Class Diagrams

The design of the XWA system uses an object-oriented notation called UML (Unified Modeling Language) [28]. This section describes the most important packages of the system. Four packages are presented in this section. They correspond to the following components: the HTML2XML Converter, the WONDEL Interpreter, the Database Adapter and the XPointer Interpreter.

5.1.1 The HTML2XML Converter Package

The class diagram of HTML2XML Converter is shown in Figure 5.1.

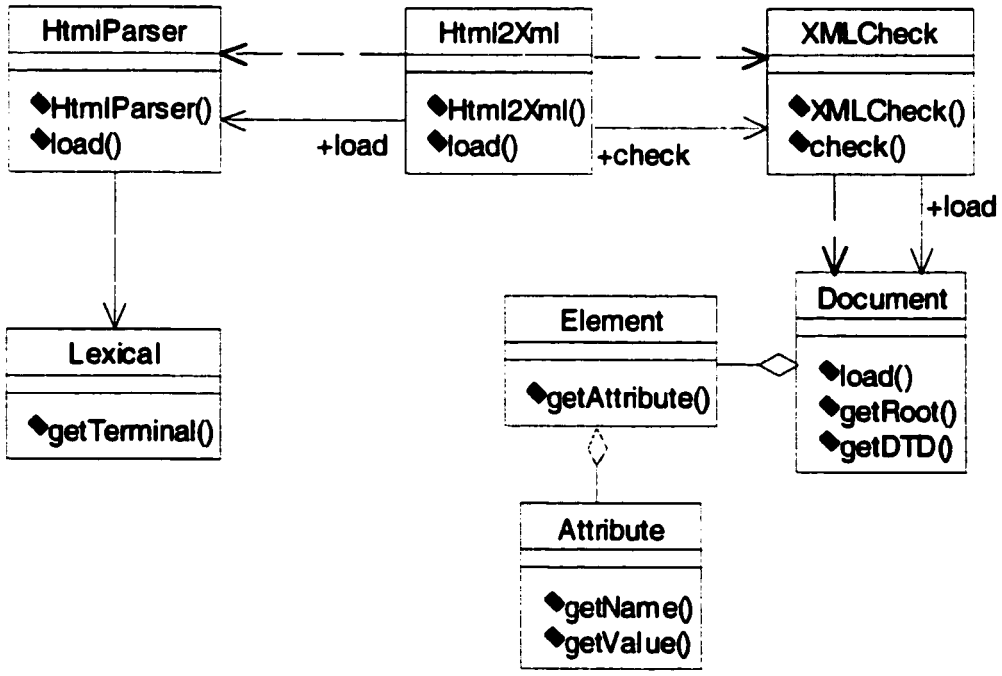


Figure 5.1: The Class Diagram of HTML2XML Converter

The main class in the package above is `Html2Xml`. When the method load() of this class is invoked, it first instantiates an object of the class `HtmlParser` and then calls the method load() of that object. `HtmlParser` is the class responsible for converting a Web page into a well-formed XML

document. Once the conversion is achieved, `Html2Xml` instantiates an object of the class `XMLCheck` and then calls the method `check()` of that class. As its name indicates, `XMLCheck` checks if the document passed to it does conform to XML syntax. It then makes other transformations on the document such as resolving relative URIs and adding the tags `TEXTFLOW`. These transformations are described in detail and illustrated by examples in Section 3.2. To parse the XML document provided in its input, `XMLCheck` uses the classes `Document`, `Element`, and `Attribute` from the Microsoft XML Parser version 1.8. The interaction between the different classes of HTML2XML Converter is summarized in the interaction diagram provided in Figure 5.2.

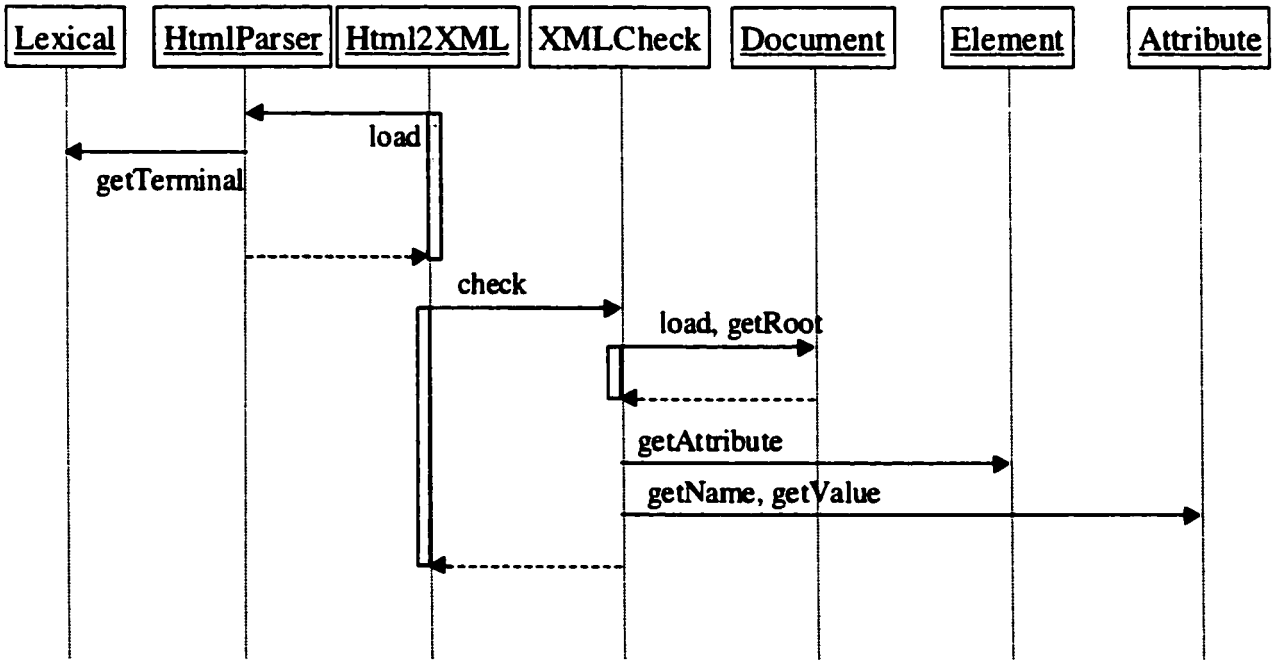


Figure 5.2: The Interaction Diagram of the HTML2XML Converter

5.1.2 The WONDEL Interpreter Package

The class diagram of the WONDEL Interpreter package is given in Figure 5.3.

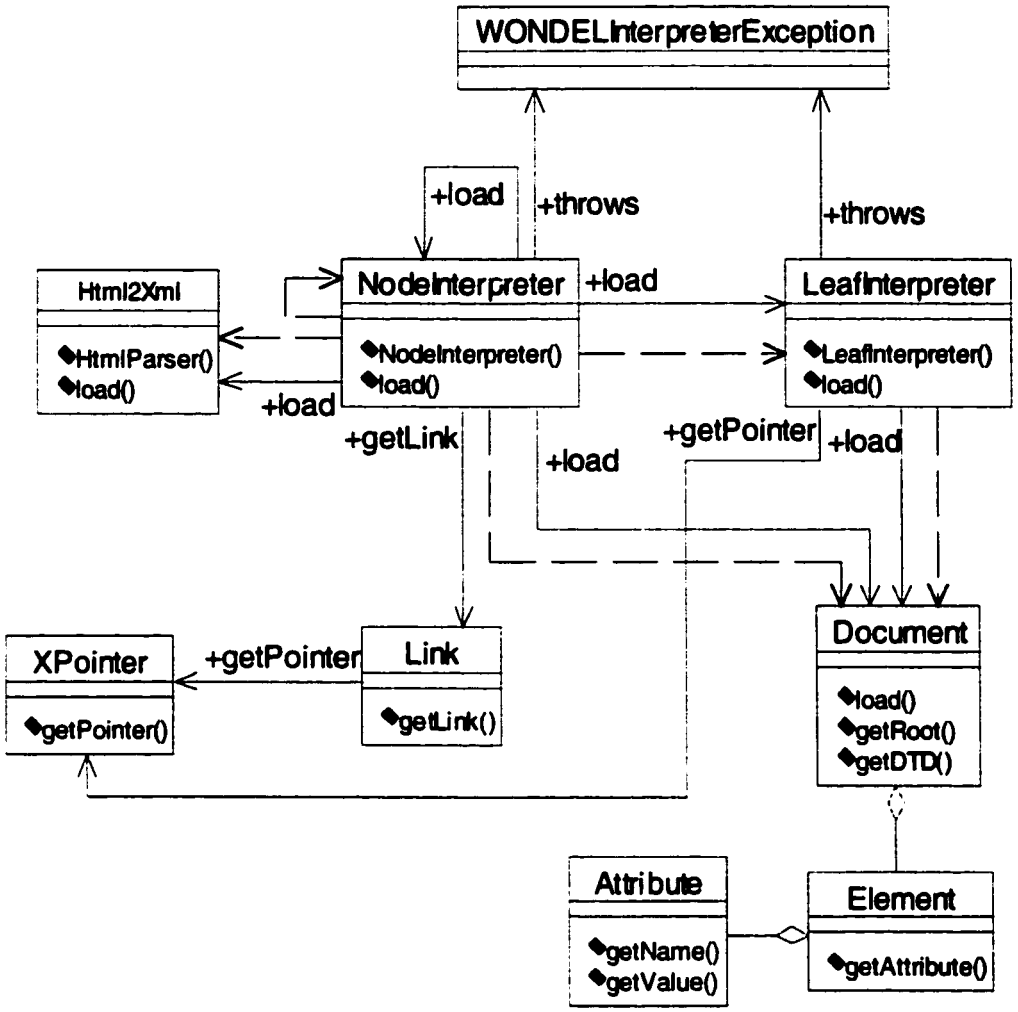


Figure 5.3: The Class Diagram of the WONDEL Interpreter

The WONDEL Interpreter package consists of two main classes: `LeafInterpreter` and `NodeInterpreter` which are used to interpret WONDEL leaf and node documents respectively.

The method `load()` of the class `LeafInterpreter` accepts a WONDEL leaf document and a well-formed XML document as input. This method interprets the given WONDEL leaf to extract information from the given well-formed XML document.

The class `NodeInterpreter` has also a method `load()` which has one parameter, a WONDEL node document. This method interprets the node document and generates an XML document with data collected from other files.

The classes `Link` and `XPointer` are used to interpret the XPointers provided in WONDEL documents. The method `getLink()` of the class `Link` extracts the root element of the XML document given in its input. This root is passed to the method `getPointer()` of the class `XPointer` along with an XPointer that is interpreted in the context of the given root.

In what follows, a typical scenario of the behavior of the WONDEL Interpreter classes is described. In this scenario, the method `load()` of the class `NodeInterpreter` is invoked to interpret a node document whose call part contains a node call followed by a leaf call. This scenario is illustrated by the interaction diagram provided in Figure 5.4.

