ONTOLOGY-BASED EXTRACTION OF RDF DATA FROM THE WORLD WIDE WEB

by
Tim Chartrand

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science
Brigham Young University
March 2003

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Tim Chartrand

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	David W. Embley, Committee Chairman
Date	Stephen W. Liddle, Committee Member
Date	Kent E. Seamons, Committee Member

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Tim Chartrand in its final form and have found that (1) its format, citations and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

David W. Embley, Committee Chairman

Accepted for the Department

David W. Embley, Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean,
College of Physical and Mathematical Sciences

ABSTRACT

ONTOLOGY-BASED EXTRACTION OF RDF DATA FROM THE WORLD WIDE WEB

Tim Chartrand
Department of Computer Science
Master of Science

The simplicity and proliferation of the World Wide Web (WWW) has taken the availability of information to an unprecedented level. The next generation of the Web, the Semantic Web, seeks to make information more usable by machines by introducing a more rigorous structure based on ontologies. One hinderance to the Semantic Web is the lack of existing semantically marked-up data. Until there is a critical mass of Semantic Web data, few people will develop and use Semantic Web applications. This project helps promote the Semantic Web by providing content. We apply existing information-extraction techniques, in particular, the BYU ontology-based data-extraction system, to extract information from the WWW based on a Semantic Web ontology to produce Semantic Web data with respect to that ontology. As an example of how the generated Semantic Web data can be used, we provide an application to browse the extracted data and the source documents together. In this sense, the extracted data is superimposed over or is an index over the source documents. Our experiments with ontologies in four application domains show that our approach can indeed extract Semantic Web data from the WWW with precision and recall similar to that achieved by the underlying information extraction system and make that data accessible to Semantic Web applications.

ACKNOWLEDGMENTS

I would like to thank the following people. First, my wife Kathleen for her help and support to me and to our son Tyler throughout my education. Second, Dr. David Embley, my graduate advisor, for helping with my research and especially with my writing. Third, the National Science Foundation for supporting this research under grant #IIS-0083127. Finally, Michael Coleman, Michael Gilleland, William C. Jones, Jr., Martin Porter, the HP Labs Jena team, and members of the BYU Data-Extraction Group for all the code that provided a basis for my project implementation.

Contents

Chapter 1

Introduction

The advent of the World Wide Web (WWW) has taken the availability of information to an unprecedented level. The simplicity of the Web has been a major factor in its proliferation [KM02]. Anyone can easily publish a document about anything or link to anyone's site. The document need not be structured according to any particular format or even contain correct information, and the link need not be valid [BLHL01]. Placing such restrictions on the Web would have increased the level of expertise necessary to create Web content and decreased the amount of useful information available.

Although the unstructured and unregulated nature of the Web makes publishing information easier, it also makes finding and using information much more difficult. The typical way to find information on the Web today is to do a keyword search using a search engine, which tries to guess the meaning of the combination of keywords in the query and the meaning of each page it has indexed to find pages relevant to the query. Even when we can find the desired information, it is usually very difficult for a machine to interpret, so the user generally has to manually review and use the information.

1.1 Semantic Web

Research is underway in universities and companies around the world to develop the next generation of the Web, called the *Semantic Web*. The Semantic Web will add

meaning, or semantics, to Web content in order to make it easier to find and use for both humans and machines. Adding formal semantics to the Web will aid in everything from resource discovery to the automation of all sorts of tasks [KM02].

As an example, consider the Web site for a medical practice. A typical WWW site might contain the names and qualifications of the doctors and the business hours and contact information of the office. The problem with such a site is that, since there is no standard format for medical-practice Web sites, a person would have to manually inspect the information. A Semantic Web site for the same practice might contain the same information in a machine-understandable format and might also specify times that are available for appointments. A person could then instruct a Semantic Web software agent to find a doctor who specializes in pediatrics with an available appointment time Thursday morning and report the doctor's name, contact information, and available appointment times [BLHL01].

The basic data model used to build the Semantic Web is called the Resource Description Framework (RDF) [LS99][MM02]. RDF is a domain-independent model for describing resources, where a resource is anything that can be represented by a Uniform Resource Identifier (URI), including Web pages, parts of Web pages, or even physical objects. RDF describes resources by making statements about resources in the form <subject><predicate><object>. The subject is the URI of some resource, the resource being described. The predicate, also represented by a URI, expresses some relationship between the subject and the object. The object is either a literal or another resource represented by a URI.

For example, if we want to say Tim Chartrand is 25 years old, we would write the statement:

<mailto:tim@cs.byu.edu><genealogy#age>"25"[1]

Or to say Tim Chartrand is the father of Tyler Chartrand:

<mailto:tim@cs.byu.edu><genealogy#fatherOf><mailto:tyler@thechartrands.com>

Thus, RDF allows us to remove ambiguity: there may be more than one person with the name Tim Chartrand, but there is exactly one person represented by the

[1]The property denoted genealogy#age is shorthand for the URI http://www.thechartrands.com/genealogy#age. The full URI is omitted for readability. The same applies for the genealogy#fatherOf property that appears later.

URI <mailto:tim@cs.byu.edu>. So some agent, either a human or a program, searching for information about Tim Chartrand would look for statements with <mailto:tim@cs.byu.edu> as the subject or object.

RDF helps give structure to Web content, but what kind of resource is `mailto:tim@cs.byu.edu` and how is the `genealogy#fatherOf` predicate related to it? Ontologies (i.e. vocabularies, schemas) that define classes of objects and their properties answer these questions. RDF Schema (RDFS) is a simple ontology language written in RDF that allows the creation of vocabularies with classes, properties, and subclass/superclass hierarchies [BG02]. DAML+OIL[2](DARPA Agent Markup Language + Ontology Inference Layer) is an extension of RDFS that allows finer-grained control of classes and properties with features such as cardinality constraints and inverses of properties [CvHH+01].

As an example of how to use an ontology, suppose we have an ontology called *genealogy* that defines a class *Person* with a property *fatherOf* whose value is a *Person*. We would add to the RDF from the above example a specification that `mailto:tim@cs.byu.edu` and `mailto:tyler@thechartrands.com` both represent objects of type *Person* (i.e. they belong to class *Person*). This tells us that everything we know about a *Person* directly applies to the given resources; if every *Person* has a *Name*, then the object identified by `mailto:tim@cs.byu.edu` must also have one. Further, suppose that our ontology defines a property called *parentOf* as a generalization of *fatherOf* and a property *childOf* as the inverse of *parentOf*. Without any extra work on our part, a fairly general Semantic Web agent would be able to infer that Tyler Chartrand is a child of Tim Chartrand, even though we only stated that Tim Chartrand is the father of Tyler Chartrand.

With all the advantages of the Semantic Web, what keeps it from reaching a critical mass where it will gain widespread acceptance and use? One reason is the newness of the area and the related tools to help people publish and use Semantic Web content. Another reason, equally important, and the one that this research addresses, is the lack of useful content. For years people have been publishing Web documents on nearly every topic imaginable and building systems to continually generate new content, so there is a vast amount of human readable information on the Web. It is hard to imagine rewriting the current Web content to be accessible to Semantic

[2]Hereafter we will refer to DAML+OIL as simply DAML.

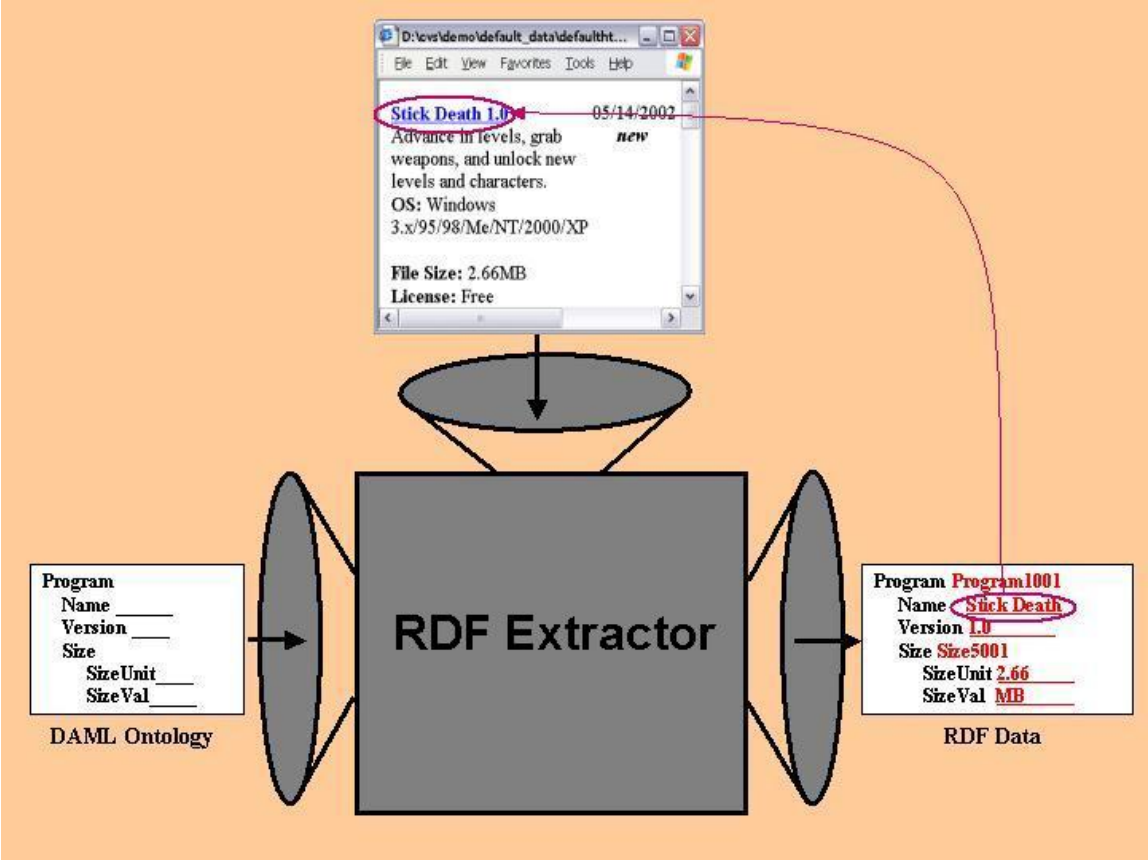

Figure 1.1: Research Overview

Web agents. As long as newspapers, for example, continue to generate simple HTML documents for their daily car advertisements or obituaries, Semantic Web agents will have a hard time performing useful tasks with that information.

1.2 Research Overview

This project helps bridge the gap between the WWW and the Semantic Web. We take advantage of information in existing Web pages to semi-automatically create Semantic Web data. Figure 1.1 shows an overview of our RDF-extraction system. The first input to the system, on the left in Figure 1.1, is a DAML ontology that describes the structure and constraints of the desired Semantic Web data. The second input, at the top of Figure 1.1, is a set of data-rich, multi-record Web pages whose data is

in the application domain described by the ontology. As output, shown on the right in Figure 1.1, the system produces RDF data, with respect to the input ontology, corresponding to the data in the input Web pages.

This RDF data is accessible to Semantic Web agents that understand the input ontology. As an example of how the extracted data can be used, we have created an application that allows a user to browse the extracted RDF as an index into the original document. Figure 1.1 shows that the *Name* `Stick Death` was extracted from the beginning of the first line of the Web page.

1.3 Supporting Research Areas

The biggest task in making WWW data accessible to Semantic Web agents is extracting the data from Web pages. There is an entire field of research called Information Extraction or Data Extraction that tries to extract unstructured or semistructured Web content so it can be stored and queried more efficiently [Eik99][LRNdST02].

The BYU Data Extraction Group (DEG) [Hom02] has developed an ontology-based data-extraction system called Ontos [ECJ+99]. Ontos uses a data-extraction ontology written in an extension of the OSM modeling language [Emb98][LEW95]. A data-extraction ontology describes the structure of the desired data and contains detailed extraction rules for each item of information to be extracted. Given an ontology for a particular domain, Ontos extracts data from Web pages containing information in that domain and stores the data in a relational database. We extend this extraction approach to extract information from Web pages and structure it as RDF.

The field of superimposed information is another area of research touched by this project. The idea of superimposed information is to make use of the connection between a document and data extracted from it so that the extracted data forms another layer that can be superimposed over the document [MD99][BDM02].

Since we are extracting Semantic Web data from unstructured documents, we keep track of the original location of each extracted data item in the source documents. Thus, the extracted data is superimposed over, or is an index into, the original data. To show how our extracted RDF can be used and to illustrate the principle of

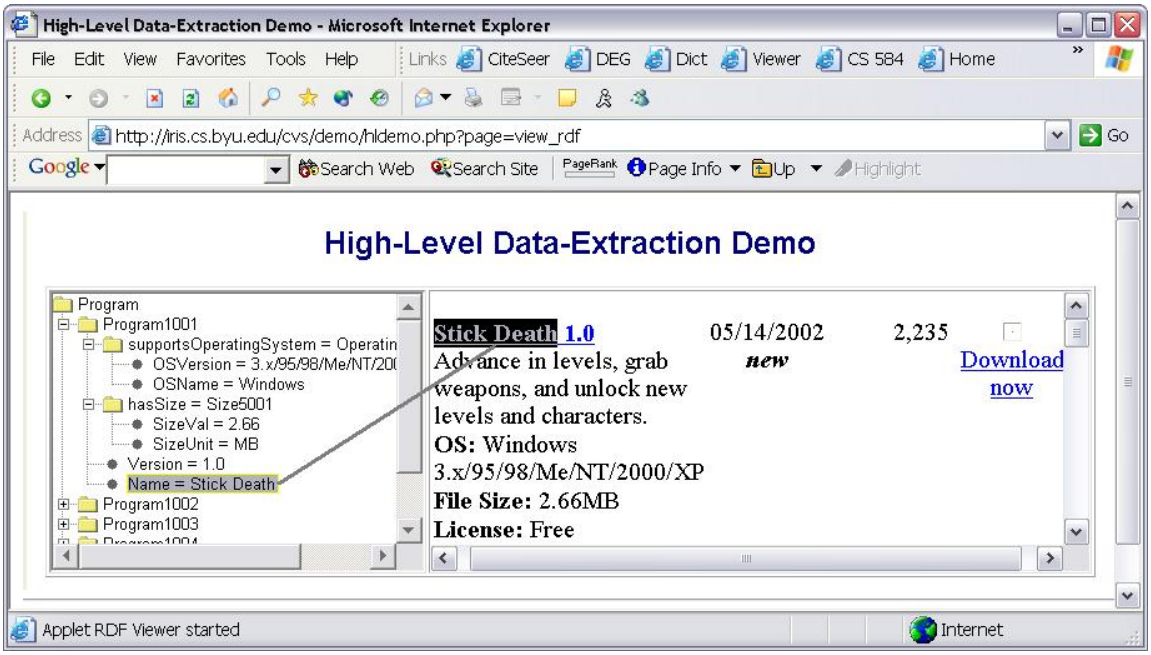

Figure 1.2: Example of RDF used as Superimposed Information

superimposing information, we have created an RDF browser that allows the user to view the RDF and the original document together in a superimposed fashion. Figure 1.2 shows our RDF browser. A user browses the RDF in the left-hand frame by navigating through the classes in the ontology. While a user navigates through the classes, our browser shows class instances extracted from a Web page. A user selects a property value in the left-hand frame to highlight in the right-hand frame the place in the original document from which the value was extracted.

