Acquiring OWL Ontologies from XML Documents

Martin J. O’Connor
Stanford Center for Biomedical Informatics Research
Stanford, CA 94305, U.S.A.
martin.oconnor@stanford.edu

Amar K. Das
Stanford Center for Biomedical Informatics Research
Stanford, CA 94305, U.S.A.
amar.das@stanford.edu

ABSTRACT
Converting non-Semantic Web encoded information to ontology-based languages such as the Web Ontology Language (OWL) is an important knowledge acquisition challenge. In many domains, a large amount of information is represented in eXtensible Markup Language (XML), which has driven the development of general-purpose tools for converting XML to OWL. These tools suffer from a variety of shortcomings, however, including a requirement for multi-stage mapping processes and limited mapping techniques. A general shortcoming is that the mapping methods are not OWL-centric and thus limit the complexity of the generated OWL ontologies. To address these shortcomings, we developed an OWL-based language that can transform XML documents to arbitrary OWL ontologies. The language is based on the Manchester OWL Syntax and extends it with XPath queries to support references to XML documents. Our language provides a compact, user-friendly approach for converting XML to OWL.

Categories and Subject Descriptors
I.2.4 Knowledge Representation Formalisms and Methods – representation languages.

General Terms
Languages.

Keywords
OWL, XML, knowledge acquisition, Manchester Syntax.

INTRODUCTION
Acquiring knowledge from a variety of information formats is a key Semantic Web challenge. An array of tools has been developed to acquire knowledge from formats such as relational databases [1], spreadsheets [2], and XML [3], and to map this knowledge to RDF and OWL [4]. Because of its extensive use in data exchange and integration, XML has been a focus of these activities. XML and its associated schema language XML Schema [5] define an approach to modeling information in a document and thus provide a standardized data exchange mechanism.

A general approach used by mapping tools is to transform an XML Schema instance to an equivalent OWL ontology, and then populate the ontology with instances from XML documents conforming to the schema. This process presents many challenges, however, because of the different modeling goals of XML Schema and OWL. The goal of XML Schema is primarily to describe the syntactic structure of a document. In contrast, the primary goal of an OWL ontology is to semantically describe concepts in a domain and the relationships between them. As a result, even if a particular XML document can be transformed to OWL, the result is often not an immediately useful ontology. Also, many XML constructs are intrinsically ordered, whereas OWL is entirely set oriented.

These different characteristics can produce a mismatch that must be reconciled by the mapping process. In general, a second transformation stage is required. This stage takes the implicit semantics of the XML document and makes them explicit in the generated OWL ontology. Much of the complexity related to content transformation occurs at this stage.

Current tools either address only the first stage of this transformation process or adopt a limited two phase approach. In general, the first stage is straightforward and can be semi-automated because of the relative simplicity of XML Schema. The second stage requires richer user-specified transformations. Ideally, this stage should support arbitrary OWL constructs but in practice, current tools limit mappings to a small predefined set. This shortcoming is a reflection of the weakness of OWL support in these tools’ mapping methods.

There is a need for a language that supports transformations to unrestriced OWL ontologies. Ideally, it would be easy to learn for users familiar with XML and OWL. It should also produce mappings that are relatively concise and maintainable. We have created XMLMaster, which we believe is such a language. It is built on the Manchester OWL Syntax [6], a widely used language for declaratively describing OWL ontologies. We extend the Manchester Syntax and combine it with the XPath XML query language [7]. The resulting language supports the extraction of content from XML documents and the generation of OWL ontologies using that content. It provides a declarative approach to transform from XML to OWL with no limitations on the type of OWL ontologies that can be generated.
RELATED WORK
The first generation of mapping tools involved relatively direct translations from XML documents to RDF or OWL. Klein presented one of the earliest approaches [8]. His approach used an RDF Schema mapping ontology to identify relevant content in a source XML document to generate a one-way mapping to RDF. Another study outlined a bi-directional mapping approach that used a mapping described in XML Schema to map XML concepts to an RDF ontology [9]. A related approach supported both RDF and OWL [10]. Another effort mapped an XML Schema to an OWL ontology and then created instances in that ontology from XML documents [11]. Garcia et al. and Kunfermann and Drumm developed similar approaches [12, 13]. All of these approaches generate new ontologies from XML using only terms in the source XML document.