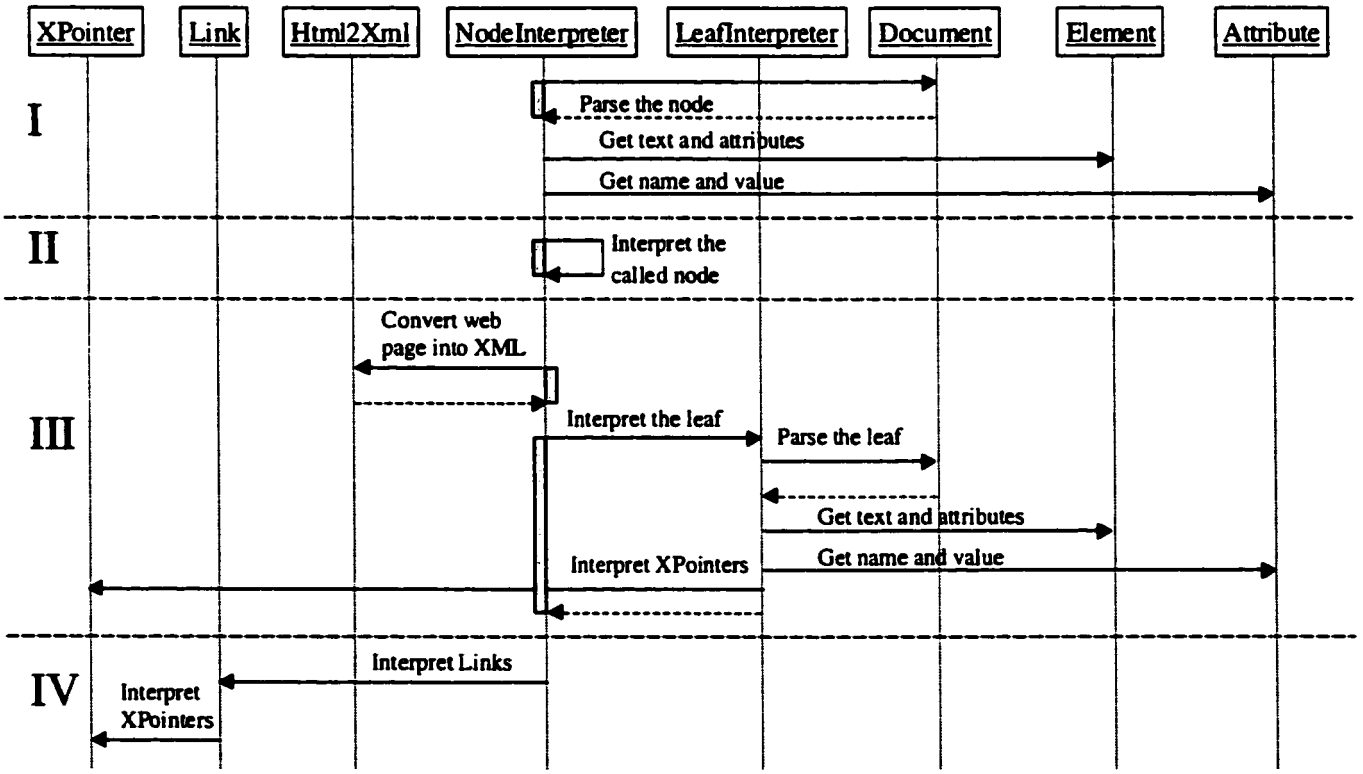


Figure 5.4: A typical Scenario of the WONDEL Interpreter Objects Behavior

The interaction diagram above is divided into four phases. In phase I, the classes `Document`, `Element` and `Attribute` are invoked to parse the given node and extract its different elements with their attributes. In phase II, a new instance of the class NodeInterpreter is created to interpret the WONDEL node document given in the call part. Phase II may include all the four phases shown in Figure 5.4 in a recursive way. Phase III includes the necessary steps to interpret a WONDEL leaf. In this phase, `NodeInterepter` first uses the `Html2Xml` class to convert the given Web page into a well-formed XML document and then invokes `LeafInterpreter` to which it passes the called WONDEL leaf and the generated well-formed XML document. `LeafInterpreter` parses the WONDEL leaf and uses the XML Parser classes to extract the XML elements with their attributes. It also interprets the XPointers contained in the WONDEL leaf using the class `XPointer`. Thereafter, it extracts the desired information from the given XML document and returns the control back to `NodeInterpreter`. At this point, `NodeInterpreter` moves into phase IV in which it interprets the collect part to marshal the information generated previously. The class `Link` is used in this phase to interpret the different links contained in the collect part.

5.1.3 The XPointer Interpreter Package

The class diagram of the XPointer Interpreter is given in Figure 5.5.

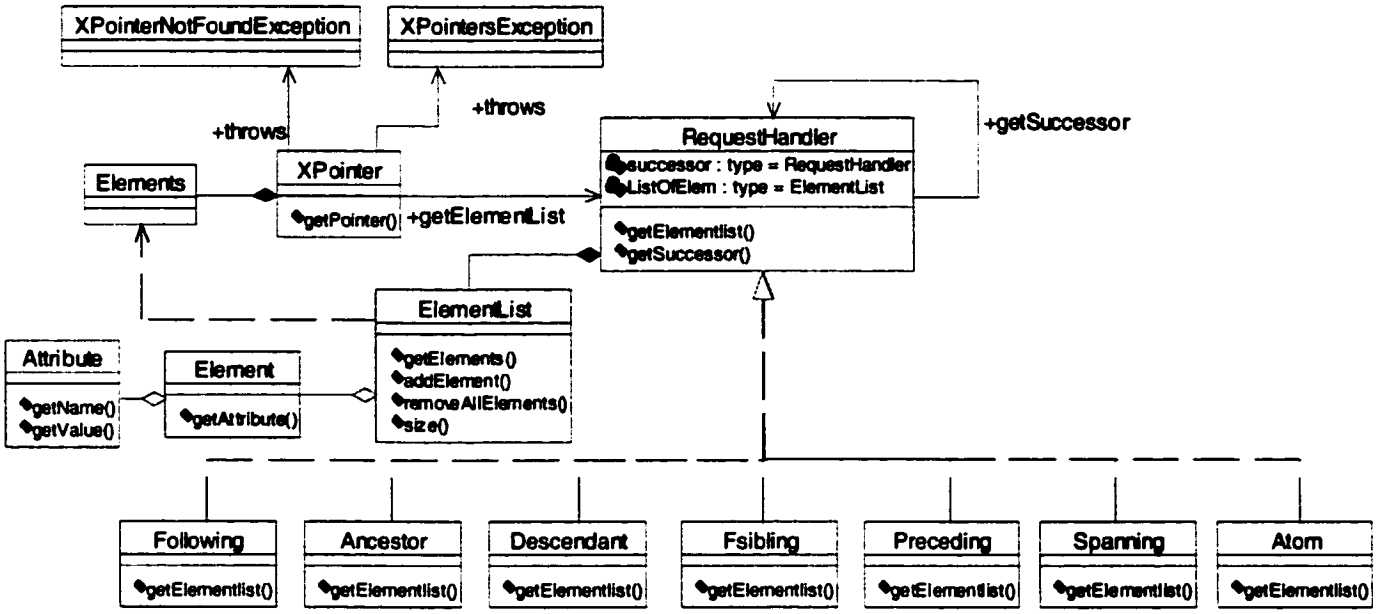


Figure 5.5: The Class Diagram of the XPointer Interpreter

The main class of the XPointer interpreter package is `XPointer`. This class has a method called `getPointer()` whose input is an XML element and a character string that represents an XPointer. When interpreting a location term of the given XPointer, this method invokes the method `getElemenList()` of the class that corresponds to the keyword of that location term. For instance, if the keyword of the location term is `FOLLOWING` then the method `getElementList()` of the class `Following` is invoked. The interaction between the class `XPointer` and the classes corresponding to the keywords follows a behavioral design pattern called Chain of Responsibility pattern [27]. To illustrate how this pattern works, assume that the XPointer interpreter finds a location term whose keyword is `DESCENDANT`. This keyword corresponds to the third class in the chain of classes above. In this case, the behavior of the interpreter is described by the interaction diagram of Figure 5.6.

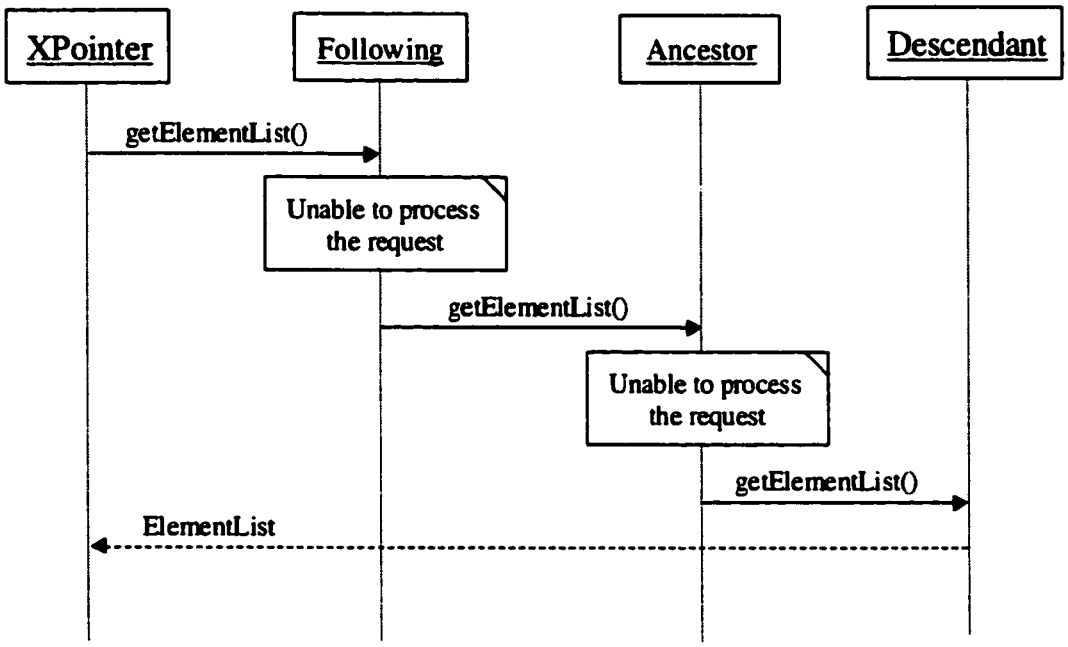


Figure 5.6: The Chain of Responsibility Pattern

The way the Chain of Responsibility pattern works is that when many objects are candidates to handle a request, they are put in a queue called the chain of responsibility. Each object has a pointer to its successor. The object that emits the request should send it to the first object in the chain. If the first object is not able to handle the request, it simply passes it to its successor. This scenario is repeated until an object is able to handle the request. In the example of Figure 5.6, the request can be handled by the object `Descendant`; this is why it has to pass through the objects `Following` and `Ancestor` since they precede the object `Descendant` in the chain of responsibility.

The major advantage of the Chain of Responsibility pattern is that other request handlers can be added without affecting the class that emits the request or the other request handlers. This provides a very high level of maintainability when adding more features to the system. In the case of the XPointer Interpreter, other keywords can be added without any changes to the existing classes. However, this pattern requires that an instance of each request handler must be present even if it's not used, which may

have a negative effect on the performance of the system. Nevertheless, this approach is acceptable in our case because the number of the request handlers is relatively small.

5.1.4 The Database Adapter Package

The last package presented in this section is the Database Adapter package. Its class diagram is shown in Figure 5.7.

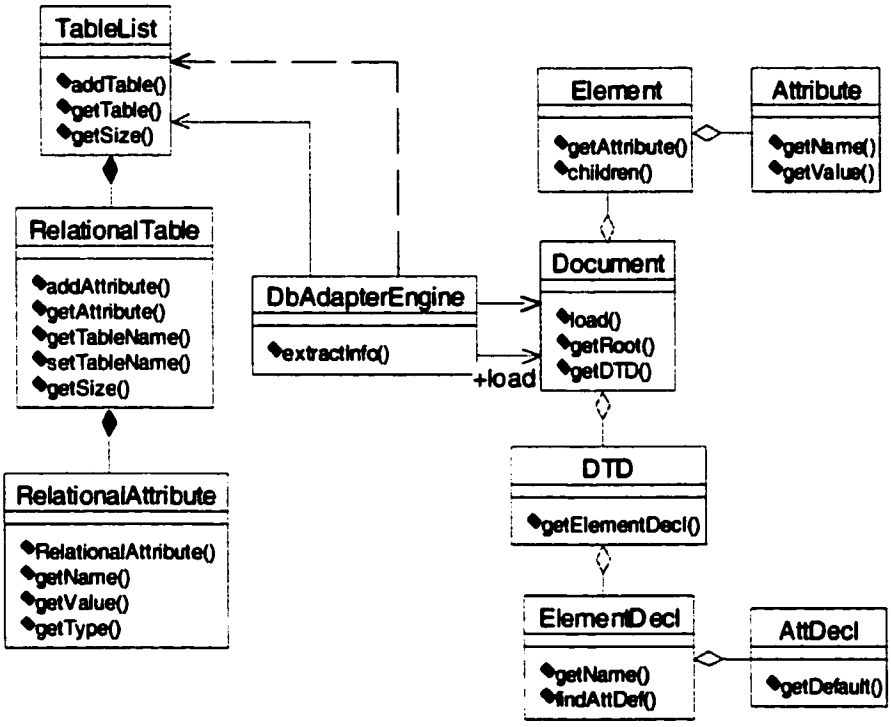


Figure 5.7: The class diagram of the Database Adapter package

`DbAdapterEngine`, the main class of the package, has a method called `extractInfo()` that maps the valid XML document given in its input into relational tables. The XML documents are parsed using the Microsoft XML parser. The method `extractInfo()` uses three other classes of the XML parser that represent the DTD and its element and attribute declarations. This is necessary to extract the attribute ROLE of the different XML elements that is stored only in the DTD as explained in

section 3.4. The output of the method `extractInfo()` is a list of relational tables whose format is implemented by the classes `TableList`, `RelationalTable`, and `RelationalAttribute`.