1.4 Overview of Thesis

Figure 1.3 presents a more detailed overview of the research embodied in this project. The inputs and outputs of the system are as explained with Figure 1.1. We take a DAML ontology as input and semiautomatically convert it into an OSM data-extraction ontology. The generated data-extraction ontology becomes the input, along with a set of multi-record Web pages, to the BYU Ontos system, which extracts relational data as output. The center of Figure 1.3 shows the inputs and outputs of Ontos. Next, we structure the extracted relational data as RDF. Finally, we create

an RDF browser as an example of how the the extracted data can be used.

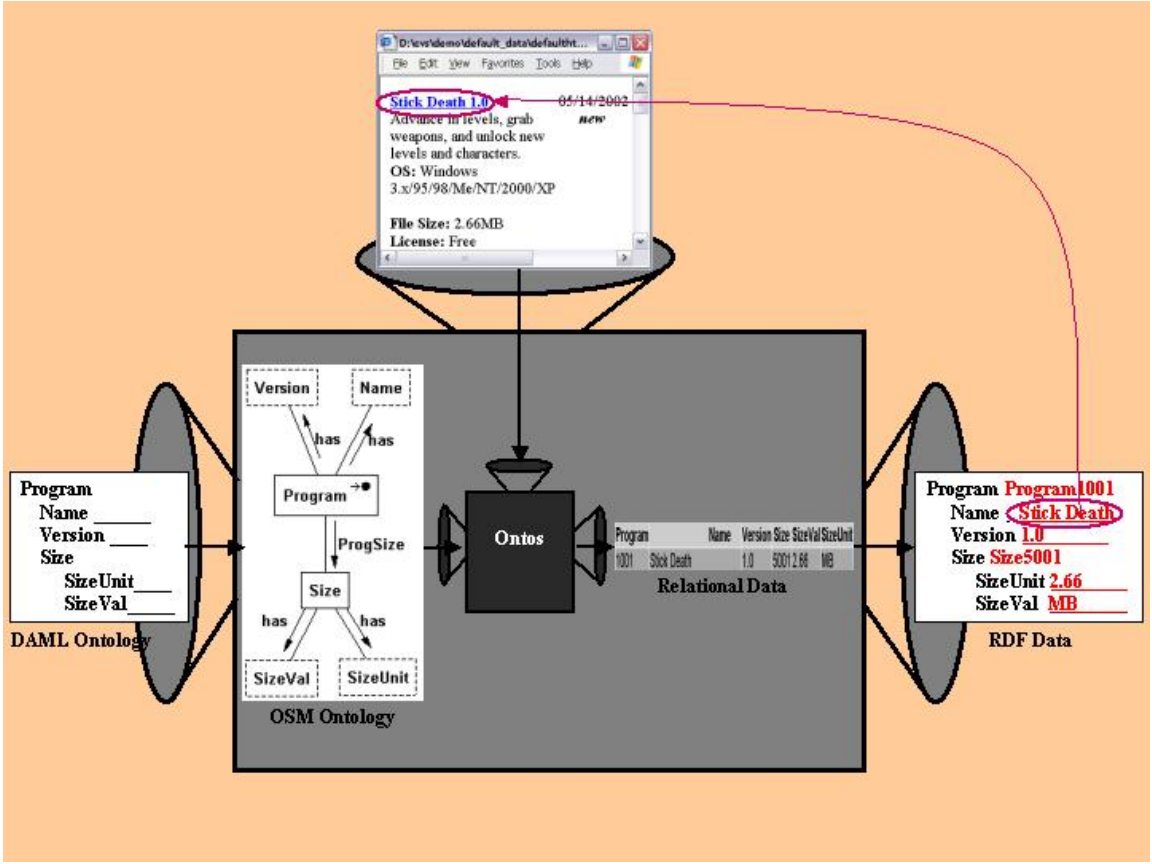

Figure 1.3: Detailed Research Overview

This thesis describes our implementation of the RDF-extraction system. Chapter 2 gives background on the DAML and OSM languages and presents an algorithm for converting a DAML ontology into an OSM data-extraction ontology. It also describes the tool we provide for a user to supply the details of an extraction ontology that are not present in the original DAML ontology. Chapter 3 discusses how we have adapted the BYU Ontos tool to extract data and how we convert that data to RDF. Chapter 4 describes our RDF browser for viewing RDF along with a Web document. Chapter 5 analyzes the performance of our RDF-extraction system. Chapter 6 discusses related work, and Chapter 7 gives our conclusions and suggests possible directions for future research.

Chapter 2

Converting DAML Ontologies to Data-Extraction Ontologies

To use a DAML ontology for data extraction, we first convert the ontology to an OSM model instance. Then we extend the model instance to include data recognizers. Finally, we allow the user to edit data recognizers and constraints of the extraction ontology.

In this chapter, we first describe the DAML language in Section 2.1 and the OSM language in Section 2.2. Section 2.3 describes an algorithm for converting from DAML to OSM including properties, restrictions, and classes. Section 2.3 also discusses limitations of the algorithm and gives some notes on our implementation. Section 2.4 describes a matching algorithm for automatically adding data recognizers to an extraction ontology, and Section 2.5 describes the tool we provide a user to edit the produced extraction ontology.

2.1 Overview of DAML

DAML is a language for creating ontologies, where an ontology is a set of concepts and the relations that exist among them [dam02]. Like many conceptual-modeling languages, DAML allows for the specification of classes of objects (i.e. entity types or object sets) and properties (i.e. associations or relationship sets) [CvHH+01]. Figure 2.1 shows an example DAML ontology.

```
1)      <daml:Class rdf:ID="Program">
2)      </daml:Class>
3)      <daml:Class rdf:ID="Size">
4)      </daml:Class>
5)      <daml:Class rdf:ID="OperatingSystem">
6)      </daml:Class>
            .  .  .
7)      <daml:DatatypeProperty rdf:ID="Name">
8)          <rdfs:domain rdf:resource="#Program"/>
9)          <rdfs:range rdf:resource="&rdfs;Literal"/>
10)         <rdf:type rdf:resource="&daml;UniqueProperty"/>
11)         <rdf:type rdf:resource="&daml;UnambiguousProperty"/>
12)     </daml:DatatypeProperty>
13)     <daml:ObjectProperty rdf:ID="hasSize">
14)         <rdfs:domain rdf:resource="#Program"/>
15)         <rdfs:range rdf:resource="#Size"/>
16)     </daml:ObjectProperty>
17)     <daml:ObjectProperty rdf:ID="supportsOperatingSystem">
18)         <rdfs:domain rdf:resource="#Program"/>
19)         <rdfs:range rdf:resource="#OperatingSystem"/>
20)     </daml:ObjectProperty>
            .  .  .
```

Figure 2.1: Example DAML Ontology

DAML is a property-centric rather than class-centric language. Whereas many languages define a class in terms of its properties, DAML defines a property in terms of the classes to which it applies. Specifically, DAML defines a property in terms of its domain (*daml:domain*) and range (*daml:range*), where the domain is the set of classes whose instances can *have* a value for the property, and the range is the set of classes whose instances can *be* the value for the property.

For example, the DAML ontology in Figure 2.1 defines a *hasSize* property (Line 13) with a domain of class *Program* (Line 14), and a range of class *Size* (Line 15), meaning that a *Program* has an associated *Size*. DAML also provides a way to specify inverse relationships between properties with *daml:inverseOf*. For example, *sizeOf* could be defined as the *daml:inverseOf hasSize*, meaning that a *Size* is associated with a *Program*.

A DAML ontology can define a property as one of two types: an object property (*daml:ObjectProperty*) or a data-type property (*daml:DatatypeProperty*). A data-type

property is one whose values are lexical, and an object property is one whose values are objects, or instances of a class. An ontology can also define a property simply using *daml:Property*, meaning that the type of values of the property is not known or not specified. A DAML ontology can also specify simple cardinality constraints on a property. If a property is a *daml:UniqueProperty*, any object can *have* at most one value for the property. If a property is a *daml:UnambiguousProperty*, any object can *be* the value for at most one instance of the property.

The ontology in Figure 2.1 defines the *Name* property as a *daml:DatatypeProperty* (Line 7), since the name of a program is lexical, and the *hasSize* property as a *daml:ObjectProperty* (Line 13), since its values are instances of class *Size*. The *Name* property is also a *daml:UniqueProperty* (Line 10), meaning that a program has at most one name, and a *daml:UnambiguousProperty* (Line 11), meaning that a name identifies at most one program.

DAML allows for arbitrary generalization/specialization hierarchies. A class can specify its superclass(es) using *daml:subClassOf* or *daml:intersectionOf*, and can specify subclass(es) with *daml:unionOf* or *daml:disjointUnionOf*. For example, we can define *Utility* and *OpenSourceProgram* as *daml:subClassOf Program*, so that every *Utility* and every *OpenSourceProgram* is also a *Program*. Further we can define *OpenSourceUtility* as the *daml:intersectionOf Utility* and *OpenSourceProgram*, so that every *OpenSourceUtility* is also a *Utility* and an *OpenSourceProgram*.

In addition to generalization/specialization, DAML also allows for other relationships among classes and properties to be specified. To say that a class or property is the same as or equivalent to another class or property, we use *daml:equivalentTo*, *daml:sameClassAs*, or *daml:samePropertyAs*. To say that one class is the global complement of another, we use *daml:complementOf*. For example, an ontology author might declare the *hasFileSize* property in one ontology the same property as or equivalent to the *hasSize* property in another ontology.

The domain and range for a property are global constraints that apply to every instance of the property, but we can also specify local constraints on a property for a given class (for a subset of the property's domain). Specifying constraints on a property effectively constrains the class by limiting the type and number of values an instance of that class can have for the given property. A restriction on a property restricts the cardinality or range of the property. A property restriction is an

```
            .  .  .
1)      <daml:Class rdf:ID="WindowsProgram">
2)          <daml:subClassOf rdf:resource="&software;Program"/>
3)          <rdfs:subClassOf>
4)              <daml:Restriction daml:minCardinality="2">
5)                  <daml:onProperty
                        rdf:resource="&software;supportsOperatingSystem"/>
6)                  <daml:toClass rdf:resource="#WindowsOperatingSystem"/>
7)              </daml:Restriction>
8)          </rdfs:subClassOf>
9)      </daml:Class>
10)     <daml:Class rdf:ID="WindowsOperatingSystem">
11)         <daml:subClassOf rdf:resource="&software;OperatingSystem"/>
12)     </daml:Class>
            .  .  .
```

Figure 2.2: Example DAML Ontology Showing a Property Restriction

anonymous class (i.e. an unnamed class used only to specify constraints) that includes any object that satisfies the constraints of the restriction for the given property. To specify local cardinality or range constraints on a property for a class, an ontology declares the class to be the subclass of a restriction, meaning that every member of the class is also a member of the restriction class and therefore satisfies the constraints of the restriction. Figure 2.2 shows a restriction on the *supportsOperatingSystem* property for the *WindowsProgram* class (Lines 3-8).

A restriction can use *daml:minCardinality* or *daml:maxCardinality*, to specify the minimum or maximum number of values that a particular object can have for the property or *daml:cardinality* to specify both the minimum and maximum. For example, the restriction in Figure 2.2 restricts the *supportsOperatingSystem* property with a *daml:minCardinality* of two (Line 4) for the *WindowsProgram* class, meaning that a *WindowsProgram* must support at least two *OperatingSystem*s.

A property restriction restricts the range of a property using *daml:toClass* and *daml:hasClass*. A restriction can use *daml:toClass* to say that *every* value for the property must be an object in the specified class (a local range constraint), or it can use *daml:hasClass* to say that any object satisfying the restriction has *at least one* value for the property that belongs to the specified class. For example, the restriction in Figure 2.2 restricts the range of the *supportsOperatingSystem* property to

WindowsOperatingSystem (Line 6), so that a *WindowsProgram* supports only *WindowsOperatingSystem*s.

2.2 Overview of OSM Data-Extraction Ontologies

OSM (Object-oriented Systems Model) is a conceptual-modeling language based on modeling objects [Emb98][LEW95]. OSM consists of three submodels: the Object Relationship Model, which represents objects and relationships among objects, the Object Behavior Model, which represents the behavior of an object, and the Object Interaction Model, which represents interactions among objects. Since we are only interested in describing the structure of data and not behavior or interaction, we use only the Object-Relationship Model. A data-extraction ontology is an OSM model instance augmented with data recognizers. Figure 2.3 shows a graphical representation of an OSM model instance corresponding to the DAML ontology in Figure 2.1[1].

In OSM, an object is either lexical or non-lexical. A lexical object represents itself, and a non-lexical object is represented by an arbitrary object identifier (OID). For example, the *Name* object *'Stick Death'* is lexical because *'Stick Death'* is the object, whereas the *Program* object *Program1001* is non-lexical because the OID *Program1001* is only an identifier for the object and not the object itself. To relate objects with each other, OSM uses relationships. For example, the relationship *Program1001 has name 'Stick Death'* says that the *Program* represented by OID *Program1001* is named *'Stick Death'*.

OSM groups similar objects into object sets, which are lexical or non-lexical depending on the type of objects in the object set. Similarly, OSM groups relationships into relationship sets. A relationship set has two or more connections to one or more object sets. The name of a relationship set contains the name of each connected object set along with a verb phrase describing how the object sets are related. For example, to relate the *Program* and *Name* object sets to say a *Program* has an associated *Name*, we would create the *Program has Name* relationship set. Figure 2.3

[1]Figure 2.1 does not show the entire DAML ontology corresponding to the OSM model instance in Figure 2.3. Because the full ontology is about 80 lines long, the figure shows only the parts necessary for illustration.

