An Integrated Ontology Development Environment for Data Extraction

Stephen W. Liddle[1,⋆] Kimball A. Hewett[2] David W. Embley[2,⋆]

[1] Information Systems Group and Rollins eBusiness Center

[2] Computer Science Department

Brigham Young University, Provo, Utah 84602, U.S.A.
liddle@byu.edu, khewett@epixtech.com, embley@cs.byu.edu

Abstract. Data extraction is a necessary technology to deal with the huge and growing collection of unstructured and semistructured information available on the World Wide Web. Ontology-based data extraction is a robust approach, but the construction of ontologies is a technical task requiring the services of a human expert. We present a Java-based tool for the graphical creation and testing of data extraction ontologies. This tool leverages standards such as Java and XML to provide a portable, extensible, maintainable, feature-rich environment. This tool reduces the burden on expert ontology developers and simplifies the task of ontology creation.

1 Introduction

The amount and variety of information available on the World Wide Web continues to grow at a dramatic pace. Unfortunately, most Web data is mainly unstructured or semistructured, making it relatively difficult to search. Keyword searches tend to be imprecise, too often finding many irrelevant candidate responses while missing highly relevant results [Ape94]. We cannot conduct traditional database queries, which tend to give precise results, because the Web lacks the regular structure necessary for such queries.

Recently the *Semantic Web* has been proposed as a possible solution to this problem [BLHL01]. However its timeline has a long horizon, and even if it is ultimately successful, not all Web participants will supply precise metadata to characterize posted information. Furthermore, different readers may characterize posted information differently, so that a single, author-supplied meta-description will not be sufficient for all potential uses of the information. ([DM99] describes approaches for accommodating various information needs with superimposed information layers.)

[⋆] Supported in part by the National Science Foundation under grant IIS-0083127 and by the Kevin and Debra Rollins Center for eBusiness at Brigham Young University.

Other approaches include virtual database technology [GHR97], Web data modeling [AMM97,DMRA97], natural-language processing [CL96,Fla98], semistructured and Web query languages [ACC$^+$97,AM98,BDHS96], and the mediator-related concept of wrappers (e.g. [AK97]). (See [LRNdST02] for a thorough survey of this related work.) Wrapper generation is the most common approach to supplying a layer of structure for unstructured and semistructured sources.

A *wrapper* adds structure to data by extracting portions of the data in a wrapper-defined way. For example, someone wishing to build a database of facts about countries might design a wrapper to extract data from the CIA World Factbook Web site [CIA02]. A visitor to the page about Ukraine would discover "*land:* 603,700 sq km" which indicates the land area of the country in square kilometers. Austria's page contains the phrase "*land:* 82,738 sq km". A wrapper could be programmed to recognize the "land:" and "sq km" tokens and then extract the bounded integer. We could then issue a structured query through the wrapper for countries with an area greater than, say, 50,000 but less than 1,000,000 square kilometers. It is common for wrappers to use specific textual items (like "land:" or "sq km") or hidden HTML tags (such as the `<i>...</i>` tags used to make "land:" appear in italics on the Web page) as markers to indicate where information of interest is located.

A major difficulty with the wrapper approach is that Web sites change frequently, often breaking the wrappers. Even if the information content is the same, changing its layout or the associated HTML markup can easily break most wrappers. Significant effort goes into implementing and maintaining site-specific wrappers. There are several projects that seek to ease the burden of wrapper implementation and maintenance [Ade98,AK97,DEW97,KWD97,Sod97], but the task remains difficult.

Our approach to data extraction [ECJ$^+$99] uses an application ontology that describes a data-rich, ontologically narrow domain in a conceptual fashion. A domain is *data rich* if relevant Web pages contain many identifiable constants such as dates, names, ID numbers, times, currency values, and so on. A domain is *ontologically narrow* if it contains relatively few concepts. From this application ontology our system automatically generates a single wrapper that can be applied to any page relevant to the application domain.

Because the ontology describes information of interest in a general way, our approach is (1) applicable to a wide variety of Web pages relevant to the given domain, and (2) resilient to changes in relevant Web pages over time. The most difficult aspect of our robust and resilient approach to data extraction is the need for a domain expert to represent domain knowledge by manually creating an application ontology.

In this paper we present an integrated ontology development environment that helps domain experts by providing a graphical interface for ontology creation and testing. The remainder of the paper is organized as follows. We explain the concepts of ontologies and our particular approach to data extraction in Section 2. Then we describe our integrated ontology development environment in

Section 3. We report on the implementation process and lessons learned in Section 4. Finally, we conclude and discuss future work in Section 5.