5.2 Applying the XWA System to Populate the GUD Database

The XML-based Web Agent System is a part of the Global University Database (GUD) project whose objective is to develop an agent to extract specific information from university Web sites and to populate a relational database [23]. This information concerns faculties, departments, courses, faculty members, and research groups and projects in each given university as well as the patents owned by the universities. This information is stored in a database, called the Global University Database (GUD), whose Entity/Relationship diagram is shown in Figure 5.8. The GUD database model is also discussed in [38] and [41].

This section reports the application of the XML-based Web Agent to university Web sites. The whole process is achieved in two phases: the data preparation phase and the data processing phase.

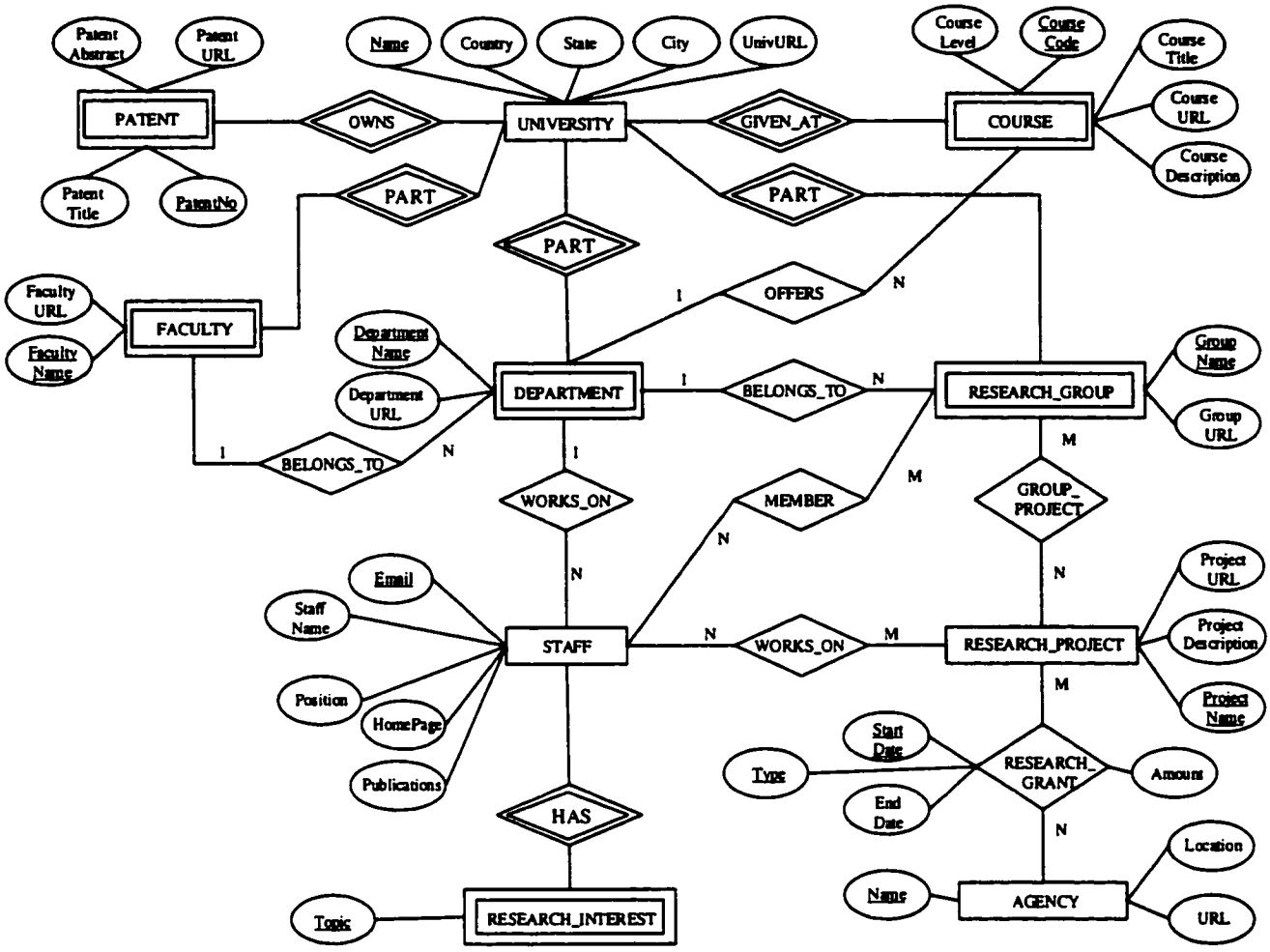


Figure 5.8: The Entity-Relationship Diagram of the Global University Database

5.2.1 The Data Preparation Phase

The data preparation phase is the only phase that requires human intervention. Given the database whose entity-relationship diagram is shown in Figure 5.8, the corresponding DTD (Document Type Declarations) should be written as prescribed in section 3.4. That DTD is provided in Appendix C. Thereafter, an ontology for the given university Web site is built in a bottom-up fashion. Each Ontology needs to be seeded by a certain number of Web pages that are discovered manually. The

XWA system uses this initial set of Web pages to find other pages from which it extracts data. Figure 5.9 shows a part of the ontology that has been created for the University of Ottawa's Web site.

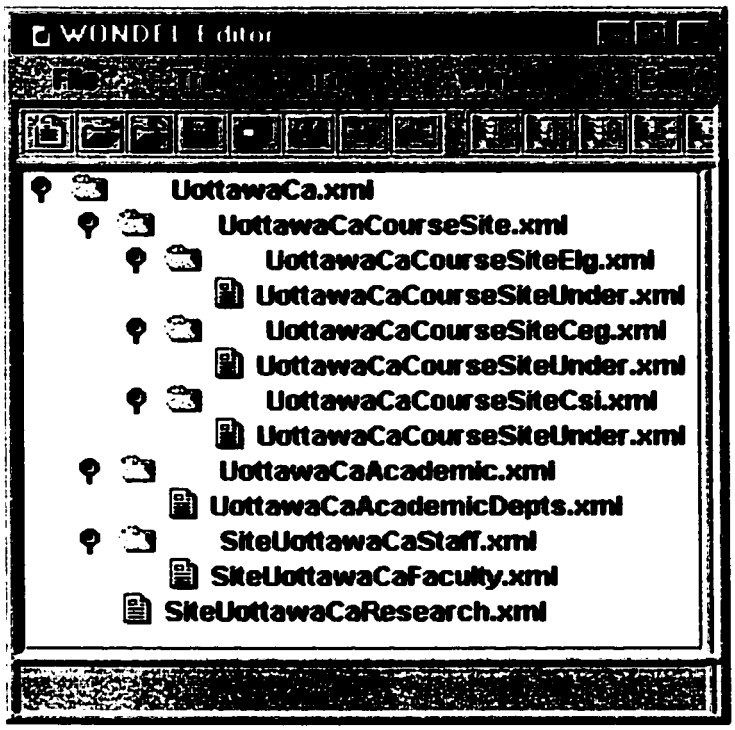


Figure 5.9: An Ontology for the University of Ottawa Web Site

In Figure 5.9, WONDEL leaves are labeled by sheet icons, whereas the WONDEL nodes are labeled by folder icons. The root of this ontology is the file "UottawaCa.xml" which invokes three WONDEL nodes and one leaf. The node "UottawaCaCourseSite.xml" collects information about the undergraduate courses given at SITE (School of Information Technology and Engineering). The node "UottawaCaAcademic.xml" collects information about the faculties and the departments of the university. The node "SiteUottawaCaStaff.xml" gathers information concerning the faculty members of SITE. Finally, the leaf "SiteUottawaCaResearch.xml" extracts information about the research groups within SITE. In the sub-tree whose root is "UottawaCaCourseSite.xml", there are three nodes that collect information about the courses provided by the electrical engineering, computer engineering, and computer science departments respectively. Note that these three nodes reuse the same WONDEL leaf,

which is "UottawaCaCourseSiteUnder.xml", to extract information from the related Web pages. Note also that the leaf "SiteUottawaCaFaculty.xml" extracts information not only from the Web page that contains the list of the faculty members but also from their homepages. In summary, the ontology above, which consists of 11 WONDEL documents, extracts information from 458 Web pages, 10 of them were discovered manually.

5.2.2 The Data Processing Phase

The XML-based Web Agent populates relational databases in two steps. It first extracts data from the Web and puts it into valid XML documents, and then maps these documents to relational tables that can be put directly into relational databases.

The data is extracted by interpreting each WONDEL document in the Web Site Ontology starting from the root to its descendants in a depth-first search. The result is a valid XML document that contains all the extracted data formatted according to the DTD built up in the data preparation phase.

The valid XML document is then mapped into relational tables by the Database Adapter using the algorithm explained in Section 3.4.

5.3 WONDEL Builder

Along with this thesis, a system to build WONDEL-based Web Site Ontlogies has been developed. This system, called the WONDEL Builder, provides the following functionality:

1. An editor for WONDEL documents.

2. A graphical viewer for Web Site Ontologies

3. A debugger to test WONDEL documents within the context of their corresponding ontologies.

A snapshot of the WONDEL builder is provided in Figure 5.10.