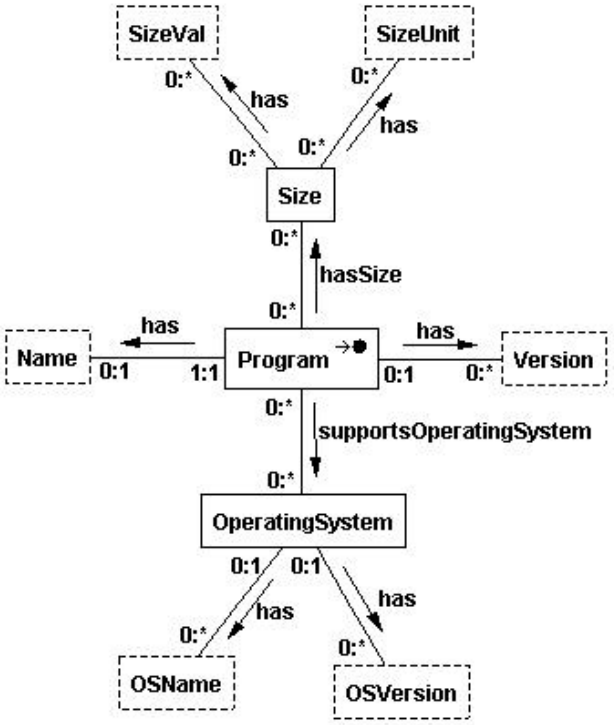

Figure 2.3: Example OSM Model Instance

shows the *Program* and *Name* object sets connected by the *Program has Name* relationship set. The reading-direction arrow next to the label for the relationship set (*has*) tells us how to construct the name for the relationship set — we concatenate the name of the tail object set, the label, and the name of the head object set to form the relationship-set name.

Object and relationship sets allow us to define participation constraints that specify how many times an object in a given object set can participate in a relationship set. For example, Figure 2.3 shows minimum and maximum participation constraints of *1* on the *Program* side of the *Program has Name* relationship set to say that every *Program* has exactly one name. The *Name* side of the relationship set has a minimum participation constraint of *0* and a maximum of *1* to say that a *Name* is the name of at most one *Program*. We write this textually as *Program [1:1] has Name [0:1]*.

OSM also provides for generalization/specialization among object sets. So we can say that *Client* is a specialization of *NetworkUtility* or *Client isa NetworkUtility*. If a generalization has two or more specializations we can say that the generalization is

the union of the specializations, for example *Client, Server isa [union] NetworkUtility* says that every *NetworkUtility* is either a *Client* or a *Server*. If the specializations of a generalization are mutually exclusive (non-overlapping) object sets we use the mutual-exclusion (mutex) constraint, for example *Client, Server isa [mutex] NetworkUtility* says that *Client* and *Server* are disjoint object sets. If both the union and mutex constraints apply, we say that the specializations partition the generalization, and we use the partition constraint as short hand for the other two, for example *Client, Server isa [partition] NetworkUtility*.

OSM also allows a specialization to have multiple generalizations, for example *OpenSourceServer isa Server, OpenSourceProgram*, meaning that every *OpenSourceServer* is both a *Server* and an *OpenSourceProgram*. We can use the intersection constraint to say that the specialization is *exactly* the intersection of its generalizations, rather than some subset of their intersection, for example *OpenSourceServer isa [intersection] Server, OpenSourceProgram* says that every object that is both a *Server* and an *OpenSourceProgram* is an *OpenSourceServer*.

For constraints not expressible as participation constraints or other types of OSM constraints, OSM provides general constraints. A general constraint is an arbitrary closed predicate-calculus formula that uses only predicates corresponding to the object and relationship sets in the model instance. For example, suppose that we have an object set *Revision* and a relationship set *Revision precedes Revision*. To specify that the relationship set is transitive, we would add the following general constraint:
$\forall x \forall y \forall z (Revision(x) precedes Revision(y) \wedge Revision(y) precedes Revision(z) \Rightarrow Revision(x) precedes Revision(z))$

To use an OSM model instance as a data-extraction ontology, we need to know how to recognize lexical objects in the text from which the data is to be extracted. For this purpose, we associate a data frame [Emb80] with each lexical object set. A data frame contains a recognizer with a detailed description of how a lexical object generally appears in an HTML document. A data frame uses regular expressions to specify how the data itself appears as well as regular expressions for the immediate context of the data. Data frames also include keywords that may appear in the document close to the data to be extracted. For example, a *ReleaseDate* data frame would have a regular expression to recognize a date and the keyword *released* [ECJ+99].

2.3 Automatic Translation of Structure and Constraints

The main constructs in DAML correspond closely to the main constructs in OSM. This section describes an algorithm for translating a DAML ontology into an equivalent or nearly equivalent OSM model instance. Since DAML is a property-centric language, we start with the translation of properties and their restrictions and then move on to the translation of classes.

It is important to note that the motivation for this algorithm is to aid in the data-extraction process. Therefore, when faced with a tradeoff between making the translated OSM more equivalent to the DAML and making the OSM more useful for data extraction, we choose the latter. We list some specific limitations and notes on our implementation at the end of this section.

2.3.1 DAML Properties

The first step in the conversion algorithm processes DAML properties. There are two basic types of properties in DAML: object properties, whose values are non-lexical and data-type properties, whose values are lexical. A property of either type translates to a binary OSM relationship set.

To determine if a property is an object property or a data-type property, we first look at the property's type; an ontology may specify the property explicitly as an object or data-type property. If the ontology does not specify the type of the property, we must look at the range of the property. If the range of the property is the special class *rdfs:Literal* or one of the XML Schema data types, we know that the range has lexical values and therefore the property is a data-type property. Otherwise the property is an object property.

In the case of an object property, whose range is not lexical, we simply create a relationship set between the object sets corresponding to the domain and range classes of the property. We assign the property name as the name of the relationship set. For example, the *hasSize* object property in Figure 2.1, with a domain of *Program* and a range of *Size* (Lines 13-16), becomes the *Program hasSize Size* relationship set

in Figure 2.3.

If the property is a data-type property, meaning that values of the property can be represented as strings, we create both a lexical object set and a relationship set. We name the lexical object set with the name of the property and give the relationship set the generic name *has*. For example, the *Name* data-type property in Figure 2.1, with a domain of *Program* (Lines 7-12), becomes the *Name* lexical object set and the *Program has Name* relationship set in Figure 2.3.

DAML provides two ways to specify cardinality constraints that apply globally to a property. A property can either specify its type as a *daml:UniqueProperty* or a *daml:UnambigousProperty* or both. If a property is unique, we place a one-max participation constraint on the domain side of the generated relationship set. If a property is unambiguous, we place a one-max participation constraint on the range side of the relationship set. For example, the *Name* data-type property in Figure 2.1, with a domain of *Program* (Lines 7-12), is unique and unambiguous, so we place a one-max participation constraint on both connections of the generated *Program has Name* relationship set in Figure 2.3. For properties that do not specify cardinality constraints, participation is unconstrained, meaning that an object can participate zero or more times in the relationship set. More complex cardinality constraints are specified by local property restrictions, which we discuss in the next subsection.

Up to this point, we have assumed that a property specifies both a domain and a range that we can use to construct a relationship set. If a property is missing a domain or a range or both, we may still be able to create a relationship set if we can infer the domain and range of the property. We can infer the domain and range of a property three different ways. First, we can use local property restrictions (discussed in the next subsection). Second, if the property is the *daml:subPropertyOf* another property, we can use the domain and range from the superproperty. If the subproperty specifies the range but not the domain, we use the domain from the superproperty and the range from the subproperty. Similarly, if the subproperty specifies the domain but not the range, we use the range from the superproperty and the domain from the subperoperty. Third, a property can define itself as the *daml:inverseOf* another property. In this case we use the same technique as for superproperties, but we interchange the domain and range. If none of these three cases apply, we cannot create a relationship set for the property.

2.3.2 DAML Property Restrictions

The second step in the algorithm applies property restrictions. A restriction constrains the range or cardinality of a property for a class when the class is a part of the domain for the property. A restriction can also help us infer global domain and range constraints of a property for which a global domain or range is not given explicitly.

A restriction is applied *to* a property *for* a class (the subclass of the restriction). If the domain was not explicitly specified for the property, then we can use the class (the subclass of the restriction) as the domain. This may not yield the correct domain for the property because a domain may have been specified outside of our current model, but we do know that the class is a subset of the property's domain. Furthermore, it is reasonable to infer the domain in this way because it is likely that the class is the part of the domain in which the ontology author is interested, so extraction based on the inferred domain will produce the desired results. As an example, the ontology in Figure 2.1 (Ontology A), defines a property *supportsOperatingSystem* with a domain of *Program* and a range of *OperatingSystem* (Lines 17-20), and the ontology in Figure 2.2 (Ontology B — the ontology we are translating) defines a restriction to class *WindowsOperatingSystem* applied to the *supportsOperatingSystem* property for the subclass *WindowsProgram* of *Program* (Line 2-9). Assuming Ontology B does not explicitly import Ontology A, Ontology B does not know the real domain of *supportsOperatingSystem*, but for translation, it is reasonable to use *WindowsProgram* as the domain.

A restriction may also specify local range constraints with values for *daml:toClass* or *daml:hasClass*. Once again, if a global range for the property is not given, we know that the local range is a subset of the global range. By the same argument as for inferring the domain, we can also infer the global range from the local range. For example, in translating ontology B in Figure 2.2, we would use *WindowsOperatingSystem* as the range for the *supportsOperatingSystem* property.

If a restriction gives a local range constraint and a range is already known for the property, the constraint is translated into an OSM general constraint. We use general constraints in several places in the translation to preserve the semantics of the original ontology, but since they currently have no effect on the extraction process, we will not describe each generated general constraint beyond mentioning that it is

created.

We handle cardinality restrictions in one of two ways. We either generate participation constraints on the relationship set generated for the restriction's property, or we create general constraints. In the case where the restriction provides a domain or range for the property so we can create a relationship set, it makes sense to create participation constraints on the relationship set. We place the generated participation constraints on the domain side of relationship set. For *daml:minCardinality* we create a minimum participation constraint, for *daml:maxCardinality*, a maximum participation constraint, and for *daml:cardinality*, both. For example, in translating the ontology in Figure 2.2, we would create a minimum participation constraint of two on the *WindowsProgram* side of the generated *WindowsProgram supportsOperatingSystem WindowsOperatingSystem* relationship set for the *daml:minCardinality* value of two for the restriction (Line 4). In the other case, where a relationship set is created before the restriction is processed, we create general constraints from the cardinality restrictions.

2.3.3 DAML classes – Generalization and Specialization

The last step of the algorithm processes all the DAML classes to build generalization/specialization hierarchies. At first glance, it might appear that we should also use the classes to simply add an object set for each class. However, classes that are not associated with any property or any generalization/specialization would translate to isolated non-lexical object sets. Isolated non-lexical object sets are not useful in the data-extraction process because data extraction focuses on lexical objects (those objects that can actually be extracted from text) and the non-lexical object sets related to them, so no instances would ever be generated for isolated non-lexical object sets. Our algorithm therefore only adds object sets for non-isolated classes.

DAML provides several attributes that a class may use to specify its relationship to other classes. Most of these attributes correspond very closely with OSM constructs. If a class, *subClass*, is a *daml:subClassOf* another class, *superClass*, we add an OSM generalization/specialization (GenSpec) with *subClass* as a specialization and *superClass* as a generalization. For example, translating the *daml:subClassOf* property for *WindowsProgram* with value *Program* in Figure 2.2 (Line 2) would

produce the GenSpec *WindowsProgram isa Program*. If a class, *subClass*, is the *daml:intersectionOf* a set *superClasses* of classes, we add a GenSpec with *subClass* as a specialization and each of *superClasses* as a generalization and add the intersection constraint. If a class, *superClass*, is the *daml:unionOf* or *daml:disjointUnionOf* a set *subClasses* of classes, we add a GenSpec with *superClass* as a generalization and each of *subClasses* as a specialization and add the union constraint. In the case of *daml:disjointUnionOf* we add the partition constraint rather than the union constraint. If a class is *daml:disjointWith* another class we save that information so we can either combine GenSpecs, as described below, or make a general constraint. If a class is the *daml:complementOf* another class, we add a general constraint.

Once we have saved all of the generalization/specialization, union, and mutual exclusion (disjoint) information, we combine it to obtain a minimal set of GenSpecs. We consolidate GenSpecs of each object set *genObjSet* as follows: first, we collect specializations of GenSpecs with *genObjSet* as the generalization and with only one specialization and remove the GenSpecs from our model instance. Those GenSpecs with more than one specialization will be be unions, and therefore cannot receive more specializations, so those GenSpecs are not affected by this algorithm. Next, we group the collected specializations in as few large groups as possible such that the members in each group are pairwise disjoint. Then, for each group, we add a new GenSpec with the mutual-exclusion constraint with *genObjSet* as a generalization and the members of the group as specializations. Next, we treat the rest of the ungrouped specializations (those not disjoint with any other specializations) as another group and add another GenSpec as before. Finally, if there are disjoint object sets without a common generalization, we add general constraints indicating that the object sets are disjoint.

2.3.4 Limitations

Translating DAML to OSM is generally an intuitive process. Delving into the details, however, the complexity of the algorithm can grow out of control. Further, usually the complex details do not yield any information useful for the extraction process. We therefore make a few limitations to keep the algorithm manageable.

1. The DAML specification says, "arbitrarily complex combinations of these [class]

expressions can be formed." Our implementation does not deal with these arbitrary combinations. In particular, it does not deal with restrictions specified by *daml:(disjoint)unionOf*, *daml:complementOf*, or *daml:sameClassAs*. Restrictions specified by *daml:unionOf* could be applied as a disjunction (whereas *daml:subClassOf* and *daml:intersectionOf* specify conjunctive restrictions), for *daml:complementOf*, we could apply the negation of the restriction, and for *daml:sameClassAs* we could apply the inverse of the restriction that says anything that satisfies the restriction belongs to the class. However, applying these restrictions will usually result in general constraints, which do not help in the extraction process. Therefore we leave these restrictions as possible future work.

2. Our implementation does not deal with *daml:hasValue* in a restriction. The value of the *daml:hasValue* property is a resource (specified by a URI). Since URIs do not help in data extraction, we ignore *daml:hasValue*.

3. A relationship set is created only for a property where a domain *and* range can be identified either directly, through a superproperty, or through a restriction on the property for some class.

4. The DAML specification allows for a property to have multiple domains and ranges. DAML interprets multiple domains or ranges as the intersection of the domains or ranges. The RDF specification, on the other hand, says that multiple domains should be interpreted as a union and that multiple ranges are not allowed. Since the two specifications are in direct conflict and for simplicity, our implementation considers only one domain and one range for a property.

2.3.5 Implementation Notes

This section discusses some issues we encountered in the implementation of the algorithm.

1. When DAML refers to a property or class that is not present in the current model (including any imported ontologies), we create a class or property to use in the translation as a preprocessing step. This has the possibility of leading to erroneous assumptions about these constructs, such as domains and ranges. We can, however, infer domains and ranges that are subsets of the actual domains

and ranges, which allows us to create object and relationship sets, so the benefits to the data-extraction process outweigh this concern.

2. DAML provides a way to specify two classes or two properties as being equivalent using *daml:sameClassAs*, *daml:samePropertyAs*, and *daml:equivalentTo*. As another preprocessing step, our implementation collapses all references to the two classes or properties into a single class or property so each equivalent set is translated into only one corresponding OSM construct.