2 Ontology-Based Data Extraction

Ontology is the branch of science concerned with the nature of being and relations among things that exist [Bun77,Bun79]. In computer science, the term generally refers to the specification of some conceptualization. While this is similar to the definition of *conceptual model*, ontologies differ from conceptual models by (1) focusing especially on extended definitions of relationships and concepts, and (2) having the explicit goal of sharing knowledge by defining a common theoretical framework and vocabulary so that interested agents can make and share a particular *ontological commitment* [Gru93].

For our purposes, *an ontology* is an instance of an augmented conceptual model that describes not only objects, relationships, and their constraints, but also rules regarding how objects may appear in an unstructured source. We start with an object-oriented conceptual model, OSM [EKW92], and add *data frames* [Emb80] to describe additional information needed for data extraction. Data frames specify patterns and keywords that may indicate the presence of an instance of a particular kind of object. They also specify data conversion rules and other useful information.

As illustrated in Figure 1, our data extraction method consists of the following five steps.

1. We develop an application ontology that describes an area of interest.
2. We parse this ontology to generate a database scheme and to generate rules for matching constants and keywords.
3. To obtain data from the Web, we invoke a record extractor that divides an unstructured Web document into individual record-size chunks, cleans them by removing markup-language tags, and presents them as individual unstructured record documents for further processing.
4. We invoke recognizers that use the matching rules generated by the parser to extract from the cleaned individual unstructured documents the objects expected to populate the model instance.
5. Finally, we use heuristics to determine which constants populate which records in the database scheme. These heuristics correlate extracted keywords with extracted constants and use relationship sets and cardinality constraints in the ontology to determine how to construct records and insert them into the database scheme. Once the data is extracted, we can query the structure using a standard database query language.

To make our approach general, we fix the ontology parser, Web record extractor, keyword and constant recognizer, and database record generator; we change only the ontology as we move from one application domain to another. A significant contribution of this approach is that we only perform the manual step, ontology development, once for a particular domain. This ontology covers

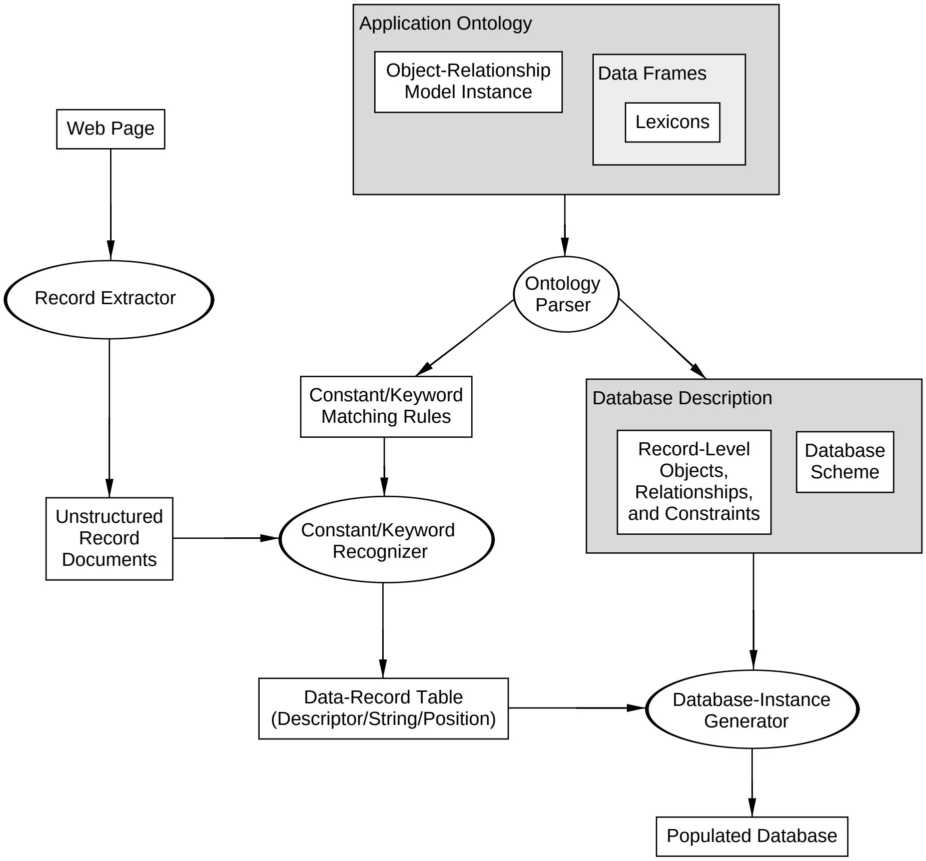

Application Ontology
Object-Relationship Model Instance
Data Frames
Lexicons
Web Page
Record Extractor
Ontology Parser
Constant/Keyword Matching Rules
Database Description
Record-Level Objects, Relationships, and Constraints
Database Scheme
Unstructured Record Documents
Constant/Keyword Recognizer
Data-Record Table (Descriptor/String/Position)
Database-Instance Generator
Populated Database

Fig. 1. Data Extraction and Structuring Process

all Web pages for that domain, and is insensitive to changes in Web-page format. Other approaches that rely on HTML structure or the order of data within an unstructured record must specify multiple wrappers (one for each structure pattern), and when a Web page undergoes a format change (a common occurrence), such wrappers must be rewritten to accommodate the new format [GHR97]. Our system generally does not rely on the order of data or the specific nature of a particular Web-page layout.