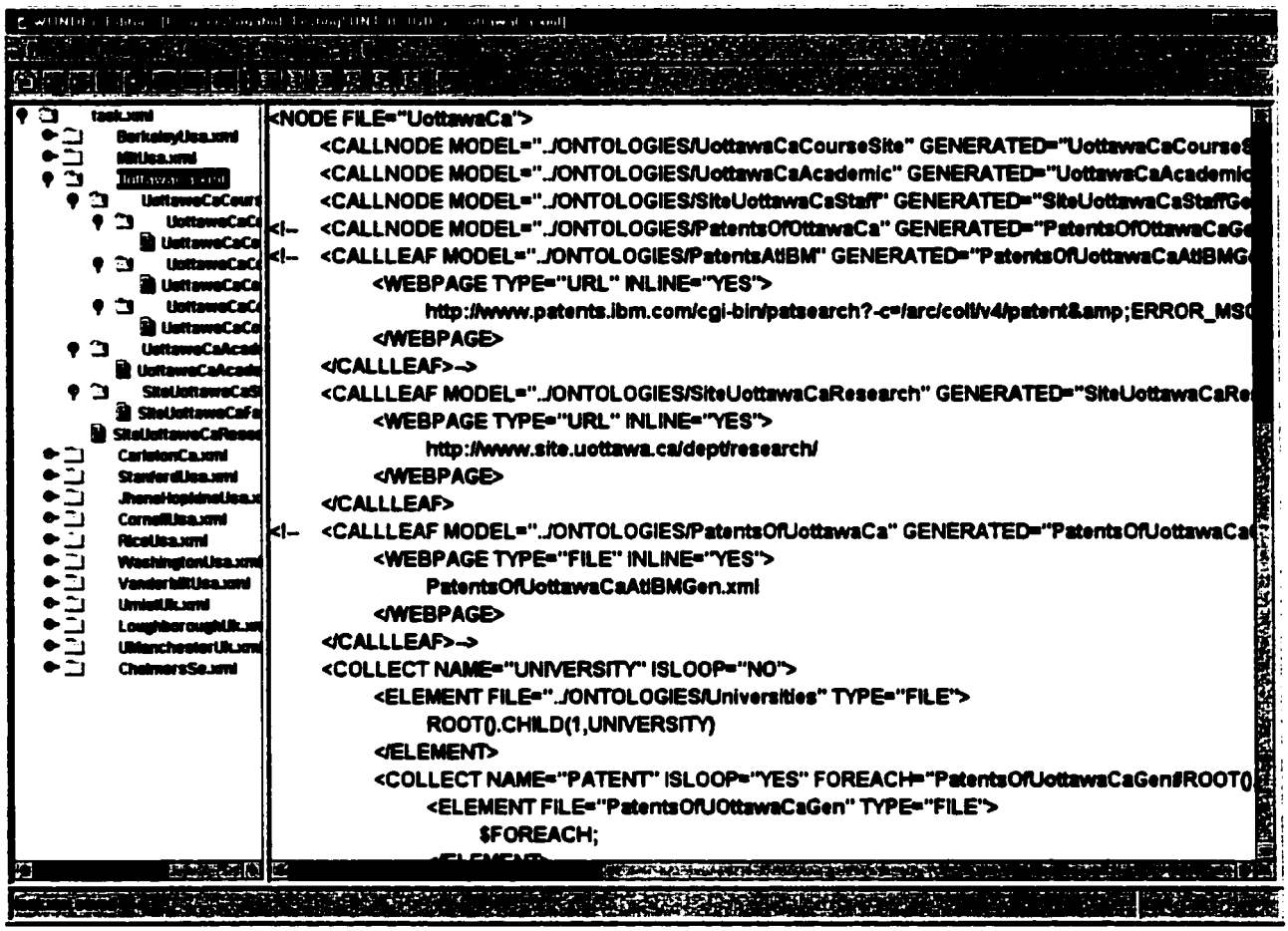


Figure 5.10: A Snapshot of the WONDEL Builder

The left frame of the WONDEL Builder displays the tree of the WONDEL documents that form each ontology. By clicking on the icon of a WONDEL document, it is edited on the right frame.

Each document of the Ontology can be interpreted separately by selecting the corresponding icon from the tree and then pressing the button with the triangle image (the most right in the button bar).

If the selected icon refers to a WONDEL leaf, then the dialog box shown in Figure 5.11 appears and prompts the user for the filename of the document to be generated and the location of the Web page from which information will be extracted. When the "OK" button is pressed, the Web page is downloaded, converted into a well-formed XML document, and then the WONDEL leaf is interpreted in order to extract data from it.

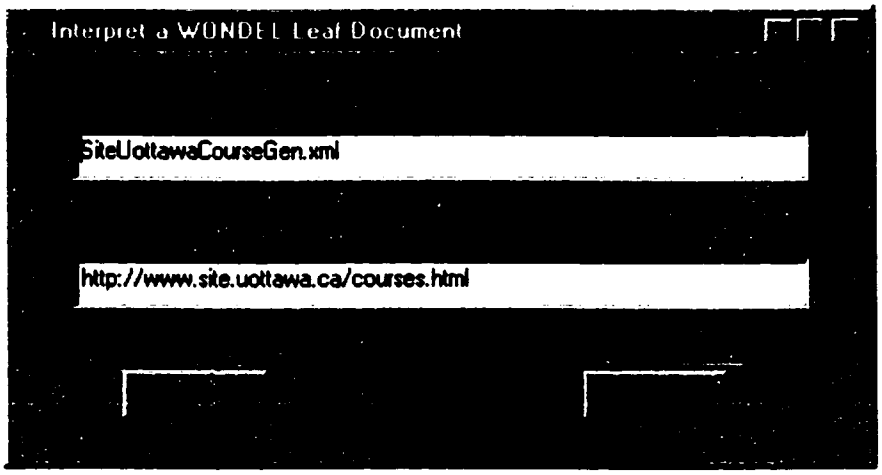


Figure 5.11: Interpreting a WONDEL Leaf Document

Figure 5.12 shows the dialog box that appears when the user requests interpreting a WONDEL node.

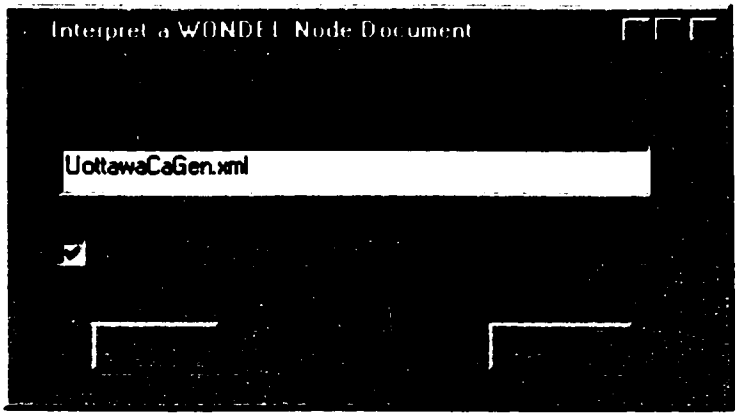


Figure 5.12: Interpreting a WONDEL Node Document

By default, interpreting a WONDEL node will trigger the interpretation of all its descendant documents, whether they are nodes or leaves. However, an option is provided to skip the interpretation of the WONDEL leaves to avoid downloading the same Web pages multiple times.

5.4 Experimental Results

5.4.1 Evaluation of the accuracy

The purpose of this section is to evaluate the accuracy of the data extracted using WONDEL. The evaluation of the accuracy does not apply to WONDEL nodes since data is extracted from Web pages using WONDEL leaves only. Nodes are only used to collect and organize data as explained in Section 4.2.

The accuracy of data extracted from a certain Web page varies with the size of the WONDEL leaf used in the extraction: the more information the WONDEL leaf has, the better is the accuracy. Therefore, a proper evaluation should take that into account. The purpose of the experimentation described in this section is to analyze how the accuracy evolves with the ratio of the size of WONDEL leaves to the size of the valid data.

Preliminary: the evaluation will be performed under the following conditions and assumptions:

1. In what follows, the term "reference data" refers to a data set that contains all and only the valid data in the Web pages selected for this experimentation.
2. The data sets considered in this experimentation are tables of records with the same field types. Each table has a field type as a primary key used to uniquely identify each record in the table.
3. The term "reference field" refers to a field in the reference data, and the term "extracted field" refers to a field in the extracted data.

4. We say that a reference field corresponds to an extracted field if they have the same type and they belong to records with the same primary keys.

5. The evaluation will be applied to 10 Web pages selected from the Web sites listed Table 5.1.

Web site	URI of the downloaded Web page	Target information
EBay (online auctions)	http://search.ebay.com/search/search.dll?MfcISAPICommand=GetResult&query=Computer&ebaytag1=ebayreg&ht=1&SortProperty=MetaEndSort&sc=1&st=0	Computer-related products available for bidding.
Amazon.com (online bookstore)	http://www.amazon.com/exec/obidos/search-handle-url/index=books&field-keywords=%20%20books&bq=1/ref=aps_more_b_1/102-7475223-1753746	Computer-related books
BigCharts.com (Stock quotes)	http://www.bigcharts.com/quickchart/quickchart.asp?symb=systems&sid=0&o_symb=systems	Stock quotes of systems-related companies
Google (Search engine)	http://www.google.com/search?q=agents&num=100&site=search&hl=en&lr=&safe=off&output=search	Agent-related Web pages
Delphion (patents)	http://www.delphion.com/cgi-bin/patsearch?-c=lblistings&-l=lbquery&Field1_Type=RAW&Field1_Text=IBM&-m=500	Patents owned by IBM
University of Ottawa	http://www.site.uottawa.ca/school/people/faculty.html	Professors at S.I.T.E
M.I.T.	http://www.ai.mit.edu/contact/people/	Member of Artificial Intelligence Lab
University of California at Berkeley	http://www.cs.berkeley.edu/Research/Projects/	Research projects at department of computer science
University of Waterloo	http://www.adm.uwaterloo.ca/infoucal/0001/COURSE/course-E_and_CE.html	Courses given at department of electrical engineering
Carleton University	http://www.sce.carleton.ca/faculty/	Professors at department of systems and computer engineering

Table 5.1: List of Web Pages Used in the Accuracy Evaluation

The reason for choosing such diversified Web sites is to show that WONDEL can be successfully applied to domains other than universities.

6. ***Precision*** and ***recall*** are the most common factors considered when evaluating the accuracy of retrieved data [42]. In this study, we will evaluate the variation of those two factors with the ratio of WONDEL leaf size to the size of valid data. Both Precision and Recall will be defined later in this section.

Methodology: Below are the steps followed in this experimentation:

1. For each Web page listed in Table 5.1:
   1.1. Define a model of data to be extracted.
   1.2. Extract all valid data from the Web page. This data has to be 100% accurate as it will be the reference on which the evaluation is based.
   1.3. Write three versions of WONDEL leaf to be applied to the Web page. The first version should be simple and straightforward. The subsequent versions should include more information to increase the accuracy of the extracted data
   1.4. Calculate the ***Precision*** and the ***Recall*** using the formulas below for the three datasets extracted using each WONDEL leaf version.
2. Draw graphs for the following two functions:

$$\underline{\text{Precision}} = F(100 \times (\Sigma S_{leaf})/(\Sigma S_{data}))$$

$$\underline{\text{Recall}} = G(100 \times (\Sigma S_{leaf})/(\Sigma S_{data}))$$

Where:

- Precision: average of the precision over all Web pages.

- Recall: average of the recall over all Web pages.
- ΣS_{leaf}: sum of the sizes of WONDEL leaves.
- ΣS_{data}: sum of the sizes of the valid data extracted in step 1.2 from all Web pages.

At each iteration, a different combination of WONDEL leaf versions will be applied to the Web pages. The initial combination includes the first version of WONDEL leaves for all Web pages. The second combination includes the second WONDEL leaf version for the first Web page, and the first version for the subsequent Web pages and so on. That way the sum of WONDEL leaf sizes will grow with each combination.

Definition of the precision and the recall: Scott Weiss defines Precision and Recall in [42] as follows:

Precision: the number of relevant data units retrieved divided by the total number of data units retrieved.

Recall: the number of relevant data units retrieved divided by the total number of relevant data units in the collection.

Relevance: an abstract measure of how well a data unit satisfies the user's information need. This is a subjective notion difficult to quantify.

Based on those definitions and given the particularities of the data sets involved in this experimentation, the precision and the recall of the data extracted using a WONDEL leaf are defined as follows:

Precision:

$$P_{data} = \Sigma R_{XY}/N_e$$

Where:

P_{data}: precision of the extracted data.

N_e: total number of extracted fields.

R_{XY}: relevance of the extracted field located at row X and column Y. It is defined as follows:

- $R_{XY} = 1$ if there is an exact match between the extracted field and the corresponding valid field.
- $R_{XY} = 0$ if the content of the corresponding valid field is not a sub-string of the content of the extracted field.
- $R_{XY} = \min(1, \max(0, 1 - (S_e - S_v)/k))$ if the content of the corresponding valid field is a sub-string of the content of the extracted field

Where:

S_e: number of characters in the extracted field.

S_v: number of characters in the corresponding valid field.

k ($1 \leq k \leq S_v$): a constant that reflects the user's tolerance towards data impurity. Its value depends on the nature of each field type and the application in which it is used.

If k is equal to its maximum value S_v, the relevance starts decreasing with each extra bad character and will reach 0 when the size of the target field is twice the size of the reference field. On the other hand, when k is equal to its minimum value 1, the accuracy is equal to 0 once the target field contains at least one bad character. The value of the tolerance factor k depends on the type of each field. For example, that factor can be high for field types in which a few bad characters are acceptable like "DESCRIPTION", and low for fields in which data impurity is not tolerable such as "URL" and "PRICE". Table 5.2 lists all the types of the fields involved in this experimentation and the value of the factor k that we chose for each field type.

