A Framework for Automatic Ontology Generation from Autonomous Web Applications

Giovanni Modica

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation
https://scholarsjunction.msstate.edu/td/129

This Graduate Thesis is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.
A FRAMEWORK FOR AUTOMATIC ONTOLOGY GENERATION
FROM AUTONOMOUS WEB APPLICATIONS

By
Giovanni Modica

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

December 2002
Copyright by
Giovanni Modica
2002
A FRAMEWORK FOR AUTOMATIC ONTOLOGY GENERATION
FROM AUTONOMOUS WEB APPLICATIONS

By
Giovanni Modica

Approved:

Hasan M. Jamil
Assistant Professor of Computer Science
(Major Professor)

Julian E. Boggess
Associate Professor of Computer Science
Graduate Coordinator
Department of Computer Science

Avigdor Gal
Senior Lecturer, Faculty of Industrial Engineering & Management
Technion Institute of Technology
(Committee Member)

Julia E. Hodges
Professor of Computer Science
and Department Head
(Committee Member)

Joerg Meyer
Assistant Professor of Computer Science
(Committee Member)

A. Wayne Bennett
Dean of the College of Engineering
Ontologies capture the structure, relationships, semantics and other essential meta information of an application. This thesis describes a framework to automate application interoperability by using dynamically generated ontologies. We propose a set of techniques to extract ontologies from data accessible on the Web in the form of semi-structured HTML pages. Ontologies retrieved from similar applications are matched together to create a general ontology describing the application domain.

Information retrieval and graph matching techniques are used to match and measure the usefulness of the ontologies created. Matching algorithms are combined together to produce global ontologies based on local ontologies inherently present in Web applications. We present a system called OntoBuilder that allows users to drive the ontology creation process using a user-friendly and intuitive interface. We also present experiments
for a well-known case of study: car-rental applications. We successfully achieve 90% accuracy on ontology extraction and 70% accuracy for ontology matching.
DEDICATION

To my family, for their patience and understanding.
ACKNOWLEDGMENTS

First of all, I would like to thank my major professor Dr. Hasan Jamil. Since my first semester at Mississippi State, Dr. Jamil has taken me as one of his students, allowing me to be part of the Intelligent Database System Research Group, a selected team of researchers formed by faculty members, students and external collaborators. Dr. Jamil introduced me to the area of research, encouraging the preparations of papers in different areas including Web databases, data warehousing and bioinformatics.

For each class I took with Dr. Jamil, I managed to publish a paper in conferences such as CoopIS, BIBE, CIKM and WWW. These publications gave me the opportunity to travel the world to present our work and to develop the skills of a researcher. During the second year of being at MSU, Dr. Jamil offered me a position as a research assistant in the Department of Computer Science, giving me stable work and economic support while advancing in my Master’s degree. Dr. Jamil is the most important reason for my success as an outstanding student of the Computer Science Department at MSU.

I would like to thank Dr. Avigdor Gal for the help and guidance provided in the realization of this work. Dr. Gal introduced me to the area of information retrieval, and although not physically present at MSU (he was working at the Technion Technical Institute of Israel at the time this thesis was written), his advice and technical background were important contributions to this thesis.
I would also like to thank Dr. Durriya Meer for the help provided revising this document.

Finally, I want to thank my family, my dad Orazio and my mom Pina, and my brother Fausto and my sister Sandra. Also to Maria Angelica, who was always there, present in good and bad times when I needed someone to understand me and give me peace of mind. I wouldn’t have made it without their support. I want to apologize to my family for not being there with them. I understand how hard it is for parents to live separated from their sons. And I understand it because it hurts the sons too, to have to live separated from their family.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE</td>
<td>xi</td>
</tr>
</tbody>
</table>

## CHAPTER

### I. INTRODUCTION

1.1 Motivation | 4
1.2 Scope of this Thesis | 8
1.3 Hypothesis and Main Goals | 9
1.4 Organization | 10

### II. DEFINITIONS AND BACKGROUND

2.1 Ontologies | 11
2.2 Data Integration | 12
2.3 Extended Hypothesis | 15

### III. ONTOLOGY EXTRACTION FROM SEMI-STRUCTURED DATA

3.1 Semi-structured Data | 18
  3.1.1 Modeling Semi-structured Data | 20
3.2 Internet Standards for Semi-Structured Data Representation | 21
  3.2.1 The Document Object Model (DOM) | 21
  3.2.2 Extensible Markup Language (XML) | 23
3.3 Ontology Extraction | 24

### IV. AN OO EXTENSION OF XML FOR ONTOLOGICAL APPLICATIONS | 28
<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 XML\textsuperscript{++}: An Object-Oriented Extension of XML</td>
<td>29</td>
</tr>
<tr>
<td>4.1.1 Static Versus Dynamic Inheritance</td>
<td>31</td>
</tr>
<tr>
<td>4.1.2 A Motivating Example</td>
<td>31</td>
</tr>
<tr>
<td>4.2 Related Work</td>
<td>33</td>
</tr>
<tr>
<td>4.2.1 XML Schema</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2 SOX</td>
<td>35</td>
</tr>
<tr>
<td>4.3 XML\textsuperscript{++} Overview</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1 Salient Features of XML\textsuperscript{++} Data Model</td>
<td>37</td>
</tr>
<tr>
<td>4.3.2 Syntax and Semantics</td>
<td>38</td>
</tr>
<tr>
<td>4.3.3 Classes</td>
<td>40</td>
</tr>
<tr>
<td>4.3.4 Inheritance</td>
<td>41</td>
</tr>
<tr>
<td>4.3.5 Support for Methods</td>
<td>44</td>
</tr>
<tr>
<td>4.3.5.1 Introspection</td>
<td>45</td>
</tr>
<tr>
<td>4.3.5.2 Usage</td>
<td>47</td>
</tr>
<tr>
<td>4.3.5.3 XPath Expressions</td>
<td>49</td>
</tr>
<tr>
<td>4.3.5.4 Caller Self Reference</td>
<td>51</td>
</tr>
<tr>
<td>4.3.5.5 Method Scope</td>
<td>52</td>
</tr>
<tr>
<td>4.3.5.6 Method Invocation</td>
<td>54</td>
</tr>
<tr>
<td>4.4 XML\textsuperscript{++} Middle-tier Engine</td>
<td>55</td>
</tr>
<tr>
<td>4.4.1 System Design</td>
<td>56</td>
</tr>
<tr>
<td>4.4.2 System Implementation</td>
<td>58</td>
</tr>
</tbody>
</table>

V. HIERARCHICAL ONTOLOGY REPRESENTATION | 63 |

5.1 Ontologies | 65 |
| 5.1.1 Ontological Structures | 66 |
| 5.1.1.1 Classes | 66 |
| 5.1.1.2 Slots | 67 |
| 5.1.1.3 Functions | 68 |
| 5.1.1.4 Axioms | 68 |
| 5.1.1.5 Composition | 69 |
| 5.1.1.6 Precedence | 70 |
| 5.1.2 An Example of a Global Ontology Creation | 70 |
| 5.2 Representing Ontologies in XML | 72 |
| 5.3 The Role of Inheritance in Ontology Representation | 76 |
| 5.3.1 Dynamic Inheritance | 78 |
| 5.4 Ontology Representation | 80 |
# Table of Contents

## Chapter VI. Ontology Matching Techniques

6.1 Related Work ........................................ 90
6.2 Metrics .................................................. 92
6.3 Information Retrieval Techniques ..................... 94
   6.3.1 Achieving Symmetry ............................. 97
6.4 Graph Matching ...................................... 99
   6.4.1 Related Work on Graph Matching .......... 99
   6.4.2 Web Resource Ontologies ................... 101
   6.4.3 The Graph Matching Algorithm .......... 102

## Chapter VII. The Ontobuilder System

7.1 System Overview .................................... 107
7.2 System Architecture .................................. 108
7.3 Label Identification .................................. 111
7.4 Utilities and Data Structures ....................... 112
7.5 Algorithms Implementation ......................... 115
7.6 Programming Language and Libraries Used .......... 116
7.7 User Interface ...................................... 118

## Chapter VIII. Experiments and Results

8.1 Label Identification Experiments .................. 121
8.2 Matching Algorithms Evolution ...................... 123
8.3 Matching Results for Different Ontologies ........ 125
8.4 Graph Matching Results ............................ 127

## Chapter IX. Conclusion

9.1 Strengths of this Thesis .......................... 133
9.2 Lessons Learned ..................................... 135
9.3 Future Work ......................................... 137

REFERENCES .............................................. 140
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Common HTML tags</td>
</tr>
<tr>
<td>3.2</td>
<td>Differences between HTML and XML</td>
</tr>
<tr>
<td>8.1</td>
<td>Label identification results.</td>
</tr>
<tr>
<td>8.2</td>
<td>Matching algorithms evolution.</td>
</tr>
<tr>
<td>8.3</td>
<td>Matching between different ontologies.</td>
</tr>
<tr>
<td>8.4</td>
<td>Graph matching between Cupid and our algorithm.</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An ideal scenario for Web applications</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Global schema creation</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Model for representing semi-structured data</td>
<td>21</td>
</tr>
<tr>
<td>3.2</td>
<td>A DOM hierarchical view</td>
<td>22</td>
</tr>
<tr>
<td>3.3</td>
<td>Ontology extraction from an HTML page</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>An example for home page design</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>XML++ language plug-ins</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>XML++ method invocation</td>
<td>55</td>
</tr>
<tr>
<td>4.4</td>
<td>XML++ framework</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>Architecture for the XML++ middle-tier engine</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>A class example</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>A slot example</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>A function example</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>A composition example</td>
<td>69</td>
</tr>
<tr>
<td>5.5</td>
<td>Local ontologies for two companies</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Global ontology</td>
<td>72</td>
</tr>
<tr>
<td>5.7</td>
<td>An ontology hierarchy diagram</td>
<td>76</td>
</tr>
</tbody>
</table>
LIST OF SYMBOLS, ABBREVIATIONS, AND NOMENCLATURE

The following is a list of the symbols used in this document:

$R$ Recall. Measures the completeness of the terms matched in the matching process.

$P$ Precision. Measures the soundness of the terms matched in the matching process.

$E$ Error. Combines the two metrics $R$ and $P$.

$b$ Importance. Indicates the importance of precision and recall in the $E$ metric.
CHAPTER I

INTRODUCTION

During the last decade we have witnessed great advances in the Internet field. Not long ago, Web applications were only used in academics, but today, Web applications are being developed for almost any area of the industry as well, including e-Commerce—with B2B and B2C applications—, research, telecommunications, databases, information systems, bioinformatics, etc. The acceptance of applications with a Web front-end has gained acceptance in the users’ community because of the intuitive point-and-click interface characteristic of these applications.

A global phenomena, the Internet accommodates a wide variety of knowledge present in different cultures around the world. This heterogeneous knowledge is not only manifested in idiomatic differences, as one may think at first, but is also present in design and structural differences as well. Heterogeneity is also present in the state of mind of the persons who design, develop and deploy Web applications. The software and hardware used by the application also influences this heterogeneity. As an immediate consequence of this, two Web application developed for the same domain differ, in the way they interact with the end user and environment in which they are deployed.
Another characteristic of Web applications is their distributed aspect, inherited from the Internet itself. A typical architectural design for Web application development is the three-tier (or more generally the n-tier) model where the application is divided into different components, each addressing a particular need in the overall application design. A typical architecture based on the three-tier model will have a layer responsible for data manipulation (usually a database), a layer responsible for modeling and enforcing business rules (the Web server and software components) and a layer that acts as a front-end to the user responsible for retrieving user input and displaying results (a Web browser in our case). This model can be as simple as a phone directory system having only one database located in the same machine for the Web server, or it can be as complex as a Bioinformatic application that needs to access genetic data spread across multiple databases and Web servers around the world.

As we mentioned before, the acceptance of Web applications is due to their ability to abstract all the heterogeneity and distributed details of the underlying domain in a way that is transparent to the user. Users do not need to be aware of what is going on behind the scenes while interacting with a Web application, and to some extent, the Web page in a browser is all what the application is for the average user with no technical background. This is the way it should be.

For instance, a biologist who is trying to mine information among a set of genetic data spread across different databases doesn’t need to worry about what is the structure of database A and the structure of database B and how does terms in A match to terms in B.
All the person needs is to perform her job as a biologist and focus on the results she wants, and not on how to get those results. This is similar to compare procedural languages such as C++ versus declarative languages such as SQL. In the former, one has to specify how to get the results, while in the latter one has to specify what the desired results are. The system will figure out (in an efficient and effective way) how to get those results.

Although we have described the ideal behavior of any Web application, we are a long way from actually having Web applications that are “intelligent” enough to behave in a declarative fashion. Different approaches have been proposed toward this goal, the most relevant being the work presented in semantic Web [33, 9], Web services [34, 78] and ontology creation [47, 62, 81]. These three approaches are not orthogonal to each other but complementary. Of the three, ontologies are the main topic of this thesis, thus, it will be our main focus for the rest of this section.

Ontologies capture the structure, relationships, semantics and other essential meta information about the application. Ontologies are generally used by software agents to “understand” the knowledge of an application. They provide a generalized view of the application semantics and behavior, allowing interaction between software agents that use ontologies for knowledge representation. Although most commonly found in e-Commerce applications, ontologies are gaining acceptance in any domain for which a Web application exists. Two prime examples of the use of ontologies in Web applications are Yahoo.com and Amazon.com [55]. Organizations like Ontology.org—an independent industry and research forum which focuses upon the application of ontologies in e-Commerce—have
emerged in the last couple of years to create standard ontologies for effective business to business (B2B) interactions. However, applications typically use and need unique ontologies to suit their business needs, causing enormous difficulties for autonomous agents trying to interoperate.

This thesis proposes to automate application interoperability by using dynamically created ontologies. We propose a set of techniques and heuristics to extract ontologies from data on the Web in the form of semi-structured HTML pages. Ontologies extracted from similar applications can be merged together to create a general ontology describing the application domain. Work in the area suggests the use of machine learning techniques [38], graph matching techniques [54] or information retrieval techniques [54, 62]. We use the latter two and provide metrics to measure the usefulness of the ontologies created.

1.1 Motivation

The motivation behind automatic ontology creation is twofold. The use of ontologies to describe Web applications has many advantages in terms of organization and information sharing among applications. However, the creation of ontologies is not a trivial matter. The need for (semi)automatic techniques applied to ontology creation has been previously considered in the literature, and many authors agree that eCommerce is driving a new paradigm in the use of ontology to describe Web services [62, 23, 55]. Although different tools exist to facilitate the process of ontology creation (e.g., Ontolingua [28, 36], Chimaera [55], Ontosaurus [81], etc.), ontologies need to be created by domain experts
(usually called cybrerians or chief ontologists [62, 55]) with in-depth knowledge of the application and the environment in which the application is executing. This gives rise to the first incentive: the creation of ontologies is a difficult, tedious, and time-consuming process. The second incentive has to do with the dynamic constitution of ontologies. Manually created ontologies need to be maintained every time changes to the application rules occur, Web applications being no exceptions.

As an example of the applicability of automatic ontology creation consider the following scenario: A biologist wants to retrieve phylogenetic information about the *Drosophila* insect by accessing information publicly available in phylogenetic databases such as *Tree of Life*\(^1\), *TreeBASE*\(^2\) or *NatureServe*\(^3\). These three phylogenetic Web applications are characterized by the heterogeneity and distributed factors explained before, *i.e.* each Web site has its own proprietary database schema and they are located on different Web servers across the globe. In order to obtain information from these applications, the biologist in our example needs to input her query three times, one for each different Web application, and after obtaining individual results, she has to figure out how to relate them in a meaningful way. An ideal scenario is the one illustrated in Figure 1.1.

In the figure, users of a Web application remain ignorant any details about heterogeneity and distributed data. All the users need to worry about is *what* are the results they want. Usually, the user has some restrictions in mind (query conditions) and optionally

---

\(^1\)http://tolweb.org/tree/phylogeny.html  
\(^2\)http://www.treebase.org/treebase  
\(^3\)http://www.natureserve.org
some pointers to where the information could reside. A query is submitted in a declarative form (some kind of SQL query) against an “intelligent” system that is able to interpret the query and take care of the technical details not relevant to the user. This system will be responsible for contacting the Web applications that act as information sources and to build a general view of their database schemas. These schemas are none other than localized ontologies describing each application domain, while the general view constructed by the system is a global ontology produced by the matching and merging of the local ontologies. In an ideal scenario, such as this, user intervention is not required at all; however, in reality the problem of ontology or schema matching is very difficult to solve without user intervention.

The most common solution to this problem is to create a global schema that maps all heterogeneous components into a common model and to let an agent perform queries against this model. The queries are then localized and executed locally for each component.
system [5, 82], as shown in Figure 1.2. While this approach works well for many applications, the creation of a global schema implies the creation of static mappings between the global schema and the component systems. Such “static ontologies” are not well suited for highly dynamic environments, since the global schema must be manually rebuilt every time a change in the underlying data sources is detected [33]. This is why Internet applications, which are constantly changing to accommodate users’ demands and new advances in Web technology, require a different approach for application interoperability.

![Diagram of Global Schema Creation](image)

**Figure 1.2 Global schema creation**

For the scenario just described, attempts such as the one proposed in [45] and [46] fall under the category of “static” systems requiring a considerable amount of effort in maintaining a synchronization between the global schemas and the local schemas.
1.2 Scope of this Thesis

The scenario described in the previous section involves a number of different tasks not mentioned in our example. Tasks such as Web application identification, information mining, ontology generation, ontology matching, and query translation are just some of the required techniques that need to be put together in order to achieve a minimum of user intervention. In this thesis, we will only focus on two such tasks: ontology generation and ontology matching.

Ontology generation is related to the retrieval of ontological structures that describe application semantics from a Web application. We will focus on generation of ontologies from information publicly available on the Internet in the form of HTML pages. For ontology matching we will implement a set of information retrieval techniques to identify syntactic similarities among ontological terms, as well as graph matching techniques for identification of structural similarity.

Although we believe that the concepts, as presented in this work, are general enough to be applied to any domain, we chose to set our experiments in the well-known application domain of car-rental. Car rental applications are simpler than Bioinformatics applications and therefore, easier to interact with. Web portals like Avis.com\(^4\) and Hertz.com\(^5\), to name a few, are being studied and used as training examples for our ontology extraction

\(^4\)http://www.avis.com
\(^5\)http://www.hertz.com
techniques. Future work will focus on the application of the concepts proposed in this thesis to the Bioinformatics area.

1.3 Hypothesis and Main Goals

Although we agree with works such as [61, 82, 10] that the process of ontology creation cannot be fully automated, we believe that it is possible to achieve a high degree of automation, requiring only little intervention from domain experts. The job of the domain expert will be greatly facilitated when provided with a well-educated guess for an application ontology.

Our main objectives are as follows:

- Provide a framework for automatic ontology creation based on publicly available information on the Internet.
- Apply metrics to measure the usefulness\(^6\) of the ontologies generated using this framework.
- Apply information retrieval techniques to merge syntactically and semantically similar ontologies from the same application domain.
- Create heuristics to find semantically similar ontologies based on their structural information.

It is our goal to combine all the objectives mentioned above into a tool for automatic ontology creation. The tool must be powerful enough to be used by domain experts and at the same time, intuitive enough to be used by normal users who wish to experiment in the area of application interoperability.

\(^6\)By usefulness we mean how good, in terms of recall and precision, a matching between a pair of ontologies is.
1.4 Organization

The rest of the document is organized as follows:

- Chapter II presents background information in the area of ontologies and data integration. An extended hypothesis is presented in this chapter.