2.4 Automatic Addition of Data Frames

Once the structure has been translated, the next step is to turn the OSM model instance into an extraction ontology by adding a data frame for each lexical object set [Emb80]. A data frame contains a recognizer with a detailed description of how a piece of information generally appears in an unstructured or semistructured document. It uses regular expressions to specify how the data itself appears as well as regular expressions for the immediate context of the data. Data frames also include keywords that should appear in the document close to the data to be extracted. For example, a *ReleaseDate* data frame would contain regular expressions to recognize a date and the keyword *released*.

To associate a data frame with each lexical object set, we draw from a library of common prebuilt data frames. Appendix A shows a graphical representation of the data-frame names and generalization/specialization hierarchy of the data-frame library we use for this project. Appendix A also shows an example data frame for the *Date* object set from the data-frame library. Mapping object sets to data frames becomes essentially a schema matching problem. Researchers have used several different techniques for schema matching, including instance- or schema-based, element- or structure-level, and constraint- or linguistic-based matching [RB01] [EJX01]. Because one of our schemas, the data frame library, has neither instances nor structure, we are limited to element-level linguistic-based matching.

To generate a mapping from object sets to data frames, we compare each lexical-object-set name with each data-frame name and generate a similarity measure from zero to one. Then for each object set, we simply choose the data frame with the

highest similarity as long as the similarity is above a given match threshold. Note that a data frame may be the best match for more than one object set; for example, the *Date* data frame would match both *ReleaseDate* and *PatentDate*.

To calculate the similarity between an object set and a data frame, we use a combination of string matching techniques on their respective names. To take advantage of the semantics of names as well as their string values in calculating similarity, we introduce an alias field in the data frame which will list possible alternate names. We can then compare an object-set name with each alias of the data frame as well as its name and choose the highest similarity as the similarity between the object set and the data frame. Some name matching techniques have also used synonyms from a thesaurus. This method, however, introduces the possibility of false positive matches between words with multiple senses [EJX01]. For example, *address* might match with *speech* even though the *address* referred to is a *mailing address*. Therefore, we use only manually specified aliases rather than synonyms from a general thesaurus. We use four character-based string matching techniques to generate similarities. First, we apply some standard information-retrieval-style stemming to get a root for each name [Por80]. Next, we combine variations of the Levenshtein edit distance [Lev65], soundex [HD80], and longest common subsequence algorithms to generate a similarity value. Finally, we apply a heuristic to detect if the object set is a specialization of the data frame.

The Levenshtein edit-distance algorithm calculates the number of characters that need to be added, deleted or changed to transform one string into another. To convert edit distance to a similarity measure, we first normalize the edit distance by dividing it by the length of the object set name and cap the result at one. Next, we subtract from one since a smaller edit distance denotes more similar strings:

$$similarity_{Lev} = 1 - min\left(1, \frac{edit\_dist(name_{os}, name_{df})}{length(name_{os})}\right) \tag{2.1}$$

The soundex algorithm was developed for automatically recognizing alternate spellings of the same surname in genealogy applications. The algorithm generates a four-character code for a string according to the following rules: 1) The first character in the code is the first letter of the string, and 2) the remaining characters in the code correspond to the next three letters of the string, excluding *A, E, I, O, U, H, W,* and *Y*. The letters are divided into six groups of letters that are considered similar, and

all the letters in a group generate the same code (i.e. *M* and *N* both generate code 5). Soundex codes are generally compared with an all-or-nothing matching approach, but we find this to be too restrictive. For example *Phone* and *PhoneNumber* have codes *P500* and *P555* respectively, so the typical matching approach yields zero percent similarity. Instead of all-or-nothing matching, we base our similarity measure on the length of the common prefix for the two four-character codes, so *Phone* and *PhoneNumber* have fifty percent similarity:

$$similarity_{Soundex} = common\_prefix\_length(soundex(name_{os}), soundex(name_{df})) * 0.25 \quad (2.2)$$

Since any two soundex codes have a common prefix length from zero to four, multiplying by 0.25 yields a similarity between zero and one.

The longest common subsequence (LCS) algorithm finds the length of the longest (not necessarily contiguous) sequence of characters that appears in both strings. As with edit distance, we normalize the LCS length by dividing by the length of the object-set name to get a similarity measure between zero and one. We find LCS to be more useful than longest common substring, which does not allow for non-contiguous sequences; however, for strings containing common letters, it is possible to recognize completely unrelated sequences. For example, *BusinessEmail* has an LCS length of five with both *Email* and *SizeArea*. Our solution is to penalize characters skipped by the LCS in one string or the other by subtracting from the LCS length, so when compared with *BusinessEmail*, *Email* still has an LCS length of five, while *SizeArea* has an LCS length of zero:

$$similarity_{LCS} = \frac{LCS\_len(name_{os},name_{df}) - min\big(skipped\_chars(LCS(name_{os},name_{df})), LCS\_len(name_{os},name_{df})\big)}{length(name_{os})} \quad (2.3)$$

To calculate the combined similarity between a an object-set name and a data-frame name or alias, we combine the similarities from the edit distance, soundex and LCS calculations. We use a weighted average, giving each of the three components a weight or importance in the combined similarity:

$$\begin{aligned} similarity_{Comb} = &(similarity_{Lev} * weight_{Lev}) + \\ &(similarity_{Soundex} * weight_{Soundex}) + \\ &(similarity_{LCS} * weight_{LCS}) \end{aligned} \quad (2.4)$$

The empirically determined component weights are given in Section 4.1.

To get the final similarity, we apply one additional heuristic. We observe that when one object set is a specialization of another, the name of the specialization often includes the name of the generalization. Since our goal with the data-frame library is to provide common, general data frames, we would like to match general data frames with specialized object sets. For example, we would like to match *OriginalReleaseDate* with *Date*. To apply the heuristic, we check to see if the name of the object set either starts with or ends with the name of the data frame; if it does, we set the similarity to the maximum of the previously computed similarity and the match threshold. Setting the similarity to the match threshold ensures that we consider the object set as a match for the data frame, but leaves room for other more specialized data frames to have a higher similarity. Therefore, *OriginalReleaseDate* matches with *Date*, but matches better with *ReleaseDate*.

$$similarity_{Final} = \begin{cases} OS \ \ specializes \ \ DF & : \ max(match\_threshold, similarity_{Comb}) \\ otherwise & : \ similarity_{Comb} \end{cases} \tag{2.5}$$

The output of this process is a suggested mapping from lexical object sets to data frames. This mapping may not be complete. If no data frame has a similarity value above the match threshold, we assume that no data frame exists for the object set, and a human must create one or provide a mapping missed by the matching process.

2.5 User Ontology Editing

DAML ontologies may not contain cardinality constraints, and the data-frame matching process may produce only a partial mapping. Therefore, the generated ontology may not be suitable for data extraction without some user intervention.

To allow a user to edit constraints and data frames, we provide an extended version of the Ontology Editor [Hew00]. When a user opens a DAML ontology, the Ontology Editor translates the ontology, matches object sets to data frames, and shows the generated extraction ontology graphically. Since a DAML ontology has

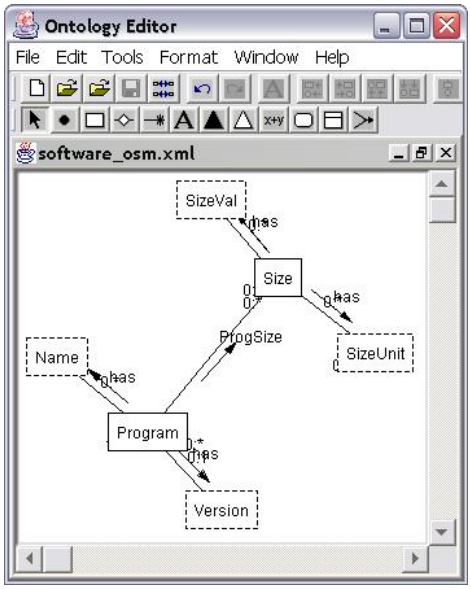

Figure 2.4: Example Automatic Layout

no display information for its classes and the Ontology Editor displays the ontology graphically, we use a modified version of the AGLO graph layout algorithm [Col93] to find a suitable screen layout. Figure 2.4 shows the screen layout produced by AGLO for an example DAML ontology opened in the Ontology Editor.

After the name matching process completes, the system presents to the user a list of suggested mappings from lexical object sets to data frames. In the simplest case, where the system finds a match for each object set and the match is correct, the user can simply accept the suggested matches. If the system does not find a data frame or finds an incorrect data frame for an object set, the user can choose from a list of existing data frames or launch the data-frame editor to create one. Figure 2.5 shows the data-frame matcher user interface after the match algorithm has been run. For each object set, the user interface shows the data frame chosen along with its similarity measure, given as a percentage. For object sets matched incorrectly or not matched, the user can choose a data frame from the data-frame list on the right-hand side of the user interface. In many cases, a generic data frame is suitable for an object set once keywords are added. In such a case, a user can choose the generic data frame from the data-frame list and edit the data frame in the data-frame editor. For example, we could use the *Date* data frame for the *MarriageDate* object set in Figure 2.5 by adding the keyword *married*.

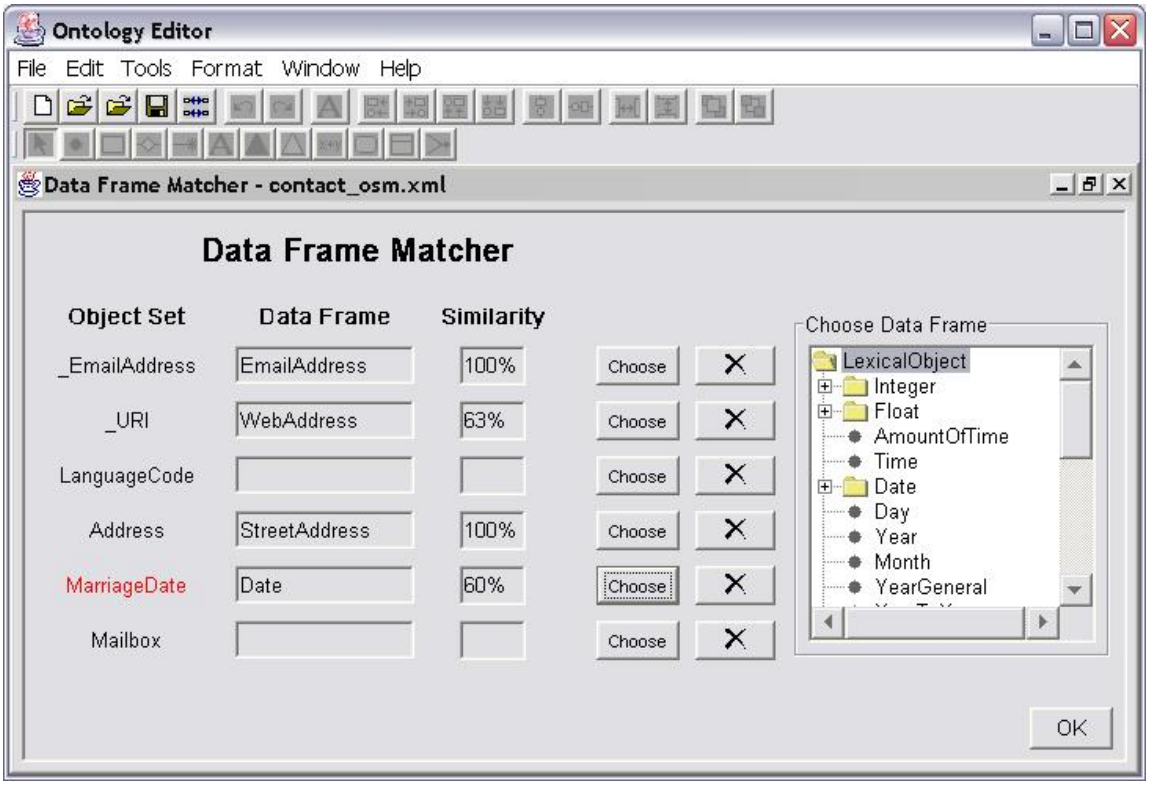

Figure 2.5: Data-Frame Matcher User Interface

After the translation process, the user can edit the ontology like any other ontology but may not change the structure. Therefore, when a user opens a DAML ontology, the Ontology Editor warns the user not to make any structural changes to the ontology[2]. The user can modify participation constraints directly and further edit and test data frames using the provided data-frame editor.

The Ontology Editor uses an XML format for its ontology serialization. The current version of the Data Extraction Group's extraction engine, however, uses an older text-based format. Therefore, when the ontology-creation process completes, we have added an option in the `File` menu for the user to save the ontology in the old text-based format.

[2]It would be possible to disable all features of the Ontology Editor that allow a user to change the structure of an ontology, but building such a restrictive user interfaces is not the goal of this project.

Chapter 3

Extracting and Viewing RDF Data

Once we have obtained a suitable data-extraction ontology from the translation and editing process, the next step is to extract RDF data from Web pages. We can then view the extracted RDF data together with the Web pages from which it was extracted.

Section 3.1 of this chapter describes how we use the BYU Ontos data-extraction tool to extract RDF data from Web pages. Section 3.2 explains how we associate extracted RDF data with the document from which it is extracted, and Section 3.3 describes the tool we have created for viewing the extracted RDF data as a superimposed layer over source Web pages.

3.1 Extracting RDF Data

Our first goal in this section is to extract data from data-rich multiple-record Web documents where each record in a page represents one instance of the primary object set in our extraction ontology. Figure 3.1 shows a sample Web page with 7 records. Each record in the page describes a *Program*, which is the primary or central object set in our software ontology from Figure 2.3. Section 3.1.1 gives the details of how Ontos can extract data from Web pages like the one in Figure 3.1. It also explains how we have extended Ontos for extracting RDF data and superimposing information.