Field Type	Tolerance Factor	Field Description
URL	1	URL of a Web page. Exact match is required to be significant.
NAME	$S_v/4$	Name of a person or an institution.
TITLE	$S_v/4$	Title of a book.
AUTHOR	$S_v/4$	Author of a book
PRICE	1	Price of a book.

DEPARTMENTNAME	$S_v/4$	Name of a department.
COMPANY	$S_v/4$	Name of a company
EXCHANGE	$S_v/2$	Stock exchange such as TSE or NASDAQ.
SYMBOL	1	Symbol of a company in a stock exchange. Exact match is required to be significant.
STAFFNAME	$S_v/4$	Name of a person (staff member).
HOMEPAGE	1	URL of a professor homepage. Exact match is required to be significant.
POSITION	$S_v/2$	Position of a faculty member
DEGREE	$S_v/2$	Degree of a faculty member
SIZE	1	Size of a Web page. Exact match is required to be significant.
CATEGORY	$S_v/2$	Category of a Web page
CATEGORY_URL	1	URL of the category. Exact match is required to be significant.
NUMBER	1	Patent number. Exact match is required to be significant.
DATE	1	Patent date. Exact match is required to be significant.
EMAIL	1	Email address. Exact match is required to be significant.
CODE	1	Course code. Exact match is required to be significant.
DESCRIPTION	S_v	Course description.
PRERIQUISITE	$S_v/4$	Course prerequisite.
OFFICE	1	Office number. Exact match is required to be significant.
BIDS	1	Number of bids. Exact match is required to be significant.
ENDS	1	Time when the bid ends. Exact match is required to be significant.
PHONE	1	Phone number. Exact match is required to be significant.

Table 5.2: Tolerance Factor for Each Field Type

Recall:

$$C_{data} = \sum R_{XY} / N_v$$

Where:

C_{data}: recall of the extracted data.

N_v: total number of fields in the valid data.

R_{XY}: relevance of the extracted field located at row X and column Y. The same as defined before.

The evaluation results:

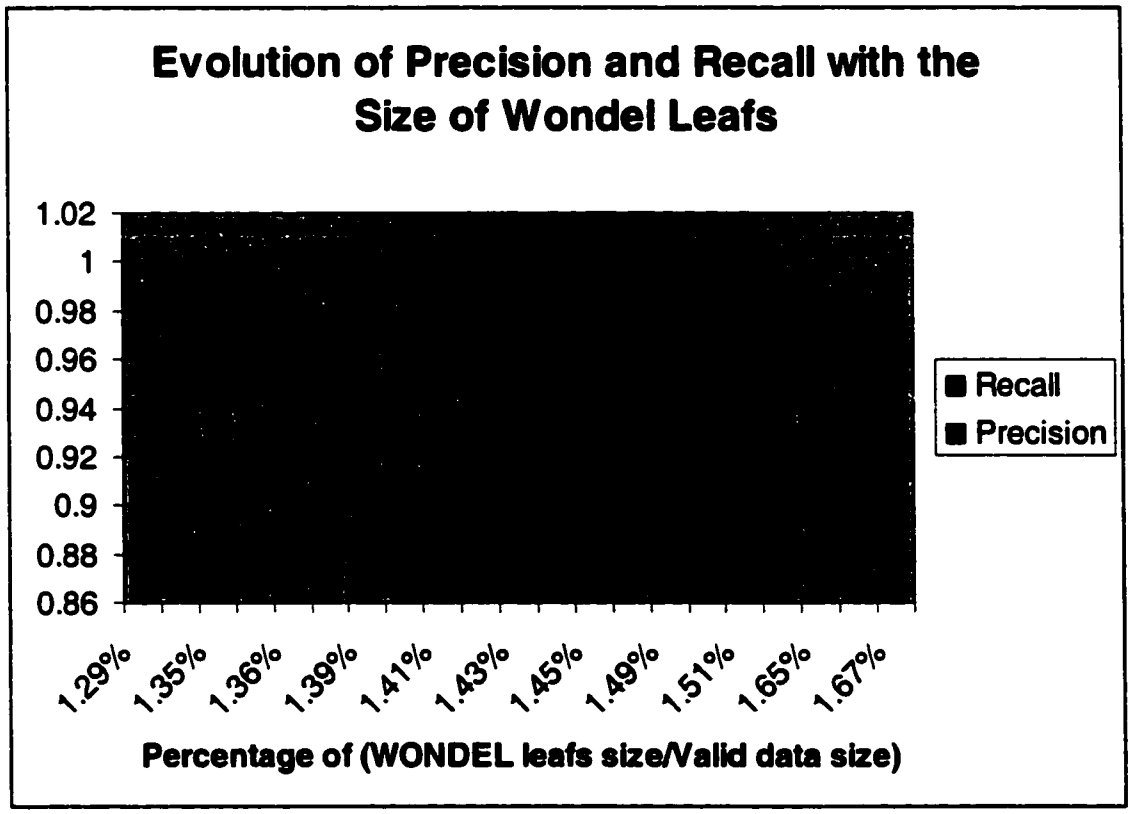


Figure 5.13: Evolution of the Precision and the Recall with the Size of WONDEL Leaves

Interpretation of the results:

This experimentation proves that with WONDEL we can extract accurate data with a reasonable amount of human effort. Indeed, both the precision and the recall of the extracted data were close to 0.92 when the total size WONDEL leaves was about 1.29 % the size of information we want to retrieve. They both were over 0.99 when the ratio was about 1.67 %. We can also notice there was a faster improvement in both the accuracy and the recall when the ratio has increased from 1.29 % to 1.41 % before it becomes more stable. One last remark is that although the recall was a bit higher than the precision, they were both increasing at the same pace. This proves that it is possible that the precision and the recall can be improved together with no conflict using WONDEL.

5.4.2 Applying WONDEL to University Web Sites

This section includes four tables that provide experimental results obtained from applying the XML-based Web Agent to 13 university Web sites. The intent here is to evaluate the human effort required to extract a certain amount of information from the Web. Table 5.3 provides numbers concerning the Web pages that were downloaded and processed for each university. It compares the number of the Web pages that were detected manually with those that were automatically found by the agent. Table 5.4 provides the size of the ontologies for each university Web site. In this table, we choose also to give the size of the WONDEL leaves and nodes so that the reader can see the amount of the code needed for the WONDEL leaves comparing to the WONDEL nodes. Table 5.5 presents the amount of information extracted from each university Web site. Finally, Table 5.6 provides a comparison between the sizes on the Ontologies and the extracted data.

Table 5.3 shows that 160 pages were discovered manually which represents approximately 2.85% of the total number of the Web pages processed by the system. However, this percentage may

vary considerably between universities. For example, it is equal to 1.02% for the Massachusetts Institute of Technology, whereas for Loughborough University, it is equal to 8.54%. This depends on the concentration of the desired information within the Web pages.

Universities	Web pages found manually	Web pages detected by the Agent	Total analyzed Web pages	% of (# of pages found manually / total # of analyzed pages)
University of California, Berkeley	8	342	350	2.29 %
Carleton University	6	128	134	4.48 %
Cornell University	7	270	277	2.53 %
Johns Hopkins University	8	242	250	3.2 %
Loughborough University	34	364	398	8.54 %
University of Manchester Institute of Science and Technology	15	219	234	6.41 %
The University of Manchester	25	329	354	7.06 %
Massachusetts Institute of Technology	16	1538	1554	1.03 %
University of Ottawa	10	448	458	2.18 %
Rice University	6	468	474	1.27 %
Stanford University	8	584	592	1.35 %
Vanderbilt University	8	124	132	6.06 %
University of Washington	9	390	399	2.26 %
Total for 13 universities	160	5446	5606	2.85 %

Table 5.3: The Web Pages processed by the XML-based Web Agent for 13 University Web Sites

The purpose of the three tables below is to evaluate the amount of WONDEL code required to extract a certain amount of data and structure it into relational tables. As shown in Table 5.6, the size of all WONDEL documents written to extract data from 13 universities represents 4.03% of the total size of the extracted data. Again, the numbers vary between universities: the percentage for the Massachusetts Institute of Technology is 1.33%, whereas for Loughborough University, it is equal to 12.32%. This percentage depends mainly on the regularity of the format of the Web pages.

Universities	The Size of WONDEL Leafs	The Size of WONDEL Nodes	The Size of the Ontology
University of California, Berkeley	4762	5203	9965
Carleton University	4165	4465	8630
Cornell University	3952	3885	7837
Johns Hopkins University	3410	4085	7495
Loughborough University	24384	17566	41950
University of Manchester Institute of Science and Technology	11823	7892	19715
The University of Manchester	16289	12089	28378
Massachusetts Institute of Technology	13797	9509	23306
University of Ottawa	4924	6786	11710
Rice University	3256	3128	6384
Stanford University	5273	4912	10185
Vanderbilt University	4954	4240	9194
University of Washington	4822	4636	9458
Total for 13 universities	105811	88396	194207

Table 5.4: The Size of the Ontologies Created for 13 University Web Sites

Universities	Number of Records	Number of Fields	The total Size of Extracted Data (Bytes)
University of California, Berkeley	543	2101	256009
Carleton University	270	976	122030
Cornell University	312	1419	162283
Johns Hopkins University	291	1114	129117
Loughborough University	723	2869	340441
University of Manchester Institute of Science and Technology	280	1331	162309
The University of Manchester	800	2792	336408
Massachusetts Institute of Technology	3203	14289	1750899
University of Ottawa	591	3179	416891
Rice university	606	2419	276579
Stanford University	768	3464	502614
Vanderbilt University	161	675	80517
University of Washington	484	2466	280699
Total for 13 universities	9032	39094	4816796

Table 5.5: The Amount of Information Extracted from 13 University Web Sites

Universities	The Size of the Ontology	The total Size of Extracted Data (Bytes)	The % of (the size of the ontology / the size of extracted data)
University of California, Berkeley	9965	256009	3.89 %
Carleton University	8630	122030	7.07 %
Cornell University	7837	162283	4.82 %
Johns Hopkins University	7495	129117	5.8 %
Loughborough University	41950	340441	12.32 %
University of Manchester Institute of Science and Technology	19715	162309	12.14 %
The University of Manchester	28378	336408	8.43 %
Massachusetts Institute of Technology	23306	1750899	1.33 %
University of Ottawa	11710	416891	2.8 %
Rice university	6384	276579	2.3 %
Stanford University	10185	502614	2.02 %
Vanderbilt University	9194	80517	11.41 %
University of Washington	9458	280699	3.36 %
Total for 13 universities	194207	4816796	4.03 %

Table 5.6: Comparison between the size of the Ontologies against the size of the extracted data

Applying the XML-based Web Agent to extract information from university Web sites has revealed some limitations that are discussed in the next section.

5.5 Limitations

The fast growth of the Web and its disorganized and ever-changing nature makes difficult the realization of a system that is able to extract information from the Web with thorough precision, and yet, without human intervention. Likewise, the XML-based Web Agent developed within this thesis has limitations that are summarized as follows:

- The agent requires human intervention to build Web Site Ontologies needed to extract information from the Web.

- The system has to be seeded with a set of Web pages locations to be able to discover other pages automatically.
- Web Site Ontologies need to be continuously maintained and tested due to the frequent changes in the Web sites. For this approach to be successful, there must be a certain level of discipline from those who update the Web pages so that the corresponding WONDEL document will remain valid. This is usually not the case.
- The amount of the WONDEL code needed to build an ontology depends on the existing regularity within Web sites. Applying this approach to Web sites without regularity may require an amount of WONDEL code which is unacceptably large and difficult to maintain. University Web sites are a typical example of irregularity since they most likely consist of a grouping of smaller Web sites owned by different departments within universities and designed in totally different ways. The situation is even worse when trying to extract information from professors' personal homepages.
- The proposed approach may eventually make wrong interpretations due to the lack of uniformity in Web sites.
- Manually building ontologies for large and complex Web sites can be a tremendous task for human users. Moreover, it is very time consuming to fix human errors in manually written WONDEL documents because it requires long hours to repeatedly download and process Web pages whose number can reach as many as hundreds per each Web site. For the thirteen university Web sites to which this system was applied, we were obliged to perform tests for several weeks, during the nighttime hours in order to have sufficient network bandwidth.