We now consider a specific example. Figure 2 shows a portion of an application ontology for the domain of car advertisements. The first four lines define object sets (such as Car, Year, Make, and Model) and relationship sets (such as Car has Year and Car has Make). The constraint [–> object] indicates that Car is the primary object set of interest for this domain, so when we extract data from a record using this ontology, we expect to find information about a single car. The numbers in square brackets are participation constraints. For example, Car

```
 1. Car [-> object];               Car [0:1] has Year [1:*];
 2. Car [0:1] has Make [1:*];      Car [0:1] has Model [1:*];
 3. Car [0:1] has Mileage [1:*];   Car [0:*] has Feature [1:*];
 4. Car [0:1] has Price [1:*];     Car [0:1] has PhoneNr [1:*];
 5. Year matches [4]
 6.      constant {extract "\d{2}";   context "\b'[1-9]\d\b";
 7.                substitute "^" -> "19"; },   ...
 8. Mileage matches [8]
 9.      keyword "\bmiles\b", "\bmi\.", "\bmi\b",
10.              "\bmileage\b", "\bodometer\b";
11. ...
```

Fig. 2. Car-Ads Extraction Ontology (Partial)

[0:1] has Year [1:*]; means that a car may have at most one associated year, but a year corresponds to one or more cars.

Lines 5–10 in Figure 2 give portions of the data frames associated with object sets Year and Mileage. Here we see that data frames include constant and keyword phrases composed of various regular expressions. On lines 6–7 we find a description of one form of year. If the text '97 were to appear in the unstructured source, this data frame would extract the constant 1997. We can interpret this Year data frame as "extract two digits in the context of a word boundary, apostrophe, a digit between 1–9, any digit, and another word boundary; then substitute 19 at the beginning of the string." The effect of this specification is to treat two-digit years from '40–'99 as 1940–1999. Presumably other specifications in the full data frame would handle other forms of years such as two-digit years from '00–'03, four-digit years, and so forth. The Mileage data frame shows how to specify keywords that might indicate proximity to a mileage constant. As indicated by this example, finding the word "miles," the abbreviation "mi," or the words "mileage" or "odometer" suggests that a mileage value could appear nearby. We use the well-accepted Perl syntax to write regular expressions.

Data frames describe specific forms of constants that may appear in a source document. The structural specifications of object sets, relationship sets, and constraints describe how concepts in the car-ads domain relate to one another. We use these descriptions to guide the process of assembling extracted constants into structured records. See [ECJ$^+$99] for more details.

The full extraction ontology for car advertisements comprises over 600 lines of code. The full Year data frame has eight different types of constants (each with its respective extract/context/substitute phrases as appropriate). The Make data frame contains dozens of different manufacturers, such as Alfa Romeo which is represented by the regular expression "\balfa(\s*romeo)?\b". Many of the regular expressions in the ontology are more intricate. As can be seen from this example, ontology development—even for a relatively narrow domain like car ads—can be a complex process. The integrated ontology development environ-