- Chapter III talks about ontology generation techniques from semi-structured data, introducing the label identification algorithm used to extract ontological terms from HTML pages.

- Chapter IV presents an object-oriented extension of XML (called XML++) for ontology representation.

- Chapter V discusses techniques for hierarchical ontology representation by means of inheritance based in XML++.

- Chapter VI introduces the information retrieval techniques being used for ontology matching. We explain a set of algorithms based on linguistic matching and the metrics used to measure the goodness of the matching. We also introduce some graph definitions for ontologies and then present a graph matching algorithm for structural matching.

- In chapter VII we talk about the OntoBuilder system, an ontology creation tool that allows users to extract and build complex ontologies directly from Web applications. We will explain how the tool works and give some details about the tool’s internals.

- Chapter VIII presents a set of experiments of ontology retrieval and ontology matching for real applications in the domain of car-rental applications.

- Chapter IX provides conclusions and future work about the concepts presented in this thesis.
CHAPTER II

DEFINITIONS AND BACKGROUND

In this chapter, we present a literature review and provide background information on the different topics covered by this thesis. We start with ontologies first, presenting a formal definition of the term and its most common uses, followed by a brief introduction to data integration and matching techniques. Finally, we present our extended hypothesis, introducing the main problem and our solution.

2.1 Ontologies

An ontology is “a specification of a conceptualization” [36], where conceptualization is an abstract view of the world represented as a set of objects. The term ontology is overloaded and has been used in different areas including philosophy (the area that coined the term), artificial intelligence, information sciences, knowledge representation, object modeling, and most recently, in eCommerce applications. An ontology can be described as a set of terms (vocabulary) with certain semantics and associated relationships [71]. A more detailed description of ontologies can be found in [36, 37].

An ontology is represented as a hierarchy of terms, much like a taxonomy. Therefore, relationships between classes are described according to a subclass-of relationships. As
an example, the following constitutes a taxonomy: human is a subclass of mammal, and mammal is a subclass of animal. However, ontologies need not be limited to hierarchical representations. They usually include a set of axioms and rules to specify the semantics of the vocabulary used to describe the ontology.

In recent years, ontologies have been widely used to develop Internet commerce applications. Ontologies are mainly used to describe application behavior and thus, achieve a high degree of interoperability by means of knowledge sharing and reusability. Internet agents, such as the ones used in most Web portals like Expedia.com or Travelocity.com, are developed with application interoperability in mind, integrating different information from heterogeneous data sources. Applications modeled after a well-defined ontology are easier to integrate and interoperate on behalf of these agents. Yet, there exist some practical barriers that constrain the use of standardized ontologies [67].

In chapter V, we will talk again about ontologies and introduce some of the structures that are commonly found in modern ontologies. We will now present a brief introduction to data integration and techniques for ontology matching.

2.2 Data Integration

A data integration system combines heterogeneous data available at different sources providing a global view of the unified data [41]. Integration of heterogeneous data sources is a challenging problem that has been widely discussed in the literature. One of the main issues in data integration is to find a relationship or mapping function between the under-
lying data sources and the global schema. In [51], three approaches to this problem are proposed:

- The *global-centric* approach requires the global schema to be expressed independently of the sources. Sources are specified as views over the global schema.
- The *source-centric* approach requires the global schema to be expressed based on the schema sources.
- The combined approach uses both approaches, depending on the integration tasks at hand.

In this project, we will be using the *global-centric* approach. This approach is useful when the sources are not known beforehand, allowing one to introduce new sources at any time during the integration process. The global schema will be constructed based on a *target* application while the local schemas, also called *candidates*, will be defined in terms of the target schema.

In order to successfully integrate heterogeneous data sources, one must build a mapping function between them. This mapping function is also called a *match*. A match is a correspondence function between elements in two data schemas [54]. The following list describes a taxonomy for matching techniques proposed in [54]:

- **Schema vs. instance based**: schema matchers only consider schema information while instance based matchers use data instances to perform the match.
- **Element vs. structure granularity**: an element level matcher provides a mapping among single elements while a structure level matcher uses groups of elements and their structure to find the match.
- **Linguistic based**: mappings are constructed based on the names of the schema elements. Information retrieval techniques like tokenization and stemming [31] are generally used in this kind of matcher, along with external aids like a thesauri.
- **Constraint based**: mappings are constructed based on schema constraints like data types or data ranges.
• **Matching cardinality**: the cardinality of a matching operation depends on the number of elements involved. As an example, a \(1:n\) match constructs a match between one term in one schema versus many terms in the other schema. Other combinations are \(1:1\), \(n:1\) and \(n:m\).

• **Auxiliary information**: auxiliary information such as dictionaries, thesauri, etc., can be used to aid the matching process.

• **Individual vs. combinatorial**: individual matchers apply only one algorithm to find a match, while combinatorial matchers use a combination of algorithms to find the best match. Algorithms like the one discussed in chapter VI can be combined and a combinatorial function (usually average) can be used to estimate the match.

The match proposed in this project falls under the following categories: *schema based*, (although we do not know the schema of the HTML pages we “guess” a schema and then perform a schema matching); *element based* and *structure based*, (we combine both techniques in the order given to maximize our matching); *linguistic based*, (we apply IR techniques on names of elements); *constrained based*, (although we do not have type information for the elements we want to match, we do have domain constraints in the case of combo boxes and radio buttons); *\(n:1\) matching cardinality*, (a term in the left data source can have multiple matching terms in the right data source); *use of auxiliary information*, (we recur to a thesauri to increase the matching between synonym terms); *combinatorial based*, (we use a combination of information retrieval algorithms and graph algorithms to find the match).

Although our main intention is to use the information retrieval algorithms outlined in chapter VI to build our matching system, we will also explore structure-based algorithms to maximize the match. Structure-based algorithms work on a hierarchical model, or using a more general term, a graph model of the data being integrated. One of the most
suitable structure-matching algorithms is the one implemented in the Cupid system called the TreeMatch algorithm[54]. The intuition behind the algorithm is very simple: elements are matched pair-wise based on a similarity function. The elements with the highest similarity function value constitute the best match. The similarity function value is constantly updated based on the similarity of the parent elements and siblings. E.g., if the parents of two elements are similar, then the similarity between those two elements is increased; if the parents are not similar, then the similarity between the elements is decreased. Initially all the elements are assigned a similarity based on their linguistic similarity using information retrieval algorithms. A detailed description of the algorithm is given in [54]. We will revisit structural matching in chapter VI where a graph representation of ontologies and a graph matching algorithm are presented.

2.3 Extended Hypothesis

We are proposing a framework to automate the creation of Web application ontologies based on public information available as HTML pages. Before we present a formal definition of the problem we are trying to solve, let us present some preliminary conjectures that will help in our definition of the problem:

- **Conjecture I**: Applications in a given domain base their information exchange on some (shared) underlying ontology.

- **Conjecture II**: Applications in a given domain might use different ontology representations.

- **Conjecture III**: Given an application $A$ such that $A$ utilizes an ontology representation $O_A$, and an ontology $O$, there exists an invertible mapping $f_A$ such that
The problem: Given two applications \( A \) and \( B \) such that \( A \) utilizes an ontology representation \( O_A \) and \( B \) utilizes an ontology representation \( O_B \), with \( O_A \) and \( O_B \) defined in the same application domain, there exists a function \( f_{BA} \) that represents the mapping between ontology \( O_A \) and \( O_B \). We define \( f_{BA} \) as follows:

\[
f_{BA}(O_A) = O_B
\]  

In a perfect world, the ontology \( O \) is known, as is also \( f_A \) (the mapping function for application \( A \) to \( O \)) and \( f_B \) (the mapping function for application \( B \) to \( O \)). Hence, it is possible to find a mapping between \( O_A \) and \( O_B \) as follows:

\[
O_A = f_A^{-1}(f_B(O_B))
\]  

However, in practice, \( O \) is unknown, or at best is approximated somehow in a standard form. The same happens with \( f_A \) and \( f_B \) due to lack of documentation or the mental state of the designer of the application.

Proposed solution: Given two applications \( A \) and \( B \) such that \( A \) utilizes an ontology representation \( O_A \), and \( B \) utilizes an ontology representation \( O_B \), introduce a mapping function \( f_{BA} \) such that

\[
f_{BA} : O_A \rightarrow O_B \times [0, 1]
\]
In Equation (2.4), $f_{BA}$ depends on the ontology representation. The match is associated with a degree of confidence where 0 identifies non-matching terms and 1 identifies a perfect match.
CHAPTER III

ONTOLOGY EXTRACTION FROM SEMI-STRUCTURED DATA

In this chapter, we introduce the ontology generation technique used to identify ontological terms from semi-structured data such as HTML pages. But before we do so, we would like to present some background information on semi-structured data and the Internet standards used to represent such data.

3.1 Semi-structured Data

According to [1], semi-structured data is that data that “is neither raw data nor strictly typed.” An example of raw data can be any binary document like an image or a sound file. In contrast with relational models where data is typed and explicitly defined, semi-structured data usually has an implicit structure that is often buried with the data itself. As an example, consider an HTML page that contains a list of cars available for rent. Each entry in the list is defined as a set of fields (e.g., car brand, price, miles, etc.). However there can be entries with additional fields that are not common to other entries in the same list. In the same way, there could also be entries with missing information.

Semi-structured data has the following characteristics (as outlined in [1]):

- **Irregular structure**: data elements do not have a regular schema associated and usually there is extra or incomplete information.
• **Implicit structure**: the underlying structure is not explicitly defined and usually specialized parsers are required in order to discover it.

• **Partial structure**: additional information can be stored in other places and linked together with the data.

• **Indicative structure**: in contrast with structured data, semi-structured data is not constrained by a specific schema.

• **A-posteriori schema**: the schema associated with semi-structured data is not defined a-priori as in the case of structured data, but a-posteriori after analyzing the underlying data.

• **Large schema**: due to heterogeneity in the data, the schema can be very large.

• **Ignored schema**: the schema is usually ignored when querying semi-structured data.

• **Changing schema**: the schema is constantly evolving.

The automatic discovery of implicit schema for semi-structured data is a challenging task, especially for HTML files, which definitely fit the characteristics outlined above. Due to little constraints imposed by Internet browsers (e.g., Microsoft Internet Explorer or Netscape Navigator) and authoring tools for Web page design, HTML pages are very difficult to parse. HTML pages are not well-formed documents and therefore are not validated against any kind of schema as in the case of XML documents (see section 3.2.2 for more details about well-formedness and validation). Although XML is emerging as a new standard for data representation, it is not fully supported in commercial applications and therefore, not widely used in user interfaces. A common use of XML is as a communication protocol for Web applications [60], or for internal data representation. However, Web applications front-ends are still designed using HTML, a trend that is likely to continue in the foreseeable future.
Still, schemas can be extracted from HTML pages by taking advantage of some tags widely used in Web page design. Table 3.1 shows a summary of some of these tags and gives a brief description for each of them.

Table 3.1 Common HTML tags

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;TABLE&gt;</td>
<td>Used to define tables</td>
</tr>
<tr>
<td>&lt;TD&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;TR&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;A&gt;</td>
<td>Link to another HTML page</td>
</tr>
<tr>
<td>&lt;H1&gt;...&lt;H6&gt;</td>
<td>Used in headings</td>
</tr>
<tr>
<td>&lt;FORM&gt;</td>
<td>Defines a set of input fields to be filled by the user</td>
</tr>
<tr>
<td>&lt;INPUT&gt;</td>
<td>Defines an input field to be filled by the user</td>
</tr>
<tr>
<td>&lt;META&gt;</td>
<td>Defines meta information about the page</td>
</tr>
</tbody>
</table>

3.1.1 Modeling Semi-structured Data

Many models have been proposed to represent semi-structured data. In this thesis, we will use the model implemented in [1, 65] to represent semi-structured data using relational tables like the one shown in Figure 3.1.a.

The model assumes that each element is identified by an object id, which can be automatically generated by the system or retrieved from the underlying data source. The data is represented by a labeled tree as the one shown in Figure 3.1.b. The atomic table is only for leaf elements in the tree, while non-leaf elements are considered complex objects and are stored in the link table.
3.2 Internet Standards for Semi-Structured Data Representation

In this section, we will discuss two standards that will play an important role in the implementation of the system proposed in this document (see chapter VII for more details).

3.2.1 The Document Object Model (DOM)

The Document Object Model, or DOM, is a standard proposed by the W3 Consortium. Currently at level 3, DOM is a language- and platform-independent interface that can be used in a wide range of programming and scripting languages to access and update different document aspects, including content, structure and style [40]. DOM provides a hierarchical view of a document (HTML, XML or any document defined using the DOM standard).

The model uses a tree where every node is represented by a Node interface. Nodes have children which can be accessed using predefined methods and functions. In the same...
way, node siblings and parents can be accessed, too. The DOM interface has been implemented in languages such as Java, C++ and even JavaScript. Many libraries that comply with the DOM interface have been designed. These libraries are based on a DOM parser that takes a document as input (in text format) like the one in Figure 3.2.a. and produces a DOM tree like the one presented in Figure 3.2.b.

```
<HTML>
  <HEAD>
    <TITLE>A Web page</TITLE>
  </HEAD>
  <BODY>
    <H1>A Web page</H1>
    <P>This is an example web page</P>
  </BODY>
</HTML>
```

![a) DOM hierarchical view](image)

**Figure 3.2 A DOM hierarchical view**

Most of the commercial DOM parsers don’t support HTML documents due to their lack of well-formedness. However, packages like HTML Tidy\(^1\) can process both XML and HTML documents. A Java implementation of HTML Tidy will be used in the scope of this work in order to extract the required information from HTML pages.

\(^1\)http://www.w3.org/People/Raggett/tidy/
3.2.2 Extensible Markup Language (XML)

The Extensible Markup Language, or XML, is also a standard proposed by the W3 Consortium. XML has been used in the past years as the *de facto* standard for data representation and electronic data interchange among Web and e-Commerce applications on the Internet. Supported by a set of related technologies, XML has covered different aspects of the data representation domain, *e.g.*, data formatting and styles (with XSL[3] and XSLT [49]), querying (with XQL, XQuery[11]), schema representation (DTDs and Schemas[27]), etc.

As in HTML, XML documents are constructed by using tags. XML documents are required to be well formed, which means they are required to comply with the following rules:

- All elements must have an open and end tag. As an example, the following XML fragment is well formed: `<name>Peter</name>`.

- Empty elements must have an end tag, *e.g.*, `<img></img>`, or using the short version, `<img />`.

- All elements, except for the root element, must be nested within a parent element.

- Elements cannot begin with `{x|X}{m|M}{l|L}`, digits, hyphen or period.

- Attributes values must be enclosed by single (`'`) or double (`"`) quotation marks. For example, the following XML fragment has its attribute value enclosed in double quotes: `<person age="26" />`.

- XML is case-sensitive, thus case should be used consistently.

- XML documents cannot contain reserved characters (`&`, `<`).

- XML documents must contain an XML prolog specifying the version being used, *i.e.*, `<?xml version="1.0"?>`.

Due to their well-formedness, XML documents are easier to parse into a DOM tree. Besides being well-formed, XML documents can be optionally validated against a prede-
fined schema using a DTD (Document Type Definition) or an XML Schema [27]. Vali-
dated documents are trusted to contain all the required information of the application that
is processing them.

XML is a simplified Web version of SGML. Although XML and HTML have the same
predecessor, there exist noticeable differences between these two languages. Table 3.2
summarizes these differences.

<table>
<thead>
<tr>
<th>XML</th>
<th>HTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tags focus on data representation semantics</td>
<td>Tags focus on data formatting</td>
</tr>
<tr>
<td>The set of tags is not predefined</td>
<td>Tags are predefined</td>
</tr>
<tr>
<td>Must be well-formed and optionally validated</td>
<td>Not even well-formed</td>
</tr>
<tr>
<td>XML is case-sensitive</td>
<td>HTML is not case-sensitive</td>
</tr>
</tbody>
</table>

We decided to use XML in order to represent the ontologies extracted from Web
applications. Using a common standard, ontologies can be translated and converted to
other knowledge-representation languages (e.g., RDF [50]) being used in applications like
Ontolingua[28] and Ontosaurus[81].

3.3 Ontology Extraction

The extraction of an ontology from a Web site is a complex process. The ontology can
be retrieved based on the forms presented in the Web site to the user. For example, the
ontology for a car-rental company can be identified by the form elements found during
online car reservation. The Web site will expect the user to specify values for its ontology: pickup time, pickup date, return location, etc. Part of the ontology can be retrieved by analyzing the forms identifying field names, its labels and its options.

The process starts with a URL for the main page (e.g., http://www.avis.com). The system retrieves the page from the Web site and parses the page in a DOM tree. The parsing of the page is done using a DOM-compliant parser that produces a DOM document as specified by the W3C DOM Level-3 specification [40]. There are many libraries that produce a DOM tree from an HTML page, so the process is independent of the parser used, which means that any available parser can be used as long as it is DOM-compliant.

Due to the late adoption of the HTML standards (HTML 4.01 is the latest one by the time this document was written) by the industry, some Web sites use browser proprietary tags to design their Web pages. Also, because most of the browsers are required to ignore errors in Web pages, some Web sites have pages with errors that will obstacle the parsing and creation of a DOM tree for the page (e.g., closing of tags using a different order than the order used to open the tags, opening of tags without the closing tag, etc.) All these factors are considered “noise” in the analysis of the DOM tree to identify HTML elements and labels for the ontology. In order to avoid this, the DOM tree is filtered before it is analyzed. The filtering process consists of the elimination of superfluous tags not required by the ontology creation process; tags used for formatting and scripting are removed from the DOM tree. Also, most of the errors in the page are identified and corrected (if possible) by the parsing library. This results in a cleaner DOM tree.
Given a DOM tree structure, the identification of the forms and the input elements is straightforward. Problems arise trying to identify the label for an input element. The lack of structure of HTML pages and the diversity of layouts by which forms are designed in a page makes the label identification process difficult. Some heuristics based on common layouts designs are used to improve label extraction. These heuristics are developed using a training set (simple HTML pages containing the most used layouts for form design) and then applied to the real HTML pages. Basically, two main layouts are identified: a table layout and a non-table layout. For each layout, the label can be specified as text or as an image. Also, for each layout, the label can be located above the form field or to the left of the form field. Not all the labels for the input elements can be successfully identified because there is no rule about the formatting and layout of forms in HTML. However, we believe that the label identification process has a high percentage of success. The label for the elements can be combined with the name of the element in order to improve the matching of terms in the ontology merging process. In addition to the label, the contents (i.e., the options) for input elements like the select, radio and checkbox can also be retrieved. The contents of a field can be successfully used in the matching of terms for the ontology merging process.

At this point, every input element in the forms should be identified by the following fields: (1) the form to which it belongs (identified by the form action URL), (2) the field name, (3) the label (if successfully identified), and (4) its contents (in case of a select, radio or checkbox field). The ontology for a Web site can be spread across multiple pages
in the site. Site navigation is performed (with the help of the user), searching for forms in the pages visited. When the navigation finalizes, we will be able to produce an XML file containing an ontology for the Web site. Figure 3.3 shows an example of an ontology extracted from an HTML page.

Figure 3.3 Ontology extraction from an HTML page
CHAPTER IV

AN OO EXTENSION OF XML FOR ONTOLOGICAL APPLICATIONS

In chapter III and in a recent research [62], we have shown that autonomous machines can create ontologies for Web portals based on information extraction techniques with acceptable accuracy (precision and recall). This development is interesting because dynamic and decoupled inheritance can be used to design better portals where generalization is the goal as opposed to specialization. In such a scenario, ontologies are created from the publicly accessible Web pages of similar applications.