Chapter 6

Conclusions

6.1 Achievements

This thesis has proposed a mechanism to retrieve information from the World Wide Web and populate a relational database. The problem was approached in three steps. First, the difficulties in processing HTML-based Web pages were addressed. Second, the notion of Web Site Ontology was introduced to represent the knowledge required to extract specific information from the Web pages and marshal it into appropriately formatted documents. Third, an algorithm that converts those documents into relational tables was defined. As proof of the concept, a system, called the XML-based Web Agent, that implements the proposed mechanism, has been developed.

XML is extensively used and strongly present in the different phases of the process. The Web pages are converted into well-formed XML documents before any further processing. Web Site Ontologies are built using WONDEL, a language based on XML and XML Pointer Language. The

information extracted from the Web is stored in non-valid XML documents, from which it is then collected to form valid XML documents. Finally, the algorithm that converts valid XML documents into relational tables, uses the DTD (Document Type Declarations) which is also provided by XML, to define constraints on the documents' structure.

The most important aspect of the proposed approach is in WONDEL. This language provides a means for users to specify the information they want to retrieve from the Web and let the XWA system run offline to extract and store the information into a relational database. This saves users the time they usually spend browsing the Web for data hunting. Furthermore, WONDEL provides a certain level of modularity to build Web Site Ontologies so that they can be fairly easy to maintain and reuse. Adopting XML and XML Pointer Language to define WONDEL has had a decisive impact on the whole system. XML is exploited not only for its ability to describe the meaning of the desired Web pages, so that they can be understood and extracted by application, but also for its aptitude to organize the extracted information into well-defined structures. The extracted information is selected using XML Pointer Language (XPointer) which provides a beautiful way to address the internal structure of Web documents. The benefit gained from the use of the XPointer language led us to fully implement its interpreter and propose new extensions that makes it more powerful and adaptable for information retrieval on the World Wide Web. To facilitate the creation of Web Site Ontologies, a WONDEL Builder has been developed to edit WONDEL documents and to debug and test WONDEL-based Web site ontologies.

6.2 Suggestions for Future Work

The limitations of this thesis and its implementation discussed in section 5.5 can be divided into two categories. There are those that relate generally to the Web paradigm, whereas some others are due

to time and resource constraints. The second category of limitations can still be investigated and would make interesting extensions to this thesis research:

- The existing uniformity within Web pages would offer a good opportunity to use machine learning techniques to create WONDEL leaf documents. However, machine learning can lead to wrong interpretations, especially if Web pages are not perfectly uniform, which is frequently the case. A way to overcome this inadequacy is to develop an interactive user interface that combines machine learning and human assistance.
- WONDEL node documents reflect the organizational aspect of Web Site Ontologies, which can be represented visually, as the WONDEL Builder does. Furthermore, WONDEL nodes can also be created visually by dragging-and-dropping icons onto an interactive canvas and joining the icons with arrows that symbolize WONDEL leaf and node calls.
- The URIs, as well as the titles of Web pages, usually contain words that provide valuable insights into the page content. For example, the URI and the title of the Web page that contains the list of faculties and departments in the University of Ottawa are respectively: http://www.uottawa.ca/academic/depts.html and "U of O: Faculties and Departments"[5]. The words "academic", "depts", "Faculties" and "Departments" could help guess that the given Web page has something to do with academic faculties and departments. It would be interesting to develop a tool that, given the URI of a Web site main page and some keywords, traverses the Web site and detects the Web pages whose URIs and/or titles contain one or many of the given keywords. It would then draw a map with the discovered Web pages and their paths, starting from the main page. This could help users find the URIs of the Web pages containing the information they are interested in.

[5] At least this was true at the time of writing this thesis

- Other features can be added to the WONDEL Builder. For example, one feature which can save a lot of testing time, is to check whether the valid XML document to be generated by a given Web Site Ontology does effectively conform to its corresponding DTD.

References

1. AltaVista, http://www.altavista.com

2. Berners-Lee T., "Realising the Full Potential of the Web", Based on a seminar given the W3C meeting, London 1997, available at http://www.w3.org/1998/02/Potential.html

3. Berners-Lee T., "W3 Concepts", Based on a talk presented at the W3C meeting, 1991, available at http://www.w3.org/Talks/General/Concepts.html

4. Booch, G., Jacobson, I., and Rumbaugh, J., "The UML specification", *Rational Software Corp.*, 1997. Available at http://www.rational.com

5. Bradshaw, J. M. (Editor), "Software Agents", *Menlo Park, Calif.:AAAI Press*, 1997.

6. Bray T., "Beyond HTML: XML and Automated Web Processing", *Netscape Communications Corporation*, 1998.

7. Bray T., Paoli J., and Sperberg-McQueen C. M., "Extensible Markup Language", *W3 Consortium*, August 7, 1997.

8. Deogun J. S., Sever H., Raghavan V. V., "Structural Abstractions of Hypertext Documents for Web-based Retrieval". 1997.
Online: http://www.cacs.usl.edu/Departments/CACS/Publications/Raghavan/DSR98.ps.Z

9. Doorenbos, R. B., Etzioni, O., and Weld, D. S. "A scalable comparison-shopping agent for the World Wide Web". *In proceedings of the ACM First International Conference on Autonomous Agents*, Marina Del Rey, CA, 1997, pp 39-48.

10. Edwards M., "XML: Data the Way You Want It", *Microsoft Corporation*, October 31, 1997.

11. El Masri, R., and Navathe, S. B. "Fundamentals of Database Systems", *The Bejamin/Cummings Publishing Company, Inc*, Redwood City, Calif. 1989.

12. Frakes W.B. and Baeza-Yates, eds., "Information Retrieval: Data Structures and Algorithms", *Prentice Hall, Englewood Cliffs*, N.J., 1992

13. Franklin S., Graesser A., "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", *In Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages. Springer-Verlag*, 1996.
Online: http://www.msci.memphis.edu/~franklin/AgentProg.html

14. From an advertisement of Lucent Technologies in *The Final Program Brochure of the IEEE International Conference on Communications*, Vancouver, June 1999, pp 19.

15. Gamma, E., Helm, R., Johnson, R., and Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison Wesley*, Reading, Massachusetts, 1995

16. Garshol L. M., "Introduction to XML".
Online: http://www.stud.ifi.uio.no/~lmariusg/download/xml/xml_eng.html

17. Gruber, T., "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", *International Workshop on Formal Ontoilogy*, March 1993, Pavoda, Italy. Available as Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University.

18. Guarino N., "Understanding, Building, and Using Ontologies", *LADSEB-CNR, National Research Council, Italy.*

19. Gudivada V. N., Raghavam V. V., Grosky W. I., Kasanagottu R., "Information retrieval on the World Wide Web", *IEEE Internet Computing*, Volume 1, N. 5, September-October 1997, pp 58-68.

20. Heflin J., Hendler J., and Luke S., "Reading Between the Lines: Using SHOE to Discover Implicit Knowledge from the Web". *Department of Computer Science, University of Maryland.* 1998.
Online: http://www.cs.umd.edu/projects/plus/SHOE/pubs/shoe-aaai98.ps

21. InfoSeek, http://www.infoseek.com

22. Ion P., and Miner R., "Mathematical Markup Language", *W3 Consortium*, February 24, 1998. Available online at http://www.w3.org/TR/PR-math/

23. Karmouch, A., "Development of a Global University Database", Project Proposal, SITE, University of Ottawa, 1997.

24. Laresgoiti I., Anjewierden A., Bernaras A., J. Corera J., Schreiber A. TH., Wielinga B.J., Ontologies as Vehicles for Reuse: a mini-experiment, LABEIN, Parque Technologico, Ed. 101, 48016 – Zamudio Bizkaia, Spain, 1998

25. Lassila O., "Web Metadata: A Matter of Semantics", *IEEE Computer*, Volume 2, N. 4, July-August, 1998, pp 30-37.

26. Lassila O., Swick R. R., "Ressource Description Framework (RDF) Model and Syntax Specification", *W3C Recommendation*, February 1999.
Online: http://ww.w3.org/TR/REC-rdf-syntax

27. Luke S., Spector L., Rager D., and Hendler J., "Ontology-based Web Agents", *In Proceedings of ACM Symposium on Autonomous Agents*, 1997.
Online: http://www.cs.umd.edu/projects/plus/SHOE/pubs/aa-paper.ps

28. Maes, P., "Artificial Life Meets Entertainment: Life like Autonomous Agents," *Communications of the ACM*, 1995, pp 108-114.

29. DeRose, S., Maler, E., Orchard, D., Trafford, B., "XML Linking Language (XLink)", *W3 Consortium*, March 03, 1998.
Online: http://www.w3.org/TR/xlink/

30. Daniel, R., DeRose, S., Maler, E., "XML Pointer Language", *W3 Consortium*, March 03, 1998.
Online: http://www.w3.org/TR/xptr

31. Murray-Rust P., "CML version 1.0", January 1997. Available online at http://www.venus.co.uk/omf/cml-1.0/intro.html

32. Quillian M. R., "Word Concepts: A Theory and simulation of Some Basic Semantic Capabilities", *Behavioral Science 12*, 1967.

33. Ragget, D., Le Hors, A., and Jacobs, I., "HTML 4.01 Specification", *W3 Consortium*, December 24, 1999.
Online: http://www.w3.org/TR/html4/

34. Luke S., Heflin J., "SHOE 1.01 Specification". Feb. 3, 2000
Online: http://www.cs.umd.edu/projects/plus/SHOE/spec.html

35. Soham, Y., "An Overview of Agent-Oriented Programming", *In Software Agents, ed J.M. Bradshaw. Menlo Park, Calif.: AAAI Press*, 1997.

36. Sperberg-McQueen, C. M., and Burnard, L., editors. Guidelines for Electronic Text Encoding and Interchange. *Association for Computers and the Humanities (ACH), Association for Computational Linguistics (ACL), and Association for Literary and Linguistic Computing (ALLC)*. Chicago, Oxford: Text Encoding Initiative, 1994.

37. Rajaraman A., Norvig P., "Virtual Database Technology: Transforming the Internet into a Database", *IEEE Internet Computing*, Volume 2, N. 4, July-August, 1998, pp 55-58.

38. Tran, J., "Intorduction to GUD", a technical report, School of Information Technology and Engineering, University of Ottawa, September 1997.

39. W3C Synchronized Multimedia Working Group, "SMIL 1.0 Specification", *W3 Consortium*, June 15, 1998. Available online at http://www.w3.org/TR/REC-smil/

40. Walsh N., "An Introduction to XML", *ArborText, Inc*, September 1997.
Online: http://www.arbortext.com/nwalsh.html

41. Wang, Y., "Implementation of a User Interface and Web Extension of a Relational Database Application", *a Master's thesis*, School of Information Technology and

Engineering, University of Ottawa, December 1999.

42. Weiss S., "Glossary for Information Retrieval", Department of Computer Science, John Hopkins University, January 1997.

Online: http://www.cs.jhu.edu/~weiss/glossary.html

43. Woods W. A., "What's in a Link: Foundations of Semantic Networks", in *Representation and Understanding: Studies in Cognitive Science,* D.G. Bobrow and A.M. Collins, eds., Academic Press, New York, 1975.

