A Generalized Framework for an Ontology-Based Data-Extraction System

Alan Wessman,[1] Stephen W. Liddle,[2] David W. Embley[1]

[1]Department of Computer Science
[2]Rollins Center for eBusiness
Brigham Young University
Provo, UT 84602, USA
alanyst@gmail.com, liddle@byu.edu, embley@cs.byu.edu

Abstract: Extraction of information from semi-structured or unstructured documents, such as web pages, is a useful yet complex task. Ontologies can achieve a high degree of accuracy in data extraction while maintaining resiliency in the face of document changes. Ontologies do not, however, diminish the complexity of a data-extraction system. As research in the field progresses, the need for a modular data-extraction system that decouples the associated processes continues to grow.

In this paper we report on the implementation of such a system. The nature of our framework allows new algorithms and ideas to be incorporated into a data extraction system without requiring wholesale rewrites of a large part of the system's source code. It allows researchers to focus their attention on parts of the system relevant to their research without having to worry about introducing incompatibilities with the remaining components. We demonstrate the value of the framework by providing an implementation that exhibits appropriate characteristics.

1 Introduction

Making sense of the vast amount of information available on the World Wide Web has become an increasingly important and lucrative endeavor. The success of Web search companies such as Google demonstrates the vitality of this industry. Yet Web search has much still to deliver. While traditional search engines can locate and retrieve documents of interest, they lack the capacity to make sense of the information those documents contain.

Data extraction addresses many of the problems associated with typical web searches based on standard information retrieval techniques. Data extraction is the activity of locating values of interest within electronic textual documents, and mapping those values to a target conceptual schema [La02]. The conceptual schema may be as simple as slots in a template (a *wrapper*) used to locate relevant data within a web page, or it may be as complex as a large domain ontology that defines hierarchies of concepts and intricate relationships between those concepts. The conceptual schema is usually linked to a storage structure such as an XML file or a physical database model to permit users to query the extracted data. In this way, the meaning of a document is detected, captured, and made available to user queries or independent software programs.

Much of the research in data extraction has aimed at developing more accurate wrappers while requiring less human intervention in the process (e.g. [Ha97] [KWD97] [AK97] [CMM01] [La02]). The primary drawback of wrappers, whether they are generated manually or semiautomatically, is that they depend on the particular syntax of the document markup to detect boundaries between relevant and irrelevant data. The main implication is that when a site's markup changes (which happens often on the web), the corresponding wrappers often break. Furthermore, since different sites in the same domain generally use distinct markup, customized wrappers are required for each site. Wrapper management can be quite complex and problematic.

Other data-extraction researchers have focused on the use of richer and more formal conceptual schemas (ontologies) to improve accuracy in data extraction (e.g. [Em99] [DMR02] [Eng02] [SFM03]). Because an ontology describes a subject domain rather than a document, ontology-based data-extraction systems are resilient to changes in how source documents are formatted, and they can handle documents from various sources without impairing the accuracy of the extraction. This contrasts with wrappers, which merely describe the locations of data values in a particular set of similarly-formatted documents. Ontology-based extractors compare unfavorably to wrappers in one important way: considerably more human effort is required up front to construct a high-quality extraction ontology, while wrappers can be constructed more easily, even to the point of automation of much of the process.

Our data-extraction system is called BYU-Ontos, or simply Ontos [Em99]. It is an ontology-based engine whose present version accepts multi-record HTML documents, determines record boundaries within those documents, and extracts the data from each record. It generates SQL DDL statements for the model structure and stores the extracted information as DML statements. This facilitates querying of the results but also removes certain metadata (such as the original location of the data within the source document) attached to the data during the extraction process. This metadata may be important for learning algorithms or for further research.

The system is based on the Object-Oriented Systems Model (OSM) [EKW92]. OSM is a set-theoretic modeling approach founded upon first-order predicate logic, which enables it to express modeled concepts and constraints in terms of sets and relations. An OSM instance can serve as an ontology: concepts are represented by *object sets*, which group values (*objects*) that have similar characteristics; and connections between concepts are expressed via *relationship sets*, which group object tuples (*relationships*) that share common structure. Generalization-specialization is a special type of relation that expresses "is-a" relationships between object sets. In Ontos, OSM is expressed by OSML (OSM Language) [LEW00].