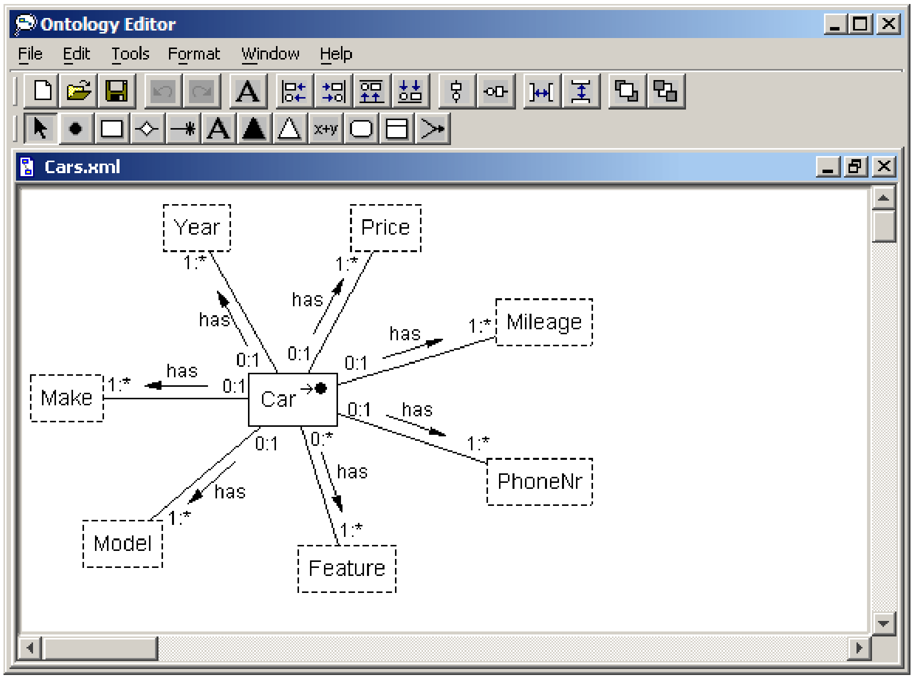


Fig. 3. Screen Shot of Main Ontology Editor Window

ment, described in the next section, provides features to facilitate this creation activity.

3 The Integrated Ontology Development Environment

The integrated ontology development environment (or “Ontology Editor”) is a graphical tool with three main components:

1. A structural model editor for defining object-relationship structures.
2. A data frame editor for graphically defining regular expressions used to identify constants and keywords.
3. A document viewer that highlights phrases matching regular expressions.

Figure 3 shows the main window of the Ontology Editor. The structure is similar to that of many graphics programs. The general architecture follows the “multiple document interface” paradigm, so the user can have several ontologies open in child windows simultaneously. A tool bar along the top provides quick, graphical access to various common functions.[3] These are grouped into file operations

[3] The toolbar in Figure 3 also includes tools for creating behavior-oriented OSM elements that have no impact on data extraction.

(new, open, save), edit operations (align, distribute, change front-to-back order), and element creation operations (create object, object set, relationship set, and so on).

The diagram in the child window of Figure 3 represents the same ontology as the one expressed textually in Figure 2. OSM has fully equivalent textual and graphical notations, as these figures indicate. However data frames do not lend themselves easily to graphical diagrams because of the nature of regular expressions. We thus represent structural aspects of extraction ontologies using the graphical notation (as Figure 3 shows), but we represent data frames primarily with a textual notation (as Figure 2 shows).

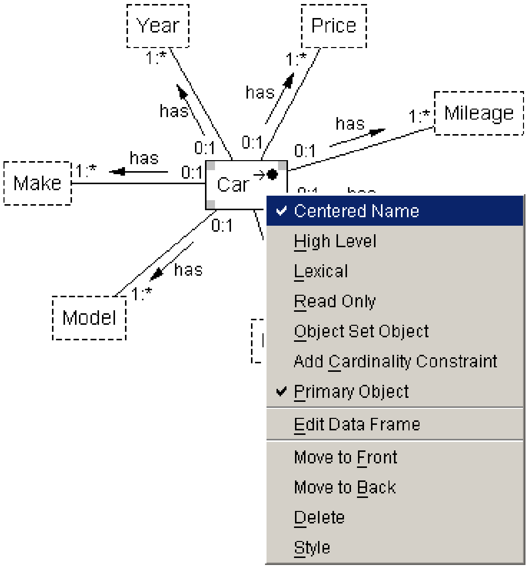
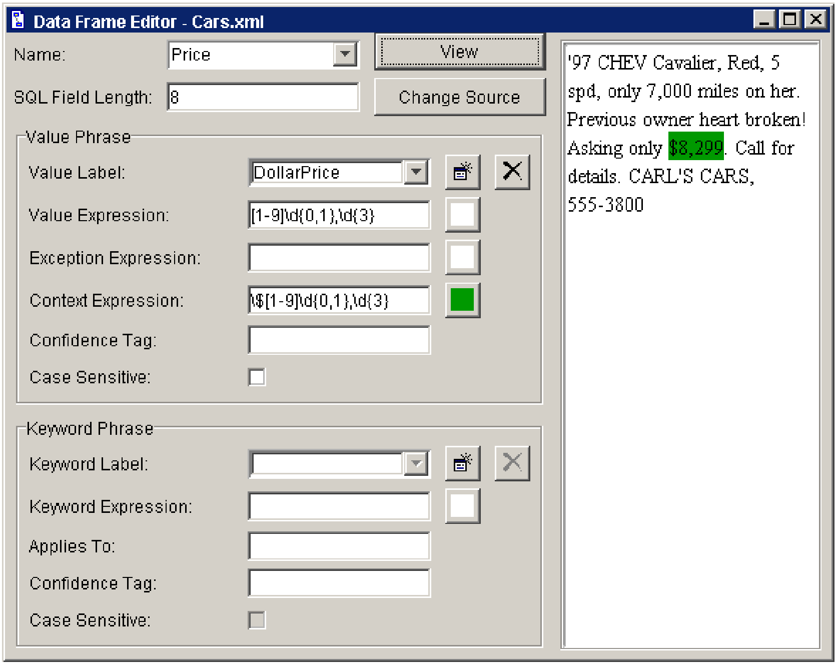