Consider building a Web portal for car reservations (similar to portals like Expedia\(^1\) or Travelocity\(^2\)) where the user can check car rental deals for different companies on one Web site. A common ontology is constructed in two steps: (i) create local ontologies for each car rental applications to be included in the portal, and (ii) create a common ontology described as the intersection of all the local ontologies created in the first step. However, there could be some interesting properties that some of the member sites share and the rest do not. In exceptional cases, it may be beneficial for the global application to add some default properties for all sites which, at some future point in time, the individual sites may want to add, and thereby override the defaults. In such cases, global ontologies will be

\(^{1}\text{http://www.expedia.com}\)
\(^{2}\text{http://www.travelocity.com}\)
created by the client portal on accessing the relevant information from the server Web site, and by properly adding the defaults. In this setup, user queries are submitted against the global ontology and are translated for the local Web servers.

4.1 XML++: An Object-Oriented Extension of XML

The reliance on XML as the de facto standard for data representation and electronic data interchange has motivated both academic research endeavors and industrial developments. These capabilities have even been exploited in areas such as ontologies and phyloinformatics, in addition to their use in document databases. XML has also played major roles in creating new languages. In fact, W3C recommendations describe XML as “a meta language to create new languages” [14]. Its popularity has led to an increasing number of standards that have been created using XML as the meta language specification, such as SOAP for information exchange [60], MathML for mathematical notation representation [6], SMIL for multimedia authoring [7], while newer standards continue to emerge. With the support of a vast array of tools and technologies, XML is also being used for data formatting and styling (with e.g. XSL/XSLT [3]), document querying (with e.g. XQuery [11]) and schema representation (with e.g., XML Schema [27]) for document databases and Web portals.

While the representation and electronic interchange capabilities of XML have been used and investigated quite well, the “re-use” aspect of XML remains rather neglected. Traditionally, inheritance has been the key to re-use and express specificity in specializa-
tion/generalization hierarchies. While there have been some efforts in incorporating the so called object-oriented features in XML, nothing substantial has been reported in XML standards or in the literature.

In this chapter, we propose an extended XML, called the $XML^{++}$, to include a subset of object-oriented features such as dynamic inheritance, and methods. Our contributions parallel other existing extensions to XML such as SOX [24], but as opposed to SOX, we extend the XML data model itself while extensions such as SOX are orthogonal to XML. By that we mean that these extensions exploit object-oriented concepts for schema validation only but fail to extend to document levels. As a consequence, they have little to do with document design and structuring. Our extensions will allow re-using existing documents in other remote documents by treating them as class templates. In this approach, documents will inherit features from other documents autonomously and without intrusion.

We take the position that our extension should also be non-intrusive to current XML documents and standards. As a result, the concept of inheritance and methods are included in the model at the instance level (XML documents) as opposed to class levels (DTDs). Since documents on the Web change frequently, and a document can re-use other documents autonomously, it is imperative that we adopt a dynamic inheritance model where the inheritance of properties will take place at runtime. Our contention here is that once

---

3The name $XML^{++}$ has been used in the literature in several contexts and may not be unfamiliar to the reader. But, to our knowledge, the features and extensions we propose here are unprecedented. We have chosen this name for our language since it seems appropriate.
we settle for dynamic inheritance as the object hierarchy cannot be determined at compile time (since the servers and clients know nothing about the hierarchy until execution time), traditional inheritance models and approaches fail.

4.1.1 Static Versus Dynamic Inheritance

Static inheritance is adopted in many programming languages and database systems for efficiency related considerations, and a critical assumption is made that the class default values in particular are stable (do not change). It is understood that in the event of a change, a massive and extremely complicated reorganization will have to be made. It is needless to point out that when the hierarchy is known only at run time, dynamic inheritance is the only choice. In Web applications where (class default) changes are very frequent and autonomous, a static inheritance model will perform very poorly. In dynamic inheritance model, class values are inherited at run time allowing the possibility for class updates, and thereby eliminating the need for reorganization altogether following a class default value update. By adopting such a model, we eliminate the traditional inheritance problems with respect to default value updates that persist in static inheritance models, and support increased autonomy and decreased management overheads.

4.1.2 A Motivating Example

Let us discuss an application where dynamic inheritance can help in the context of document structuring. Consider designing a personal Web page by re-using someone else’s
Web page as a template – not by copying but by inheriting the properties such as layout, background color, and style of the source Web page. The expectation here is that if the source page properties change, the page using the source should also change except for the ones that are specifically overridden. The basic idea is depicted in Figure 4.1. In this figure, an abstract class representing a general home page is designed using XML++. Several other Web pages are then created remotely and autonomously as instances of the top page that is regarded a template. The lower pages appropriately override a subset of the template properties to customize themselves. Any changes applied to the template or the source page will now extend to the lower pages too, which are now considered subclasses or instances of the source page. This inheritance is user transparent at least for the source page which is assumed to be unaware of the re-use or inheritance. The lower Web pages in the hierarchy can override attributes or properties of the superclass such as background color or location of the photo in the page.

In Figure 4.1, the Web pages for Sandra and Kate override page attributes such as background, name, and photo while they inherit the rest of the page structure from the superclass. Subclasses may also add additional properties or object classes. For example, John’s page inherits all the properties (with overriding) and also adds additional declarations for new object classes such as Photoalbum and Hobbies.

This seemingly simple concept can be applied to form many complex applications in eCommerce. Take the case of a multinational company which has a number of different branches distributed all over the world. Each of its branches has the autonomy to manage
and maintain its own version of order and billing documents in XML. In this scenario, it is conceivable that the headquarters could create an $\text{XML}^{++}$ template of an order document and let the branches extend it with the appropriate information specific to each branch. As we will show later, this extension is simple yet powerful enough to add value to many applications, specially for ontology representation.

### 4.2 Related Work

Object-orientation is not a new concept in programming language and database paradigms, but in the context of XML, it remains fairly unexplored. It is worth pointing out again that most research and practical systems that addressed this issue considered only static inheritance models. Recently, several researchers have addressed the issue of object-orientation in XML documents. Many of these proposals and notes were made to the W3 Consortium [85]. We will discuss only two major proposals: XML Schema [27] and SOX [24], keep-
ing object-orientation in our sight. We have adapted some of these concepts in XML++
with substantial enhancements that strengthen XML Schema and SOX.

4.2.1 XML Schema

XML Schema [27] is an approved recommendation from the W3 Consortium. Designed
to overcome the limitations of DTDs, XML Schemas are used to define the structure,
contents and semantics of XML documents. XML Schema is divided in two parts: XML
Schema Structure to specify the structure and constraints of XML documents, and XML
Schema Datatypes used to represent and validate complex data types in the definitions of
XML documents. Of these two, the latter is the one that presents a few object-oriented
capabilities from the point of view of abstract data types. As an example, consider the
following XML Schema fragment:

```xml
<complexType name="personType">
  <attribute name="fname" type="string" />
  <attribute name="lname" type="string" />
  <attribute name="gender" default="male">
    <simpleType base="string">
      <enumeration value="male" />
      <enumeration value="female" />
    </simpleType>
  </attribute>
</complexType>

<element name="person1" type="personType" />
<element name="person2" type="personType" />
```

As the reader can see from the above fragment, XML Schema data types provide fa-
cilities to specify the declaration of data types similar to object-oriented languages like
C++ and Java. To make an analogy, the \texttt{personType} type constitutes a class in a language like Java, while the elements \texttt{person1} and \texttt{person2} constitute instances of the \texttt{personType} class. However, XML Schema is far from being considered an object-oriented language extension to XML, since it lacks most of the capabilities (see section 4.3.1) considered essential for object-oriented languages. As we discuss next, SOX extends the XML language a bit further compared to XML Schemas by including inheritance and polymorphism.

4.2.2 SOX

SOX (Schema for Object-Oriented XML) [24], currently on version 2.0, is a language (or as defined in its note “a meta grammar”) which extends the XML DTDs by supporting object-oriented features such as: (i) data types, (ii) inheritance, (iii) namespaces, (iv) polymorphism, (v) embedded documentation and (vi) distributed schema management. It is not our intention to describe each one of these features (for that we refer the reader to [24]), but instead we will present the example below adapted from [24] that shows how inheritance is handled in SOX.

```xml
<elementtype name="note">
  <model>
    <element type="p" occurs="+" />
  </model>
</elementtype>

<elementtype name="datednote">
  <extends type="note">
    <append>
      <element type="date" name="adate" />
      <element type="time" name="atime" />
    </append>
  </extends>
</elementtype>
```
In the above example, a base type note is defined as containing only one paragraph element, while the subtype datednote extends the definition of note by adding two new elements: a date and a time. Using the above declaration, a new data type of type note can contain instances of both the base class and the subclass as shown in the following code fragment:

```xml
<elementtype name="multinode">
  <element type="note" occurs="+" />
</elementtype>

<multinode>
  <note>
    <p>This is the base class</p>
  </note>
  <datednote>
    <p>This is the subclass</p>
    <adate>20010918</adate>
    <atime>19:30:00</atime>
  </datednote>
</multinode>
```

It is, however, possible to simulate similar functionalities in XML Schema using a different syntax. But it is a simulation nonetheless, not a language feature. Although SOX provides a great deal of object-oriented capabilities for XML documents, these capabilities are defined from the point of view of XML elements, and not from the point of view of XML documents as a whole. It is here that XML++ introduces a new set of features that enriches the language defined in SOX.
4.3 XML++ Overview

In this section, we highlight the major features of XML++ that make our system unique and useful in contrast to XML, XML Schema, and SOX. We first start with an intuitive description of the features captured in the XML++ data model. Then we present an overview of the syntax and semantics of some of the major features in XML++.

4.3.1 Salient Features of XML++ Data Model

Following are the major object-oriented concepts we adapt for XML++. A survey of object-oriented modeling constructs may be found in [43, 39, 76].

- **Object Classes**: objects that have the same variables (or attributes), data types and structure, respond to the same messages, and use the same methods are grouped to form *classes*. Objects derived from the class are called *instances*.

- **Inheritance**: Usually, different classes share a common structure while differing in just specific attributes. It would be desirable to abstract the common structure of those classes under the concept of a superclass from which subclasses can be extended by inheriting the entire structure of the superclass. Subclasses can be extended with specific and additional attributes and data types to further specialize the subclasses. Inheritance is defined in the literature as *Is-a* relationship.

- **Multiple Inheritance**: An object can extend more than one class when required. However, conflicts may occur in the inheritance process due to multiplicity of properties inheritable from multiple classes in distinct branches in the inheritance hierarchies. As an example, suppose that a class subclass inherits from two superclasses superclass1 and superclass2, both defining an attribute X. Suppose that in superclass1, X has a default value of A and in superclass2, it has a default value of B. Which default value should the subclass object inherit for the attribute X? Similar situations arise when class methods are concerned. In section 4.3.2, we take up this issue again to explain how XML++ deals with these issues.

- **Default Values**: Class attributes can specify default values in the domain of the attribute type. These default values must be inherited from superclasses to subclasses, and then to instances. Optionally, subclasses (or instances) can override the default
values of attributes by specifying a new default on the subclass declaration (or a specific value at the instance).

- **Object Identity**: Every object in the system has an unique identifier either generated by the system or specified as part of the object definition. Object identities are used in object references between classes in the system. Object identity is currently part of the XML specification [14, 25]. XML elements can be created with an ID attribute, which is enforced for uniqueness in the document. Elements can reference existing elements identified by a unique ID using the IDREF or IDREFS element types in the DTD declaration. External elements (elements defined in other XML documents) can be referenced using standards like the XLink and XPointer [25].

- **Methods**: Classes define methods (or functions) that execute specific actions to the instance of the class. As we will explain in section 4.3.5, methods should implement a required interface in order to be used by classes. Methods must be defined outside of the XML++ document using an existing programming language.

- **Encapsulation**: Encapsulation provides an abstraction on the definition of the objects. Objects are seen as black boxes with a well-defined interface. *Data hiding* is the main concept behind encapsulation. Encapsulation applies to both classes and methods. By using encapsulation, class implementations can be modified as long as the interface to the external world is respected. It is worth noting that XML++ documents are created on the fly by the middle-tier engine (see section 4.4) and a runtime final document is never materialized. On the other hand, documents extending existing XML++ templates that define methods can make use of them in a user transparent way; subclasses do not need to be aware of the method implementation details in order be able to use them, extending the superclass is enough.

### 4.3.2 Syntax and Semantics

In this section, we present the XML++ syntax and its semantics through examples. The XML++ syntax is largely based on XML, and the extensions introduced in this thesis can be suitably used to describe XML++ documents. This means, XML engines cannot, in general, interpret XML++ documents, and thus, to be able to interpret XML++, a new interpreter is necessary. Our solution to this issue is that we exploit existing XML engines and support tools such as XML Schema and DTDs. While we actually use XML Schema
for the implementation, for lack of space and simplicity, we will describe XML++ features using only DTDs. We chose to present our ideas using DTDs for the simple reason that it is generally believed that documents designed using Schema are more structured but less intuitive than the ones designed based on DTDs. Though complex, XML Schema provide a richer set of structure definitions and data types which can be used to enhance the creation of XML documents.

```xml
<!ELEMENT document (templates?, element)>  
<!ELEMENT templates (template+)>  
<!ELEMENT template EMPTY>  
<!ATTLIST template  
   url NMTOKEN #REQUIRED  
   prefix NMTOKEN #REQUIRED>  
<!ELEMENT element (attribute*,element*,methods?,elementContent?)>  
<!ATTLIST element  
   name NM_TOKEN #REQUIRED  
   extends NM_TOKENS #IMPLIED  
   final (yes | no) #IMPLIED  
   exclusive (yes | no) #IMPLIED>  
<!ELEMENT attribute EMPTY>  
<!ATTLIST attribute  
   name NM_TOKEN #REQUIRED  
   value CDATA #IMPLIED>  
<!ELEMENT elementContent ((useMethod|text|xmlText|anyContent)+)>  
<!-- Enforced to be valid XML--->  
<!ELEMENT xmlText ANY>  
<!-- Not enforced --->  
<!ELEMENT anyContent ANY>  
<!ELEMENT methods (method+)>  
<!ELEMENT method (class,constructor?,parameters?)>  
<!ATTLIST method  
   name NM_TOKEN #REQUIRED  
   element-reference (true|false) #IMPLIED>  
<!ELEMENT class EMPTY>  
<!ATTLIST class  
   language CDATA "Java"  
   class NM_TOKEN #REQUIRED  
   method NM_TOKEN #REQUIRED>  
<!ELEMENT constructor (constructorParameter+)>
```
As shown in the DTD above, XML\textsuperscript{++} object classes are declared using the \texttt{element} tag. Object classes can contain attributes (using the \texttt{attribute} tag), simple text or XML fragments (using the \texttt{elementContent} tag), or subclasses. Subclass definitions are nested within the parent class. Classes can also contain any kind of data not specified in
the DTD by using the anyContent tag. This approach allows extensibility beyond the designed schema. An example object class declaration is shown below:

```xml
<element name="home-page">
  <element name="title">
    <elementContent>
      <text>Welcome to my Home Page</text>
    </elementContent>
  </element>
  <element name="photo">
    <attribute name="url" value="photo.gif"/>
  </element>
</element>
```

4.3.4 Inheritance

Inheritance is supported by allowing object classes to extend other classes. The element tag uses an optional extends attribute which is used to specify the object class for which it inherits the contents. The inheritance is based on template definitions (which are XML++ documents) by specifying the template prefix and the superclass name in the template. Superclasses in templates must have a unique name in the template scope, so that they can be identified in the subclass document. We will discuss templates in some details. The following example class definition is a subclass of the home-page class identified in the HomePage template:

```xml
<element name="home-page" extends="HomePage:home-page"/>
```

Multiple inheritance is allowed by providing a space separated list of template:class pairs. When multiple inheritance introduces the kind of conflicts explained in section 4.3.1, the conflict resolution algorithm gives priority to templates based on the order in
which they are listed in the `extends` attribute. Of course, there are some cases where this kind of conflict cannot be resolved by using order. As an example, consider the classes `Professor` and `Student`, both defining a salary attribute. The professor class defines a default salary of 50K, while the student class defines the salary as 20K. If John is an instance of `TA` class, which is defined as a subclass of both `Professor` and `Student` (multiple inheritance), then telling John that his salary is 20K, because the student class was listed before in the class declaration, will not be sufficient. In this case, the best solution to handle multiple inheritance is really to not handle it, but just avoid it (either by throwing an error or by not using the attribute at all). In order to accommodate both possibilities, the `element` definition accepts an optional parameter: `exclusive`, which means that multiple inheritance is not supported by this type of element.

In order to use the class definition for other classes, the XML++ document must declare the templates containing the superclass definitions. A document can use more than one template in case different subclasses are defined in the document. Also, by allowing the use of multiple templates we also support multiple inheritance, as explained before. Templates are defined by specifying two required attributes: the `url` containing the XML++ document definition, and the `template prefix`. Prefixes are used to differentiate between multiple templates used in the same document. Following is an example of a document that uses two templates:

```xml
<templates>
  <template url="mariaHomePage.xml" prefix="homePage"/>
  <template url="personInfo.xml" prefix="personInfo"/>
</templates>
```
Alternate ways of specifying templates are possible, and are currently under investigation. The current XML++ constructs for templates makes it difficult to use traditional XML pages (templates that are not designed using the XML++ syntax) for the purpose of inheritance as we discuss here. However, XML content can be used as part of the templates and classes as described in section 4.3.3. By using XML namespaces and entities, traditional XML content can be referenced by means of XLink/XPointer [25, 8] links inside XML++ documents. This approach makes it possible to exploit the extensive collection of XML documents along with XML++ documents. As an example, consider the following fragment where homePage.xml and personInfo.xml are common XML pages:

```xml
<!DOCTYPE home-page [ 
  <!ENTITY hp 'homePage.xml#'>
  <!ENTITY pi 'personInfo.xml#'>
]> 
<home-page xmlns:xpp="http://www.xpp.org#"
            xpp:extends="&hp;home-page πperson-information" />
```

The references to the superclasses are XPointer links to the pages indicated as entities in the DTD. This approach of using templates is similar to the way templates are used in the ZOPE system [77].

Inheritance can also be avoided (which means that elements cannot be extended) by using the attribute final of the element element. If a class tries to extend another class that defines the final attribute, the XML++ parser will throw an error indicating that the class extension is invalid. This modifier can be useful in cases where the owner of the template wants to restrict the use of its document classes as superclasses of extended document instances. This concept has applications in enforcing security in XML documents.
Similar modifiers are implemented in some object-oriented programming languages like Java [64] and ORLog [43, 44].

4.3.5 Support for Methods

XML++ elements can support the use of methods\(^4\) in their declarations. Methods are represented by the `method` element in the XML++ DTD. The implementation of the method must be externally defined using a programming language supported by the XML++ engine. Currently, the only programming language supported is the Java programming language, however, different implementations of the engine can add support for other common languages like C++, Perl, Visual Basic, etc. Figure 4.2 shows how XML++ engines can be extended using language plug-ins to support different programming languages for method execution. The principle behind this approach is reminiscent of the way common database wrappers work (e.g. Java JDBC [64]). The execution of the methods defined in a document is delegated to the method executioner module. This module can be extended by declaring an XML configuration file that contains the list and parameters of each plug-in available to the engine. Every time the module wants to execute a method, it will call the appropriate plug-in depending on the `language` attribute of the method element declaration. The information is exchanged using an XML protocol similar to SOAP [60] between the executioner module and the plug-in. Each plug-in must respect a specific interface

\(^4\)We would like to mention here that we only consider methods without side-effects for now. While methods with updates are still possible in XML++, to effectively and safely use such methods will require additional machineries such as concurrency control, version management, and so on.
in order to be able to communicate with the executioner module. Besides the exposed interface, plug-ins must support some sort of introspection, as defined for the Java language [64]. By using introspection, the executioner module is able to query at runtime for methods interfaces defined inside the class library.