For use in extraction, OSM has been augmented by *data frames*, which describe characteristics of objects, similar to an abstract data type [Emb80]. Data frames are attached to object sets, and provide a means to recognize lexical values that correspond to objects in the ontology.

OSM and data frames together provide the modeling power necessary for effective ontology-based data extraction. Experiments on small- to medium-size ontologies (two to twenty object sets) have demonstrated that Ontos exhibits a rather high degree of accuracy with the resiliency and robustness to maintain that accuracy even when the structure of the source records varies considerably [Em99].

OntologyEditor is a predominantly WYSIWYG tool for editing OSM-based data-extraction ontologies which we presented an earlier ISTA Conference [LHE03]. It performs no true data-extraction work itself, but provides a way for the user to preview the effect of the value recognition rules defined in the data frames of the ontology on a source document. As part of the preparation for this paper, the authors spent a significant amount of effort in refactoring OntologyEditor to accommodate new extraction ontology capabilities and to support a new ontology storage format.

Over the last several years, we have performed considerable amounts of successful data-extraction research based on the tools just described. But our experiences have served to exercise these tools in ways that were difficult to predict when they were first developed. Much of the research conducted after the development of Ontos has, as a side effect, shown the system to be inflexible when certain fundamental operational parameters are changed. For instance, the system expects multiple-record documents as input, and thus performs poorly on single-record or tabular document structures. Research conducted on such document structures has required parallel versions of Ontos to be developed, or significant portions of Ontos code to be extracted and customized for newly developed systems, in order to perform effective experiments. Furthermore, Ontos is not easily adapted when new features are added to the ontology specification.

Although there are many possible algorithms for extracting data based on an ontology, and it is not yet clear which are best under which circumstances, Ontos is heavily tailored to a single extraction algorithm and cannot readily be modified to execute a substantially different one. For example, one of the authors devised an extraction algorithm that involved inferring hidden Markov models from the ontology and using those to map values to concepts. The new algorithm could not be readily tied back into Ontos because the system was too strongly coupled with its original extraction algorithm, and the project was eventually abandoned.

This inflexibility makes it difficult to evaluate different ideas or approaches for data extraction. A higher number of hard-coded assumptions about operational parameters makes it more likely that reimplementation of the system is required when these assumptions are contradicted. In contrast, reducing the number of *a priori* assumptions encoded into the system should make it easier to experiment with or improve certain aspects of the process while keeping the rest of the system constant. This allows us to make scientifically rigorous claims about the performance of the system and the impact of the changes made.

Examples of such operational parameters include the chosen ontology language (e.g. OSML in our case) and supported document types (e.g. multiple-record HTML documents). We would like to experiment with ontology languages that conform to W3C standards, and we would like to be able to experiment with recognition techniques that can leverage the DOM tree structure of a document (which we currently ignore), or take as input other document types (e.g. PDF and XML). At this point, expanding the power of Ontos requires building a more general framework with fewer *a priori* assumptions.

To address the need for high flexibility, modularity, and extensibility in a data-extraction system, we propose a new framework for data extraction. We assert that such a framework will provide the support for customization and experimentation needed to efficiently conduct continued data-extraction research. Frameworks and design patterns constitute the heart of our approach.

Frameworks provide the means to address the macro-level problems of a system while allowing details to be implemented or varied after the initial design, so that components may be interchanged and system performance may be tuned. A framework thus provides a skeletal implementation for a general problem, and establishes parameters for a set of solutions based on that partial implementation. Solution providers can concentrate on satisfying those requirements that are unique to a particular approach, using existing mechanisms provided by the framework to handle the rest of the system.

The theory of design patterns has contributed to the rise of robust software frameworks. *Design patterns* are collections of templates and principles for designing code that solve commonly-encountered software design scenarios in an abstract manner. An example of a design pattern is the **`Factory Method`** [Ga95], which provides a model for instantiating new objects whose actual classes are known only at runtime (dynamic binding) rather than at design time (static binding). These design patterns are often encountered in frameworks due to their highly generalized architecture, which enables frameworks to defer non-essential design decisions to their implementations.