Mappings in the second generation of tools were more complex, and allowed interoperetion with existing ontologies. In general, the mapping process was two-phased. The first phase generated an intermediate ontology from source XML schemas or instances. The second mapped the intermediate ontologies to an existing ontology [14-16]. A system called WEESA [17], for example, supported mapping from XML to RDF with manually created rules. A similar system was described by Kobeissy et al. [18]. It mapped an XML Schema to OWL concepts and XML instances to OWL individuals. This approach allowed selective extraction of XML content using XPath. Tous et al. employed a related approach, also using XPath [19]. A recent system developed by the authors adopted a variant of this approach [3]. It mapped an XML document to instances of an OWL ontology describing the serialization of the document. It then used the Semantic Web Rule Language (SWRL; [20]) to map these instances to instances in a domain ontology. This approach provided great flexibility, but managing and debugging rule sets for non-trivial mappings was a challenge. Yet another variant used a meta-model-based approach to map between the different models provided by XML Schema and OWL [21].

A general limitation of these approaches is that none provided a custom mapping language. In general, mapping rules were specified using a selection of predefined templates. Thus, ad hoc mappings were not possible, which limited expressivity. Systems using XPath could support essentially arbitrary selection of XML content, but the range of possible transformation of content to OWL was limited. As a result, developers could generate only very basic OWL ontologies, and a decoupled post-processing phase was often required.

CORE LANGUAGE FEATURES
XMLMaster is a new declarative OWL-centric mapping language that addresses these limitations. It is a domain-specific language (DSL) that supports XML documents that do not necessarily have an XML Schema describing them. It can also process unstructured text content in these documents. Full coverage of all OWL constructs is a primary feature. In addition to defining simple OWL entities such as named classes, properties, and individuals, class expressions and potentially complex necessary and sufficient declarations are expressible. Additionally, the language aims to be concise and simple to learn for users familiar with OWL and XML. It also provides debugging support, which is a general usability requirement when developing a custom language. The high levels of complexity when mapping from XML to OWL make this support crucial. In particular, previewing the final result of a mapping expression before executing it can greatly assist in debugging. A key feature is thus to provide instantaneous preview of mapping results before they are inserted into the OWL ontology.

Rather than designing a DSL from scratch, we built our language on the Manchester OWL Syntax [6], a widely used DSL for declaratively describing OWL ontologies. This language has concise clauses for defining common OWL entities. It also provides full coverage of all OWL constructs and, because it is the standard presentation syntax in many OWL-based ontology editing tools, it is familiar to most users of OWL. It also has a clean language definition, allowing it to be extended in a principled way. Our language is a superset of the Manchester OWL Syntax. It extends it to allow the use of XPath expressions to refer to XML content. Primarily, it introduces a new reference clause to support this extraction.

![XML document describing books in a bookstore.](image)

**Figure 1.** XML document describing books in a bookstore.

**Basic Reference Clause**
A reference clause uses an XPath expression to indicate content in an XML document. In our DSL, this reference clause can substitute for any clause in a Manchester Syntax expression that indicates an OWL class, property, individual, data type, or data value. Reference clauses are prefixed with @ and are followed by an XPath expression.

For example, Figure 1 is an XML document showing books in a bookstore. A reference to the first book’s name element is written:

```
@/store/book[1]/name
```

To return element content, the `text` function can be used:

```
@/store/book[1]/name/text()
```

1 Our language wiki has a full description of the language [22].
The element name can be obtained in a similar way.

The reference clause can be used in an expression to define OWL constructs using XML content. For example, the following expression takes the text in the first book element in Figure 1 and declares an OWL named class as a subclass of an existing Book class:

\[
\text{Class: } @/\text{store/book[1]/name/text()}
\text{SubClassOf: Book}
\]

This expression declares an OWL class named by the contents of the first book element (“Huckleberry Finn” in this case) and asserts that it is a subclass of class Book. If the class has previously been declared and is not already a subclass of Book, then the subclass relationship will be established. In this way, references can be used to define new OWL entities or to refer to existing entities.

The language has default options to automatically extract either an element’s content or its name. As discussed later, this default can be changed globally. In the following examples, the default is to use an element’s content, so the text function qualification is omitted.