Appendix A

Publications based on this Thesis

1. "A Web Ontology Description Language", Hicham Ouahid and Ahmed Karmouch, *Proceedings of the IEEE GlobCom'99*, December 1999

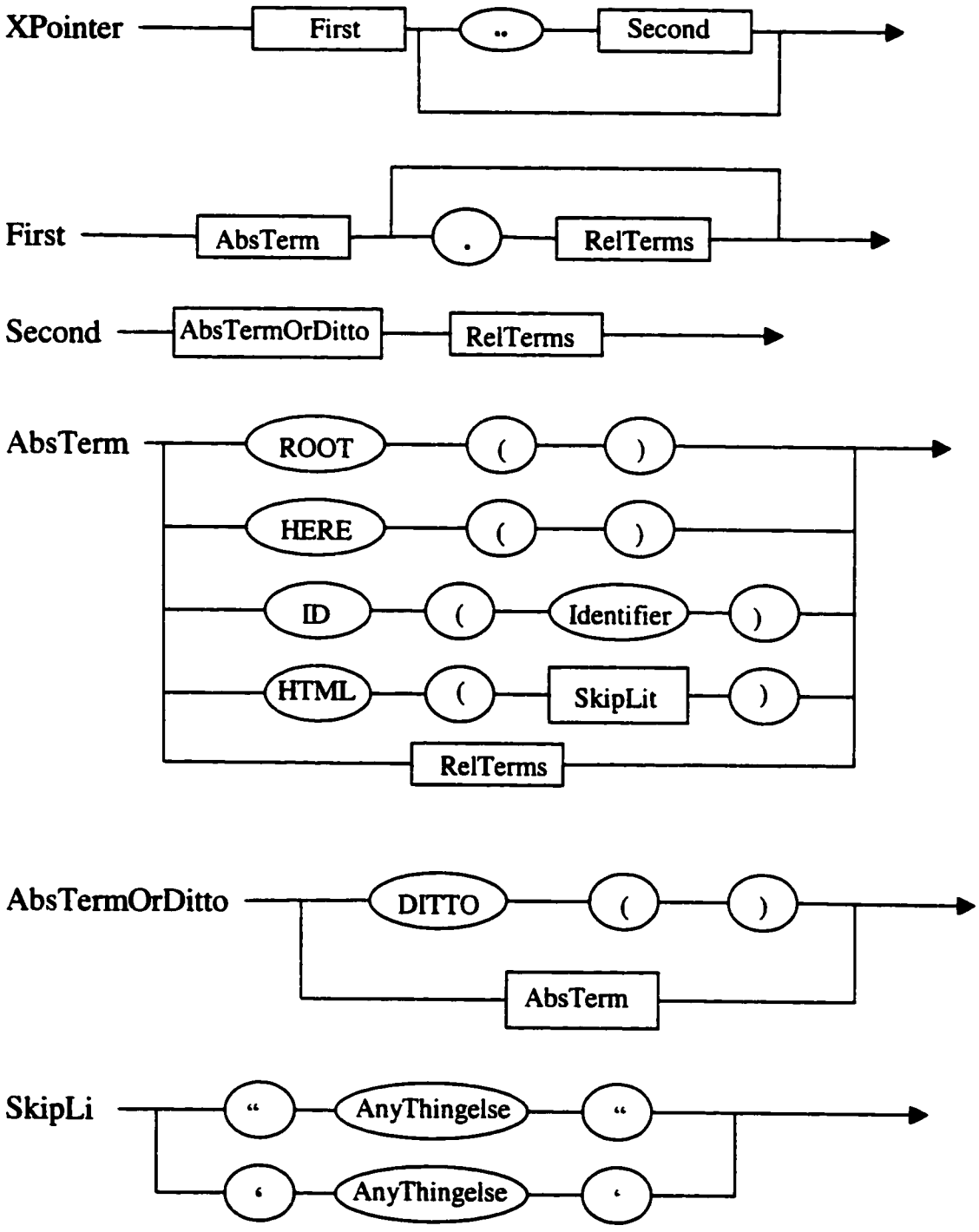
2. "An XML-based Web Mining Agent", Hicham Ouahid and Ahmed Karmouch, *Proceedings of MATA'99*, October 1999

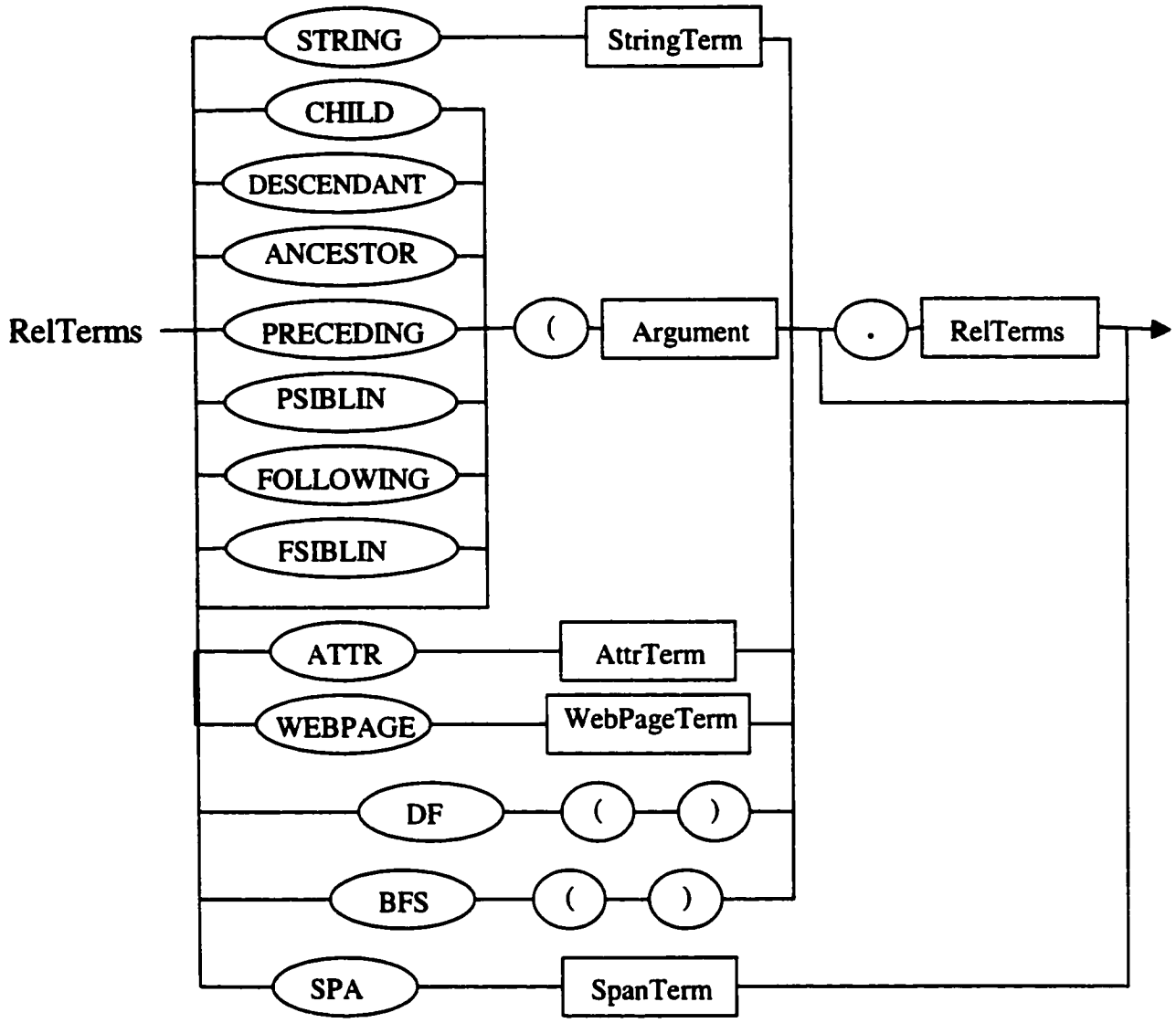
3. "WONDEL: an Efficient Way to Add Semantics to the Web Pages", Hicham Ouahid and Ahmed Karmouch, *Proceedings of Metastructures'99*, August 1999

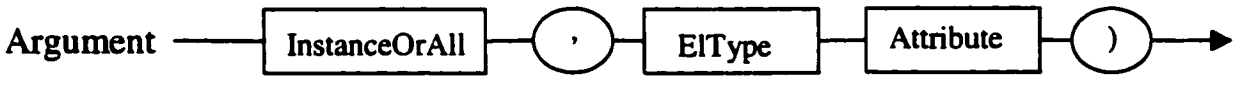
4. "Converting Web Pages into Well-formed XML Documents", Hicham Ouahid and Ahmed Karmouch, *Proceedings of the IEEE ICC'99*, June 1999

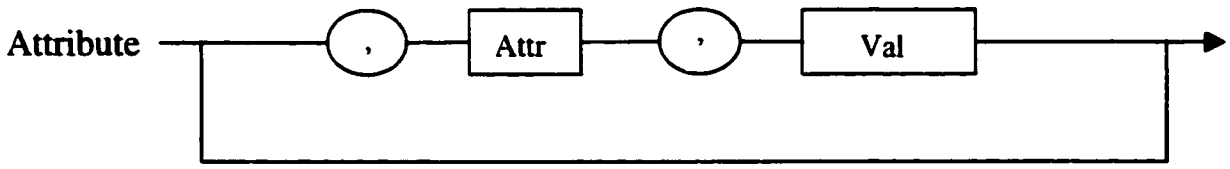
Appendix B

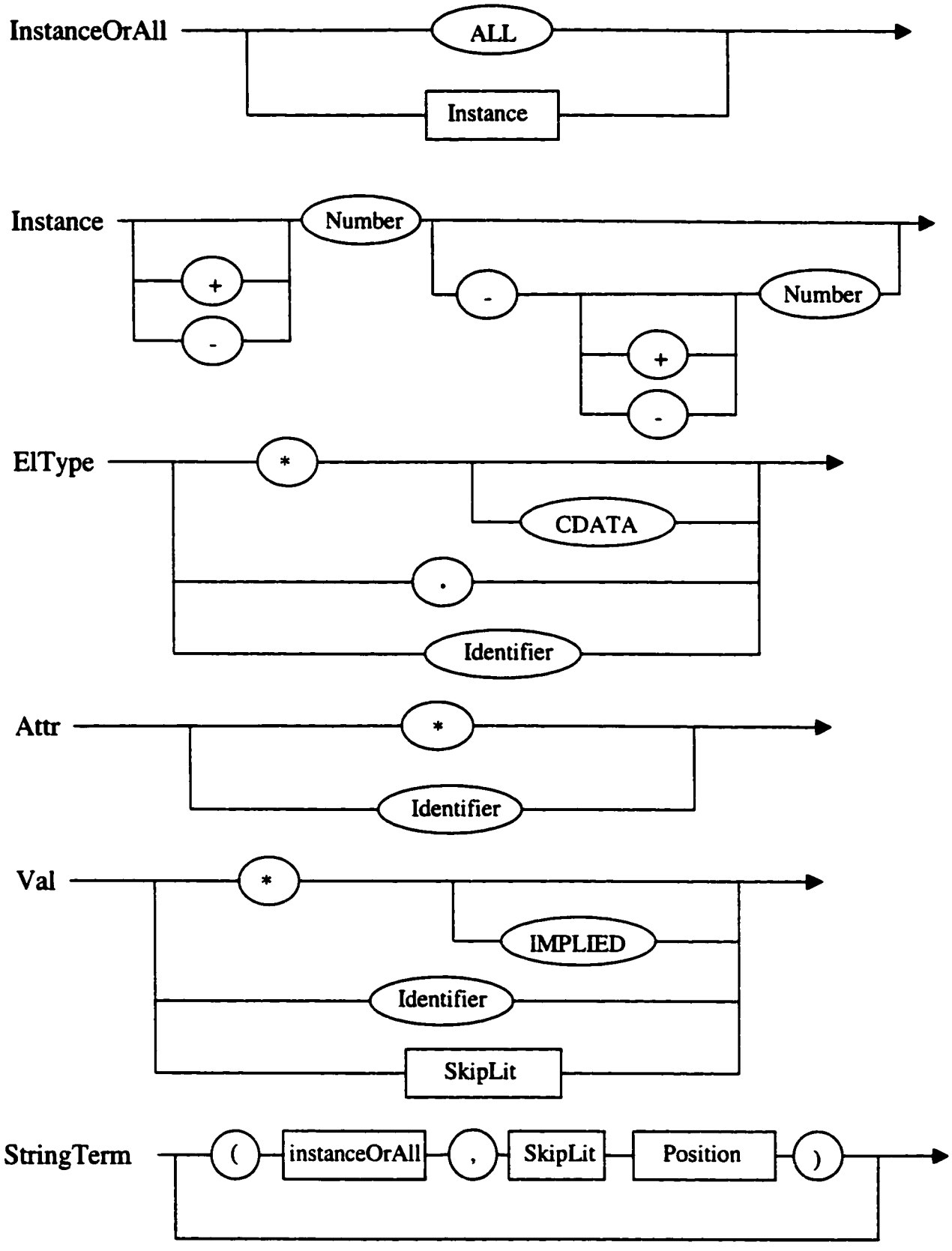
Syntax diagrams for the extended XPointer language grammar