A generalized framework of interfaces and abstract classes written in an object-oriented language (such as Java) can decouple each operational module from the rest of the engine to produce a highly flexible and configurable data-extraction system. The framework can be sufficiently flexible both to allow the current heuristics to be re-implemented under the framework and to enable new heuristics and other modules to be created and incorporated into the Ontos system without requiring significant rewrites of unrelated code.

The contributions of this paper include (1) an ontology-based data-extraction framework written in Java, (2) a means for explicitly modeling extraction plans, (3) an extraction-ontology schema, OSMX, written in XML Schema, and (4) a reimplementation of the legacy version of Ontos within the new framework. The new Ontos serves as a reference for future implementations of the framework. It retains support for all essential features of the old Ontos system but also adds many new capabilities.

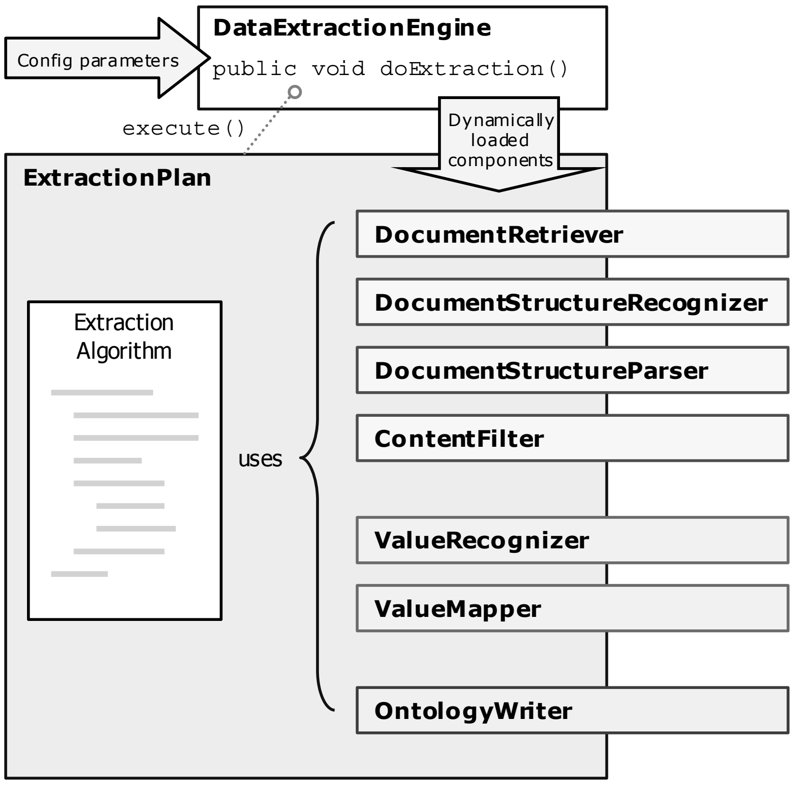
The remainder of this paper is organized as follows. Section 2 describes our new data-extraction framework at the architectural level. Section 3 briefly highlights the main points of OSMX. In Section 4 we give a description of the reference implementation of the framework. Section 5 describes the results of validating the reference implementation, and we conclude in Section 6.

2 A framework for performing ontology-based data extraction

This section explains the design and construction of the data-extraction framework. We describe each interface and abstract class, explain the essential contracts they define, and discuss how they might be implemented.

A graphical overview of the framework appears in Figure 1. Control begins at the engine at the top of the diagram, and passes to the extraction plan. The narrow boxes running down the right side represent modules involved in the extraction process.

The `DataExtractionEngine` abstract class represents the overall data-extraction system. Its primary purpose is to accept operational parameters, locate and load the appropriate modules, perform any additional initialization steps, and initiate the



extraction process. It then performs cleanup as necessary and terminates. `DataExtractionEngine` follows the **`Facade`** design pattern [Ga95]. A facade is a simplified interface to a complex system; in this case, `DataExtractionEngine` defines simple methods for initializing the system and executing the extraction process. This pattern also allows clients such as the OntologyEditor to interact with the system at a very high level without being coupled to specific components of the system.