A similar expression to declare an individual of type Book using the element contents as its name can be written:

\[
\text{Individual: } @/\text{store/book[1]/name Types: Book}
\]

Of course, XPath expressions can match multiple elements. If the item selector is omitted from the above expression, multiple book elements are returned. For example, if the item selector is omitted, as in the following expression, individuals will declared for all book elements found:

\[
\text{Individual: } @/\text{store/book/name Types: Book}
\]

The Manchester Syntax supports a clause for associating property values with individuals. This clause contains a list of property value declarations. For example, the following is an expression specifying that an individual created from the first book element obtains a value for a data property value from the associated element and a category property value from its category attribute:

\[
\text{Individual: } @/\text{store/book/name Types: Book}
\text{Facts: } \text{price } @/.\text{/price,}
\text{category } @/.\text{category}
\]

As can be seen, relative references are used here to refer to the element and the attribute. The language interprets relative references in terms of the most recent non-relative reference in an expression. As with element text qualification, value qualification is the default, and can be omitted for attribute nodes.

Any XPath expression can be used in a reference. For example, expressions can select nodes that meet particular criteria. Using this approach, the previous example can be modified to select books with a price over $25:

\[
\text{Individual: } @/\text{store/book/name Types: Book}
\text{Facts: } \text{price } @/.[\text{price}>25.0]/\text{price}
\]

Similarly, the XPath expression can be modified to select only the first three book elements in the XML document:

\[
\text{Individual: } @/\text{store/book[position()<=3]/name Types: Book}
\]

Document structural information can also be recorded. For example, the position of a book element in a document can be retrieved using the XPath position and count functions. Maintaining positional information from XML documents is often essential, as OWL has no native list support. Summary information about documents, such as the number of elements of a particular type, can be recorded in a similar way using XPath counting functions.

XMLMaster supports the full range of Manchester Syntax expressions. For example, using the standard Manchester Syntax, annotation properties can also be associated with declared entities. Using this clause, a string data type annotation property called source can be used to associate a declared book individual with an annotation as follows:

\[
\text{Individual: } @/\text{store/book/names Types: Book}
\text{Annotations: } \text{source } "\text{From: book XML document"}
\]

OWL class and property expressions can also be used. In general, a class or property expression may occur anywhere a named class or property can occur. The following expression defines conditions for a class SalesItem; it uses the contents of a book element’s price and name sub-elements:

\[
\text{Class: SalesItem}
\text{EquivalentTo: } @/\text{store/book/name}
\text{SubClassOf: price value } @/.\text{/price}
\]

Using this approach, any OWL axiom can be declared using the appropriate Manchester Syntax clause, with XPath references used in these clauses to extract XML content.

Reference Mapping Directives

The basic reference clause outlined above can deal with straightforward mappings. In many cases, however, references need to be qualified to resolve ambiguity or indicate additional processing directives. XMLMaster provides a directives clause to support this specification.

The most basic directive controls the type of a generated OWL entity. In general, XMLMaster tries to infer the type of an entity in a reference, but this inference is not always possible. To deal with this case, our language supports explicit entity type specifications. A reference may be followed by a parentheses-enclosed entity type specification to explicitly declare the type of the referenced entity. This specification can indicate that the entity is a named OWL class, an OWL object or data property, an OWL individual, or a data type. The keywords to specify the types are the standard Manchester Syntax keywords Class, ObjectProperty, DataProperty, and Individual, plus any XSD type name (e.g., xsd:int). The following uses this specification to write the book individual declaration above:

\[
\text{Individual: } @/\text{store/book/name(Individual)}
\text{Types: Book}
\]

In many cases, specifying the super class, super property, individual class membership, or data type of referenced entities is also desired. While these relationships can be
defined using standard Manchester Syntax expressions, doing so often entails the use of multiple mapping expressions. To concisely support defining these relationships, a reference may be followed by a parentheses-enclosed list of type names. Using this approach, the above drug declaration can be written:

```xml
<Individual: @/store/book/name(Individual Book)
```

Type specifications can themselves be references. Super properties, individual class membership, and data types can be specified in the same way.

**Global Mapping Directives**

XMLMaster supports several global processing directives to specify default configuration options for the mapping process. They were designed to allow specification of common defaults for a set of mappings so that they do not have to be repeated in each expression.

The most common default controls element processing. This default specifies whether an element’s content or name is automatically extracted for elements resolved in an XPath expression. Thus, explicit `text` or `name` function qualification is not needed. Other defaults include the ability to declare a default namespace for both source XML documents and generated OWL entities. Prefix-to-namespace mappings can also be specified. In addition, directives control the handling of missing values in documents. In the default case, if an XPath expression evaluates to an empty value, the clause containing it is skipped. However, in some cases, users may wish to generate warnings or terminate the mapping when values are missing.