![Figure 4.2 XML++ language plug-ins](image)

4.3.5.1 Introspection

Although supported only by Java, introspection is a general concept that allows programs or libraries to retrieve runtime information about the methods, parameters, datatypes, etc. defined in the program or library. Using again Java as an example, the language provides a Reflection API in its native implementation to support introspection. Some of the methods available in this API are listed below:

```java
Class.getConstructor(...)
Class.getMethod(...)
Method.getName(...)
```
Method.getParameterTypes(...)  
Method.invoke()  
Constructor.newInstance()  

The details about the parameters and exact use for each of the methods and objects in the API is beyond the scope of this thesis, but as we can see from the short list presented above, the API provides methods for retrieving constructor information for classes, parameter information for methods, and even invoke methods based on a string representation of the method. Introspection has also some drawbacks from the point of view of XML++, and the most important one is that since XML++ is just plain text, complex data values are not supported as values for method parameters. In other words, only primitive data types (namely, int, long, float, double, char, boolean and byte) can be used as values in method parameters. Complex datatypes like arrays or complex objects are difficult to represent as text, however the use of an appropriate standard (again defined as XML) for complex datatypes representation can overcome this limitation. Some examples of how to transform complex objects using data binding techniques for XML can be found in [56, 53, 26]. The other drawback is that introspection is not trivial to simulate in such languages that don’t support introspection natively. C++ has support for some sort of introspection, although limited, by using RTTI (RunTime Type Information). Introspection must be simulated for other languages not supporting it.
4.3.5.2 Usage

In order to use a method, the method declaration must contain the name of the class (library) implementing the method represented by the `class` attribute of the `method` element. It is also required to specify the name of the method which is going to be used inside the class, using the `method` attribute. If the class is implemented in a language other than Java, the `language` attribute can be used to indicate it. Both functions and procedures can be used as XML++ methods. Method parameters are specified using the `parameter` element.

The following example is a simple method declaration for a built-in method in the `java.util.Date` class.

```xml
<methods>
  <method name="getDate">
    <class language="Java"
      class="java.util.Date"
      method="toString"/>
    <parameters>
      <parameter mode="return" name="date"
        type="java.lang.String"/>
    </parameters>
  </method>
</methods>
```

Methods can be used in XML++ documents where an `elementContent` element is defined. As an example, the method presented above can be used in a document as follows:

```xml
<element name="date">
  <elementContent>
    <useMethod method="getDate">
      <value name="date">
```
In the XML document resulting from the parsing and processing of the XML\textsuperscript{++} document, the above element declaration will be translated to the following XML fragment:

\begin{verbatim}
<date> 10/13/2001 </date>
\end{verbatim}

Parameters are of two types: input parameters and return parameters, which are used to specify return values for functions. Parameters are identified by a name and a value. At method invocation, all the input parameters must define either a value or an XPath expression for the attribute value. Specifying a value for a return parameter does not make any sense since function results are assigned a value after the method is executed. Parameter values can be defined at declaration time, in which case they are considered the default values for the attributes, or at invocation time. If a method parameter has a default value, then a value specification at invocation time is not required, since the method will use the default as the parameter’s input value.

During declaration time, parameters are required to be listed in the same order as defined in the method definition, in other words, parameter specification is position dependent. To illustrate this, suppose we want to make use of a mathematical method that calculates the power of a number. The method parameters are the base and the exponent, in that order. In this case, the method declaration in XML\textsuperscript{++} must list both parameters in the same order as shown in the following example:

\begin{verbatim}
<method name="power">
</method>
\end{verbatim}
Notice, in the above code fragment, how we first listed the base parameter followed by the exponent and finally, the return parameter. On the other hand, during method invocation the use of parameter is position independent, meaning that method parameters can be listed in any order as long as the name of the parameter is used to specify its value. As an example, consider the following code fragment that invokes the method defined above:

```
<useMethod method="power">
  <argument name="exponent" value="4" />
  <argument name="base" value="2" />
  <value name="power">
```

As the reader can guess, the result is $2^4$. Notice how we inverted the parameters of the method without affecting the final result.

4.3.5.3 XPath Expressions

As mentioned in previous sections, the designer can choose between specifying a hard-coded value for the method parameter or using an XPath expression for the same purpose. The use of XPath expressions as parameter values is very useful for dynamic content creation. As an example consider the following XML++ element definition:

```
<element name="param">
```
The previous code describes an element that defines two parameters: a decimal and a grouping separator. Each parameter has a value that is in accordance with the United States (US) number formatting, e.g., where numbers are expressed using this pattern: #,###.##. Other countries, e.g. Latin-American countries like Venezuela, use a different set of parameters, defining numbers using the following pattern: #.###,## (notice how the decimal and grouping separators are inverted). If we create documents that use number formatting depending on whether the document is for the US or Venezuela, then we need a method that formats these numbers according to the parameters defined in the document. As an example, consider the following method definition:

```xml
<method name="formatMoney">
  <class class="OrderUtilities" method="formatMoney"/>
  <parameters>
    <parameter mode="in" type="double" name="amount" />
    <parameter mode="in" type="char" name="group" xpath="element[@name='param']/element[@name='grouping-separator']/elementContent/text" />
    <parameter mode="in" type="char" name="decimal" xpath="element[@name='param']/element[@name='decimal-separator']/elementContent/text" />
  </parameters>
</method>
```
The method `formatMoney` takes three parameters: the amount to format and the decimal and grouping separators used to format the amount. By using XPath expressions as the parameters default values, we assure that whatever is the value of the decimal and grouping parameters at invocation time, that value will be used for formatting purposes. Documents for the US can inherit US parameters while Venezuelan documents can inherit the Venezuelan counterparts, using only one method definition for both set of parameters. The framework also allows dynamically retrieving values returned by method calls to be used as dynamic content, or even as parameters to other method calls.

### 4.3.5.4 Caller Self Reference

Methods can retrieve information from the XML++ object caller. The object caller is passed as an implicit parameter to the methods in the form of an XML fragment. The method will not have access to other objects in the XML++ document besides the calling object and the subobjects defined in the same scope. XML++ methods must implement a specific interface in order to be able to handle the XML++ fragment being passed to the method. Since not all the methods being used in XML++ use the calling XML++ object, the default behavior is to obviate the passing of the object caller to the method. If a method requires a reference to the object caller, the attribute `element-reference`
must be used with a value of true. The object caller is passed as an instance of a DOM element structure [40].

4.3.5.5 Method Scope

Methods can be defined in three different scopes:

- Template inherited scope.
- Element inherited scope.
- Local scope.

The most specific scope overrides the most generic one, the template inherited scope being the most generic, and the local scope the most specific. For the first scope, methods are inherited from elements defined in other documents defined as templates in the local document. As explained in section 4.3.4, subclasses inherit all the content from its superclass, including method declarations. Following is an example of a method template scope. Suppose that the method getDate presented earlier in this section is defined for an element method in a file called methods.xml, then in a subclass of the document called methodsInstance.xml, an element date can make use of the inherited getDate method as showed in the following code fragment:

```xml
<templates>
  <template url="methods.xml" prefix="m"/>
</templates>

<element name="date" extends="m:methods">
  <elementContent>
    <text>Current date is</text>
    <useMethod method="getDate">
      <value name="date"/>
    </useMethod>
  </elementContent>
</element>
```
The second method scope, the element inherited scope, uses the same concept of inheritance as in the previous scope, with the exception that in this case methods are not defined in another document, but instead in the same document in an element defined higher in the hierarchy. The following code fragment shows how element scope is used:

```xml
<element name="methods">
    <element name="date">
        <elementContent>
            <text>Current date is</text>
            <useMethod method="getDate">
                <value name="date"/>
            </useMethod>
        </elementContent>
    </element>
    <methods>
        <method name="getDate">
            ...
        </method>
    </methods>
</element>
```

The element methods defines the getDate method and contains a sub element called date. Notice how the subelement date is allowed to use the method result as its local content because it is automatically inherited from its superelement methods. The third and last method scope is the local scope, where methods are defined at the innermost element overriding any other scope as shown in the following example:

```xml
<element name="methods">
    <element name="date">
        <elementContent>
            <useMethod method="getDate">
```

Notice how the outer element methods and the inner element date both define the same method getDate. When the inner element uses the method, it is using the method defined in its methods section and not the method defined in the method section of the outer element. This is an example of how method overriding is handled in XML++.

4.3.5.6 Method Invocation

The method invocation procedure is depicted in Figure 4.3. XML++ documents can make references to class libraries defined in other XML++ repositories spread across the Internet. When a local document references a remote class in its methods definition, the XML++ engine will retrieve the remote class using a simple HTTP request. Once the class is locally stored in the engine, the processing of the page continues as if all the methods libraries were local. There is one issue, however, that arises when dealing with methods defined remotely, which is the following: What happens if the method to be
invoked is defined in a programming language other than those supported by the local XML++ engine? In this case, two approaches can be used, the first approach being to simply ignore the method call since the local engine will not be able to handle it. And the second approach being to handle the problem as a common RPC (Remote Procedure Call) invocation. Different protocols can be used for this type of remote calls, including the CORBA [66] standard.

![XML++ method invocation](image)

Figure 4.3 XML++ method invocation

### 4.4 XML++ Middle-tier Engine

In this section, we present details about the middle-tier XML++ engine, its architecture, and the design aspects of its implementation. The framework follows the multi-tiered application framework as specified in [48]. In this framework, the MVC (Model-View-Controller) programming model for Web applications development is used. Although the XML++ engine is implemented using Java Enterprise Edition (J2EE) [12] platform
specifications, its design and implementation can be applied to similar platforms, like for example the Microsoft .NET architecture.

### 4.4.1 System Design

The framework for the engine is shown in figure Figure 4.4. The figure shows a typical three-tier implementation of a Web application: the data tier containing all the information in a database or file system, the middle-tier represented by the Web server and any other business logic components, and the client tier, represented in the figure by a Web browser, which handles the user interface and is responsible for retrieving user input and present system output (in general any client type can be used in the client side).

![Figure 4.4 XML++ framework](image)

As the reader can observe in the figure, the engine is implemented in the middle-tier as an extension to the Web server. The Web server is configured to forward requests for
.xml and .xpp documents containing XML++ instructions to the engine, which after
the appropriate parsing and loading of the XML++ document templates, interprets all the
object-oriented instructions (i.e., complex data types, inheritance, encapsulation, etc.) of
the XML++ document requested. Note that XML++ documents will be required to have
either of the two extensions mentioned before in order to be processed by the engine, all
other documents will be handled by the host Web server as normal requests. Of course,
the transport protocol used between clients and the engine is the typical HTTP protocol.

The strategy implemented by the XML++ engine will be similar to the strategy used
in object-relational databases [76, 52]. Relational databases provide object-oriented capa-
bilities to DBMS by providing an extended language built on top of the traditional SQL
language. The object-relational engine is responsible for translating between the object-
oriented database language and the SQL language, objects and type definitions are stored
in relational tables providing only an object-oriented view to the user. In this same fashion,
XML++ documents are converted (or, by using a more appropriate term, “rendered”) to
traditional XML documents. All the object-oriented instructions are handled by the en-
gine and converted to XML documents which are then processed by an XML parser such
as JDOM [42] with Xerces.

One side effect of the adopted execution model\(^5\) of XML++ engine is scalability. In
complex scenarios, a deep hierarchy of classes spread across multiple servers can slow the
response time for the displaying of Web pages at the bottom of the hierarchy. Two ap-

\(^5\)Which of course is intimately dependent upon the XML++ data model that features dynamic inher-

ance.
proaches can be implemented in this kind of scenario. The first approach consists of re-collecting all the participating classes or pages in the Web server that initiated the request, resolving inheritance locally and then producing the result. The other approach consists of the application of inheritance in a hierarchical way. To illustrate both approaches, suppose that a user is requesting a Web page A, which inherits information (or uses methods defined) in a page B, which in turn is a subclass of page C. In the first approach, the server containing page A retrieves both pages B and C from their respective servers, and uses them locally to resolve references to these pages in page A. In the second approach, the server containing A issues a request to the server containing B, which in turn resolves any reference that B may contain to objects in page C in another server. Each scenario has its advantages and disadvantages in terms of processing time and communication overhead. We are currently evaluating optimization strategies that include object caching to leverage system scalability and response time.

4.4.2 System Implementation

The engine was developed using the Java programming language for portability, abundance of predefined libraries and tools, and because Java supports better abstraction capabilities for object-orientation [64]. Another relevant feature inherent to the Java language is the built-in support for introspection, defined as the ability for a Java class to inspect itself, allowing to query at runtime, the methods and parameters implemented by specific classes. As we mentioned in section 4.3.5, this feature is very important for the execution
of methods in XML++ documents. The choice of Java is also obvious since the object-oriented concepts implemented in XML++ are based on this language. As mentioned before, the J2EE specification is used for the implementation of the XML++ engine. This means that all the different components of the engine are developed using Java Servlets and JavaServer Pages. Following the MVC model, Figure 4.5 shows how each component is developed in the J2EE platform.

![Figure 4.5 Architecture for the XML++ middle-tier engine](image)

Each object-oriented feature (i.e., data types, inheritance, etc.) is handled by a different module. The XML++ parser is responsible for loading the appropriate module depending on the type of object-oriented instruction being parsed. Using this modular approach al-
allows for the implementation of new object-oriented features in future versions of the engine, without compromising existing ones. The XML++ servlet is the controller for the engine. It is registered to process every request with an .xml or .xpp (standing for XML plus plus) files. Before the file is delivered to the client response object, the requested file is retrieved from the engine Web context and passed to the parser. If the file defines templates, each template is treated as a new request. Once all the templates are resolved and parsed, the requested XML++ document is returned to the client. If an error occurs (such as a parsing error), the application forwards the control to a predefined error page, which is responsible for showing an appropriate error message to the client. Templates can be resolved either in the engine context (local to the Web server where the engine is installed) or to an external context through the Internet (e.g., templates contained in other XML++ engines).

In order to parse XML++ documents, the JDOM [42, 57] library using Xerces XML parser is used. JDOM is compatible with the SAX [58] and DOM [40] parsing models for XML, combining the best features of both models. It combines the speed of a SAX parser with the versatility and convenience of a light data structure similar to the DOM structure for element manipulation. Again, the use of JDOM as an XML parser is not a requirement at all; other XML parsers like XT or MSXML can also be used for the same purpose. The pseudocode used by the engine to parse XML++ documents is presented next:

Algorithm: XML++ parsing
Input: URL request for the XML++ document
Output: XML document stream
buildDocument(URL)
{
    validate_XML++_document(URL)
    templates=getDocumentTemplates(URL)
    create_rendered_XML_document(doc)
    processElements(URL.getRoot(),templates,doc)
}

getDocumentTemplates(URL)
{
    HashTable templates
    for each template t in URL
        prefix=getPrefix(t)
        url=getURL(t)
        templates.add(prefix,url)
    return templates
}

processElements(element,templates,doc)
{
    if element is <element>
        if elements is subclass
            element=inheritance(element,templates)
            doc.create_element_tag(element)
        if element is <method>
            doc.execute_method_tag(element)
    ....
    for each child c of element
        processElements(c,templates,doc)
}

The algorithm “renders” an XML++ document into simple XML document. The first step is to parse and validate the XML++ document against a Schema or DTD specification like the one presented in section 4.3.2. If the document is valid, then all the templates defined, if any, are retrieved and stored in a hash table using the template prefix as the key, and the template URL as the value. The root of the document is retrieved and then processed by a recursive subroutine. Note also how the result XML document is created and
passed as a parameter to the `processElements` subroutine. This routine is responsible for resolving each element according to the type (i.e., element, method, attribute, etc.). Inheritance is resolved by retrieving the URL using the templates’ hash table. If an element extends from some template, then it must declare the prefix of that template. Methods are handled as explained in section 4.3.5. The routine is repeated recursively for each child of the element being processed.
CHAPTER V

HIERARCHICAL ONTOLOGY REPRESENTATION

Most eCommerce applications (including B2B, B2C, and P2P) rely heavily on numerous Web services and technologies for their day to day operations. Of them, creation and maintenance of ontologies is a key function for the survival of the majority of the enterprises that depend on some form of interoperability and information interchange over the Web. Typical real life ontologies consist of numerous intricate rules, exceptions, and integrity constraints, in addition to the generalized “global schema” of the domain of discourse. The global schemas usually capture the structure, relationships, semantics and other essential meta information about the application along with site specific mapping functions.

We would like to point out here that P2P and B2B applications truly rely on well designed ontologies so that higher-order ontologies (ontologies of ontologies) can be built. For these kinds of applications, such as Expedia.com or Travelocity.com, standardized ontologies show promise. However, despite perceived advantages there are several practical barriers in using such standardized ontologies. Readers may refer to [67] for a comprehensive exposition to these issues. However, one can summarize the issues cited in [67] as follows: wide variation in commercial practices, business complexity, possible limitation
of action on the market, and Internet security. These are considered to be the most common obstacles in using standardized ontologies. Consequently, a significant number of research groups are addressing these issues, and reasonable solutions are yet to be found. So, customized ontology design still remains the focal point for interoperable systems.

While research and developments in ontology design, creation and maintenance are requiring much less customizations than before, enterprises are still forced to design and maintain custom built ontologies to suit their unique business needs. Such case specific ontology designs entail substantial constraints, both in terms of time and cost. Several recent research, including [62], have attempted to reduce such constraints by developing almost automated agents to create ontologies with sufficient accuracy. Semi-automatic mapping function generators [20] and their integration with traditional relational databases [19] show true promise in the direction of fully automated ontology generation. Such promise also insists on sophisticated abstractions that autonomous agents are capable of exploiting during ontology generation and representation process. One such abstraction is inheritance that we leverage in our research on automated ontology generator agent, OntoBuilder.

Recent XML approaches in ontology designs based on RDF [50] and DAML [22] lack the power of superior modeling constructs, inheritance for instance, available in object-oriented paradigm. Consequently, large ontologies generated by these systems are complex, less intuitive, error prone due to maintenance and update complications, and expensive. These systems fail to leverage the inherent structuredness and extensibility of
ontologies in general, and remain unaware of the associated maintenance overheads entailed by the ontologies they generate that does not exploit higher levels of abstractions such as inheritance.

5.1 Ontologies

It is probably very difficult to find a definition of ontology that most researchers can agree on. In its simplest form, Gruber defines ontology as “a specification of a conceptualization” [36]. In particular, in the context of Web applications, the global schema and related meta information required for the effective use of the global schema can be perceived as the “conceptualization”, and the instrument to encode and represent that conceptualization can be regarded as the “specification” in Gruber’s definition [36, 37].

Usually, ontologies can be conveniently described as a set of terms (vocabulary), having defined relationships among them, and an assumed semantics. Descriptions of relationships may take several different forms, and the terms and relationships may be combined to construct higher level structures. In the following few sections, we will introduce several such structures that are commonly found in modern ontologies. But before we present these structures, we would like to draw readers’ attention to a distinction we make between hierarchy of terms and taxonomy. Taxonomy (subclass-of relationships) is a grouping mechanism for objects with shared properties and it is one among many other constructs used in describing an ontology. Constructs other than taxonomy used in defining semantics of ontology includes axioms, rules, constraints, and exceptions.
In the next few sections, we will present a brief discussion on the ontological structures before we address the main issue of this chapter – ontology representation based on inheritance.