Fig. 4. Context-Sensitive Pop-Up Menu

The Ontology Editor makes good use of context-sensitive pop-up menus. For example, right-clicking on the Car object set results in the pop-up menu in Figure 4. Right-clicking on a relationship set displays a different pop-up menu with features tailored to the properties of relationship sets. The Ontology Editor supports various intricate details of editing OSM model instances such as the high-level, lexical, read-only, and object-set object designations to which the pop-up menu in Figure 4 alludes.

The data frame editor in Figure 5 is central to the task of developing extraction ontologies. Because data frames can be fairly complex, it is helpful to the human—especially one who is a domain expert but not necessarily an ontology-development expert—to use a GUI template to guide the creation of data frames. The other primary benefit of the data frame editor is that it supports quick de-



Data Frame Editor - Cars.xml

Name: Price

SQL Field Length: 8

Value Phrase

Value Label: DollarPrice

Value Expression: [1-9]\d{0,1},\d{3}

Exception Expression:

Context Expression: \$[1-9]\d{0,1},\d{3} 🟩

Confidence Tag:

Case Sensitive: □

Keyword Phrase

Keyword Label:

Keyword Expression:

Applies To:

Confidence Tag:

Case Sensitive: □

'97 CHEV Cavalier, Red, 5 spd, only 7,000 miles on her. Previous owner heart broken! Asking only $8,299. Call for details. CARL'S CARS, 555-3800

Fig. 5. Screen Shot of Data Frame Editor Window

bugging of regular expressions by letting the user select (1) a source file to view, and (2) colors to associate with various regular expressions. In Figure 5 we see that the user has selected a color for the context expression `\$[1-9]\d{0,1},\d{3}`, and in the text pane to the right the one matching string is highlighted in the same color. This simple idea is very helpful to the ontology developer, because regular expressions can be difficult to read, and minor changes in complex regular expressions often have large effects on the results.

The other Ontology Editor feature we highlight here is its ability to generate high-resolution PostScript output. Graphical desktops make it easy to capture screen shots, but the resulting bitmap images are not ideal for publishing. We chose to support exporting graphical diagrams to PostScript or Encapsulated PostScript (the latter format is best suited for embedding images in other documents, especially papers written in LaTeX). Figure 6 demonstrates the quality of this feature for the car-ads ontology shown in Figure 3. Because our tool is most likely to be used primarily in an academic environment, this capability is significant to us.

4 Implementation Issues

In this section we describe the lessons learned during our implementation of the Ontology Editor and its predecessor tools. We began developing graphical

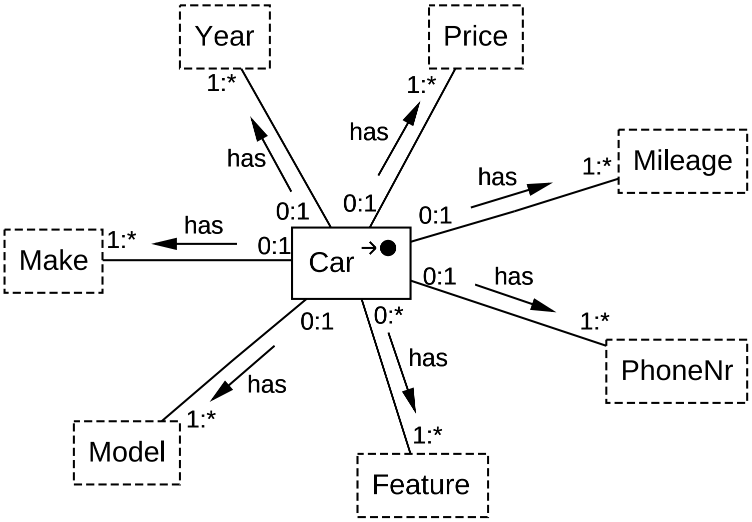


Fig. 6. High-Resolution Encapsulated PostScript Output

editors for the OSM conceptual model in the early 1990's. We wrote our first implementation, OSM Composer, in C++ for the Unix/X-Window environment. This tool used an experimental, high-level graphical user interface library developed by the user interface research community. We gave up some control over specific low-level features, but gained much by using an abstract interface library. However, evolving language, compiler, and operating system issues made it difficult (but possible) to port Composer from its original HP-UX platform to the other open-source, Gnu-based platforms (we currently can run Composer on machines running Sun's Solaris operating system). For the sake of historical continuity, we may yet port Composer to Linux, but it will require some effort to do so.