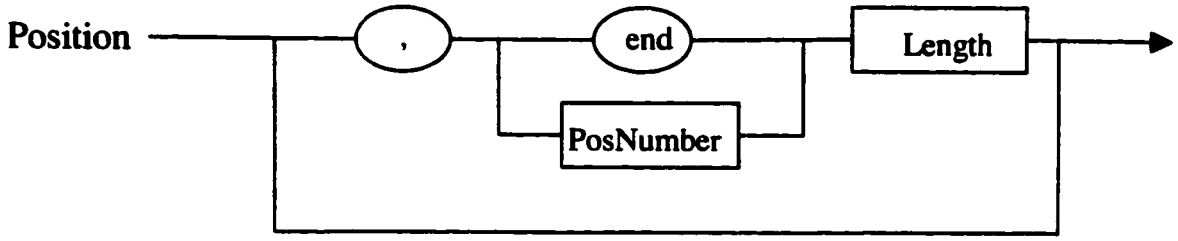


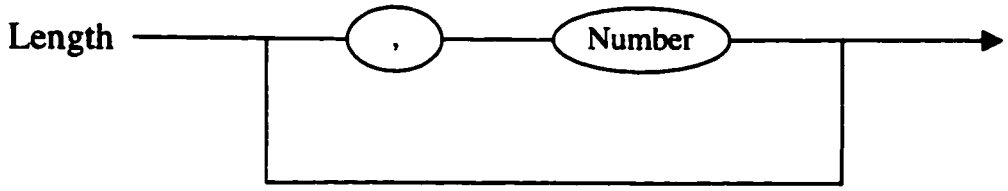

RelTerms
STRING
StringTerm
CHILD
DESCENDANT
ANCESTOR
PRECEDING
PSIBLIN
FOLLOWING
FSIBLIN
(
Argument
.
RelTerms
ATTR
AttrTerm
WEBPAGE
WebPageTerm
DF
(
)
BFS
(
)
SPA
SpanTerm

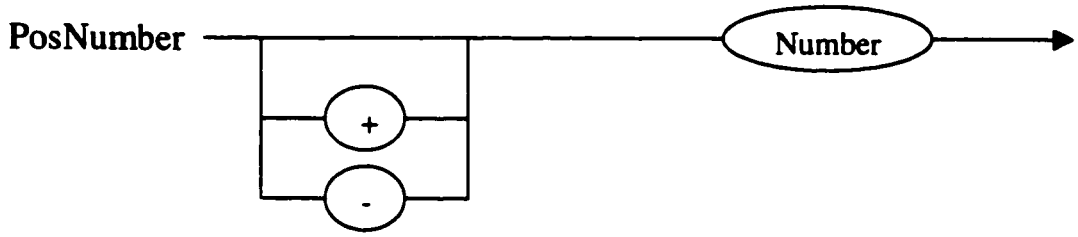

Argument
InstanceOrAll
,
ElType
Attribute
)

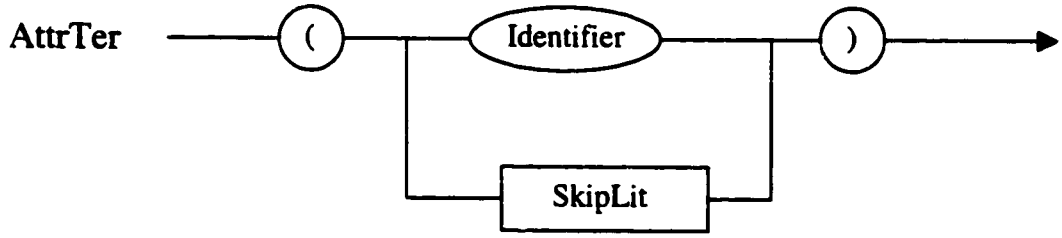

Attribute
,
Attr
'
Val

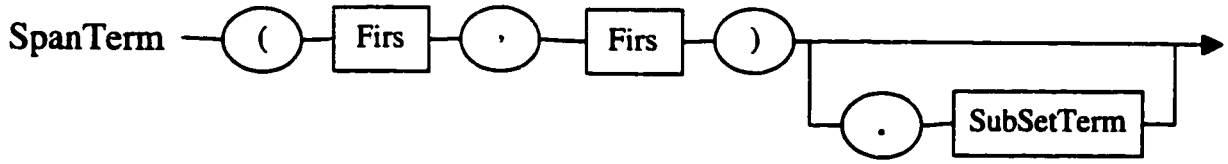

InstanceOrAll
ALL
Instance
Instance
Number
+
-
-
Number
+
-
ElType
*
CDATA
.
Identifier
Attr
*
Identifier
Val
*
IMPLIED
Identifier
SkipLit
StringTerm
(
instanceOrAll
,
SkipLit
Position
)


Position
,
end
Length
PosNumber


Length
,
Number


PosNumber
Number
+
-


AttrTer
(
Identifier
)
SkipLit


SpanTerm
(
Firs
,
Firs
)
.
SubSetTerm


SubSetTerm
SUBSET
(
Argument

Appendix C

The DTD of the Global University Database

```
<!ELEMENT GUD                (UNIVERSITY|RESEARCH_PROJECT|AGENCY)+>
<!ELEMENT AGENCY             (NAME,URL,LOCATION)>
<!ELEMENT RESEARCH_PROJECT   (NAME,URL,DESCRIPTION,RESEARCH_GRANT*,WORKS_ON*)>
<!ELEMENT WORKS_ON           (STAFFEMAIL)>
<!ELEMENT RESEARCH_GRANT     (TYPE,STARTDATE,ENDDATE,AMOUNT,AGENCYNAME)>
<!ELEMENT UNIVERSITY         (NAME,COUNTRY,STATE,CITY,UNIVURL,
                                (PATENT|FACULTY|DEPARTMENT|COURSE|RESEARCH_GROUP)*)>
<!ELEMENT PATENT             (NO,TITLE,ABSTRACT,URL)>
<!ELEMENT FACULTY            (NAME,URL)>
<!ELEMENT DEPARTMENT         (NAME,URL,FACULTYNAME,STAFF+)>
<!ELEMENT COURSE             (CODE,TITLE,URL,DESCRIPTION,LEVEL,DEPARTMENTNAME)>
<!ELEMENT RESEARCH_GROUP     (NAME,URL,DEPARTMENTNAME,GROUP_PROJECT*,MEMBER*)>
<!ELEMENT MEMBER             (STAFFEMAIL)>
<!ELEMENT GROUP_PROJECT      (RESEARCH_PROJECTNAME)>
<!ELEMENT STAFF
(EMAIL,FAMILYNAME,GIVENNAME,POSITION,HOMEPAGE,RESEARCH_INTEREST*)>
<!ELEMENT RESEARCH_INTEREST (TOPIC,DESCRIPTION?,URL?)>

<!ELEMENT   NAME                 (#PCDATA)>
<!ELEMENT   URL                  (#PCDATA)>
<!ELEMENT   LOCATION             (#PCDATA)>
<!ELEMENT   DESCRIPTION          (#PCDATA)>
<!ELEMENT   STAFFEMAIL           (#PCDATA)>
<!ELEMENT   TYPE                 (#PCDATA)>
<!ELEMENT   STARTDATE            (#PCDATA)>
<!ELEMENT   ENDDATE              (#PCDATA)>
<!ELEMENT   AMOUNT               (#PCDATA)>
<!ELEMENT   AGENCYNAME           (#PCDATA)>
<!ELEMENT   COUNTRY              (#PCDATA)>
<!ELEMENT   STATE                (#PCDATA)>
<!ELEMENT   CITY                 (#PCDATA)>
<!ELEMENT   UNIVURL              (#PCDATA)>
<!ELEMENT   NO                   (#PCDATA)>
<!ELEMENT   TITLE                (#PCDATA)>
<!ELEMENT   ABSTRACT             (#PCDATA)>
<!ELEMENT   FACULTYNAME          (#PCDATA)>
<!ELEMENT   CODE                 (#PCDATA)>
<!ELEMENT   LEVEL                (#PCDATA)>
<!ELEMENT   DEPARTMENTNAME       (#PCDATA)>
<!ELEMENT   RESEARCH_PROJECTNAME (#PCDATA)>
<!ELEMENT   EMAIL                (#PCDATA)>
<!ELEMENT   FAMILLYNAME          (#PCDATA)>
<!ELEMENT   GIVENNAME            (#PCDATA)>
<!ELEMENT   POSITION             (#PCDATA)>
<!ELEMENT   HOMEPAGE             (#PCDATA)>
<!ELEMENT   TOPIC                (#PCDATA)>
```

```
<!-- ATTRIBUTES ROLE -->
<!-- REGULARS -->
<!ATTLIST  URL
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  LOCATION
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  DESCRIPTION
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  ENDDATE
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  AMOUNT
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  COUNTRY
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  STATE
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  CITY
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  UNIVURL
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  TITLE
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  ABSTRACT
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  LEVEL
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  FAMILLYNAME
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  GIVENNAME
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  POSITION
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">
<!ATTLIST  HOMEPAGE
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "REGULAR">

<!-- PRIMARY KEYS-->
<!ATTLIST  NAME
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  TYPE
```

```
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  STARTDATE
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  NO
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  CODE
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  EMAIL
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  TOPIC
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  STAFFEMAIL
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  AGENCYNAME
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">
<!ATTLIST  RESEARCH_PROJECTNAME
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "PRIMARYKEY">

<!-- FOREIGN KEYS -->
<!ATTLIST  DEPARTMENTNAME
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "FOREIGNKEY">
<!ATTLIST  FACULTYNAME
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "FOREIGNKEY">

<!-- STRONG ENTITIES -->
<!ATTLIST  UNIVERSITY
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "STRONGENTITY"
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  RESEARCH_PROJECT
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "STRONGENTITY"
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  AGENCY
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "STRONGENTITY"
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  STAFF
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "STRONGENTITY"
            PREFIX (YES|NO) #FIXED "NO">

<!-- WEAK ENTITIES -->
<!ATTLIST  PATENT
            ROLE   (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
```

```
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  FACULTY
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  DEPARTMENT
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  COURSE
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  RESEARCH_GROUP
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  RESEARCH_INTEREST
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "WEAKENTITY"
            PREFIX (YES|NO) #FIXED "NO">

<!-- RELATIONS -->
<!ATTLIST  WORKS_ON
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "RELATION"
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  RESEARCH_GRANT
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "RELATION"
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  GROUP_PROJECT
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "RELATION"
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  MEMBER
            ROLE  (REGULAR|PRIMARYKEY|FOREIGNKEY|WEAKENTITY|STRONGENTITY|RELATION)
#FIXED "RELATION"
            PREFIX (YES|NO) #FIXED "NO">

<!-- ATTRIBUTE PREFIX -->
<!ATTLIST  PATENT
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  COURSE
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  FACULTY
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  DEPARTMENT
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  RESEARCH_GROUP
            PREFIX (YES|NO) #FIXED "YES">
<!ATTLIST  RESEARCH_PROJECT
            PREFIX (YES|NO) #FIXED "YES">

<!ATTLIST  UNIVERSITY
            PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  STAFF
```

```
             PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  AGENCY
             PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  RESEARCH_INTEREST
             PREFIX (YES|NO) #FIXED "NO">

<!ATTLIST  MEMBER
             PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  WORKS_ON
             PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  GROUP_PROJECT
             PREFIX (YES|NO) #FIXED "NO">
<!ATTLIST  RESEARCH_GRANT
             PREFIX (YES|NO) #FIXED "NO">
```