5.1.1 Ontological Structures

As we have mentioned before, ontologies are usually described with the help of a vocabulary of related terms that have a well-defined semantics. Structures that help define the semantics is the focus of this section. We are adapting a subset of structures discussed in [15] for our purpose, as not all are relevant for describing ontologies in XML. Interested readers may refer to [15, 16, 86] for a more complete discussion on this subject.

5.1.1.1 Classes

The structure *class* is similar in concept to the notion of classes in object-oriented systems. A *class* (also known as a term) represents a group of similar objects that share common properties. Classes can be further organized in a specialization and generalization hierarchy in a way identical to object-oriented systems. In such a hierarchy, classes higher in the hierarchy are called the superclasses of the classes below them, and those classes below the superclasses are known as the subclasses. Objects that belong to the classes are called instances. Figure 5.1 depicts an example of a class structure.

In the example in Figure 5.1, the class *Car* represents a general car object while the subclass *Ford* is a specialization (as objects in *Ford* class have properties special to them
that no other car in Car class shares) of the general car class. Finally, an individual car MyFord is an instance of the subclass Ford. But by virtue of the fact that Ford is a subclass of Car, MyFord is also a car.

5.1.1.2 Slots

A slot structure captures the relationship between two terms or classes of objects and can be viewed as a specialized function. Slots accept an object from a domain $D$ and maps it to an object in another class (range $R$), i.e., $s : D \mapsto R$. This is actually analogous to the concept of a property or attribute in relational databases. Figure 5.2 shows an example of a slot. In this example, the slot Father maps every object of the class Person to an object in the class MalePerson.
5.1.1.3 Functions

A function structure is a generalization of slots. It is also essentially a special kind of mathematical relation. Like a slot, a function maps terms to other terms, but unlike slots, it now accepts more than one argument, i.e., \( f : D_1 \times \ldots \times D_n \mapsto R \). Figure 5.3 shows an example of a function. In this example, the function \text{Car-Price} takes two terms as its arguments and returns, say a real number (an element of the range \$).

![Figure 5.3 A function example](image)

5.1.1.4 Axioms

Axioms are usually used to capture relationships otherwise difficult to express using slots or functions. For example, domain knowledge, constraints, implications, exceptions, and so on, are examples of relationships commonly represented as axioms. Axioms are usually expressed in some form of logic based languages such as first-order logic or description logic. For example, the assertion "all cars (i.e., instances of the class \text{isCar}) that costs
less than $15,000 (i.e., \( \text{cost} \leq 15,000 \)) belong to the economy class category (i.e., member of the class \text{isEconomyCar})” can be expressed in first-order logic as follows:

\[
\forall x (\text{isCar}(x) \land \text{cost}(x) \leq 15,000) \implies (\text{isEconomyCar}(x))
\]

In this example, \text{isCar} and \text{isEconomyCar} are classes, \( x \) is an instance, and \text{cost} is a slot.

5.1.1.5 Composition

Similar to user defined data types in programming languages, the composition structure allows grouping of related terms together to form a complex new term or class. The criteria used to form compositions vary widely. They can be based on similarity of domain values, semantic similarities, etc. As an example consider the composition constructor shown in Figure 5.4. here the term Reservation is a composition of two other terms, Pickup and Return, which in turn are compositions of other terms.

![Figure 5.4 A composition example](image-url)
Composition can be very useful in the identification of similarities between structures. Terms that cannot be related using linguistic methods (e.g. substring matching, etc.) during ontology generation process can be related based on their position in a structure tree and the structural similarity of the compositions.

5.1.1.6 Precedence

It was observed in [62] that many Web interfaces use page structures that depend on the user input or responses on the previous page. This page precedence in many cases determine the ontological make-up for most Web applications. This holds true when automated ontology creation is considered, as in OntoBuilder (see chapter VII). This observation led to the introduction of the precedence structure that was not considered in the Bunge [15, 16] approach. To facilitate page navigation and automated interaction with already established Web forms, we keep page precedence relationships. Basically such structure is a binary relationship of the form $p_1 < p_2$ between Web pages or terms where $p_1$ and $p_2$ are terms. Intuitively, it means $p_1$ precedes $p_2$.

5.1.2 An Example of a Global Ontology Creation

We now introduce a simple example on global ontology generation based on a set of component data sources. Consider two fictitious car rental companies CompanyA and CompanyB, as shown in Figure 5.5. These two companies have identical concepts of a customer or client. However, the terminology used to describe it is not identical. Figure 5.5 shows
the local ontologies for both companies. Notice the difference in vocabulary used in the
description.

![Local ontologies for two companies](image)

**Figure 5.5 Local ontologies for two companies**

Using information retrieval techniques, it is possible to establish the correspondence of the terms (or slots) such as *first_name* and *fname*, *email* and *address*, and so on. It is also possible to discover the fact that the term *Client* is a synonym of the term *Customer*, based on the structural similarities of both local ontologies. However, it may not be possible to determine the correspondence between *Member* and *VIP Customer* so easily. In case the system fails to identify them as the same concept, it may decide to treat them as two distinct subclasses of the class *Customer* as shown in Figure 5.6 which essentially is a merging of the two local ontologies. It is important to note here that terms selected for the merged ontology may depend on the order in which the local ontologies are merged or how the domain expert interacts with the system during the merge process. In *OntoBuilder*, the terms selected for the merged ontology are taken from the first ontology.
in the order they are chosen for the merge. However, the final ontology (as a structure) is independent of this choice.

5.2 Representing Ontologies in XML

Several standards for ontology representation exist that allow knowledge processing and manipulations. Some of the most popular ones include Stanford’s Open Knowledge Base Connectivity (OKBC) API [68], Ontolingua [28], and the CLASSIC Knowledge Representation System [13]. More recently proposed XML and RDF [50] based DARPA Agent Markup Language (DAML) [22] are also showing promise and gaining acceptance.

In general, any language capable of expressing and manipulating a required set of ontological structures, such as the ones presented in section 5.1, can be used to represent ontologies. For the purpose of this thesis, we do not particularly commit to any such language for ontology representation. Instead, we attempt to present a more abstract ontolog-
ical representation framework based on a more commonly used language such as extended XML. Such an approach makes it possible to develop and use a suitable mapping function to map the XML representation to any (or all) desired representation standard. We believe that the way ontologies are represented in our approach as XML files makes it easier to convert our representation to target standards with the help of languages such as XSL and XSLT [49] compared to many other contemporary approaches.

We present below an XML DTD\(^1\) that we use to validate ontological structures presented in section 5.1. The validation against this DTD ensures that the structures represented for an ontology are the ones expected and the ones that the system understands.

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ontology (classes, terms)>
<!ATTLIST ontology
   name CDATA #REQUIRED
   title CDATA #REQUIRED
   site CDATA #IMPLIED>
<!ELEMENT classes (class*)>
<!ELEMENT terms (term*)>
<!ELEMENT class (domain, attributes, axioms, subclasses)>
<!ATTLIST class
   name CDATA #REQUIRED>
<!ELEMENT attributes (attribute*)>
<!ELEMENT axioms (axiom*)>
<!ELEMENT subclasses (class*)>
<!ELEMENT domain (entry | term)>
<!ATTLIST domain
   name CDATA #IMPLIED>
<!ELEMENT entry ANY>\(^1\)

\(^1\)In general, XML Schemas provide a better representation and validation support compared to DTDs. However, representations using XML Schema tend to be less compact and more complex than their DTD counterparts. Hence, in this thesis, we chose to use DTDs as opposed to XML Schema although in our actual implementation we use XML Schema for the power and flexibilities offered by it.
Readers may notice the way the above DTD captures the ontological structures such as classes, terms, axioms, relationships (i.e., functions) and attributes (i.e., slots) as introduced in section 5.1. We can now exploit this DTD to express a simple ontology involving only one class and one term as shown below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ontology SYSTEM "dtds/ontology.dtd">
<ontology name="Company" title="Ontology"
    site="http://www.company.com">
    <classes>
        <class name="input">
            <domain />
            <attributes>
                <attribute name="name" value="">
                    <domain />
                </attribute>
            </attributes>
        </class>
    </classes>
</ontology>
```
From a conceptual point of view, ontologies in our framework can be generated from the visual production rules presented in Figure 5.7. This figure captures our approach to ontological structure representation in which an ontology is basically represented as a hierarchy of classes (and subclasses) and terms (and subterms). Notice that in this view, a term is an instance of a class, and as such, it inherits all the properties (axioms and attributes) from its parent class. Classes can be further specialized in subclasses and here too, subclasses inherit all the properties in the superclass. In both cases, rule of specificity applies, and thus, subclasses (or term or subterm) override properties of parent classes if they have properties local to them in a way similar to object-oriented languages. Furthermore, terms can be expressed as a grouping of complex terms through subterms relationships.
5.3 The Role of Inheritance in Ontology Representation

The power of inheritance can be exploited in ontologies in two principal ways. First, inheritance can be used as a tool for hierarchical ontology design in modules. In this approach, global ontologies are created as an abstraction of local ontologies and local ontologies actually become specialization of the global ontology. Global ontologies can be created as a collection of abstract classes that encapsulate all the common characteristics of the application being modeled. Local ontologies can then be rewritten as specializations of the global ontology for that application. In this scenario, inheritance allows propagation of updates and changes at the global ontology to the local ontologies in an automated fashion, and thereby reduces maintenance overhead and complications. Figure 5.8 shows
an example where two ontologies (for an application) are expressed as specializations of a global ontology that exploits inheritance.

![Ontology Inheritance Diagram]

**Figure 5.8 An example of ontology inheritance**

Secondly, in many B2B, B2P, P2P and also B2C applications, where ontologies are created almost autonomously from a set of source Web pages or ontologies \(S_1, \ldots, S_n\), the responsibility of assigning the meaning for the global ontology \(O\) and its proper functioning rests with the user (or the generator) of the global ontology \(O\). This is mostly because the sources in this scenario are truly unaware of such generalization being made at \(O\) and remain autonomous and independent of \(O\). For this reason, generalization of the sources usually will result in a loss of Membership Info of CompanyA in our ex-
ample in Figures 5.8.a and 5.8.b. Instead, if we choose to structure the ontology $O$ as shown in Figure 5.8.c, this loss can be prevented. All that now remains to be done is to design suitable mapping functions to actual sources CompanyA and CompanyB from the corresponding ontologies in Figure 5.8.c.

In terms of ontology maintenance, the second approach will have advantages, specially when systems such as CoopWare [33], WebCQ [21], or WebMonitor [72] are used to autonomously maintain the global ontology. In this case, most changes in any particular site can be accommodated without major changes in global view as source specific information can be better represented by pushing the changes in the component subontology corresponding to the site undergoing changes unless it truly forces a total reorganization. However, in the remainder of this chapter, we do not particularly opt for either one of these possible approaches, rather we present the tools necessary for the design ontologies using any one of these methods.

5.3.1 Dynamic Inheritance

Most object-oriented systems in practice use a static inheritance model where class hierarchies are usually known ahead, making it possible to close the inheritance for the propagation of the default values along the hierarchy and compiling the classes. Database systems such as DB2 and Oracle take this approach, and so do programming languages such as C++. This approach is efficient and has less run time undesirable consequences such as exceptions, and inefficient execution. In this approach, overriding of default values is han-
dled at the object creation time using object constructors. But when object hierarchies are not known ahead (such as in the case of autonomous internet sites), inheritance and overriding can only be resolved at run time giving rise to the idea of dynamic inheritance.

The unsuitability of static inheritance model in frequently and autonomously changing class hierarchies, such as Web applications, can be shown using a simple object-relational database example. In object-relational databases, users can create complex types to be used in table schemas [76]. Types that can declare default values for some of the attributes that will be used in case specific values are not supplied at tuple creation (or insertion) time. In other words, class default values are inherited by instance tuples. Once the classes and the associated hierarchy is compiled and the database has been used for some time, class default cannot be changed easily even if the application necessitates such a change (because of changes in the business rules). This is because the tuples that were inserted using the old default value before the change cannot be identified as some tuples may have an identical value but are not inherited from the class. Notice that had the default inheritance taken place every single time when the tuples are accessed, as opposed to when they were created and stored, the issue of class default update would have been rendered irrelevant. The only drawback now would be the performance issue as query processing will slow down quite significantly.

However, in autonomous Web and ontology based applications, dynamic inheritance provides an edge. In particular, if dynamic inheritance is employed in ontologies, the semantics of the local sites can be kept untouched and the global ontology can have a
“virtual” semantics of the local sites that is different from the actual site viewed globally. This is because the semantics for the local site is usually superimposed by the global ontology in the global context, as we have discussed in the previous section (the second scenario).

5.4 Ontology Representation

We will now revisit the ontology example introduced in the previous section to illustrate how XML++ can be used to model global/local ontologies and how the idea of inheritance can be exploited to design better ontologies. At this point, we will assume that a suitable matching process has taken place using certain standard IR and/or graph based techniques such as the ones presented in chapter VI. Let us consider the representation of the global ontology first. We proceed as follows. Our XML++ document for this purpose is basically an XML document with the document element as the root element:

```xml
<?xml version="1.0"?>
<!DOCTYPE document PUBLIC '-//XML++ DTD 1.0//EN'
'http://www.xml.org/xpp.dtd'>
<document>
...
</document>
```

It must be mentioned that the global ontology must be validated against the XML++ DTD (or schema) and any root level class will take the place of the dots in the above code snippet. In our case, we have only one root class – ontology. Classes are created using the element tag as shown below:

```xml
[element name="ontology"]
```
As shown in Figure 5.8, the ontology class has two member elements: Personal Information and Reservation Information. These two elements will be the subelements of the ontology class. In XML++, subelements are declared using the same element class embedded inside the superelement, in this case the ontology class. In a recursive fashion, the First Name and Last Name elements are also subelements of the Personal Information element, and so on. The following code shows how the subelements are declared:

```xml
<element name="ontology">
  <element name="personal-info">
    <element name="first-name" />
    <element name="last-name" />
  </element>
  <element name="reservation-info">
    <element name="pickup">
      <element name="pickup-location" />
      <element name="pickup-date">
        <element name="pickup-date-month" />
        <element name="pickup-date-day" />
        <element name="pickup-date-year" />
      </element>
    </element>
    <element name="return">
      <element name="return-location" />
      <element name="return-date">
        <element name="return-date-month" />
        <element name="return-date-day" />
        <element name="return-date-year" />
      </element>
    </element>
  </element>
</element>
```
It is easy to observe that there are structures in the above representation that are reusable and perhaps are common across many local car rental ontologies. For example, the Pickup Date and Return Date elements have identical structures and subelements. Consequently, factorization is possible using XML++ constructs (complex terms and their inheritance) to create a more reusable set of elements to be used in the ontology. The following code fragment shows how to represent a library of reusable elements:

```xml
<element name="car-rental-types">
  <element name="date">
    <element name="month" />
    <element name="day" />
    <element name="year" />
  </element>
  <element name="time">
    <element name="hour" />
    <element name="minute" />
    <element name="am_pm" />
  </element>
</element>
```

We can now import these objects in our global ontology declaration using a template definition as follows:

```xml
<templates>
  <template url="car-rental-types.xml" prefix="car-rental"/>
</templates>
```
Using the reusable element definition, or the template, we can rewrite the global ontology in a more concise manner, as shown below. In this representation, we use the elements defined in the car-rental template by extending the date and time elements. In order to extend an element the extends attribute can be used, by specifying the template name and the element that we want to extend in that template, separated by a colon (:).

```xml
<element name="ontology">
    <element name="personal-info">
        <element name="first-name" />
        <element name="last-name" />
    </element>
    <element name="reservation-info">
        <element name="pickup">
            <element name="pickup-location" />
            <element name="pickup-date" extends="car-rental:date"/>
            <element name="pickup-time" extends="car-rental:time"/>
        </element>
        <element name="return">
            <element name="return-location" />
            <element name="return-date" extends="car-rental:date"/>
            <element name="return-time" extends="car-rental:time"/>
        </element>
    </element>
</element>
```

We can now describe the local ontologies for CompanyA and CompanyB by using our previous definition of a global ontology. The following code fragment shows both local ontologies. Notice how the representation is structured in a modular fashion and how inheritance plays a critical role in this representation.

```xml
<?xml version="1.0"?>
<!DOCTYPE document PUBLIC '-//XML++ DTD 1.0//EN'
The skeletons presented until now can be used to develop a real representation of the ontologies in section 5.2 in XML++ that actually conforms to the XML++ structure definitions as shown in Figure 5.7. While the full representation of the structure definitions (against which the ontologies must be validated) is beyond the scope of this chapter (for the want of space), it will look similar to the one shown below. And when the following code
fragment is “rendered” by an XML++ engine, it will produce the sample code fragment presented in the example.

```xml
<?xml version="1.0"?>
<!DOCTYPE document PUBLIC '-//XML++ DTD 1.0//EN' 'http://www.xml.org/xpp.dtd'>
<document>
  <element name="ontology">
    <attribute name="name" value="Company"/>
    <attribute name="title" value="Ontology"/>
    <attribute name="site" value="http://www.company.com"/>
  </element>
  <!-- Classes -->
  <element name="classes">
    <element name="class">
      <attribute name="name" value="input"/>
      <element name="domain"/>
      <element name="attributes">
        <element name="attribute">
          <attribute name="name" value="name"/>
          <attribute name="value" value=""/>
          <element name="domain">
            <attribute name="name" value="Text"/>
          </element>
        </element>
      </element>
      <element name="axioms"/>
      <element name="subclasses"/>
    </element>
  </element>
  <!-- Terms -->
  <element name="terms">
    <element name="term">
      <attribute name="name" value="Name"/>
      <attribute name="value" value=""/>
      <attribute name="class" value="input"/>
      <element name="domain"/>
      <element name="attributes">
        <element name="attribute">
          <attribute name="name" value="name"/>
          <element name="domain">
            <attribute name="name" value="Text"/>
          </element>
        </element>
      </element>
    </element>
  </element>
</document>
```
<element name="attribute">
  <attribute name="name" value="size"/>
  <attribute name="value" value="50"/>
  <element name="domain">
    <attribute name="name" value="Number"/>
  </element>
</element>
<element name="axioms"/>
<element name="relationships"/>
<element name="subterms"/>
</element>
CHAPTER VI

ONTOLOGY MATCHING TECHNIQUES

*Information seeking* is the process in which human beings recourse to information resources in order to increase their level of knowledge with respect to their goals. Information seeking has affected the way modern libraries operate (using tools such as catalogs, classifications, and indexing) and perpetrated the World Wide Web in the form of search engines. While the basic concept of information seeking remains unchanged, a growing need for automation of the process has called for innovative tools to propagate some of the tasks involved in information seeking to the machine level. Therefore, databases are widely used for the efficient storage and retrieval of information. Also, techniques from the area of information retrieval [73] were refined over the years to predict the relevance of information to a person’s needs and to identify appropriate information for a person to interact with. Finally, the use of computer-based ontologies [79] was suggested to classify the available information based on some natural classification scheme that would allow a more focused information seeking.

Most Internet portals (including Yahoo! and OpenDirectory) use “cybrerians” to maintain internet directories. Common practice nowadays assume that once ontologies are created, computer-supported tools can utilize them as part of the information seeking process.
The next natural step is then to let the machine generate the ontologies. One may consider two incentives in doing so. The first is rooted in the initial creation of ontologies, which is a tedious, time-consuming process. The second incentive is rooted in the rapid evolution of ontologies. If ontologies are managed manually, any change to them requires human intervention. This can bring to a halt an electronic process in the absence of constant human support. While the latter attracted a little attention in previous years, it has become a major sticking point with the introduction of eCommerce and electronic exchange markets, a rapidly changing environment in which virtual manufacturers, retailers, and consumers join in to perform activities in cyberspace.