Our second goal in this section is to make the extracted data accessible to Semantic Web agents. Section 3.1.2 presents our algorithm for generating RDF data

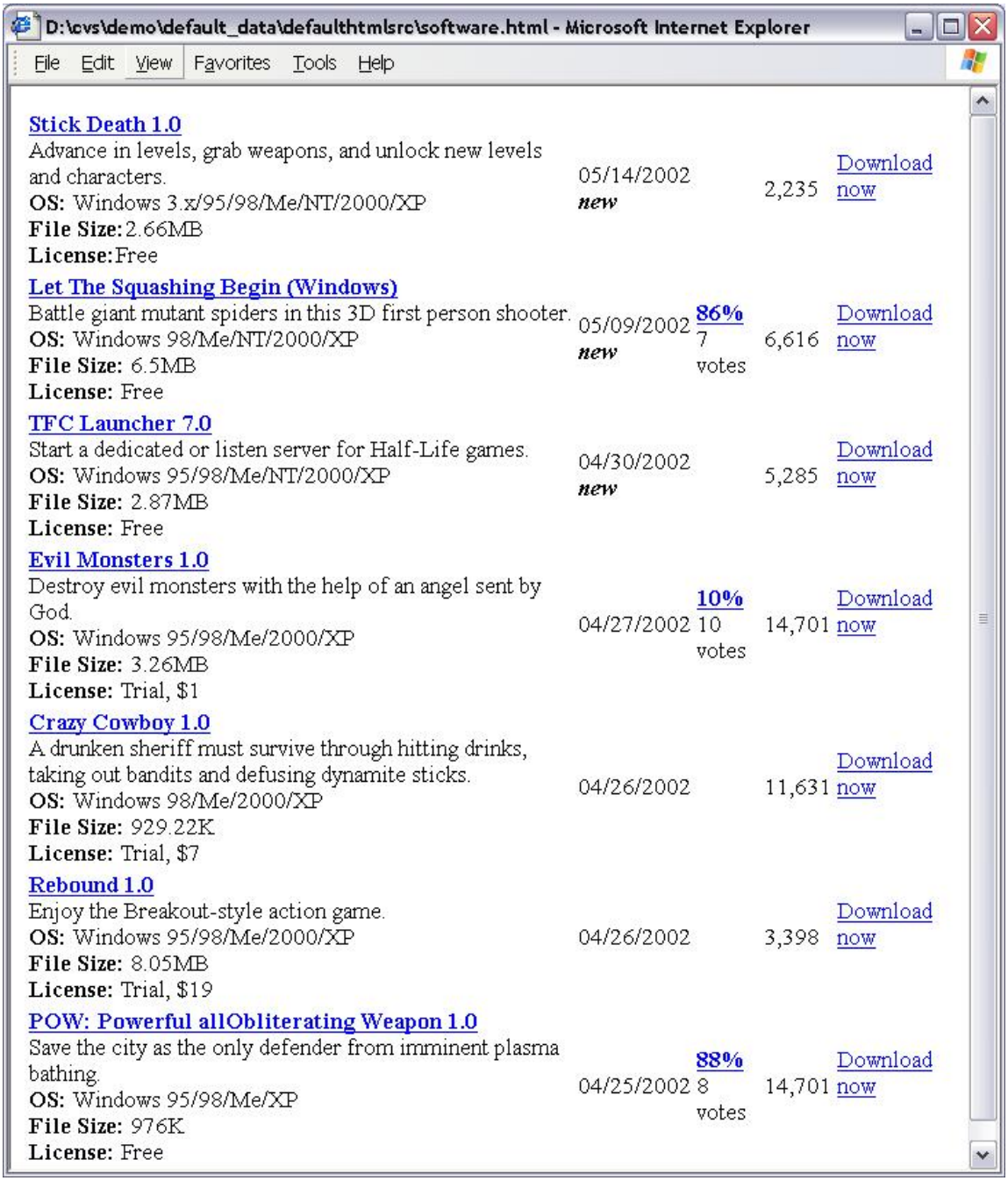

Figure 3.1: Example Multiple-Record Web Page

from the data extracted by Ontos.

3.1.1 Data-Extraction Process

The first step in the process of extracting data from an HTML page is to separate it into distinct records. The record separator attempts to automatically find first, an HTML element (the container element) containing all the records in the page, and second, an HTML tag (the boundary tag) that separates the records such that there is one data record between each occurrence of the tag within the container element. It then separates the page into records based on that tag and strips out all markup in the record, leaving only plain text. [EJN99] explains the details of this process.

Figure 3.2 shows part of the HTML source tree for the Web page in Figure 3.1. In this case, the record separator finds that all of the records in the page are contained as the content of a $< table >$ element (Line 5), so that the $< table >$ element becomes the container element. The separator also finds that each $< tr >$ element that is a direct child of the container $< table >$ element contains one record, so $< tr >$ is the boundary tag. The separator then removes the HTML markup, leaving only the plain-text content. Figure 3.3 shows part of the output from the record separator for the page in Figure 3.1. Lines 4-10 show the plain-text content of the first record.

Our RDF-extraction system needs three additional pieces of information about each record. The first piece of information we need for each record in order to generate RDF data from the extracted data is the URL of the input document for use in creating URIs for extracted data instances. We obtain this by simply passing the input document URL into the record separator which includes the URL in its output. Line 1 of Figure 3.3 shows the input document URL for the Web page in Figure 3.1.

The second piece of information we need for each record is an address to the HTML element containing the record. We use this address to create a pointer to the extracted data to preserve the link between the extracted data and the source document. We use an XPath [CD99] as the address of an element. For example, for the HTML tree in Figure 3.2, the XPath `/html[1]/body[1]/table[1]/tr[2]` addresses the second $< tr >$ (Line 22) contained in the first $< table >$ (Line 5) in the $< body >$ (Line 4) of the HTML document.

```
1)   <html>
2)       <head>
             .  .  .
3)       </head>
4)       <body>
5)           <table>
6)               <tr>
7)                   <td>
8)                       <a href="..."><b>Stick Death 1.0</b></a><br />
9)                       Advance in levels, grab weapons, and unlock new levels
10)                      and characters.<br />
11)                      <b>OS:</b> Windows 3.x/95/98/Me/NT/2000/XP<br />
12)                      <b>File Size:</b>2.66MB<br />
13)                      <b>License:</b>Free<br />
14)                  </td>
15)                  <td>05/14/2002<br />
16)                      <i><b>new</b></i>
17)                  </td>
18)                  <td></td>
19)                  <td>2,235</td>
20)                  <td><a href="...">Download now</a><br /><br /></td>
21)              </tr>
22)              <tr>
                     .  .  .
23)              </tr>
                 .  .  .
24)          </table>
25)      </body>
26)  </html>
```

Figure 3.2: Partial HTML Source Tree for Figure 3.1

```
1)   <records IputDocumentURL="http://www.deg.byu.edu/software.html">
2)       <record XPath="/html[1]/body[1]/table[1]/tr[1]"
3)           charOffsetInParent="0">
4)           <![CDATA[Stick Death 1.0
5)     Advance in levels, grab weapons, and unlock new levels and characters.
6)     OS: Windows 3.x/95/98/Me/NT/2000/XP
7)     File Size:2.66MB
8)     License:Free
9)      05/14/2002
10)    new   2,235 Download now]]>
11)      </record>
12)      <record ...>
13)          .  .  .
14)      </record>
15)      .  .  .
16)  </records>
```

Figure 3.3: Example Record-Separator Output

The type of boundary tag that the record separator finds dictates how we determine the parent element for each record. There are two cases. In each case, we consider only instances of the boundary tag that are direct children in the HTML tree of the container element (the element containing all the records). The first case is a tag that allows element content such as $< tr >$. For this case, we consider each instance of the element to be a record. We use the content of the element as the text of the record, and we generate an XPath for the element as the address of the parent of the record. The second case is a tag that does not allow element content, such as $< hr >$ (**h**orizontal **r**ule). For this case, we use the text between each instance of the tag as the record text, and we save an XPath to the container element (the element containing all the records) as the parent element of each record.

For the HTML in Figure 3.2, the first case applies. Since the record separator finds $< tr >$ as the boundary tag, and the $< tr >$ element allows content, each record's parent element is the $< tr >$ element in which it is contained. The first record in the page has an XPath of `/html[1]/body[1]/table[1]/tr[1]`, which we save along with the text of the record (Line 2 in Figure 3.3).

The third piece of additional information we need from the record separator is the character offset of each record within its parent element (the element to which

we saved an XPath). In the first case above, where a separator finds a boundary tag that allows content, we save boundary tag instances as the parent elements for their records. When the entire content of a boundary tag instance is the content for a record, the character offset of the record is always zero. Line 3 in Figure 3.3 shows that the character offset for the first record in Figure 3.1 is zero. In the second case, for boundary tags that do not allow content, all the records in the page have a common parent element (the container element). Hence, for each record we save the character offset from the beginning of the parent element's content to the beginning of the record text[1].

The second step of the extraction process, the matcher, takes as input the output of the record separator along with matching rules from the data frames in an extraction ontology. The matcher produces several outputs. Figure 3.4 shows sample output from the matcher, based on the input in Figure 3.3. First, the matcher generates an object identifier (OID) for each record. The record in Figure 3.4 has an OID of `1001` (Line 1). We note also that, for each record, the matcher passes the input document URL (Line 2) and the XPath of the parent element (Line 3) to its output. Finally, the matcher applies the matching rules from the extraction ontology to each record found by the record separator. Based on these rules, the matcher produces a set of matches for each record, where each match includes the matched data and the character offsets within the record text to the beginning and end of the match [2] . To get character offsets within the record's parent element, we add the character offset *of* the record to the start and end character offsets *within* the record. For example, `Stick Death` is a possible instance for the *Name* object set (Lines 4-6). Adding the character offset zero of the record from Line 3 of Figure 3.3 to the start and end character offsets of the data (zero and nine respectively), we get zero and nine for the start and end offsets of the data within the parent element of the record (Line 4).

[1]All character offsets discussed are zero-based. They also ignore whitespace to facilitate the correct interpretation of offsets within a Web browser, because a browser may insert whitespace for some HTML tags, which we have already stripped out at this point.

[2]Zero-based, non-whitespace counting makes the 'S' in `Stick Death` character 0 and the 'h' character 9.

```
1)   <record_data object_id="1001"
2)       URL="http://www.deg.byu.edu/software.html"
3)       XPath="/html[1]/body[1]/table[1]/tr[1]">
4)       <constant object_set="Name" start_pos="0" end_pos="9">
5)           <![CDATA[Stick Death]]>
6)       </constant>
7)       <constant object_set="Version" start_pos="10" end_pos="12">
8)           <![CDATA[1.0]]>
9)       </constant>
10)      <constant object_set="Name" start_pos="13" end_pos="72">
11)          <![CDATA[Advance in levels, grab weapons, and unlock new levels and characters.]]>
12)      </constant>
13)      <constant object_set="Name" start_pos="73" end_pos="82">
14)          <![CDATA[OS: Windows]]>
15)      </constant>
16)      <constant object_set="OSName" start_pos="76" end_pos="82">
17)          <![CDATA[Windows]]>
18)      </constant>
19)      <constant object_set="OSVersion" start_pos="83" end_pos="105">
20)          <![CDATA[3.x/95/98/Me/NT/2000/XP]]>
21)      </constant>
         .  .  .
22)  </record_data>
```

Figure 3.4: Partial set of Matches for the First Record in Figure 3.1

Program	Name	Size	SizeVal	SizeUnit	Version
1001	Stick Death	5001	2.66	MB	1.0
1002	Let The Squashing Begin	5002	6.5	MB	3D

▪ ▪ ▪

OperatingSystem	Program	OSVersion	OSName
7001	1001	3.x/95/98/Me/NT/2000/XP	Windows
7002	1002		Windows
9002	1002	98/Me/NT/2000/XP	Windows

▪ ▪ ▪

Figure 3.5: Extracted Relational Data

The final step of the extraction process takes the matches generated in the second step and applies a set of heuristics to determine which matches correspond to actual object instances. [ECJ+99] explains the details of this process. Ontos then creates

SQL insert statements to populate a relational database with the object instances. Figure 3.5 shows the relational data output by Ontos for the first two records in Figure 3.1. For example, Figure 3.4 shows that `1.0` (Lines 7-9) is a possible instance for the *Version* object set. Ontos, therefore, inserts `1.0` in the database as the value for *Version* field (corresponding to the *Version* lexical object set) in the record (row) with the OID `1001` as the value for the *Program* field (corresponding to the *Program* non-lexical object set). Similarly, Ontos inserts `Windows` (Lines 16-18) as the value for the *OSName* field and `3.x/95/98/Me/NT/2000/XP` (Lines 19-21) as the value for the *OSVersion* field. Figure 3.4 shows that `Stick Death` (Lines 4-6) and `OS: Windows` (Lines 10-12) are both possible instances for the *Name* object set. The heuristics explained in [ECJ+99] select `Stick Death` as the actual name of the program in the record with OID `1001`, so Ontos inserts `Stick Death` as the value for the *Name* field.

3.1.2 Generating RDF Data

Once the system finishes extracting the data, converting the data to RDF is straightforward. Figure 3.6 shows the RDF data generated from the relational data in Figure 3.5. Appendix B contains pseudocode for our algorithm for generating RDF data from a populated relational database corresponding to a schema generated by Ontos according to the given original DAML ontology.

The database generated by Ontos has a main table, where each row represents an object in the primary object set. Each row in all other tables in the database is related to a row in the main table by object identifiers. Thus, each row in the main table along with rows in related tables contain the extracted values for an object. For example, the first table in Figure 3.5 is the main table; its first row, along with the first row of the second table, related by OID `1001`, contain the data for the *Program* with OID `1001`.

Each row in the main table becomes an instance of the DAML class corresponding to the primary object set, and the values in that row and its associated rows become the property values for the generated instance. We generate URIs for class instances by concatenating the URL of the input HTML page, the name of the DAML class, and the OID from the database. Line 1 of Figure 3.6 shows the instance of *software:Program* generated for the first row of the main table with URI

```
1)   <software:Program
         rdf:about='http://www.deg.byu.edu/software.html#Program1001'
2)       software:Version='1.0'
3)       software:Name='Stick Death'>
4)       <software:hasSize>
5)           <software:hasSize
                 rdf:about='http://www.deg.byu.edu/software.html#Size5001'/>
6)       </software:hasSize>
7)       <software:supportsOperatingSystem>
             <software:OperatingSystem rdf:about=
                 'http://www.deg.byu.edu/software.html#OperatingSystem7001'/>
8)       </software:supportsOperatingSystem>
9)   </software:Program>
10)  <software:Size rdf:about='http://www.deg.byu.edu/software.html#Size5001'
11)      software:SizeUnit='MB'
12)      software:SizeVal='2.66'>
13)  </software:Size>
14)  <software:OperatingSystem
         rdf:about='http://www.deg.byu.edu/software.html#OperatingSystem7001'
15)      software:OSVersion='3.x/95/98/Me/NT/2000/XP'
16)      software:OSName='Windows'>
17)  </software:OperatingSystem>
```

Figure 3.6: Extracted RDF Data

http://www.deg.byu.edu/software.html#Program1001.

The method we use to generate property values for generated instances depends on the type of the property. In the case of a data-type property, where the range of the property is lexical, we simply add the value from the corresponding field in the database as the value of the property. For example, the value `1.0` in the *Version* field in Figure 3.5 becomes the value of the *software:Version* data-type property in Line 2 of Figure 3.6.