The `Ontology` interface describes an in-memory representation of the extraction ontology. The interface is designed to be independent of the language the ontology is written in; thus we eliminate from the framework any assumption that the extraction ontology is built with OSM, DAML, OWL, or any other specific ontology language. Without knowing the features of the ontology language, how can we define the capabilities of the `Ontology` interface? The answer is that we defer such details to implementation classes, and use `Ontology` as primarily a marker interface for ensuring that parameters and member variables representing the ontology are of the correct data type.

`Document` is an interface that represents cohesive units of unstructured or semi-structured information that may be interspersed with data that is not of interest. We narrow the scope of our problem by choosing to focus only on textual data extraction.

We do not attempt to decode images or other non-text sources of information. Taking our cue from XML's successful general technique, we represent a document as a sequence of (possibly zero-length) strings interleaved with sub-documents.

A sub-document is itself a `Document`, and thus may contain other sub-documents. This definition implies a tree structure with one `Document` at the root, and we formally define the term *sub-document* to indicate any non-root node in a `Document` tree. We note that `Document` is an instance of the **`Composite`** design pattern [Ga95], which allows objects to be composed into treelike hierarchies while providing a uniform interface both for interior nodes and leaves.

The `ExtractionPlan` abstract class represents the overall algorithm for carrying out the extraction activity. In this way, it is similar to an SQL extraction plan in a relational database management system. `ExtractionPlan` eliminates assumptions about the order of operations for a data-extraction system: one implementation might proceed in a linear fashion, from retrieval straight through to mapping and output, while another implementation might discover some new information (such as a relevant URL) during extraction and immediately branch into a recursive execution based on that data. In this sense, `ExtractionPlan` adheres to the **`Strategy`** design pattern [Ga95], which encapsulates an algorithmic process and standardizes its invocation so that different algorithms may be interchanged.

Before actual extraction work can occur, the extraction engine must provide a way for documents to be located and prepared for extraction. The following interfaces define important aspects of this process: `DocumentRetriever`, `DocumentStructureParser`, `DocumentStructureRecognizer`, and `ContentFilter`.

The `DocumentRetriever` interface defines the module responsible for supplying the extraction engine with source documents. It may, for instance, represent a view of a local file system, or it may wrap the functionality of a Web crawler or even a Web search engine such as Google. The module accepts a URI as input and produces a set of `Documents`. The `DocumentRetriever` will usually perform its functions at the beginning of the extraction process. However, a sophisticated implementation might use it to retrieve additional documents using extracted URLs from previously retrieved web pages, in an intelligent spidering process.

`DocumentStructureRecognizer` is an optional component of the system. Its role is to analyze a document to determine which available `DocumentStructureParser` is best suited to decompose the document. This is useful when the extraction engine is operating on a mixture of source document types.

We may be interested in breaking up a document into sub-documents in order to make extraction easier or more accurate. For example, dividing a multi-record document into sub-documents, each constituting an individual record, allows us to process one record at a time without having to worry about missing a record boundary and extracting values from an adjacent record. We define the `DocumentStructureParser` interface as a solution to this problem. It is an optional component of the system; left unspecified, the input document will be treated as an indivisible unit.

Most documents contain a combination of meaningful data and formatting information. An HTML document contains many tags that indicate how the document may be represented in a browser, but these tags usually do not lend additional meaning to the content. We find it convenient therefore to remove text that is exclusively for formatting purposes from the document before proceeding with extraction. We define the `ContentFilter` interface to support this requirement. This is another optional component of a data-extraction system, as going without a filter simply means extracting from the document's original content.

Filtering out the formatting data from a document is a process that demands a substantial degree of flexibility. For example, we may at times wish to strip all HTML tags from a document, leaving behind only the text content found between those tags. On the other hand, we might desire to preserve quasi-meaningful pieces of information found within certain HTML tags, such as the contents of the ALT attribute of an IMG tag. Consider the example of a series of icons used to depict amenities provided at a campground. The icons express information that we wish to extract into an ontology for campgrounds, and by extracting from the ALT attribute of those IMG tags we hope to glean the desired knowledge without having to attempt to decode the graphics themselves. By providing a flexible means to implement various filters, we allow implementers to target those portions of the document that they deem most likely to yield useful data, while discarding data that simply gets in the way.