It is the evolution of ontologies which we offer to automate in this thesis. In particular, we suggest utilizing ontologies to support a user in seeking information using interactive systems. As an example, consider a researcher who is interested in renting a car to attend her favorite conference. Using Web services, the researcher attempts at comparing available rates from many different car rental companies in order to reach an educated decision in obtaining her goal. Alas, this process of information seeking is tedious as well as frustrating. Information has to be typed in over and over again, and in most cases a manual comparison of terms and conditions is needed in evaluating the outcomes. An alternative exists in the form of car rental portals (e.g., Travelocity.com). However, as most general-purpose tools, such portals cater to popular needs, and therefore may only offer a limited set of options (such as the cheapest car available). Therefore, if our researcher is interested
in a deal which offers no mileage constraints she has to resort to manual search of terms and conditions.

Figure 6.1 outlines the various stages of ontology creation and adaptation, as suggested in this chapter. An initial ontology is created, using either extraction tools or an existing ontology. Equipped with the ontology, the user performs an information seeking session, in which the machine captures the inserted data and matches it with the ontology. This step is followed by an iterative process, in which new information seeking sessions are performed automatically. Each such session requires the fine tuning of the available ontology to the one currently in use, to be followed by automated information seeking. The results become available to the user and additional feedback is used to enhance the existing ontology and to improve the system’s capabilities for the next session. We term the initial session a training session, since this is the session in which the machine learns the data needs of the user. There is also a continuous learning process, which enables the machine to improve the ontology with each additional session.

Figure 6.1 Overview of dynamic ontology creation
In our experiments, we extract ontologies from HTML documents. We recognize the fact that XML may serve as a better candidate for ontology exploration. In fact, while one can *exarch* terms and structure from XML documents, one has to *mine* for ontologies in HTML documents. However, current trends in deploying XML as part of the organization data management scheme suggest that while XML may be used for B2B communication, and to some extent as a storage mechanism, interactive sessions still use HTML. Therefore, it is possible that XML data on the server side is “translated” into HTML before being shipped out to the client. It is also worth noting that once an ontology is extracted (from either XML or HTML documents), the process of ontology adaptation remains unchanged.

### 6.1 Related Work

The problem we tackle in this thesis falls into the category of *semantic heterogeneity*, which is well documented in the literature. The area of information science has an extensive body of literature and practice on ontology construction using tools such as thesauri and on terminology rationalization and matching of different ontologies [4, 74, 80, 84].

In the area of databases and information systems, many models were suggested to support the process of *semantic reconciliation*, including the SIMS project [5], SCOPES [69], dynamic classificational ontologies [47], COIN [63], and CoopWARE [33], to name a few. What is common to these solutions is their reliance on the designer’s intervention, rather than supporting a fully automatic semantic reconciliation. However, redesign and
re-implementation of metadata can incur tremendous cost. Therefore, automatic reconciliation becomes a must in such an environment.

Database research has extensive literature on data integration, including [17], [2], and [59], yet there is little impact of this research on the state of the art in commercial systems. We believe this chasm can be attributed to the fact that most of these approaches rely on semantic reconciliation to be resolved first (probably manually), before attending to the more “technical” aspects of the integration. However, researchers and practitioners alike are coming to realize that there can be no solution to the delivery of integrated information unless one tackles head-on the semantic heterogeneity problem [70]. This research works towards this goal.

Some research was devoted to automatic schema analysis and integration (e.g., [75], [65], and [18]). In [75], the analysis is based on a hand-crafted attribute hierarchy, which we avoid. The work of [18] and [65] are similar in that they analyze a schema, given in an abstract form of a graph, using formal methods of graph analysis. The tools and methodologies suggested in [18], when applied to schema integration are “not sufficient and must be enriched with semantic consideration, such as the interpretation of terms within an application domain in order to correctly compare elements.” Our experiments, as shown in chapter VIII, show that it is possible to automatically (and correctly) derive matchings, without reverting to manual interpretation. In [65], it is shown that the process of finding an optimal typing for semi-structured data is NP-hard. Therefore, a method is
presented based on heuristics to approximately type a large collection of semi-structured data. No extension of the method to deal with schemata matching is given in [65].

Like many before us, we attempt to perform semantic reconciliation using syntactic comparisons. However, we also enhance our model to include a measure of accuracy, which becomes a powerful tool whenever automated reasoning is involved. The provision of a measure of accuracy allows a user to determine her own tolerance to imprecision and to instruct the system to request for help once imprecision becomes too great. As our experiments demonstrate, if one narrows down the scope of the domain, ontologies can be extracted with a very high level of accuracy. In the next section, we present the metrics used to measure this accuracy.

6.2 Metrics

In this section, we provide a formal definition of the metrics used to evaluate the goodness of a term match. Two metrics are commonly used in IR to estimate the soundness and completeness of the merging process, precision and recall, respectively. In order to formally define these two metrics, let us first introduce some notation:

- \( t_r \): Number of terms retrieved in the extraction process.
- \( t_m \): Number of terms matched by the matching operations (see section 6.3).
- \( t_e \): Number of terms effectively matched. Two terms are considered to be effectively matched if their semantics are the same. The semantic similarity between two terms is defined by the domain expert, thus it is not possible for a machine to determine if two terms are correctly matched by the matching operations.
**Recall:** The *recall* is represented by the symbol $R$. It measures the completeness of the terms retrieved. It is defined as follows [31]:

$$ R = \frac{t_m}{t_r} \quad (6.1) $$

As an example, suppose that we are trying to merge two ontologies $O_1$ and $O_2$, containing 20 and 25 terms, respectively. After applying the matching operations we find that only 15 terms were successfully matched between the two ontologies. The recall of the operation, from the point of view of $O_1$, is $\frac{15}{20}$ or 75%.

**Precision:** The *precision* is represented by the symbol $P$. It measures the soundness of the terms retrieved. It is defined as follows [31]:

$$ P = \frac{t_e}{t_r} \quad (6.2) $$

Using the previous example, suppose now that of the 15 terms matched, only 10 of those terms are correct matches, while the other 5 were erroneously matched by the matching algorithms. Then the precision of the match is $\frac{10}{15}$ or 66%.

**Error:** In most of the cases there is no direct correlation between recall and precision. This is why we introduce another metric, denoted by $E$, to combine the metrics. The error is defined as follows [31]:

$$ E = 1 - \frac{(1 + b^2)PR}{b^2P + R} \quad (6.3) $$
In Equation (6.3), the variable $b$ measures the importance of the precision or the recall in the equation. A value for $b$ of 0.5 means that both precision and recall have the same importance. Intuitively, the lower the value of $E$ in the merge, the better the match is.

6.3 Information Retrieval Techniques

In this section, we briefly outline some of the matching operations to be used in the merging of two ontologies. Ontologies are merged pair-wise, applying a combination of operations, as described below. During the matching process, terms with a recall below a threshold are not considered in the merging. We now introduce the matching operation, listed in order of complexity.

**Textual Matching:** In this step, all the terms are compared pair-wise and tested for identical textual match (equality test). Usually the recall after this step is very low as labels and terms are unlikely to be named identically even though they represent the same term.

**Ignorable Character Removal:** Characters such as ‘*’, ‘/’, ‘-’, etc., are treated as “noise” and are removed from the terms. Hence, after this step, terms such as “*Country” and “country” will be considered identical. The argument here is that such characters do not contribute towards a meaningful identifier for a database field name.

**De-hyphenation:** Labels such as “pick-up” and “pick up” are considered identical (e.g., [30]). Hence, hyphens in labels are removed to improve matching. By merging the hyphenated words, it is possible to achieve better recall than replacing hyphens with
white space. Hence, we merge hyphenated words by removing the hyphens. For example, “pick-up” will be replaced by “pickup”.

**Stop Terms Removal:** Common terms such as ‘a’, ‘to’, ‘in’, and ‘the’ are considered *stop terms* in the literature (e.g., [32]). The removal of stop terms improves recall and does not adversely affect precision.

**Substring Matching:** Labels are matched pair-wise for substring matching. A parametric threshold for term matching is used. The match effectiveness for two terms $t_1$ and $t_2$ is defined as the ratio between the number of words in term $t_2$ that are substrings of terms $t_1$ and the number of words in term $t_2$, providing a measure of the semantic similarity of these two terms. The more words of one term contained by another term, the more similar they are. For example, the match effectiveness of $t_1=$Pickup Location and $t_2=$Pick-up location code can be computed at 66%.

**Name Matching:** In general, two properties are used to identify terms, the label and the name. The label is a string usually expressed in natural language to describe the purpose of the field to humans. The name can be any string that is constrained by some name rules (depending on the language being used to describe the term). Names are usually intended to identify terms in operations carried by the computer agents. As an example, consider an input field in an HTML page. The input field has two attributes: name and value. The value attribute is the string being rendered by the browser and is the string that the user sees. The name attribute is used in the CGI application and HTTP commu-

---

1 Usually a 50% match is considered a good measure, and hence we have used this threshold in all the steps described in this section. However, this threshold should be subject to adjustments.
communication between client (the browser) and the server (the Web server). The value can be something like *First Name*, while the corresponding name could be *fname*.

Matching can also be achieved by finding similarities between terms’ names. For this purpose, we use a name matching strategy that compares the names of two terms based on the ratio between the number of “parts” of the second name contained in the first name, and the number of “parts” in the second name. A part is a substring of name of length $c$. For example, to match two terms with names *address* and *addr1* for a $c = 3$, the algorithm first finds the parts of the second name (*add*, *ddr* and *dr1*) and sees how many are substrings of the first term (only *add* and *ddr*) producing a matching confidence of $\frac{2}{3} = 66\%$.

Other strategies for name matching can also be used. For example, in cases where the name of the term is expressed as an acronym of the label (e.g., an input with label *Pickup Location Code* can be named *PLC*), an algorithm that combines the first letters of the label and tries to match it with the name can be used. Another strategy can use a different definition of “part”, using only consonants, and not vocals, in the construction of the part. These strategies are not explored in this research and are left as future work.

**Content Matching**: Fields with select, radio, and check box options are processed using their value-sets. A match effectiveness is applied here too, calculated as the number of values in the second term that match (using substring matching) values in the first term, divided by the number of values in the second term. For example, suppose that $t_1$ is a *Return-time* term and $t_2$ is a *Dropoff-time* term with values such as \{10:00am,
10:30am, 11:00am} and \{10:00am, 10:15am, 10:30am, 10:45am, 11:00am\} respectively. Then, if we inspect each value of \(t_2\) for a match in \(t_1\) (using the substring matching technique described above), we will not find a match for values such as 10:15am, 10:45am and so on. Hence the match effectiveness of \(t_2\) (with respect to \(t_1\)) is calculated as \(\frac{3}{5} = 60\%\).

The power of content matching can be further highlighted using the case of terms \texttt{Dropoff Date} in Alamo and \texttt{Return Date} in Avis. These two terms have associated value sets \{(Select), 1, 2, \ldots, 31\}, and their match effectiveness is 100\%, and hence, are identified by our method as semantically identical concepts.

**Thesaurus Matching:** Finally, terms and labels that were not matched using one of the previous operations are matched using an ever expanding thesaurus. The thesaurus is constructed automatically by the result of matching operations like content matching, or manually from user interactions. Mismatched terms are presented to the user for manual matching. Every match the user identifies is accepted as a synonym. Each such manual match expands and enriches the thesaurus. This thesaurus is consulted in future matching to improve recall and precision.

## 6.3.1 Achieving Symmetry

The reader may have noticed that some of the operations presented above are not symmetric operations. Consider the same example used to explain the content matching operator: the match effectiveness of \(t_2\) with respect to \(t_1\) is calculated as \(\frac{3}{5} = 60\%\), but the match effectiveness of \(t_1\) with respect to \(t_2\) is calculated as \(\frac{3}{3} = 100\%.\) It is not always appropriate
to say that two terms are similar depending on the order in which the terms are listed. In
natural language, similarity is usually a symmetric operation. In this section, we redefine
the substring, name and content matching operations to be symmetric. Notice that all other
operations are symmetric by definition of the equality on strings (if \( s_1 = s_2 \) then \( s_2 = s_1 \)).

**Symmetric Substring Matching:** Let \( W_{t_1} \) and \( W_{t_2} \) be the set of words of term \( t_1 \) and
\( t_2 \) respectively. The match effectiveness between \( t_1 \) and \( t_2 \) is defined by

\[
e = \frac{|W_{t_1} \cap W_{t_2}|}{|W_{t_1} \cup W_{t_2}|}
\]

(6.4)

**Symmetric Name Matching:** Let \( P_{t_1} \) and \( P_{t_2} \) be the set of “parts” of term \( t_1 \) and \( t_2 \)’s
names respectively. The match effectiveness between \( t_1 \)’s name and \( t_2 \)’s name is defined
by

\[
e = \frac{|P_{t_1} \cap P_{t_2}|}{|P_{t_1} \cup P_{t_2}|}
\]

(6.5)

**Symmetric Content Matching:** Let \( V_{t_1} \) and \( V_{t_2} \) be the set values in the domain of \( t_1 \)
and \( t_2 \) respectively. The match effectiveness between \( t_1 \) and \( t_2 \) is defined by

\[
e = \frac{|V_{t_1} \cap V_{t_2}|}{|V_{t_1} \cup V_{t_2}|}
\]

(6.6)

The use of symmetric operators in terms matching has the effect of decreasing the
recall and increasing the precision of the match. We refer the reader to chapter VIII where
some experiments on using symmetric operators are presented.
6.4 Graph Matching

In this section, we present the graph matching technique. We first outline some related work in the area of graph isomorphism and graph matching and introduce a set of definitions that will be used in the presentation of our graph matching algorithm.

6.4.1 Related Work on Graph Matching

The problem of graph matching is not new. It has been presented in the literature as graph isomorphism and weighted graph match. Due to the representational power of graphs, there has been a lot of work put into graph matching [83, 35, 29]. Graph matching algorithms have been used in different research areas, in addition to finding structural matching. As an example, graph matching techniques are used in computer graphics to find objects (with points represented as a labeled graph) in a scene containing a set of different objects.

The problem of graph isomorphism consists of finding a matching function $f$ for two input graphs $G' = (V', E')$ and $G'' = (V'', E'')$. The idea is to find if two graphs are equal based on the matching between pair of nodes $(u, v)$, where $u \in V'$ and $v \in V''$. Most of the time this problem is changed to find subgraphs of $G'$ and $G''$ that are isomorphic, such as in the case of the example about localizing an object in a scene presented above.

The problem of (sub)graph matching problem is an NP-complete problem, as pointed out in [35]. Different algorithms have been proposed using heuristics to approximate a solution to this problem. One of these algorithms, applied to the domain of structural
matching, was recently presented in [54] with the name, *Cupid*. *Cupid* is a general purpose schema matcher similar to the *OntoBuilder* system presented as part of this thesis (see chapter VII).

*Cupid* makes use of both information retrieval techniques as well as graph matching techniques to find the best match between two input schemas. The graph matching technique used by this system is based on the calculation of a “structural similarity” metric, which is initialized according to the “linguistic similarity” between a pair of nodes (here the linguistic similarity is obtained based on information retrieval techniques similar to the ones presented in section 6.3). The algorithm starts comparing pairs of nodes in the leaves of both graphs, incrementing or decrementing the structural similarity according to the similarity between parents and children nodes.

*Cupid* is based on three intuitions [54]:

- Leaf elements are similar if they are “linguistically similar” and if the neighboring nodes are similar, too.
- Non-leaf elements are similar if they are “linguistically similar” and the subtrees rooted at the two elements are also similar.
- Non-leaf elements are “structurally similar” if their leaves sets are similar (even if their immediate children are not).

We borrow some of the ideas from *Cupid*; however, we reject the third intuition. Immediate children are not considered important in the overall similarity function unless they are leaves. This intuition is well suited for highly grouped schemas where intermediate nodes are not semantically important. However, we think that this is not the general case. In the case of semistructured data retrieved from HTML pages, as in our case, where
the schema is inherently flat, the Cupid algorithm may not function correctly. Besides, precedence between schema terms is not considered in Cupid, but is an important factor in the semantic behaviour of any Web application.

6.4.2 Web Resource Ontologies

In this section, we present a graph representation of a Web ontology. Intuitively, an ontology can be represented as a graph with a vertex for each term in the ontology and edges between a pair of vertex for precedence and grouping relationships. The following is a formal definition of this concept.

Definition 1 (Precedence)

Let \( u \) and \( v \) be two terms of a Web ontology. \( u \) precedes \( v \) if there exists a script in the Web ontology such that:

- The user inputs \( u \) before the script is activated.
- \( v \) can be updated only after the script is activated.
- There is some evidence in the script that the available values for \( v \) depend on \( u \).

Definition 2 (Web Resource Ontology)

A Web resource ontology is a directed ordered graph \( G = (V, E) \) such that:

- \( V \) is a set of terms.
- \( E = E_1 \cup E_2 \), where
  - \((u, v) \in E_1 \) if \( u \) belongs to group \( v \).
  - \((u, v) \in E_2 \) if \( u \) precedes \( v \).
Using the previous definitions, we can now introduce a graph matching algorithm that takes as input two Web resource ontologies, producing as output a third graph, which we call an extended Web resource ontology. This is basically the union of the two input graphs with edges between pairs of terms in both ontologies representing matches.

Definition 3 (Extended Web Resource Ontology)

Let $G' = (V', E')$ and $G'' = (V'', E'')$ be two Web resource ontologies. An extended Web resource ontology of $G'$ and $G''$ is a directed ordered graph $G = (V, E)$ such that:

- $V = V' \cup V''$.
- $E = E' \cup E'' \cup E_3$, where $(u, v) \in E_3$ if $u \in V'$ and $v \in V''$.

We define a function $w : E_3 \rightarrow [0, 1]$ as a confidence function, assigning each $(u, v) \in E_3$ a level of confidence $w(u, v)$.

6.4.3 The Graph Matching Algorithm

Before presenting the graph matching algorithm, let us first introduce some notation and definitions that will help explain how the algorithm works.

Given two Web resource ontologies $G' = (V', E')$ and $G'' = (V'', E'')$, and an extended Web resource ontology $G = (V, E)$ for $G'$ and $G''$, the following definitions apply:

Definition 4 (Children)

Let $u$ be a term of $G$ ($u \in V$), then the children of $u$, denoted $\text{children}(u)$, is the set $C = \{c\}$, such that $\exists e = (u, c) \in E$ meaning that $c$ belongs to the group $u$, or $u$ precedes $c$. 

Definition 5 (Parents)

Let $u$ be a term of $G$ ($u \in V$), then the parents of $u$, denoted $\text{parents}(u)$, is the set $P = \{p\}$, such that $\exists e = (p, u) \in E$ meaning that $u$ belongs to the group $p$, or $p$ precedes $u$.

Definition 6 (Siblings)

Let $u$ be a term of $G$ ($u \in V$), then the siblings of $u$, denoted $\text{siblings}(u)$, is the set $S = \{s\}$, such that $\exists p \in \text{parents}(u)$ and $s \in \text{children}(p)$ and $s \neq u$.

Each of the previous definitions can be optionally defined with a second parameter $r$, which indicates how deep the definition extends. For example, $\text{children}(u, r)$ is the set of all the children of $u$ up to a level $r$ in the graph. By definition, $\text{children}(u) \leftrightarrow \text{children}(u, 1)$. The same concept applies to the other two definitions.