In the case of an object property, where the range is a class, we generate an instance of the range class as the value of the property. We generate the URI for the instance of the range class as well as its property values as described above. For example, we use the value `5001` of the *Size* field in Figure 3.5 to generate an instance of the *software:Size* class as the value of the *software:hasSize* object property with URI *http://www.deg.byu.edu/software.html#Size5001* in Lines 4-6 of Figure 3.6. The values of the *SizeUnit* and *SizeVal* fields in Figure 3.5 (`MB` and `2.66`) become property

values for the generated instance of *software:Size* in Lines 11-14 of Figure 3.6.

Once we have the extracted data formatted as RDF, it is accessible to various kinds of Semantic Web agents. An agent that understands the ontology referred to by the RDF data could, for example, search for instances of a class with a specific property value. An agent that understands our sample software ontology could find the names of all programs that support a Windows operating system. Even an agent that does not understand the ontology can make use of the extracted RDF data without understanding the semantics of the data. We provide such an application, which we describe in Section 3.3.

3.2 Associating RDF Data with HTML

In order to superimpose our extracted information, we need to associate the extracted RDF data with the HTML. This section describes two different levels of association that we use. First, we associate each extracted property value to the place in the original document where it was extracted. Second, we associate our generated RDF data with the document from which it was generated.

For each extracted lexical property value, we keep a link to the place in the source document from which it was extracted. For each value, we create an XPointer [DDG+02] based on the XPath to the parent element of the current record saved by the record separator and the character offsets saved by the matcher. For example, for the extracted value `1.0` of the *software:Version* in Line 2 of Figure 3.6, we combine the XPath of its record, from Line 3 of Figure 3.4, and its start character position and the number of non-whitespace characters computed from its start and end character positions, from Line 7 of Figure 3.4, to form the XPointer `xpointer(string-range( /html[1]/body[1]/table[1]/tr[1], '', 10, 3))`, which yields the tenth, eleventh and twelfth non-blank characters in the first row of the table in the HTML document in Figures 3.1 and 3.2.

To associate XPointers with property values, we have created a DAML class called *deg:XPointer*[3] and two properties: *deg:XPTValue* and *deg:XPTPointer* (whose domain is *deg:XPointer*). For each data-type property value in the extracted RDF

[3]The *deg* prefix used here refers to http://www.deg.byu.edu/ontologies/deg.daml, the default namespace for the BYU Data Extraction Group.

```
1)   <software:Program
         rdf:about='http://www.deg.byu.edu/software.html#Program1001'
2)       software:Version='1.0' ...>
3)       <software:VersionXPT rdf:resource='#XPT1'/>
         .  .  .
4)   </software:Program>
5)   <deg:XPointer rdf:about='#XPT1'
6)       deg:XPTValue='1.0'
7)       deg:XPTPointer=
             'xpointer(string-range(/html[1]/body[1]/table[1]/tr[1],'',10,3))'>
8)   <deg:XPointer>
```

Figure 3.7: Extracted RDF Data with an Associated XPointer

data, we append *XPT* to the property name to create a new property. As the value of the new property, we create an instance of *deg:XPointer*[4] and assign the extracted data value as the value of the *deg:XPTValue* property for the *deg:XPointer* instance and the XPointer to the extracted value as the value of the *deg:XPTPointer* property. Figure 3.7 shows part of the RDF data in Figure 3.6 with an associated XPointer. Along with the value `1.0` of the *software:Version* property (Line 2), the *software:Program* instance shown has a value of `#XPT1` for the *software:VersionXPT* property (Line 3). `#XPT1` is an instance of the *deg:XPointer* class (Line 5) with `1.0` as the value for *deg:XPTValue* (Line 6) and `xpointer(string-range(/html[1]/body[1]/table[1]/tr[1],'',10,3))` as the value for *deg:XPTPointer* (Line 7).

Once we have extracted RDF data and generated the RDF data containing XPointers for the extracted data, we need to associate all the RDF data with the source document. [Pal02] presents several methods for associating RDF data and HTML documents; we choose to store the RDF data inside the HTML document. Figure 3.8 shows part of the HTML source tree from Figure 3.2 augmented with the RDF data inserted as the content of an HTML script element (Line 3) at the end of the head section of the HTML page. Inserting the RDF data into a script element keeps the RDF data from being rendered by a Web browser as suggested by [Pal02].

[4]The obvious way to associate a property value and an XPointer is to make the XPointer the value of the newly created property. The problem with this approach is that some properties may have more than one value, so there would be no way to know which XPointer associates with which value. We thus introduce a *deg:XPointer* for every property value.

```
1) <html>
2)    <head>
         .  .  .
3)       <script type="application/rdf+xml">
4)          <rdfRDF ...>
5)          <software:Program
                 rdf:about='http://www.deg.byu.edu/software.html#Program1001'
6)               software:Version='1.0'
7)               software:Name='Stick Death'>
8)               <software:VersionXPT rdf:resource='#XPT1'/>
9)            </software:Program>
              .  .  .
10)           <deg:XPointer rdf:about='#XPT1'
11)              deg:XPTValue='1.0'
12)              deg:XPTPointer=
                    'xpointer(string-range(/html[1]/body[1]/table[1]/tr[1],'',10,3))'>
13)           <deg:XPointer>
              .  .  .
14)         </rdfRDF>
15)      <script type="application/rdf+xml>
16)   </head>
17)   <body>
18)      <table>
19)         <tr>
20)            <td>
21)               <a href="..."><b>Stick Death 1.0</b></a><br />
                  .  .  .
22)            </td>
               .  .  .
23)         </tr>
            .  .  .
24)      </table>
25)   </body>
26)</html>
```

Figure 3.8: Partial HTML Source Tree for the Web Page in Figure 3.1 with Extracted RDF Data and XPointers

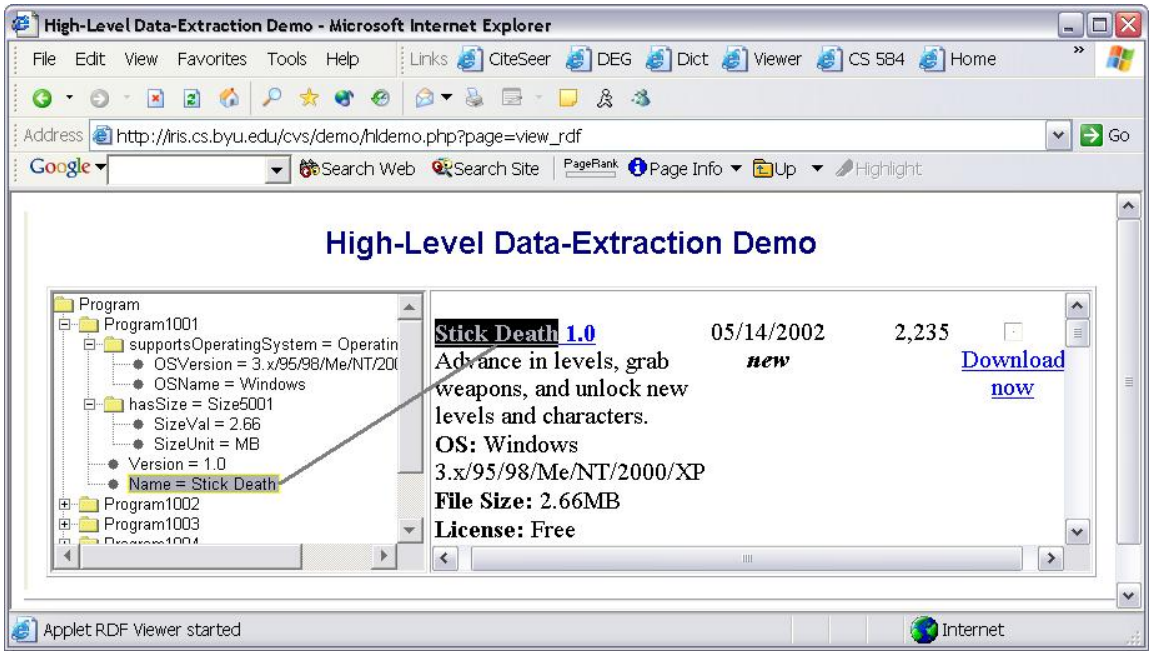

Figure 3.9: Example of RDF Data used as Superimposed Information with our RDF Browser (connecting line added for emphasis)

3.3 Viewing RDF Data

To verify that a Semantic Web agent can use our extracted RDF data and to illustrate the principle of superimposed information, we provide a sample application — a simple Semantic Web agent — that allows the user to browse the RDF data by DAML class. Expanding a class in the browse tree shows its subclasses and instances, and expanding an instance shows its property values. When the user selects a property value for an instance (i.e. an extracted data item), the application uses the XPointer associated with that value to highlight the data item in the original document.

Figure 3.9 shows our RDF browser with the RDF data from Figure 3.6 in the left-hand frame and part of the HTML page from Figure 3.1 (from which the data was extracted) in the right-hand pane. The record shown in the HTML page corresponds to the instance *Program1001* of class *software:Program*. Expanding the tree node for *Program1001* shows the property values for that instance. Selecting a property value highlights the corresponding value in the HTML page on the right. For example, selecting the value of the *Name* property for *Program1001* highlights `Stick Death` in

the Web page. Therefore, a user can browse both layers of information, the structured (RDF) layer and the unstructured (HTML) layer, together in a superimposed fashion.

Chapter 4

Analysis and Results

This chapter describes the experiments we performed to test our system and their results. Section 4.1 gives the precision and recall for our data-frame matcher. Section 4.2 discusses the challenges and successes we had in creating data-extraction ontologies from DAML ontologies. Section 4.3 describes an experiment we used to test the tools we provide to build data-extraction ontologies, and Section 4.4 discusses our results for RDF extraction using the extraction ontologies that we created.

4.1 Data-Frame Matching

To evaluate the accuracy of the data-frame matcher, we used 24 DAML ontologies from the DAML repository [dam02]. We arbitrarily chose eight ontologies as "training" documents and used the rest as test documents. We used the training documents to tune the relative weights of the three string comparison algorithms and to determine the best match threshold value. Table 4.1 summarizes the results of tuning the matching process. We also used the training documents to adjust the data-frame library to make sure the names and aliases of the data frames were appropriate, removing, for example, very general aliases like *'x'* for *PhoneExtension* that produced too many false-positive matches.

After we tuned the matcher and the data-frame library, we evaluated the matching process on the 16 test documents, using precision and recall [BYRN99] as performance metrics. For precision we use the number of correct matches divided by

Levenshtein Weight	40%
Soundex Weight	20%
LCS Weight	40%

Match Threshold	60%

Table 4.1: Matcher Tuning Parameters

the total number of matches generated, and for recall, we use the number of correct matches divided by the number of object sets that should have been matched. The test ontologies contained a total of 128 lexical object sets, 45 of which should match to data frames in our library. Our matcher generated a total of 52 matches, 40 correct and 12 incorrect, yielding a precision of 77%. Dividing the number of correct matches, 40, by the total number of object sets that should have been matched, 45, yields a recall of 89%.

Of the possible matches the matcher missed, it missed most of them not because the correct data frame did not yield a similarity above the match threshold, but because it considered another data frame more similar. For example, *ZoneNumber* matched with *PhoneNumber* rather than *Integer*. The matcher did not simply miss the actual match *Integer*, but we penalized the recall because the object-set name was more similar to *PhoneNumber*, a different data frame.

We attribute the excellent recall of the matching process mainly to two factors. First, the manually specified data-frame aliases allow us to match the data-frame *Integer* to an object set like *Number* with a completely different name, and since the data-frame library is so small (31 data frames), adding aliases did not impose a significant work load. (We added aliases for all of the data frames in less than one hour.) We note also that using manually specified aliases rather than an automated thesaurus minimizes the impact on precision, as explained in Section 2.4. Second, the specialization heuristic, described in Section 2.4, boosts recall by catching many of the matches that we miss using only string matching algorithms.

The specialization heuristic, however, does lower the precision of the matching process. For example, the *Age* data frame matches incorrectly with object-set names like *Language* and *Coverage* since both end in 'age'. Our justification for favoring recall over precision is that the user has to manually inspect the matches generated, and we provide a very simple way in the user interface to remove incorrect matches

(simply click the X button next to the data frame), so the penalty for low precision is small. We also provide a way for the user to declare missed matches by browsing the data-frame library, but it is more difficult to browse the library to find a missed match than it is to remove an incorrect match.

4.2 Data-Extraction Ontology Creation

To test our system, we built four data-extraction ontologies. We built three extraction ontologies — one each for describing software, car advertisements, and apartment rental listings — by first creating DAML ontologies specifically for the purpose of data extraction and then converting the DAML ontologies to OSM data-extraction ontologies. We built the fourth extraction ontology, which describes university courses, by converting an existing DAML ontology from the DAML repository [dam02] to an extraction ontology.

The three DAML ontologies that we created converted very easily into data-extraction ontologies, just as we expected they would. Converting from existing DAML ontologies, however, was much more difficult. Translation into OSM model instances worked fine, but we had a difficult time finding DAML ontologies suitable for data extraction. We found several reasons for our difficulty in using DAML ontologies. First, and probably most importantly, many DAML ontologies relate only abstract (non-lexical) concepts, so the conversion produces only non-lexical object sets. Non-lexical object sets are useful for data extraction only if they have related lexical object sets. Second, we found that most DAML ontologies have a scope that is either too large or too detailed; most generate forty or more object sets when we convert them to OSM. The final reason, and one that encompasses the other two, is that most DAML ontologies are simply not created with data extraction in mind. Our extraction process assumes that we have multiple-record Web documents where each record represents an object. This assumption limits us to ontologies that have one or a few non-lexical object sets, each with a few associated lexical object sets that correspond to the data fields commonly found in the records of Web pages. The DAML ontology that we did use is the one DAML ontology we could find that exhibits the properties of a good data-extraction ontology. In fact, it was written to model records in Web pages from a particular site.

Object Set	Precision	Recall	Time (min.)
PhoneNumber	100%	100%	6
	100%	100%	1
	100%	100%	7
	100%	88%	1
	100%	85%	1
	100%	100%	1
Average	**100%**	**95%**	**2.8**
Bedrooms	95%	85%	8
	98%	96%	10
	98%	98%	7
	100%	92%	15
	100%	85%	50
	100%	86%	15
Average	**99%**	**90%**	**17.5**
MonthlyRate	94%	88%	20
	96%	91%	20
	92%	83%	12
	98%	93%	5
	98%	91%	2
	84%	86%	40
Average	**94%**	**88%**	**16.5**

Table 4.2: Ontology Creation Experiment Results

4.3 Data-Extraction Ontology Creation Tools

In order to evaluate the effectiveness of our tools for creating data-extraction ontologies from DAML ontologies, we asked six students from the BYU data extraction group (who have experience creating data-extraction ontologies) to convert a DAML ontology to an OSM ontology and add extraction rules. We supplied the students with a small DAML ontology describing apartment rental listings and Web pages from three different sites, each listing apartments for rent. Our ontology had one class, *Apartment*, and three data-type properties, *PhoneNumber*, *Bedrooms*, and *MonthlyRate*, which converted to non-lexical object sets. We gave a 45 minute tutorial to the students on how to use our conversion tool, data-frame matcher, and ontology editor and asked them to report the time it took them to create each data frame (a data frame for each lexical object set) along with the precision and recall each data frame achieved over the three Web pages.