Directives are also provided to deal with references to OWL entities. For example, a directive can be set to indicate that an error should be thrown if a name refers to an existing entity in the target ontology. There is also a directive to indicate that an error should be thrown if the name does not refer to an existing entity. A related option deals with potential ambiguity introduced by annotation value references. It can be set to produce an error if more than one existing OWL entity could be named by the value.

Our language provides an option specification clause for each option type. The general form of this clause is a keyword followed by a value. For example, the default name encoding for all mappings can be written:

```xml
mm:DefaultNameEncoding = rdfs:label
```

Our online documentation has a full list of options [22].

**ADVANCED LANGUAGE FEATURES**

The language features outlined above support basic mappings of XML documents to OWL. Additional features are required for documents with unstructured text and missing elements. There is also a need for fine-grained control of the names and namespaces of created OWL entities.

**OWL Entity Name Encoding and Resolution**

Users can employ a variety of name-encoding and resolution strategies when they are creating or resolving OWL entities. The primary strategies are to use direct URI-based names (equivalent to using `rdf:about` or `rdf:ID` clauses in an RDF serialization of OWL) or `rdfs:label` annotation values. With `rdf:ID` encoding, an OWL entity generated from a reference is assigned its `rdf:ID` from the referenced content. If the content does not represent a fully qualified URI or a prefixed name, it is appended to a URI representing the namespace of the active ontology. Clearly, when using `rdf:ID` encoding, the content must represent a valid identifier—spaces are not allowed, for example. With `rdfs:label` encoding, the generated OWL entity is given an automatically generated (and non-meaningful) URI and its `rdfs:label` annotation value is set to the content specified by the XPath expression. A third encoding type is provided to support the case where the actual contents of the element are to be ignored. In this case, the generated OWL entity is again given an automatically generated URI, but its label is not assigned. The elements location in the XML document is used to track it during processing.

The default naming encoding uses the `rdfs:label` annotation property. As discussed, this default may also be changed globally. A name encoding clause explicitly specifies a desired encoding. As with entity type specifications, the clause is enclosed by parentheses after the XPath expression. The keywords to specify the encoding types are `rdf:about`, `rdf:ID`, `rdfs:label`, and `mm:UseLocation`. The following is a specification of `rdf:ID` encoding for the previous book example using this clause:

```xml
Class: @/store/book/name(rdf:ID) SubClassOf: Book
```

As mentioned, the default behavior is to use the text specified by the XPath expression when encoding a name. However, the text can first be processed with an optional value specification clause. This clause is indicated by the `=` character after the encoding specification keyword, and is followed by either a single value specification or a comma-separated list of value specifications in parentheses. Value specifications are a quoted string, a reference, or a function.

XMLMaster includes a predefined set of functions for manipulating text. These include `mm:prepend`, `mm:append`, `mm:trim`, `mm:replace`, and `mm:replaceAll`. These functions take zero or more arguments and return a value. Arguments may be quoted strings, references, or functions. For example, the following is an expression that extends the earlier book class declaration to specify `rdfs:label` name encoding. It specifies that the extracted name should be preceded by the underscore character:

```xml
Class: @/store/book/name(rdfs:label-mm:prepend("_")) SubClassOf: Book
```

A similar declaration that uses the `mm:trim` function to strip leading and trailing spaces can be written:

```xml
Class: @/store/book/name(mm:trim) SubClassOf: Book
```

Here, the content specified by the XPath expression is the implied argument to the `trim` function. It is processed by the function and then assigned to the class’s label. The
The prefix earlier expressions should use the namespace identified by surname @../author(mm:capturing("\S+\s\(\S+\))

be used in any position in a value specification clause. The strings are to be extracted. In some cases, more than one clause specify capturing groups and indicate that matched parentheses around sub-expressions in a regular expression are extracted in the order that they are matched and are appended to each other.

A more complex variant to convert comma-separated floating point numbers to dot-specified is:

Individual: @/store/book/name(Book)
Facts:
  price @../price(mm:ProcessIfEmptyLocation)

The mm:replace method would also work here:

Individual: @/store/book/name(Book)
Facts: price @../price(mm:replace("\",\"\"))

The syntax of capturing expressions follows that supported by the Java Pattern class. It provides quite a degree of flexibility when processing semi-structured text. Obviously, there are limitations to this method. Completely unstructured text may require a separate pre-processing stage.