Even though C++ is an object-oriented language and OSM Composer uses an object-oriented design, the architecture of Composer did not leverage the model-view-controller approach sufficiently. Also, the event model was fairly monolithic, introducing too much modality into the event handling code. As was common for its era, Composer used a proprietary file format for storing model instances, making interchange with other tools difficult. In spite of these weaknesses, Composer was an excellent first step. We were able to test numerous ideas in graphical model development and see which ones best suited our needs.

Our second tool project, OSM Allegro and Design Assistant [Car99], built on the ideas of Composer but followed a different philosophy. Allegro was written again in C++, but this time was targeted to the ubiquitous 32-bit Microsoft Windows platform. Allegro did leverage the model-view-controller architecture more effectively, and provided a more robust end-user experience. Because Composer's architecture was too rigid, the Allegro project was especially attuned to the issue of providing an extensible platform where tools could be plugged in. To demonstrate the success of Allegro in this regard, our research group created the OSM Design Assistant, which leverages the COM aspect of Allegro to wrap a model-design tool around the diagram editor. Our Design Assistant is a synergistic tool that helps users apply the design transformations (such as functional dependency-based data reductions) described in [Emb98].

Allegro turned out to be a great tool, but it also had limitations we wanted to avoid. Like Composer, Allegro used a proprietary file format, making interchange a bit more complex. Its Windows-based print capability was page-oriented, making it harder to turn Allegro diagrams into figures we could embed in our publications. Though most of our community has access to a Windows desktop, we still wanted to be able to edit diagrams on Unix and Macintosh platforms. But the largest issue was integration with our other tools.

4.1 Standardizing on Java

We implemented our original data extraction toolkit in a combination of Perl, C++, and then Java code, running on a Unix platform. Given the diversity of platforms, languages, and tools we were using, it was clear we needed to simplify and do a better job of using emerging standards which were beginning to solidify. Java was becoming sufficiently robust for sophisticated graphical application development, and XML was rapidly gaining acceptance (and importantly, programming support in the form of high-level, stable code libraries). We had created a Java-based Pattern Editor tool to help with debugging complex regular expressions, and newer elements of our code base were increasingly gravitating toward Java. So it was a natural choice to decide on Java as the common platform for our tools. We saw the following benefits to adopting Java:

- It has considerable momentum in industry and academia. In a laboratory where graduate students come and go every year, it is easier to maintain code across generations of programmers when we use a popular language.
- Because of its virtual machine strategy, Java is highly portable across the platforms we are interested in supporting.
- Java is a wonderfully robust language, supporting concurrency, exception handling, regular expressions, graphics, GUI frameworks, XML, and a slew of other features. Significantly, this large API support base continues to grow over time. The new Java Web Services developer pack, for example, promises to simplify our future XML development efforts.
- Though not an ideal language, Java simplifies certain critical aspects of complex systems, most notably memory management and handling of pointers.

To be certain, we have experienced our share of frustrations with the Java platform. Mostly this was due to our desire to access features not yet available, but Java's rapid evolution has also created maintenance difficulties. For example, at the time we began our work on the Ontology Editor, the Java Foundation Classes, or Swing interface, had just been released and worked with JDK 1.1.7.

Later, JDK 1.2 integrated Swing into the core API. The use of Swing and its advanced event model was a major help to our project. Such features as support for drawing lines of variable widths was not available until JDK 1.2's Java 2D API. This advanced graphics API also provided very useful capabilities to rotate, scale, and translate graphical objects. Ready support for highlighting text (for our regular expression debugging feature) was not available until JDK 1.3. Since we finished the first version of the Ontology Editor, the OROMatcher regular expression library has been rewritten and incorporated directly into JDK 1.4.

Java performance, while not quite up to native implementation speeds, is acceptable, especially after the just-in-time compiler has done its work. Similarly, integration with the Windows environment is sufficient (and ever improving), though not quite as refined as a native implementation might be.

4.2 Standardizing on XML

When we created OSM, we developed graphical and textual notations for all components of the model. A textual notation can be stored directly in a text file, but a graphical notation requires representation in a well-defined, structured format. Our first model analysis tool used a proprietary, textual format that was well organized for the purposes of performing formal verification of model constraints. OSM Composer used this same format and added a layer of presentation information in a separate text-based format. In the early stages of tool development, it was quite helpful to be able to modify these files in an ad-hoc way in a text editor. Allegro used a proprietary, binary format that was more efficient for the tool to read and write, but was impractical to modify in an ad-hoc fashion.