Table 4.2 shows the results reported by the students. The results correspond with what we expected. First, the data frame for *PhoneNumber* took an average of less than three minutes to build and achieved very high precision and recall. This is because a phone number is a common concept, so our data-frame library contains a data frame for phone numbers. The students simply had to accept the match suggested by the matcher and test the provided data frame. In most cases, though, the data frame library does not contain a data frame that matches so closely with an object set. The *MonthlyRate* object set is an example of this. The data-frame library contains a data frame for prices, which most of the students used (even though it was not suggested by the matcher). The data frame for prices gave the students a good starting point, but it was much more general than prices for apartments, so they had modify and remove expressions to specialize the data frame. They also had to account for ranges of prices that appeared in some of the records in the Web pages. We believe the data-frame library along with the graphical Ontology Editor contributed to the excellent time of 16.5 minutes for the somewhat complex data frame for *MonthlyRate*. The last object set, *Bedrooms* was probably the simplest of the three to recognize, since in most cases in the Web pages, the number of bedrooms was clearly labeled with something like `bd` or `bdrm`. Because there was no good data frame from which to start, though, the data frame for *Bedrooms* took the longest time on average to build. We believe that this result shows the advantage of using our data-frame library to help build extraction ontologies.

We note that our experiment has some limitations. First, the time we reported is the time to build and test each data frame, not the time to build an ontology from start to finish. We do not claim (or believe) that an ontology builder, even using our tools, can generally build an ontology in less than forty minutes. Second, in order to simplify the students' task, we provided them with Web pages for testing. Normally, finding Web pages is part of building an ontology and finding representative Web pages can be time consuming. Also, we asked the students to extract data from only three Web pages. To make a robust data-extraction ontology, a student would need to gather and test more than three Web pages. Finally, although we tried to choose data frames that are representative of most data frames in terms of complexity and time to build, we note that there may be other data frames that take more time to build. For example, some data frames use lexicons, or lists, such as a list of makes of cars. These lexicons can either take almost no time to build because an appropriate

lexicon is readily available or take a long time to build. Since building a lexicon would not show anything about our tools, we did not include such a data frame in our experiment.

4.4 RDF Extraction

We are encouraged by our extraction results. For each of the four application domains for which we created ontologies, we successfully extracted data from Web pages from two or three Web sites. We were able to use our RDF browser to verify the correctness of the extraction. Since this project does not attempt to make any enhancements to the data-extraction process, we do not report numeric results for the extraction process. We note that since we produce the same kind of extraction ontologies as are normally used by Ontos, our extraction process does not have any advantages or disadvantages in terms of precision and recall. We refer the interested reader to [ECJ+99], which gives detailed results and analysis of data extraction using Ontos. Generally, Ontos can attain recall of 90% or better and precision around 98% [ECJ+99].

Chapter 5

Related Work

This chapter discusses research areas and projects related to this project. Aside from the Semantic Web, which we discussed in Chapter 1, this project builds on research in two different areas of computer science. This chapter gives an overview of the fields of information extraction and superimposed information in Sections 5.1 and 5.2 respectively. Section 5.3 describes three research projects, RDF Web Scraper, On2Broker, and S-CREAM, whose aims are very similar to the goals of this project.

5.1 Information Extraction

Much like the Semantic Web, information extraction tries to improve usability of information on the Web by making it more structured. Unlike the Semantic Web, however, information extraction tries to take advantage of the huge amount of information already available on the Web. The idea of information extraction is to extract data from unstructured or semistructured documents and structure it for easy querying.

The most common approach to extracting information from the Web is wrapper-based extraction. A wrapper for a Web site is essentially a grammar for its pages[1]. This wrapper uses extraction rules to describe where each data field is in the page so that a program can extract data from any number of pages as long as they follow the

[1]We note that there are other more general definitions of a wrapper. Indeed, even an OSM data-extraction ontology is sometimes considered a wrapper, but for purposes of characterizing information extraction techniques, we use the definition given here.

format prescribed by the wrapper. Extraction rules in a wrapper are generally based on some kind of delimiter for a data field. Some common delimiters are characters, HTML tags, and parts of speech.

The main drawback of wrapper-based information extraction is the work required to build and maintain wrappers. Extraction wrappers work very well for sites with a large number of very similar pages, but whenever a site changes or a user needs to extract information from pages in another site, a new wrapper is necessary. [Eik99] and [LRNdST02] survey several approaches to the representation and the semiautomatic and automatic generation of wrappers for information extraction.

The BYU Data Extraction Group (DEG) has taken a different approach to Information Extraction. Instead of describing the page of interest with a wrapper, they describe the application domain of interest with an ontology. The effect of this difference is that, given an ontology for a particular domain, the BYU Ontos system can extract from Web pages containing data in that domain even if the pages are from different Web sites or if the format of a site changes [ECJ+99].

This ontology-based approach is ideal for this project because of the prevalence of ontologies in the Semantic Web. The biggest problem with the ontology-based approach is the difficulty in manually building data-extraction ontologies. This project helps alleviate the problem of building extraction ontologies by semi-automatically converting the structure from a DAML ontology and by facilitating the addition of data frames.

5.2 Superimposed Information

The idea of superimposed information is very simple: given a set of information, create another layer of information that shows a different organization or view of the original information. Indexes, commentaries, concordances, and Web search engines are examples of superimposed information. Information extraction generally considers the extracted data as the only result of the extraction process and ignores the relationship between the extracted data and the source documents. However, if the extraction system preserves the relationship between the extracted data and an original document, it achieves another important result; the structured data becomes an

index into or becomes superimposed over the source documents [MD99][BDM02].

[MD99] and [BDM02] suggest a conceptual-model-based approach for superimposing information where the superimposed layer is a set of instances of the concepts in the model instance. The conceptual-model-based (or ontology-based) information extraction technique that we use in this project is a perfect fit with this type of superimposed information. Since the data is extracted as instances of object sets in an OSM ontology, all that is needed is to keep track of the location of each extracted data item in the original document. Our RDF browser shows how this connection between the superimposed layer and the original layer can be used.

5.3 Related Projects

This section discusses three projects closely related to ours: RDF Web Scraper, On2Broker, and S-CREAM.

5.3.1 RDF Web Scraper

The RDF Web Scraper is a tool for building and using wrappers to extract data from HTML pages and converting the data to RDF [GGP+02]. To construct a wrapper for a Web site, a user manually analyzes the structure of a page and specifies the location of each field relative to an HTML tag. Next, the user maps each field to a class or property in a DAML ontology. Once the wrapper and mapping are complete, a user can use the tool to extract RDF data from any number of HTML pages from the site for which the wrapper was constructed.

The Web Scraper approach has two advantages compared with our approach. First, since the wrapper is based on HTML tags, information from inside HTML tags (i.e. attributes) as well as tag content can be extracted. This allows a user to extract things like URLs and email addresses which are usually the value of an HTML href attribute. Second, the tool allows a user to use extracted content as the URI for an object; for example, a user might use a home page URL for a person as the URI for a *Person* object.

The RDF Web scraper also has two disadvantages compared with our approach.

The first is the effort required to manually create a wrapper for each site and maintain the wrapper when the site changes. We note that there is still a significant manual component to building ontologies, but this manual effort is not repeated for every site or every version of a site. The second disadvantage is the manual mapping from fields to ontology concepts. This mapping is automatic in our approach because during the conversion from DAML ontologies to data-extraction ontologies, we save the reverse mapping from OSM ontology concepts to DAML ontology concepts.

5.3.2 OntoBroker and On2Broker

OntoBroker provides semantic-based access to heterogenous, distributed, and semi-structured information sources [FAD+00]. OntoBroker uses a Frame-Logic [KLW95] ontology to annotate Web pages and gathers annotations into a central knowledge base to provide a query and inference interface to annotations. It is important to note that while OntoBroker has goals very similar to those of the Semantic Web, it is not a Semantic Web application because it does not use Semantic Web standards (i.e. RDF, RDFS, DAML, etc.).

OntoBroker relies on a simple, proprietary extension of HTML that allows a user to annotate existing Web documents. Since an annotation attaches directly to the data it describes, the annotation does not duplicate the data elsewhere, and therefore does not introduce redundancy. The Web crawler component gathers facts from annotated Web pages into a central knowledge base, which provides a query and inference interface.

Three approaches have been used to annotate Web documents with semantic information [EMSS00]. First, a user can manually attach annotations to data elements in a document either with a text editor or with the provided graphical user interface. Second, a user can write a wrapper for a Web site in a general-purpose programming language such as Java or Perl to extract semantic information from Web documents. This wrapper approach does not actually insert annotations into the document; instead it inserts the extracted data directly into the knowledge base. Third, a user can use the SMES [NBJBB97] shallow text processor for German to automatically suggest possible annotations to the user that must still be created manually.

The OntoBroker project and our project have a very similar goal — make existing

WWW information semantically accessible. There are two main differences — the methods and the format of the results. Semantic annotation with OntoBroker is a manual process. The authors of OntoBroker have tried to address this problem by allowing a user to write a wrapper, however, writing and maintaining wrappers in a general-purpose language is even more difficult than for wrapper-based information extraction systems. Their best attempt at automation is with the text processor, but a user must still manually create the annotations suggested by the text processor. Our project on the other hand uses one of the latest information extraction systems, so the extraction is fully automatic. We do note, however, that the creation of ontologies is still mostly manual and can be time consuming.

The second difference between our approaches is that our project structures the information in a very standard way (RDF), whereas OntoBroker relies on a proprietary extension to HTML. Their data format decreases redundancy, but limits access to the semantic data, so a general Semantic Web agent cannot understand it. However, OntoBroker does collect semantic data in a central knowledge base with query and inference capabilities, so a program can access the information if it understands OntoBroker's interface.

The second version of OntoBroker, On2Broker, recognizes the need for interoperating with Semantic Web standards. The On2Broker crawler component has been updated to store facts from RDF and XML as well as annotations from Web pages, but no attempt has been made to annotate Web documents with RDF.

5.3.3 CREAM and S-CREAM

CREAM (CREAtion of Metadata) is a framework for manually creating relational metadata in RDF by annotating existing Web pages [HS02]. A user creates annotations in a drag-and-drop user interface by dragging text from a Web page onto a concept in an RDFS ontology. A meta-ontology specifies each property as either a quote or a reference. In the case of a quote, content from the Web page is copied as the value of the property, and in the case of a reference, an XPointer [DDG+02] is saved as the property value. Our approach is an extension of this idea that saves *both* the content and location of the extracted data. Annotations are stored as RDF inserted into the document. Like OntoBroker, CREAM provides a crawler component

that collects the annotations from multiple documents into a knowledge base on a server from which queries can be made.

A recent extension of CREAM, called S-CREAM (Semi-automatic CREAM), automates annotation using Amilcare [Cir01], a wrapper-induction and information-extraction system [HSC02]. Amilcare uses a set of manually annotated documents and a learning algorithm to induce a wrapper that annotates documents by inserting XML tags around items to be annotated. A user maps Amilcare XML tags to ontology concepts, and the system automatically converts the XML annotations into RDF annotations. The framework also provides a tool to manually verify and adjust the annotations produced; this tool is similar to our RDF browser.

The S-CREAM project is the most similar to ours. The main difference is in the information extraction method used — wrapper based versus ontology based. The wrapper is generated automatically, but the user must provide manually marked up examples and must manually map XML tags to ontology concepts. Our approach eliminates the manual mapping; since we obtain an extraction ontology by translating from a DAML ontology, we simply save the reverse mapping during the translation process. However, a user still has to do some work in adding and editing data frames. One advantage an extraction ontology has over a wrapper is that it can be used for multiple sites. The authors do not report results for S-CREAM, so it is difficult to compare precision, recall, or man-hours involved in extraction with our approach.

Chapter 6

Conclusions and Future Work

6.1 Summary

In this project, we designed and implemented a system for extracting data from the World Wide Web to build Semantic Web data based on DAML ontologies. We used BYU Ontos, an ontology-based data-extraction system, to extract data from Web pages. We developed an algorithm for converting DAML ontologies into OSM model instances. We obtained data-extraction ontologies by adding data recognizers to OSM model instances converted from DAML ontologies. We added data recognizers based on name matching between the lexical-object sets in an OSM model instance and data frames in our library of common prebuilt data frames, which we created for this purpose. Our matching algorithm performed well, with a precision of 77% and a recall of 89%.

We also provide an extended version of the Ontology Editor to allow a user to add and edit data frames and participation constraints in the extraction ontology. We applied a graph layout algorithm to the converted ontology to find a suitable screen layout for the ontology in the graphical editor.

We used our system to build four data-extraction ontologies. Three of the extraction ontologies that we built — software descriptions, car advertisements, and apartment rentals — are based on DAML ontologies that we created specifically for data extraction. The other extraction ontology — describing university courses — was based on an existing DAML ontology from the DAML repository [dam02]. We

conclude that DAML ontologies created for data extraction work better in our application than most existing DAML ontologies. We note, however, that we see nothing that prevents people from creating DAML ontologies, just as we have done, whose purpose is to describe data found in multiple-record Web documents.

Given a converted data-extraction ontology, we applied Ontos to Web pages to extract relational data. Next, we converted the extracted relational data to RDF data, making it accessible to Semantic Web agents. In addition to structuring the data, we also saved an XPointer to each item of extracted data, so that the extracted data could be superimposed over the original documents from which it was extracted. We created an RDF browser that allows a user to browse the extracted RDF data together with the original Web pages, so that a user can see the context of the extracted data.

Finally, we created a Web demo that integrates all the components of our RDF-extraction system. As a test of our system, we asked six students familiar with the BYU Ontos system to build part of a data-extraction ontology based on a DAML ontology, which we provided. The results of the experiment indicate that our data-frame library and our graphical ontology editor simplify the task of building extraction ontologies.

6.2 Future Work

As we have completed this project, we have identified four areas that we believe merit future work. The first two areas contribute to the data-extraction process, and the remaining two address the quality of the generated RDF data. 1) We believe it is possible to enhance our name matcher with value-based schema matching techniques based on sample values for the data frames in our library and sample values provided by the user for properties in the user's DAML ontology. 2) DAML properties always translate to binary relationship sets, but the BYU Ontos system can take advantage of n-ary relationship sets for data extraction. A possible future enhancement to our system is to recognize automatically or let the user specify DAML properties that can be combined to form ternary relationship sets in the data-extraction ontology. 3) Our system generates a unique URI for each instance, as required, but if an instance appears in more than one Web page, we generate more than one URI for it. It would

be useful if we could recognize previously extracted instances of an object in order to give them the same URI or relate the two URIs in some way. 4) We observed that most existing DAML ontologies do not work well for data extraction. A possible solution is to build DAML ontologies specifically for data extraction and then relate the elements of those ontologies with elements of more standard existing ontologies so that more Semantic Web agents could understand the semantics of the extracted data.