Definition 7 (Matching Matrix)

The matching matrix for $G$, denoted as $M_G$, is a bidimensional matrix of $|V'| \times |V''|$ cells. A cell at position $i, j$ represents the matching confidence between a term $u_i \in V'$ and a term $v_j \in V''$, and is defined by a function $m_{i,j}$ as follows:

$$m_{i,j} = \begin{cases} 
0 & \text{if } \neg \exists e = (u_i, v_j) \in E_3 \\
\omega & \text{if } \exists e = (u_i, v_j) \in E_3, \omega : E \rightarrow [0, 1] 
\end{cases}$$

The intuition for the algorithm is very simple, two terms $u_i \in V'$ and a term $v_j \in V''$ match if their children, parents and siblings also match, to some degree. We will now formalize this intuition.

The graph match algorithm uses the previous definitions to determine the matching confidence between two terms $u_i \in V'$ and a term $v_j \in V''$. The matching confidence is
a function of three factors, defined by the matching between children, parents and siblings of $u$ and $v$. Each of these three factors has an importance parameter associated with it. Importance parameters are usually domain dependent and can be changed to reflect different behaviors of the algorithm. The following notations define the importance parameters for each factor.

- **Children importance**: denoted as $C_{imp}$, specifies the importance of children matching in the overall matching operations.

- **Parents importance**: denoted as $P_{imp}$, specifies the importance of parents matching in the overall matching operations.

- **Siblings importance**: denoted as $S_{imp}$, specifies the importance of siblings matching in the overall matching operations.

Additionally, there are two more importance factors associated with each matching technique, namely IR technique and the graph technique (this algorithm). The reason for having an importance parameter for each technique is the following. Suppose there are two terms that are not semantically similar but have the same name. The terms will be erroneously matched by the IR technique but with no support in the graph technique. In these cases, we want to determine the overall matching after applying both techniques, expressed by a function of the confidence level obtained in each technique. The function is defined by associating importance parameters to each technique involved in the match. The following notation defines the importance parameters for each technique.

- **Linguistic importance**: denoted as $L_{imp}$, specifies the importance of linguistic (IR) matching in the overall matching operations.

- **Graph importance**: denoted as $G_{imp}$, specifies the importance of graph matching in the overall matching operations.
We can now introduce the graph matching algorithm.

Algorithm: Graph matching
Input: \( G', G'', G \), and \( M_G \)
Output: \( \hat{G} \), a new extended Web resource ontology

01. Initialize \( M_G \)
02. for each term \( u \in V' \)
03. for each term \( v \in V'' \)
04. \( C_{conf} = \text{getChildrenConfidence}(u, v) \)
05. \( P_{conf} = \text{getParentsConfidence}(u, v) \)
06. \( S_{conf} = \text{getSiblingsConfidence}(u, v) \)
07. \( O_{conf} = C_{conf} \times C_{imp} + S_{conf} \times S_{imp} + S_{conf} \times S_{imp} \)
08. \( M_{aux}[u][v] = O_{conf} \)
09. end for
10. end for
11. \( M_G[u][v] = L_{imp} \times M_G[u][v] + G_{imp} \times M_{aux}[u][v] \)
12. Build \( \hat{G} \) by creating edges \( \hat{e} = (\hat{u}, \hat{v}) \) \((\hat{u} \in V', \hat{v} \in V'')\) such that \( M_G[\hat{u}][\hat{v}] = \max\{M_G[x_k][\hat{v}]\}_{0 \leq k < |V'|} \)
13. return \( \hat{G} \)

The \text{getChildrenConfidence} function is defined as follows

Algorithm: Get Children Confidence
Input: \( u \in V' \) and \( v \in V'' \), and \( M_G \)
Output: a confidence between \( u \) and \( v \)

01. \( conf_{u,v} = 0 \)
02. \( matches = 0 \)
03. if \( |\text{children}(u)| = 0 \) and \( |\text{children}(v)| = 0 \) then return 1
04. for each \( c_u \in \text{children}(u) \)
05. for each \( c_v \in \text{children}(v) \)
06. if \( M_G[c_u, c_v] > 0 \)
07. \( conf_{max} = \max\{conf_{max}, M_G[c_u, c_v]\} \)
08. \( matches = matches + 1 \)
09. end for
10. \( conf_{u,v} = conf_{u,v} + conf_{max} \)
11. end for
12. return \( \frac{conf_{u,v}}{matches} \)
The `getParentsConfidence` and `getSiblingsConfidence` functions are defined similarly.
CHAPTER VII

THE ONTOBUILDER SYSTEM

In this chapter, we will present the OntoBuilder system. OntoBuilder is the tool we use to implement all the ontology matching techniques discussed in previous chapters. Built entirely from scratch by using standard libraries, the tool presents a user-friendly interface for users to apply the concepts developed in this thesis. We also used the tool to perform all the experiments presented in chapter VIII. We will first present a general overview of how the tool works, followed by a detailed explanation of the tool architecture and its internals.

7.1 System Overview

We developed a tool that extracts the ontologies from a Web application and creates a global ontology which can be used to answer user queries against data sources in the same domain as the global ontology. The whole process is divided into four phases, as depicted in Figure 7.1. In phase 1, the system parses the page in its input producing a DOM tree which is then used to identify all the form elements and their label in phase 2 (see section 7.3). In phase 3, the system produces an initial version of the target and candidate ontologies, i.e., the global ontology and local ontologies, respectively. Later, in
phase 4, the ontologies are merged in an iterative way to produce a refined global ontology which can be queried by users.

[Figure 7.1 Phases of the ontology creation process]

### 7.2 System Architecture

The architecture is depicted in Figure 7.2. The tool has a modular design where every module can be replaced as needed. The two main modules are the DOM analyzer and the ontology module. The input to the system is an HTML page representing the eCommerce Web site main page (e.g., [http://www.avis.com](http://www.avis.com)). The HTML page is parsed by the HTML parser module using a parser library for HTML/XML documents and produces a DOM tree representing the page. During the parsing process, the HTML page is filtered, removing tags that can produce "noise" in the extraction of the elements in the DOM
analyzer module; i.e., tags used for formatting (e.g., `<font>` tags) and scripting (e.g., `<script>` tags) are removed from the resulting DOM tree. The DOM analyzer is the module responsible for identifying the HTML elements that will be used in the ontology extraction process. These elements are identified by `<a>`, `<form>` (including all the `<input>` and `<select>` elements that comprise the form), `<meta>` and `<frame>` tags. This module is also responsible for the extraction of labels for each element.

The DOM analyzer encapsulates all the elements identified in the page as a complex `HTMLElement` structure and passes this to the visualization module. This module constitutes the main interface to the user. The visualization module receives input from every
other module in the system (even if not shown in the figure) so the user can drive the process of ontology extraction in an interactive way. The figure shows a subset of the submodules comprising the visualization module, each submodule in charge of a specific function. For example, the HTML viewer and the source viewer receive an HTML page as input and produce a browser-like view of the page, so the user can interact with the page clicking on hyperlinks to navigate the site. The DOM tree produced by the DOM analyzer is also used as input to the DOM viewer so the user can see a parsed version of the page. The HTML element viewer submodule is used to show to the user all the HTML elements identified by the DOM analyzer. Finally, the form viewer produces a preliminary view of the identified ontology for the page; from here the user can see how the different form elements (e.g., `<input>`, `<select>`) were automatically labeled by the system. The user can also input values into the fields to submit the form and navigate through the site.

The ontology module is responsible for the creation of the ontology. Its input is the form elements identified in the DOM analyzer. The output is an XML file describing the ontology for the Web site. The main function of this module is the merging and matching of ontologies for different Web sites. This module contains statistical submodules as well as string matching routines and thesaurus used for the ontology merging. The algorithms used in this module were discussed in chapter VI. The navigation module is responsible for logging all the activities performed by the user like link navigation and form submission. It produces a navigation file that can be used as input in later sessions with the system to automate the ontology extraction, \textit{i.e.}, if a Web site has a navigation file associated, then
the system is able to simulate the user interaction and navigate the Web site searching for an ontology.

### 7.3 Label Identification

The identification of form labels is one of the most important processes in the ontology extraction procedure. Problems arise trying to identify the label for an input element. The lack of structure of HTML pages and the diversity of layouts by which forms are designed in a page renders the label identification process difficult. Some heuristics based on common layout designs are used to improve the label extraction. These heuristics were developed using a training set (simple HTML pages containing the most used layouts for form design) and then applied to the real HTML pages. Basically two main layouts were identified: a table layout and a non-table layout. For each layout the label can be specified as text or as an image. Also, for each layout, the label can be located above the form field or to the left of the form field.

Figure 7.3 shows some layouts used as the training set. Notice how the layouts are divided in two dimensions based on structure (table vs. paragraph) and alignment (horizontal vs. vertical). Not all the labels for the input elements are successfully identified because there is no rule about the formatting and layout of forms in HTML. However, the label identification process has a high percentage of success.
OntoBuilder uses more than 450 classes divided into two categories: interface, data structures and utilities. Interface classes provide a GUI (Graphic User Interface) on top of the data structure and utilities classes, using common GUI objects such as windows, frames, combo boxes, etc. (see more details about the user interface in section 7.7).

Utility classes are support libraries for the overall functionality of the system. Following is a brief description of the most important utilities:

- Network Utilities: This set of utilities provide support for access to remote resources identified by URL (Universal Resource Locators). Resources can be located either by the HTTP and the HTTPS (Secure HTTP) protocols.
- DOM Utilities: Utilities that provide support for parsing and interpretation of HTML pages to/from DOM structures.
• XML Utilities: This set of utilities provide support for parsing and creation of XML files. These utilities also provide support for entity resolution of public DTDs and XML Schemas.

• HTML Utilities: HTML utilities provide a set of libraries to support HTML related operations such as label identification, form submission and HTML elements construction. These libraries are used by the HTMLElement structure providing common functionality shared among the different subclasses of the HTMLElement structure.

• Ontology Utilities: This set of libraries provide support to the construction, matching and merging of ontologies. The Ontology structure makes extensive use of these libraries.

• Algorithm Utilities: These utilities provide a set of basic functionality for the different matching operators presented in chapter VI. Matches are performed between two similar ontologies producing a MatchInformation structure that encapsulates the information resulting from the application of the match operators. These utilities also provide configuration functionality to each algorithm plugged into the system (see section 7.5).

Data structures are the heart of the system. They encapsulate the functionality and behaviour of different entities in the overall procedure of ontology extraction and matching.

We will now briefly describe some of the most important data structures in our system:

• HTMLElement: This structure models elements found in an HTML page such as FORM, A (anchors), INPUT and FRAME elements. Elements are grouped in a hierarchy as shown in Figure 7.4.

![Figure 7.4 The HTMLElement structure](image-url)
• **Ontology:** The ontology structure, as the names implies, models an ontology. This data structure complies with the ontological representation presented in chapter V. Ontologies can be serialized as XML documents, or as binary files using a Java 2 SDK native representation. Ontologies delegate some functionality to the ontology and algorithm utilities, specially for matching and merging purposes. Figure 7.5 shows a graphical representation of an ontology, for more details see chapter V.

• **Thesaurus:** The thesaurus is a dictionary of similar terms, used by the matching operators to aid the matching process. The thesaurus structure provide a set of methods to identify similarities between terms and to create thesaurus entries given two similar terms. This structure is a general component that provides an API that can be used by any system with thesaurus requirements. The thesaurus is serialized as an XML file and can be edited either by using the API or by editing the XML file directly. Figure 7.5 shows how the thesaurus structure is composed as a tree of words with links to synonyms and homonyms.

![Figure 7.5 a) The Ontology structure, b) The Thesaurus structure](image-url)
7.5 Algorithms Implementation

 OntoBuilder implements every algorithm introduced in chapter VI. New algorithms can be implemented and added to the tool as plug-ins. All the algorithms are extensions of an abstract algorithm class called AbstractAlgorithm, which in turn is an implementation of the Algorithm interface. The interface describes the signature (methods and functions) that matching algorithms must implement in order to be used in the tool. Common functionality such as setting the name and description of the algorithm and setting algorithm properties are implemented in the AbstractAlgorithm class. New algorithms can either extend the default implementation of the algorithm, and thus use the default functionalities provided, or implement directly the algorithm interface.

 Here are the methods supported by the Algorithm interface:

 public interface Algorithm
 {
     public MatchInformation match(Ontology targetOntology,
                                        Ontology candidateOntology);
     public void setMode(int mode);
     public int getMode();
     public String getName();
     public String getDescription();
     public void configure(Element element);
     public JTable getProperties();
     public void setThreshold(double threshold);
     public double getThreshold();
     public boolean usesThesaurus();
     public void setThesaurus(Thesaurus thesaurus);
     public Thesaurus getThesaurus();
 }

 As the reader can see, the interface provides a standard set of routines to match two ontologies and to configure and set algorithm properties and auxiliary utilities such as
a thesauri. Each particular algorithm must override the appropriate method according to the algorithm specification. Also notice that in order to configure an algorithm, the configure method expects an Element parameter in the format of a standard W3C DOM element. This means that algorithm’s parameters are specified as XML files such as the one presented below:

```xml
<algorithm name="Textual Match">
  <class>className</class>
  <parameters>
    <parameter>
      <name>param1</name>
      <value>false</value>
      <default>false</default>
    </parameter>
    <parameter>
      <name>param2</name>
      <value>2</value>
      <default>5</default>
    </parameter>
  </parameters>
</algorithm>
```

The configuration file provides the name of the algorithm (as seen in the tool wizards) and a list of parameters. Each parameter must specify its name, current value and a default value.

### 7.6 Programming Language and Libraries Used

*OntoBuilder* was developed using the Java language. Java is a platform-independent language which makes the tool portable to platforms like Sun Microsystems SPARC Stations and Intel x86 or Pentium based platforms, and to different operating system environments
like DOS, Windows, Linux, Unix, etc. Java also has a set of predefined libraries (like the util, AWT, Swing, JDBC, to name a few) [64], that make the tool development much easier than in other languages. Language features like the object-oriented paradigm and the declaration of interfaces are very useful to achieve the functionality specifications required by the tool. Another useful advantage of Java is that the tool can be programmed as an applet and used in most common Java-enabled Web browsers like Netscape Navigator or Microsoft Internet Explorer. *OntoBuilder* provides an applet version with all the same features of the standalone version with the added functionality that allows users to access and use it within a Web client. The tool also runs under the Java Web Start technology.

NOTE: This version of the tool runs under the Java 2 JDK version 1.4\(^1\) or greater only. Previous versions of Java are not supported at this time.

Besides the standard libraries that are part of the Java Standard Development Kit, additional libraries were used. These libraries are all covered by OpenSource licenses, which usually allows free use of the libraries for academic and non-profit purposes. Here we present a list of these libraries:

- Xerces, XML parser ([http://xml.apache.org](http://xml.apache.org)).
- JDOM, Document Object Model library for XML ([http://www.jdom.org](http://www.jdom.org)).
- JGraph, graph visualization library for Swing ([http://www.jgraph.com](http://www.jgraph.com)).
- JFreeChart, charts visualization ([http://www.jfree.org](http://www.jfree.org)).
- Hypertree, hyperbolic trees ([http://hypertree.sourceforge.net](http://hypertree.sourceforge.net)).
- Tidy, HTML parser ([http://sourceforge.net/projects/jtidy](http://sourceforge.net/projects/jtidy)).

\(^1\)At the time this thesis was written, version 1.4 was the latest release of the JDK from Sun Microsystems.
7.7 User Interface

The interface was entirely created using the Java Swing UI library [64]. Figure 7.6 presents a screenshot of the tool’s interface identifying all the components and panels.

![The OntoBuilder interface](image)

Figure 7.6 The OntoBuilder interface

The functionality of the tool is based on the concept of wizards. Wizards are very common in modern UIs since they drive the process in a user friendly fashion, asking for the appropriate user input and interacting depending on this input. Two main wizards are used in the tool:
• Ontology Creation Wizard: This wizard helps the user to extract ontologies from an HTML page. Figure 7.7 shows a screenshot of this wizard presenting an HTML form that requires user input.

![Figure 7.7 The ontology creation wizard](image)

• Ontology Merging Wizard: Once ontologies are created by the ontology creation wizard, they can be matched and merged to create global ontologies. This process is implemented by the merging wizard. The wizard asks for which algorithm and with what parameters the user wants to merge the input ontologies. Figure 7.8 shows a screenshot of the final stage of this wizard presenting the results of the merging.

The system also makes use of common display techniques such as graph representations and hyperbolic views for Web site maps and document structures, as shown in Figure 7.9.
Figure 7.8 The ontology merging wizard

Figure 7.9 a) Graph view of an ontology, b) Hyperbolic view of a document structure
CHAPTER VIII

EXPERIMENTS AND RESULTS

In this section, we present the results of running the algorithms introduced in previous chapters against real Web applications in the domain of car-rental companies. We conducted experiments in each of the following areas:

- Label identification algorithm.
- Usefulness of matching algorithms.
- Matching results for different ontologies.
- Graph matching results.

8.1 Label Identification Experiments

As explained in chapter VII, Web application ontologies are retrieved based on HTML form elements and their associated label. In general, HTML pages are not well-formed nor validated against any schema specification. This makes the label identification process hard and inexact. Heuristics based on structure (table vs. paragraph) and alignment (left vs. right alignment) are very helpful in cases like this.

In the following graph, we show how the label identification algorithm is able to identify a high percentage of elements and their associated labels. The x axis shows different
Web applications while the $y$ axis shows the number of labels correctly identified vs. the number of total labels in the application.

Table 8.1 Label identification results.

<table>
<thead>
<tr>
<th></th>
<th>Correct Labels</th>
<th>Incorrect Labels</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avis</td>
<td>33</td>
<td>1</td>
<td>97.06</td>
</tr>
<tr>
<td>Hertz</td>
<td>32</td>
<td>7</td>
<td>82.05</td>
</tr>
<tr>
<td>Alamo</td>
<td>18</td>
<td>4</td>
<td>81.82</td>
</tr>
<tr>
<td>National</td>
<td>14</td>
<td>0</td>
<td>100.00</td>
</tr>
<tr>
<td>Interamerican</td>
<td>12</td>
<td>1</td>
<td>92.31</td>
</tr>
<tr>
<td>Orbitz</td>
<td>12</td>
<td>0</td>
<td>100.00</td>
</tr>
<tr>
<td>Travelocity</td>
<td>11</td>
<td>1</td>
<td>91.67</td>
</tr>
<tr>
<td>Galileo</td>
<td>11</td>
<td>0</td>
<td>100.00</td>
</tr>
<tr>
<td>Budget</td>
<td>17</td>
<td>4</td>
<td>80.95</td>
</tr>
</tbody>
</table>

The results presented above do not include hidden fields, submits, resets and buttons in general, and images. Although these elements are used in the ontology extraction, they usually do not have a label associated (e.g., hidden fields are not even shown to the user). On average, the label identification algorithm achieves more than 90% effectiveness. With the new release of the W3C standard for XHTML (basically well-formed HTML), Web application developers will have a solid foundation to make HTML pages easier to parse, and in our case, to extract ontologies from.
8.2 Matching Algorithms Evolution

During our research for ontology matching algorithms we started with simple matching algorithms from the information retrieval field, evolving into complex algorithms that combine different approaches including algorithms that make use of external aids (such as thesauri) and graph matching algorithms. In this section, we show how the evolution of our matching algorithms produces better results when compared with previous algorithm versions.