Missing Value Handling

To deal with missing values, default values can also be specified in references. A default value clause is provided to assign these values. This clause is indicated by the mm:default keyword and is followed by at least one value specification. For example, the following expression uses this clause to indicate that the value 0.0 should be used as a price if the price sub-element is missing:

Individual: @/store/book/name(Book)
Facts: price @../price(mm:default="0.0")

XMLMaster also has additional behaviors to deal with missing values. The default behavior is to skip an entire expression if it contains any references with empty content. Four keywords are supplied to modify this behavior. They indicate that when a reference resolves to empty content:

(1) an error should be thrown and the mapping process should be stopped (mm:ErrorIfEmptyLocation); (2) the expression should be skipped (mm:SkipIfEmptyLocation); (3) a warning should be generated and the reference should be skipped (mm:WarningIfEmptyLocation); and (4) expressions containing these references should be processed (mm:ProcessIfEmptyLocation). The last option allows processing of documents that contain many missing values. The option indicates that the language processor should, if possible, conservatively drop the sub-expression containing the empty reference rather than dropping the entire expression.

Consider, for example, the following expression declaring book individuals and their prices:

Individual: @/store/book/name(Book)
Facts: price @../price(mm:ProcessIfEmptyLocation)

Using the default skip behavior, a missing price element will cause the entire expression to be skipped. However, the process directive for the price property will instead drop only the sub-expression containing it if that element is empty. As a result, the expression will still declare an individual. More fine-grained empty value handling is also supported to specify different empty value handling behaviors for rdf:ID and rdfs:label values.
IMPLEMENTATION
We have developed a parser, an editor, a processor, and a debugger for the mapping language as an open source plug-in to the Protégé-OWL development environment [23].

User Interface
The user interface provides an editor for defining, managing and executing expressions. The plug-in also supports loading XML documents and previewing them. It also allows users to define expressions interactively and then execute them. Users can also control the configuration options supported by the language interactively. The plug-in also includes a persistence mechanism to save and reload mappings. Additionally, we have written a development environment that includes Java APIs for interacting with the language from software applications.

Expression Processing Engine and Debugging
The expression processing engine takes an XML document, a base OWL ontology, and a set of XMLMaster expressions as input. It then generates a target OWL ontology from these expressions. The expressions are processed in three phases. In first phase, every expression is preprocessed and content specified by references is retrieved from the XML document. This content, which will either specify an OWL entity, a data type name, or a data value, is substituted for each reference in a mapping expression to generate one or more Manchester Syntax expressions.

At the end of phase one, a summary display allows users to preview the expressions that were generated. This display allows users to see the final entity names expanded within their enclosing expressions. It also indicates how each reference was resolved within an expression. For example, a reference may resolve to an existing OWL entity or may name a new entity that it created on demand.

If the user is satisfied with the generated expressions, they can activate the second stage of the mapping process. This phase declares all OWL entities referenced in mapping expressions that are not already declared in the target ontology. The type specification for each reference—be it explicit or inferred—is used to generate the appropriate declaration clause. Any super class, super property, individual class membership, or data type specifications in the reference are also declared in this phase.

The third phase occurs once the referenced entities have been declared. At this point, the expressions are sent to a Manchester OWL Syntax processor. It generates an OWL ontology containing the OWL axioms specified by the expressions. The generated ontology can be saved separately to produce a new ontology, or it can be used to expand an existing one.

USE CASE
The Annotation and Image Markup project (AIM) recently developed an information model that describes the semantic contents of radiological images [24]. AIM defines an XML-encoded information model that describes anatomic structures and visual observations in the images. Information about image annotations is recorded in AIM’s information model, which is described using XML Schema, with the goal of enabling the consistent representation, storage, and transfer of the semantic meaning of image features. A variety of tools have been developed to produce image annotations in AIM format.

Our goal was to develop OWL- and SWRL-based reasoning methods for automated calculation and classification of tumor response from image annotations. To perform this reasoning, we first had to produce an OWL equivalent of the AIM XML-based information model. This model represents all the concepts in the AIM XML Schema, and it can be used to store OWL instances of AIM annotations. We then required a mechanism to transform annotations from AIM XML document instances to instances in the OWL model.

Our initial approach used a tool developed by the authors [2]. Unfortunately, this tool required a cumbersome mapping process that involved interactive specification of mappings to generate an OWL ontology from an XML Schema. It then required manual generation of SWRL rules to transform instances to OWL. These rules were difficult for non-specialists to write and were cumbersome to maintain and debug for greater than a small number of mappings. Using XMLMaster, we defined an equivalent mapping process.