6.3 Conclusion

We conclude that this thesis has in large part achieved its aims of promoting the Semantic Web and illustrating the use of superimposed information. We believe that this and other similar projects will help provide the basis for the next generation of the Web.

Appendix A

Data-Frame Library

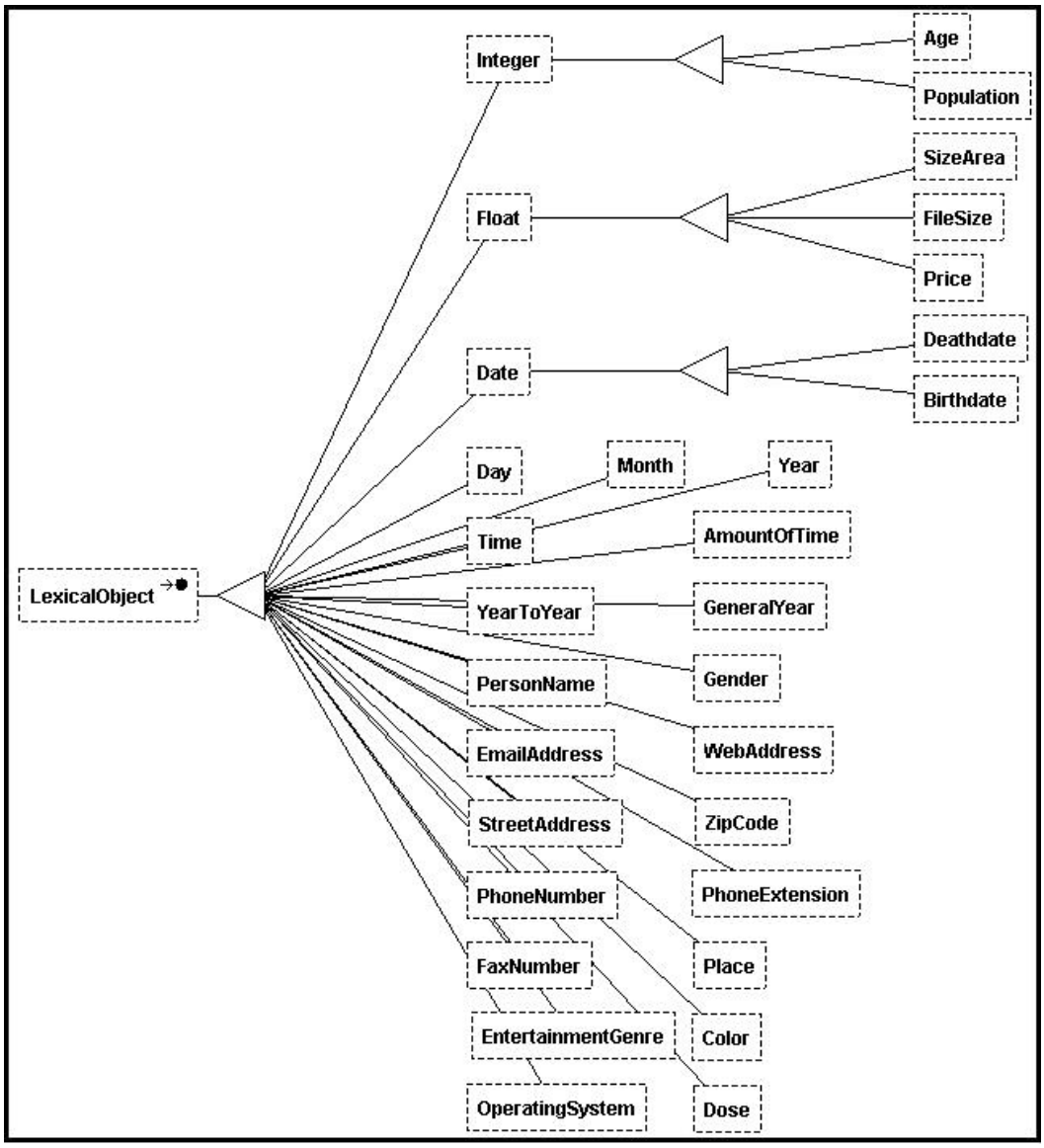

Figure A.1: This figure shows the names of the object sets in our version of the data-frame library as well as the generalization/specialization relationships among them.

```
<ObjectSet Lexical="Y" Name="Date"
     Aliases="day,year">
     <DataFrame SQLFieldLen="20">
         <ValuePhrase Label="date_1"
             ValueExpression="\b(January|February|March|April|May|June|
                 July|August|September|October|November|December|Jan|Feb|
                 Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec|Febr|Sept|Octob)
                 \s+[0-3]?\d,?\s+[1-9]\d\d\d"
             RContextExpression="(\s|\.)" />
         <ValuePhrase Label="date_2"
             ValueExpression="\b(January|February|March|April|May|June|
                 July|August|September|October|November|December|Jan|Feb|
                 Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec|Febr|Sept|Octob)
                 \s+[0-3]?\d(st|nd|rd|th|d),?\s+[1-9]\d\d\d"
             RContextExpression="(\s|\.)"
             SubFromExpression="(st|nd|rd|th|d)" />
         <ValuePhrase Label="date_3"
             ValueExpression="\b[0-1]?\d\s+[0-3]?\d(\s|,)\s*[1-9]\d\d\d"
             RContextExpression="(\s|\.)" />
         <ValuePhrase Label="date_4"
             ValueExpression="\b[0-1]?\d/[0-3]?\d/\d\d(\d\d)?\b"/>
         <ValuePhrase Label="date_5"
             ValueExpression="Month\.?\s*(1\d|2\d|30|31|\d),?\s*(\d\d\d\d)"
             ExceptionExpression="" />
         <ValuePhrase Label="date_6"
             ValueExpression="(1\d|2\d|30|31|\d)\s*Month\.?\s*(\d\d\d\d)" />
         <ValuePhrase Label="date_7"
             ValueExpression="\b\d\d?\d\d?\d\d\b" />
         <ValuePhrase Label="date_8"
             ValueExpression="\b\d\d?\d\d?\d\d\d\d\b" />
         <ValuePhrase Label="date_9-DayMonth"
             ValueExpression="(1\d|2\d|30|31|\d)\s*Month\.?" />
             <ValuePhrase Label="date_10-MonthDay"
             ValueExpression="(January|February|March|April|May|June|July|August|
                 September|October|November|December|Jan|Feb|Mar|Apr|May|Jun|Jul|
                 Aug|Sep|Oct|Nov|Dec|Febr|Sept|Octob)\s+[0-3]?\d"
             RContextExpression="(\s|\.)" />
     </DataFrame>
</ObjectSet>
```

Figure A.2: As an example, we give the data frame for the *Date* object set.

Appendix B

RDF Data Generation Algorithm

```
1)   generateRDFInstances()
2)       FOR EACH non-lexical object set domain_os DO
3)           FOR EACH relSet DO
4)               IF domain_os on domain side of relSet THEN
5)                   property := URI for the DAML property corresponding to relSet
6)                   range_os := range object set of relSet
7)                   table := table containing both domain_os and range_os
8)                   field1 := field in table for domain_os
9)                   field2 := field in table for range_os
10)                  domain := URI for domain_os
11)                  range := URI for range_os
12)                  FOR EACH row in table
13)                      value1 := URI for field1 in row
14)                      value2 := URI or lexical value of field2 in row
15)                      IF value1 and value2 are not null THEN
16)                          add statement:  '< value1 >< rdf : type >< domain >'
17)                          add statement:  '< value1 >< property >< value2 >'
18)                          IF range_os is non-lexical
19)                              add statement:  '< value2 >< rdf : type >< range >'
```

Figure B.1: This figure illustrates our algorithm for generating RDF data from an OSM model instance and a set of tables populated with relational data.

Bibliography

[BDM02] S. Bowers, L. M. L. Delcambre, and D. Maier. Superimposed schematics: Introducing E-R structure for in-situ information selections. In S. Spaccapietra, S. T., and Y. Kambayashi, editors, *21st International Conference on Conceptual Modeling (ER2002)*, pages 90–104, Tampere, Finland, October 2002.

[BG02] D. Brickley and R. V. Guha. RDF vocabulary description language 1.0: RDF Schema. W3C working draft, World Wide Web Consortium, Cambridge, Massachusetts, USA, April 2002. http://www.w3.org/TR/rdf-schema/.

[BLHL01] T. Berners-Lee, J. A. Hendler, and O. Lassila. The semantic web. *Scientific American*, pages 28–31, May 2001. http://www.sciam.com/2001/0501issue/0501berners-lee.html.

[BYRN99] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*, chapter 3: Retrieval Performance Evaluation. Addison Wesley, Menlo Park, California, 1999.

[CD99] J. Clark and S. J. DeRose. XML path language (XPath) version 1.0. W3C recommendation, World Wide Web Consortium, Cambridge, Massachusetts, USA, November 1999. http://www.w3.org/TR/xpath.

[Cir01] F. Ciravegna. Adaptive information extraction from text by rule induction and generalisation. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, Washington, USA, August 2001.

[Col93] M. K. Coleman. Aesthetics-based graph layout for human consumption. Master's thesis, University of California, Los Angeles, Los Angeles, California, USA, March 1993.

[CvHH+01] D. Connolly, F. van Harmelen, I. Horrocks, P. F. Patel-Schneider D. L. McGuinness, and L. A. Stein. DAML+OIL (March 2001) reference description. Technical report, DARPA Agent Markup Language Program, December 2001. http://www.daml.org/2001/03/reference.

[dam02] The DARPA Agent Markup Language Homepage, 2002. http://www.daml.org.

[DDG+02] S. J. DeRose, R. Daniel, P. Grosso, E. Maler, J. Marsh, and N. Walsh. XML pointer language (XPointer). W3C working draft, World Wide Web Consortium, Cambridge, Massachusetts, USA, August 2002. http://www.w3.org/TR/xptr/.

[ECJ+99] D. W. Embley, D. M. Campbell, Y. S. Jiang, Y. Ng S. W. Liddle, D. Quass, and R. D. Smith. Conceptual-model-based data extraction from multiple-record web pages. *Data Knowledge Engineering*, 31(3):227–251, 1999.

[Eik99] L. Eikvil. Information extraction from world wide web - a survey. Technical Report 945, Norweigan Computing Center, Oslo, Norway, 1999.

[EJN99] D. W. Embley, Y. S. Jiang, and Y. Ng. Record-boundary discovery in Web documents. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 467–478, Philadelphia, Pennsylvania, USA, June 1999. ACM Press.

[EJX01] D. W. Embley, D. Jackman, and L. Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Proceedings of the International Workshop on Information Integration on the Web (WIIW'01)*, pages 110–117, Rio de Janeiro, Brazil, April 2001.

[Emb80] D. W. Embley. Programming with data frames for everyday data items. In *Proceedings of the 1980 National Computer Conference*, pages 301–305, Anaheim, California, May 1980.

[Emb98] D. W. Embley. *Object Database Development: Concepts and Principles*. Addison-Wesley, Reading, Massachusetts, 1998.

[EMSS00] M. Erdmann, A. Maedche, H. Schnurr, and S. Staab. From manual to semi-automatic semantic annotation: About ontology-based text annotation tools. In K. Hasida P. Buitelaar, editor, *Proceedings of the COLING 2000 Workshop on Semantic Annotation and Intelligent Content*, Luxembourg, Luxembourg, August 2000.

[FAD+00] D. Fensel, J. Angele, S. Decker, M. Erdmann, H. Schnurr, R. Studer, and A. Witt. Lessons learned from applying AI to the web. *International Journal of Cooperative Information Systems*, 9(4):361–382, 2000.

[GGP+02] J. Golbeck, M. Grove, B. Parsia, A. Kalyanpur, and J. A. Hendler. New tools for the semantic web. In *Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Siguenza, Spain, October 2002. Published on line: http://www10.org/cdrom/papers/488/index.html.

[HD80] P. A. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[Hew00] K. Hewett. An integrated ontology development environment for data extraction. Master's thesis, Brigham Young University, Provo, Utah, USA, April 2000.

[Hom02] Home Page and Extraction Demo for BYU Data Extraction Group, 2002. http://www.deg.byu.edu.

[HS02] S. Handschuh and S. Staab. Authoring and annotation of web pages in CREAM. In *The Eleventh International World Wide Web Conference (WWW2002)*, Honolulu, Hawaii, USA, May 2002. Published on line: http://www2002.org/CDROM/refereed/506/index.html.

[HSC02] S. Handschuh, S. Staab, and F. Ciravegna. S-CREAM — semi-automatic CREAtion of metadata. In *Proceedings of 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Siguenza, Spain, October 2002.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42:741–843, 1995.

[KM02] M. Koivunen and E. Miller. W3C Semantic Web activity. In E. Hyvonen, editor, *Semantic Web Kick-Off in Finland*, pages 27–44, Helsinki, Finland, May 2002. Helsinki Institute for Information Technology, HIIT Publications.

[Lev65] V. I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[LEW95] S.W. Liddle, D.W. Embley, and S.N. Woodfield. Unifying Modeling and Programming Through an Active, Object-Oriented, Model-Equivalent Programming Language. In *Proceedings of the Fourteenth International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95), Lecture Notes in Computer Science, 1021*, pages 55–64, Gold Coast, Queensland, Australia, December 1995. Springer Verlag.

[LRNdST02] A. H. F. Laender, B. A. Ribeiro-Neto, A. S. da Silva, and J. S. Teixeira. A brief survey of Web data extraction tools. *ACM Sigmod Record*, 31(2):84–93, June 2002.

[LS99] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation REC-rdf-syntax-19990222, World Wide Web Consortium, Cambridge, Massachusetts, USA, February 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[MD99] D. Maier and L. M. L. Delcambre. Superimposed information for the Internet. In *SIGMOD Workshop on the Web and Databases (WebDB'99) (Informal Proceedings)*, pages 1–9, Philadelphia, Pennsylvania, USA, June 1999.

[MM02] F. Manola and E. Miller. RDF primer. W3C working draft, World Wide Web Consortium, Cambridge, Massachusetts, USA, April 2002. http://www.w3.org/TR/rdf-primer/.

[NBJBB97] G. Neumann, R. Backofen, M. Becker J. Baur, and C. Braun. An information extraction core system for real world german text processing. In *Proceedings of ANLP-97*, pages 208–215, Washington, D.C., USA, 1997.

[Pal02] S. B. Palmer. RDF in HTML: Approaches. http://infomesh.net/2002/rdfinhtml/, May 2002.

[Por80] M. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[RB01] E. Raham and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):344–350, 2001.