The idea was to combine different techniques, which used individually would not have produced significant results, but combined together are very powerful when trying to identify matching terms between a pair of ontologies. The following table and graph shows the error produced when matching two ontologies such as the one obtained from Avis and Hertz car-rentals applications.
Table 8.2 Matching algorithms evolution.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Terms&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Matches&lt;sup&gt;b&lt;/sup&gt;</th>
<th>Eff. Matches&lt;sup&gt;c&lt;/sup&gt;</th>
<th>Recall&lt;sup&gt;d&lt;/sup&gt;</th>
<th>Precis.&lt;sup&gt;d&lt;/sup&gt;</th>
<th>E&lt;sup&gt;d&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Textual</td>
<td>43</td>
<td>9</td>
<td>5</td>
<td>20.93</td>
<td>55.56</td>
<td>58.26</td>
</tr>
<tr>
<td>Ign. Char. Removal</td>
<td>43</td>
<td>9</td>
<td>5</td>
<td>20.93</td>
<td>55.56</td>
<td>58.26</td>
</tr>
<tr>
<td>Hyphen Removal</td>
<td>43</td>
<td>15</td>
<td>7</td>
<td>34.88</td>
<td>46.67</td>
<td>56.29</td>
</tr>
<tr>
<td>Stop Terms Removal</td>
<td>43</td>
<td>15</td>
<td>7</td>
<td>34.88</td>
<td>46.67</td>
<td>56.29</td>
</tr>
<tr>
<td>Word Separator</td>
<td>43</td>
<td>15</td>
<td>7</td>
<td>34.88</td>
<td>46.67</td>
<td>56.29</td>
</tr>
<tr>
<td>Substring</td>
<td>43</td>
<td>34</td>
<td>16</td>
<td>79.07</td>
<td>47.06</td>
<td>48.80</td>
</tr>
<tr>
<td>Subtrings and Name</td>
<td>43</td>
<td>35</td>
<td>19</td>
<td>81.40</td>
<td>54.29</td>
<td>41.84</td>
</tr>
<tr>
<td>Content</td>
<td>43</td>
<td>35</td>
<td>20</td>
<td>81.40</td>
<td>57.14</td>
<td>39.24</td>
</tr>
<tr>
<td>Thesaurus</td>
<td>43</td>
<td>31</td>
<td>21</td>
<td>72.09</td>
<td>67.74</td>
<td>31.43</td>
</tr>
<tr>
<td>Symmetric</td>
<td>43</td>
<td>35</td>
<td>21</td>
<td>81.40</td>
<td>60.00</td>
<td>36.67</td>
</tr>
<tr>
<td>Graph&lt;sup&gt;d&lt;/sup&gt;</td>
<td>49</td>
<td>29</td>
<td>22</td>
<td>59.18</td>
<td>75.86</td>
<td>28.19</td>
</tr>
</tbody>
</table>

<sup>a</sup>Total terms in candidate ontology

<sup>b</sup>Matches identified by the algorithm

<sup>c</sup>Effective matches that are actually correct

<sup>d</sup>Normalized ontologies

Figure 8.2 Matching algorithm evolution
We used a value of 0.5 for $b$ (see chapter VI), thus giving equal importance to recall and precision. It is worth noting how the algorithm evolution reduces the error in the matching process. Matches between different Web applications behaved similarly to the one presented above so we will not present them for space considerations.

Although not 100% accurate, our matching algorithm produces very good results, achieving confidence above 70% threshold. Higher confidence levels can be achieved by applying the same techniques to structured data.

### 8.3 Matching Results for Different Ontologies

In this section, we present some results of the application of the matching algorithms between different ontologies. We used the ontology from the Avis Web site as our target ontology in all the results shown in this experiment. The target ontologies considered are the ones retrieved from applications such as Hertz, Alamo, National, Interamerican, Orbitz, Budget, Galileo, Thrifty, and Travelocity.

The algorithm used is the symmetric graph matching algorithm, which provides the best confidence level as shown in the previous experiment. The following table and graph show the different levels of matching confidence for each candidate ontology. Again we use the $E$ (error) to calculate the confidence with a $b$ of 0.5.

Ontologies such as Hertz, National and Interamerican achieve a very high degree of confidence (above 80%) while others such as Travelocity and Budget have a very low degree of confidence. These discrepancies are due to several factors, the most important
Table 8.3 Matching between different ontologies.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Terms&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Matches&lt;sup&gt;b&lt;/sup&gt;</th>
<th>Eff. Matches&lt;sup&gt;c&lt;/sup&gt;</th>
<th>Recall%</th>
<th>Precis.%</th>
<th>E%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hertz</td>
<td>49</td>
<td>29</td>
<td>22</td>
<td>59.18</td>
<td>75.86</td>
<td>28.19</td>
</tr>
<tr>
<td>Alamo</td>
<td>21</td>
<td>18</td>
<td>11</td>
<td>85.71</td>
<td>61.11</td>
<td>35.17</td>
</tr>
<tr>
<td>National</td>
<td>18</td>
<td>15</td>
<td>13</td>
<td>83.33</td>
<td>86.67</td>
<td>14.02</td>
</tr>
<tr>
<td>Interamerican</td>
<td>15</td>
<td>8</td>
<td>8</td>
<td>53.33</td>
<td>100.00</td>
<td>14.89</td>
</tr>
<tr>
<td>Orbitz</td>
<td>17</td>
<td>8</td>
<td>7</td>
<td>47.06</td>
<td>87.50</td>
<td>25.33</td>
</tr>
<tr>
<td>Travelocity</td>
<td>17</td>
<td>10</td>
<td>5</td>
<td>58.82</td>
<td>50.00</td>
<td>48.45</td>
</tr>
<tr>
<td>Budget</td>
<td>23</td>
<td>23</td>
<td>12</td>
<td>100.00</td>
<td>52.17</td>
<td>42.31</td>
</tr>
<tr>
<td>Thrifty</td>
<td>23</td>
<td>10</td>
<td>7</td>
<td>43.48</td>
<td>70.00</td>
<td>37.61</td>
</tr>
<tr>
<td>Galileo</td>
<td>20</td>
<td>8</td>
<td>5</td>
<td>40.00</td>
<td>62.50</td>
<td>43.82</td>
</tr>
</tbody>
</table>

<sup>a</sup>Total terms in candidate ontology  
<sup>b</sup>Matches identified by the algorithm  
<sup>c</sup>Effective matches that are actually correct

Figure 8.3 Matching between different ontologies
being the difference between the target and candidate ontologies and the accuracy of the label identification process. In the case of Travelocity, the ontology for this Web application includes not only car-rental concepts but also flight and hotel reservation, concepts not available in the target ontology. On the other hand, the ontology of the Budget Web application has a very poor accuracy in terms of label identification (see Figure 8.1, where Budget has the lowest label identification accuracy), producing a very low match confidence.

8.4 Graph Matching Results

In the case of Web applications, the ontologies produced have a very flat nature, i.e., the degree of nesting of terms within one another is very low. Terms are nested first by the Web page and then by the form in which the terms are found. Some applications use terms that have a very high degree of nesting, producing ontologies with groupings of similar terms into categories localized deep down in the hierarchy.

It is in these cases that the graph matching techniques play an important role in the ontology matching process. In order to measure the effectiveness of our graph matching algorithm, we performed a new set of experiments. We used Cupid [54] as our reference algorithm to compare ours. At the time we performed these experiments, Cupid was no more than a research activity in the Microsoft Research laboratories, and although we were able to obtain a working version of the algorithm with some examples, we were not able to reproduce our ontologies in the Cupid language due to a lack of an ontology editor.
For the experiments, we chose a sample ontology that came with Cupid. Although not from the car-rental domain, the sample ontology will suffice to illustrate how our graph matching algorithm behaves. The sample ontology represents an order processing application managing the contact information, billing and shipping address, and the details of the items ordered. Figure 8.4 shows two similar ontologies that model this kind of application. Notice how the terms are nested into grouping terms (composition), e.g., the term contactName is at depth four nested within the terms PurchaseOrder, Header and Contact.

The following graph shows the results of matching both ontologies using Cupid and using our graph matching algorithm with different thresholds\(^1\). Again, we measure our experiments using recall, precision and the error with a \( b \) of 0.5.

Table 8.4 Graph matching between Cupid and our algorithm.

<table>
<thead>
<tr>
<th>Algorithm(^a)</th>
<th>Terms(^b)</th>
<th>Matches(^c)</th>
<th>Eff. Matches(^d)</th>
<th>Recall%</th>
<th>Precis.%</th>
<th>E%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cupid</td>
<td>40</td>
<td>40</td>
<td>33</td>
<td>100.00</td>
<td>82.50</td>
<td>14.51</td>
</tr>
<tr>
<td>Graph match (40%)</td>
<td>40</td>
<td>14</td>
<td>13</td>
<td>35.00</td>
<td>92.86</td>
<td>30.21</td>
</tr>
<tr>
<td>Graph match (10%)</td>
<td>40</td>
<td>23</td>
<td>16</td>
<td>57.50</td>
<td>69.57</td>
<td>33.24</td>
</tr>
<tr>
<td>Graph match (5%)</td>
<td>40</td>
<td>26</td>
<td>17</td>
<td>65.00</td>
<td>65.38</td>
<td>34.69</td>
</tr>
</tbody>
</table>

\(^a\)The percentage in parenthesis indicates the threshold  
\(^b\)Total terms in candidate ontology, the ontology to the right in Figure 8.4  
\(^c\)Matches identified by the algorithm  
\(^d\)Effective matches that are actually correct

\(^1\)The threshold determines whether two terms are selected as a match or not. If the confidence is above the threshold, the terms are considered a match.
Figure 8.4 Two ontologies modeling an order processing application
As the reader can see in Figure 8.4, Cupid has very low error compared with our graph matching algorithm. However, it is important to note that Cupid implements some techniques not used by our graph algorithm. First, Cupid produces $n : m$ matches, i.e., terms in the target ontology can be matched to more than one term in the candidate ontology, as in the case of the Address subterms, which are repeated three times in the first ontology in Figure 8.4. Our graph matching algorithm produces $1 : 1$ matches. In the case of the Address subterms, only one of the three subsets of terms is matched. Another technique used by Cupid is the specification of initial matches to aid the graph matching algorithm. In our case, the algorithm does not make use of any predefined matches.

In spite of these disadvantages, we consider our algorithm to have produced good results when we focus on the precision achieved, as depicted in Figure 8.5. The precision tends to increase as we increase the threshold level for which matches are considered.

Figure 8.5 Graph matching between Cupid and our algorithm
CHAPTER IX

CONCLUSION

Ontologies are being widely used to represent application knowledge on Internet applications. Different Web applications sharing compatible ontologies will be able to interact in an automatic way by means of applications agents cooperating among them, to achieve a high degree of automation. The use of a common ontology to describe similar applications for a specific domain is crucial when we talk about application interaction.

The problem of ontology matching, also known as schema matching, is not new. In this thesis, we presented a novel approach to two of its subproblems: first, we presented a solution to (semi)automatic ontology generation from semi-structured data, and second and most important, we presented a set of algorithms and techniques to find matches between pairs of ontologies.

For the first subproblem, we developed a set of heuristics to retrieve and identify the relevant terms present in an HTML page and considered part of the application ontology. For this, we implemented an algorithm that takes advantage of the structure and alignment of input terms contained in form elements inside a Web page. Results showed that we were able to achieve a very high degree of confidence in the identification of terms and their
labels, reaching levels averaging 90% accuracy. Ontologies generated by this method are expressed in XML++, an object-oriented extension to XML, for document representation.

For the ontology matching, we adapted some of the algorithms from the information retrieval area, such as textual, ignorable character removal, stop terms removal, substring and thesaurus match. We also developed our own set of algorithms including hyphen removal, word separator, content and symmetric matching. These algorithms, combined together, are able to identify linguistic/syntactic matches between pairs of ontologies. To identify structural matches, we developed a new technique based on graph matching. This novel approach is based on the calculation of nodes’ weights based on the structural similarity of a node’s children, parents and siblings. All these algorithms were combined to achieve better results, obtaining an average error around 30%.

All these techniques and algorithms were implemented as part of a system called OntoBuilder. This tool was developed using the Java programming language using an expandable architecture based on the Model-View-Controller programming model. It was our intention to make this system a full-featured automatic ontology creation tool which could be used by domain experts working with ontologies. The tool presents a very intuitive and browser-like user interface with visualization aids such as graph and hyperbolic representation of ontologies. The tool also includes an ontology editor that allows users to build ontologies from scratch and to retrieve ontologies from Web sites through a network connection.
Our experiments presented in previous chapters were conducted using our OntoBuilder and were applied to 10 different Web sites from the same application domain, which in our case was the domain of car-rental applications. Some of the Web sites used to test our techniques and tools were Avis, Hertz, Budget and Alamo.

In this work, we also presented a new language for ontology representation based on the XML standard, which we called XML++ due to its object-oriented capabilities. Although not a standard, XML++ has features to handle inheritance, attributes and support for methods, features not supported in their counterparts being used in some ontology editors. XML++ by itself is a whole new research project which can be extended and used not only for ontology representation, but also for document management on the Internet.

9.1 Strengths of this Thesis

It is very important to note that although we are performing ontology extraction and matching from semi-structured data, we believe that the algorithms and concepts introduced in this thesis (especially in chapter VI) are well suited for any kind of data format. Ontology matching can be performed on structured data like XML schemas or relational databases as well. We chose semi-structured data for the challenges it presents and because of the large percentage of Web applications that are still using semi-structured (HTML) pages as a front-end solution to represent their business rules. Future work can be oriented to the adaptation of these techniques to structured data repositories.
Ontology matching, or more generally, schema matching is not limited to Internet applications only. Different areas will benefit from the techniques presented in this work. For example, matching techniques can be used in data warehouse applications where heterogeneous data needs to be consolidated in a central location. In this scenario, the schemas of the different data sources can be mapped using our matching techniques to the schema being used in the central location. Another area of interest for the application of schema matching is Bioinformatics. There are currently different data sources published on the Internet containing information about animal genes and phylogenetic data. Bioinformatics is a dynamic science, with new information being discovered almost every day and datasources needing to be updated constantly due to the nature of the information. In this scenario, manual intervention needs to be minimized as much as possible, presenting an opportunity for automatic data consolidation tools like the one presented in this thesis.

Although different solutions have been proposed in this area, only a few combine linguistic and structural techniques for ontology matching, Cupid [54] being the most representative of all the work implemented. Our work is the only work (to our best knowledge) that applies matching techniques to ontologies automatically generated from Web pages. Current work in the area of ontology matching does not deal with ontology generation but instead works with pre-generated ontologies. This constitutes the main strength of our work.

To conclude, our main contribution to this area of research is twofold: first, we developed techniques to retrieve and dynamically construct Web application ontologies from
semi-structured data, and second, we developed and combined a set of algorithms for ont-
tology matching. We implemented these techniques and algorithms in the OntoBuilder
system, an intuitive and easy to use tool that will alleviate the work of domain experts in
the construction and maintenance of complex ontologies.

We believe that in spite of the use of semi-structured data as the source for ontologies,
we achieved an acceptable degree of confidence identifying and matching ontologies on
the Web. As we outline in the section about future work, it would be interesting to apply
these same techniques to structured data such as XML data files or to relational schemas.

9.2 Lessons Learned

During the elaboration of this thesis, we came across a set of issues that, due to time con-
straints, were excluded from this work. Most of these issues were present in the ontology
generation process. The extraction of ontologies from HTML pages is a very difficult
process, as we have mentioned in different places in this document, mostly because of the
irregularities allowed in HTML pages (HTML is not well-formed). Although the heuristics
developed as part of this thesis achieve a very high degree of confidence for the ontology
extraction, we believe that such methods can be improved.

As time goes by, Web pages will present more dynamic content in the form of DHTML,
plug-ins (such as Flash and Shockwave animations, etc.), server side includes, JavaScript
etc. The implementation and support for all these technologies is outside the scope of
this thesis, however, we strongly believe that it will considerably increase the ontology
generation confidence. Scripting is being supported by most major vendors of browser applications and therefore, Web page designers are using JavaScript as the *de facto* scripting language to handle dynamic content and data validation in their pages. JavaScript can be embedded in OntoBuilder by using scripting engines such as Rhino\(^1\) and some others, currently available as Java libraries.

Better parsing engines for HTML documents can be used to generate better ontologies. Although the use of Java as a programming language was obvious at the beginning, we discovered that HTML parsing in Java is still in an early stage (by the time this thesis was written, there were no high-end browsers in Java). HTML parsing performed by the two most used browsers on the Web (Netscape and Microsoft Internet Explorer) is the best parsing available. We would suggest that future releases of the tools be developed by using Visual C++ and Internet Explorer (IE) components. IE browser components have built-in support for scripting, plug-ins, cookies, etc., minimizing the amount and quality of work required to reimplement such features in Java.

Another issue that could improve our results is the identification of deeper hierarchies within a Web page. Terms can be grouped together based on the visual area in which they are displayed in a browser. Section identification in Web pages can be achieved by analyzing the formatting structure of the HTML document (by using tags such as headings (H), table headers (TH), color attributes, etc.).

\(^1\)http://www.mozilla.org/rhino
9.3 Future Work

The problem of schema matching is a very broad area of research. In this thesis we only presented an approach to schema matching on semi-structured data retrieved from the Internet. The work presented here opens a wide possibility of research extensions for information retrieval and database areas.

New matching techniques, such as machine learning techniques, can be adapted and implemented into the OntoBuilder tool. Graph matching techniques such as the ones implemented in Cupid can also be adapted to our matching algorithms. For instance, the graph matching algorithm can be modified to make use of user specified matches that aid the matching process; the user can specify beforehand a few matches that are key in the ontologies being used, or the system can make use of historical information from previous matchings. Another technique that can be very helpful is the use of external dictionaries that are specific to the domain of the ontologies being matched; such dictionaries are a little more complex than the current thesauri being used, including information about terms relationships. To illustrate how a dictionary can improve the matching process, suppose that there are two ontologies that define information about clients; in the first ontology, the client name is represented by a single term named name, while in the second ontology, the same concept is represented by two terms firstName and lastName. With a dictionary specifying compound terms such as name=firstName+lastName, it is possible to match multiple terms to single terms and improve the recall and precision of the overall operation.
Another interesting research extension is oriented towards the calculation of matching expressions. In this thesis, we presented algorithms that produce single matches only between a pair of terms. As shown in the previous paragraph when we discussed the compound name term, complex matches are possible. In these cases, it is very useful to also produce an expression to achieve the match. For example, in the case of the name term, a valid expression would be a concatenation of two or more terms such as \( \text{name} = \text{concat}\{\text{firstName}, \text{lastName}\} \). Expressions can also be calculated using arithmetic expressions, such as terms that are a product of some other terms, \( \text{total} = \text{unitPrice} \times \text{qty} \). In these cases, it may be useful to look at instance data to “guess” some data pattern.

One of the extensions left for future work is the implementation of query translation capabilities into the system. By query translation, we mean the translation of queries against the global ontologies to their equivalent query expressed in terms of the local ontologies. This kind of extension is very useful for data integration and interoperability. Although very easy to implement once the global ontology is built, there are some issues that need further attention. One of the most important issues to take into consideration is what happens when the user performs a query that includes terms that are not common to all the underlying local ontologies, but for some reason were included in the global ontology. In this case, the system has to decide whether to use those terms for those ontologies defining it, and not to use them for those ontologies not defining them, or, to
ignore those terms by not using them in the translation. More issues similar to this one can also arise and are left for further studies in this area.

As we pointed out, resulting ontologies (ontologies produced as a result of a matching between two or more ontologies) can be expressed in different formats. For this thesis, we chose to represent ontologies using a proprietary format extended by a proprietary language extension such as XML++. Future work would represent ontologies using some of the standard languages, currently under revision or at draft stage, such as RDF and DAML+OIL.

We encourage the application of the concepts presented in this thesis to a wider range of domains besides car-rental and flight reservation. Some of the domains for which ontology matching would be very useful is in Bioinformatics, where similar phylogenies can be identified automatically, database schema integration, where heterogeneous data sources can be mapped to one another, and Web service discovery, where similar services can be identified to fulfill particular needs.
REFERENCES