XML was a natural choice for Ontology Editor's external data representation format. Many of the same issues arguing for the use of Java likewise favor the use of XML. There is considerable momentum behind XML. It supports easy information interchange; it is highly flexible. Drawbacks, such as the space inefficiency of lengthy, redundant markup tags, are insignificant in the context of our application.

To implement XML support, we had to create a Document Type Definition (DTD) to specify what constitutes a well-formed XML representation of an OSM model instance. OSM was one of the first conceptual models to have a complete, formal definition [Cly93]. From the beginning, the OSM metamodel provided a solid foundation for our theoretical research and tool development. When it came to constructing a DTD, the process was a straightforward representation of the metamodel in DTD syntax. Figures 7 and 8 respectively show portions of the OSM DTD and XML version of the car-ads extraction ontology.

Again like Java, XML standards continue to evolve rapidly. The newer, more powerful XML Schema allows for more precise characterization of constraints. We have developed an XML Schema description of OSM that will be used in the next release of the Ontology Editor. We will also use the improved XML support available in the new Java Web Services API.

```
<!ELEMENT OSM (Style?,(ObjectSet | Object | GeneralConstraint | Note |
              RelationshipSet | GenSpec | Association | Aggregation |
              State | Transition | Conjunction | Macro | Lexicon)*)>
<!ATTLIST OSM  x                     CDATA        #IMPLIED
              y                     CDATA        #IMPLIED
              order                 CDATA        "0"
              width                 CDATA        #IMPLIED
              height                CDATA        #IMPLIED
              ID                    NMTOKEN      "1" >
<!ELEMENT ObjectSet (DataFrame?, Style?, OSM?)>
<!ATTLIST ObjectSet
              ID                    NMTOKEN      #REQUIRED
              Name                  CDATA        "ObjSet"
              x                     CDATA        "0"
              y                     CDATA        "0"
              order                 CDATA        "0"
              width                 CDATA        "0"
              height                CDATA        "0"
              Lexical               (Y|N)        "N"
              ReadOnly              (Y|N)        "N"
              HighLevel             (Y|N)        "N"
              Primary               (Y|N)        "N"
              Centered              (Y|N)        "Y"
              ObjectSetObject       NMTOKEN      #IMPLIED
              CardinalityConstraint CDATA        #IMPLIED >
```

Fig. 7. DTD for OSM Extraction Ontologies (Partial)

4.3 Implementation Goals

We established three broad goals when we set out to create the Ontology Editor:

1. *Portability.* We wanted to be able to use the tool on a wide variety of platforms, including Windows, Unix, and Macintosh. Java's virtual machine approach let us realize this goal.
2. *Extensibility.* By following Allegro's lead and using a strong model-view-controller architecture, we created a clean infrastructure that has room for extension. Also, use of a pure Java solution enables integration of all our tools (as they are converted to pure Java) within a single application framework.
3. *Maintainability.* The Ontology Editor's code uses javadoc comments extensively for internal documentation. Also, for improved communication we created high-level conceptual models of the various Ontology Editor classes.

Extensibility and maintainability are difficult to quantify, but anecdotally, we are satisfied with the level of achievement in these areas. For example, menus and toolbar buttons are configured by a simple entry in a properties file (without the need to recompile the application). Also, we explicitly tested the extensibility of the major DrawObj class from which all drawable components inherit by asking