Transforming AIM Schema to OWL
We first defined a set of mappings to take an AIM XML Schema and generate an OWL ontology containing equivalent definitions of the entities it describes. An XML Schema document is represented in XML so can be directly processed by our language. Figure 2 shows a portion of the AIM schema.

```
<xs:schema targetNamespace="gme://caCORE...">
  ...
  <xs:simpleType name="CalculationResultIdentifier">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Scalar"/>
      <xs:enumeration value="Vector"/>
    </xs:restriction>
  </xs:simpleType>
  ...
  <xs:complexType name="Annotation"...>
    ...
    <xs:attribute name="id" type="xs:integer">
    <xs:attribute name="version" type="xs:string"/>
  </xs:complexType>
  ...
  <xs:complexType name="ImageAnnotation">
    <xs:complexContent>
      <xs:restriction base="Annotation">
        <xs:attribute name="version" type="xs:string"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:complexType>
</xs:schema>
```

Figure 2. Portion of AIM XML Schema.
Figure 3 shows a selection of the XMLMaster expressions. The first expression generates an OWL class declaration for simple schema types. The second expression deals with extended complex types and generates classes and subclass relationships for them. The third expression declares OWL enumerated classes from an XML Schema value enumeration. Each enumeration value will become an individual in the enumerated class. The fourth expression declares OWL object properties from element definitions. The final example defines OWL data property declarations for attribute of type string. Similar expressions are written for other types.

1. Class: @//xs:simpleType/@name
2. Class: @//xs:complexType/@name
   SubClassOf: @./@xs:extension/@base
3. Class: @//xs:simpleType/@name
   EquivalentTo: [ @./@xs:enumeration/@value ]
4. ObjectProperty: @/xs:element/@name
5. DataProperty:
   @/xs:attribute[@type="xs:string"]/@name
   Range: xsd:string

Figure 3. Selection of AIM XML Schema to OWL mappings.

As can be seen from these examples, mappings can be incrementally constructed. The basic class declaration in example 1, for example, is supplemented by the more complex declaration in example 3. The key is that references to the same element or attribute in different mapping expressions resolve to the same OWL entity declared from them. Expressions can thus be independently evaluated in any order. As a further consequence, individual mapping expression can be relatively concise. Users can also decide how much of the XML Schema description they want to capture in OWL and add mappings later if more expressivity is desired.

Like schema mappings, instance mappings can be constructed incrementally. For example, the first mapping declares an image annotation individual and associates it with anatomic entity individuals declared from its AnatomicEntity sub-elements. The details of each anatomic entity are defined in the second mapping. These elements resolve to the same declared individual in both expressions and can thus be linked in this way. As the examples also show, an associated XML Schema is not required when defining instance mappings.

DISCUSSION

Previous approaches for mapping information from XML to OWL suffered from a variety of limitations. Although recent approaches have addressed shortcomings in selective extraction of information from XML documents, they have weak OWL ontology generation capabilities. Typically, they support a small number of predefined mappings, which limits the expressivity of the resulting ontologies. Other weaknesses include requiring an associated XML Schema in order to process a document, and limited support for dealing with unstructured element content.

To address these limitations, we developed XMLMaster, a custom domain specific language. It combines the Manchester OWL Syntax and XPath, two well-known technologies and standards from the semantic and syntactic worlds. The resulting language supports flexible extraction of information from XML documents and a compact, user-friendly, OWL-centric approach for generating OWL ontologies. It can deal with XML documents that may not have an associated XML Schema description. XMLMaster can be used to generate new ontologies from scratch or to extend existing ones with new mapped content. These ontologies can contain individual and class axioms. Our experience with a similar language for mapping from spreadsheets to OWL [2] leads us to believe that users who are familiar with OWL and XML will learn it easily.

Future work includes the development of a visual tool to allow non specialist users to define mappings from XML to OWL. Using a sample set of XMLMaster mappings defined for a variety of XML data sources, we plan to identify common transformations. We will use these transforma-
tions to develop a set of template mappings. These templates will be used to drive the design of a GUI-based tool to interactively define mappings. Other extensions include supporting XQuery [26] to provide even more expressivity in processing XML documents. Similarly, adding support for user-defined functions will enhance text processing options.

The software will be released shortly as an open source plug-in to the Protégé-OWL ontology development environment [23].

ACKNOWLEDGMENTS
This research was supported in part by the N.L.M. under grants LM007885 and LM009607 and by STTR Award #0750543 from the N.S.F. We thank Valerie Natale for her valuable editorial comments.

REFERENCES
[22] XMLMaster: http://protege.cim3.net/cgi-bin/wiki.pl?XMLMaster
[23] Protégé: http://protege.stanford.edu