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE OSM SYSTEM 'osm.dtd'>
<OSM x="0" y="0" ID="21" width="578" height="337">
  <ObjectSet x="146" y="118" ID="1" Name="Car" Primary="Y" order="1">
  </ObjectSet> ...
  <ObjectSet x="213" y="9" ID="7" Name="Price" Lexical="Y" order="7">
    <DataFrame SQLFieldLen="8">
      <ValuePhrase Label="DollarPrice"
                   ValueExpression="[1-9]\d{0,1},{3}"
                   ReqContextExpression="$[1-9]\d{0,1},\d{3}"
                   ReqContextExpColor="ff00ff00" />
    </DataFrame>
  </ObjectSet> ...
```

Fig. 8. XML Version of Car-Ads Extraction Ontology (Partial)

another student programmer to implement the OSM *association* and *aggregation* relationship sets. The student accomplished the task in just a few days, spending most of his time on the relatively complex paint() method. Since then we have successfully added major components to the Ontology Editor without compromising its architecture.

5 Conclusion

In this paper we have reported on the results of a project that represents the culmination of approximately a decade of tool development based around our theoretical research in object-oriented conceptual models and ontology-based data extraction. Our goal in this development effort has not been to create a commercial-quality tool, but rather an effective test bed for our academic research. We have reported on some of the history of that development, and lessons we have learned along the way.

Because it is an important activity for Web users today, data extraction continues to be a ripe area for research. We believe we will be able to further our research effectively by using our integrated ontology development environment as the platform for future work. We are confident of this because our graduate students continue to be able to leverage the Ontology Editor platform several years following its initial version.

A more complete description of the initial version of our integrated ontology development environment is found in [Hew00]. Improvements (such as automatic graphical layout of ontologies specified using the older textual notation) were also introduced in [Cha03], and we have current projects that will continue the evolution.

We expect to accomplish the following activities with the Ontology Editor in the near future:

- Migrate from DTD to XML Schema for XML metadata.

- As they become available in pure Java (rather than the current mixture of Java and C++) integrate our data extraction tools more thoroughly into the Ontology Editor framework.
- Experiment with alternate, high-bandwidth means for creation of data frames. Two ideas we are considering include using a spreadsheet-style data-input window, or using a tree view for more rapid navigation. A masters thesis that is currently in progress will experiment with these possibilities.
- Experiment with the structure of data frames and the process of debugging them. Current techniques help debug "in the small" by identifying individual matches to regular expressions. We could also highlight the results of the concept-structuring step of the data extraction process, so the user can see a more global picture of the effect of the data frame.

References

[ACC^{+}97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1:5–9, 1997.

[Ade98] B. Adelberg. NoDoSE—a tool for semi-automatically extracting structured and semistructured data from text documents. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 283–294, Seattle, Washington, June 1998.

[AK97] N. Ashish and C. Knoblock. Semi-automatic wrapper generation for Internet information sources. In *Proceedings of the CoopIS'97*, 1997.

[AM98] G.O. Arocena and A.O. Mendelzon. WebOQL: Restructuring documents, databases and webs. In *Proceedings of the Fourteen International Conference on Data Engineering*, February 1998.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To weave the web. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases*, pages 206–215, Athens, Greece, August 1997.

[Ape94] P. M. G. Apers. Identifying Internet-related database research. In *Proceedings of the 2nd International East-West Database Workshop*, pages 183–193, Klagenfurt, 1994. Springer-Verlag.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of SIGMOD'96*, June 1996.

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

[Bun77] M.A. Bunge. *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World.* Reidel, Boston, 1977.

[Bun79] M.A. Bunge. *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems.* Reidel, Boston, 1979.

[Car99] E.H. Carter. A design assistant for generating relational database schemes. Master's thesis, Computer Science Department, Brigham Young University, 1999.

[Cha03] T. Chartrand. Ontology-Based Extraction of RDF Data from the World Wide Web. Master's thesis, Department of Computer Science, Brigham Young University, Provo, Utah, February 2003.

[CIA02] CIA World Factbook, 2002. URL: http://www.cia.gov/cia/publications/factbook/.

[CL96] J. Cowie and W. Lehnert. Information extraction. *Communications of the ACM*, 39(1):80–91, January 1996.

[Cly93] S.W. Clyde. *An Initial Theoretical Foundation for Object-Oriented Systems Analysis and Design.* PhD thesis, Department of Computer Science, Brigham Young University, Provo, Utah, 1993.

[DEW97] R.B. Doorenbos, O. Etzioni, and D.S. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proceedings of the First International Conference on Autonomous Agents*, pages 39–48, Marina Del Rey, California, February 1997.

[DM99] L. Delcambre and D. Maier. Models for superimposed information. In P.P. Chen, D.W. Embley, J. Kouloumdjian, S.W. Liddle, and J.F. Roddick, editors, *Advances in Conceptual Modeling: Proceedings of the Workshop on the World Wide Web and Conceptual Modeling (WWWCM'99)*, volume LNCS 1727, pages 264–280, Paris, France, November 1999. Springer Verlag.

[DMRA97] L.M.L. Delcambre, D. Maier, R. Reddy, and L. Anderson. Structured maps: Modeling explicit semantics over a universe of information. *International Journal on Digital Libraries*, 1(1):20–35, April 1997.

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

[EKW92] D.W. Embley, B.D. Kurtz, and S.N. Woodfield. *Object-oriented Systems Analysis: A Model-Driven Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[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.

[Fla98] S. Flank. A layered approach to NLP-based information retrieval. In *Proceedings of COLING'98*, pages 397–403, 1998.

[GHR97] A. Gupta, V. Harinarayan, and A. Rajaraman. Virtual database technology. *SIGMOD Record*, 26(4):57–61, December 1997.

[Gru93] T.R. Gruber. A translation approach to portable ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Hew00] K.A. Hewett. An Integrated Ontology Development Environment for Data Extraction. Master's thesis, Department of Computer Science, Brigham Young University, Provo, Utah, April 2000.

[KWD97] N. Kushmerick, D.S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, pages 729–735, 1997.

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

[Sod97] S. Soderland. Learning to extract text-based information from the World Wide Web. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 251–254, Newport Beach, California, August 1997.