With the document retrieved, parsed, and filtered, we can perform the actual task of extracting values from the document and mapping them to the ontology. Two interfaces divide this work: `ValueRecognizer`, and `ValueMapper`.

`ValueRecognizer` occupies a key role in a data-extraction system, and is a required component of the framework. Its responsibility is to apply the value-recognition rules associated with the extraction ontology to the input document, producing a set of candidate extractions. We say "candidate" because it does not resolve conflicts about which matched values belong to which parts of the ontology; it merely identifies from the document the values we can find that *might* belong in the final data instance.

Locating and interpreting the extraction rules is a process that can differ according to the ontology language used, so at the framework level we do not restrict how this is done. Nor do we specify how the rules are to be applied to the document or how the matching results are to be stored. Our reference implementation associates match values with the ontology through composition; but other methods of handling the match results (such as annotations inline with the document content) may be equally valid.

The `ValueRecognizer` also bears the responsibility of maintaining location information for each candidate value. This provides a traceable path back to the document content and also can supply useful data for the algorithms that resolve match conflicts and create mappings from candidate values to elements of the ontology. We do not specify a format for the location data, but <*start position, end position*> or <*start position, length*> pairs generally make the most sense for character-based text sources.

Perhaps the most important and difficult part of the extraction system is the process that takes candidate value matches and uses them to build a data instance consistent with the constraints specified by the ontology. Since this process maps candidate values to elements of the ontology, we name this interface `ValueMapper`. There are four tasks that a `ValueMapper` must perform to transform candidate value matches into a data instance: (1) resolve conflicting claims that different elements of the ontology make upon the same matched value, (2) transform lexical values into objects (instances of concepts defined in the ontology), (3) infer the existence of objects that have no direct lexical representation in the text, and (4) infer relationships between objects. The `ValueMapper`'s work yields a data instance: a collection of objects and relationships between those objects.

When the `ValueMapper` process has finished, the `OntologyWriter` abstract class provides a standard way for an implementation to export the objects and relationships to a useful storage format via the Java `Writer` interface. The particular storage format is up to the implementation to define.

3 Constructing extraction ontologies with OSMX

Fundamental to an implementation of the data-extraction framework is the language used to define the ontologies involved in the extraction process. The ontology language establishes the capability of the ontology to represent a given subject domain. The previous ontology description language, OSML, is adequate for representing extraction ontologies, but because it is a proprietary language with a highly ambiguous grammar that is difficult to parse correctly, interchange with other tools has always been difficult. Our new XML-based language, OSMX, is more portable and much easier to integrate with modern Java implementation environments.

The official OSMX specification is defined by an XML Schema document (http://www.deg.byu.edu/xml/osmx.xsd). This document defines the standards for creating a well-formed and valid OSMX document. We use the Java Architecture for XML Binding (JAXB) technology to generate, from the OSMX specification, Java classes and interfaces that represent OSMX constructs. Modifying the OSMX definition is generally a straightforward process: we adjust the definitions in the XML Schema document, and then execute a JAXB program that rebuilds the classes and interfaces automatically. We use these classes and interfaces to access and manipulate portions of an ontology from within the data-extraction framework reference implementation. This standards-based approach is far more convenient and modifiable than the proprietary solution had been.

In the process of designing OSMX, we augmented the prior ontology language in several important ways. First, we enhanced data frames by allowing the specification of an internal representation for a lexical object's value. This allows us to interpret the value as a particular data type, such as `String` or `Double`. We may also designate a *canonicalization method* that converts extracted values into a canonical format compatible with the internal representation. The prior framework only allowed limited regular-expression based string substitutions for manipulating extracted values. The new approach lets us integrate complex methods written in Java to manage canonicalization.

The most basic element of an extraction rule for a data frame is a matching expression. This is an augmented form of a Perl-5 compatible regular expression. The level of regular expression support is defined by the Java regular expression package `java.util.regex`. We augment regular expressions by allowing the rule designer to embed macro and lexicon references within the expression itself.

A macro defines a simple string substitution rule. For example, we might define a macro named "DayOfWeek", which can be used in a regular expression as follows:

```
((from|on|starting|beginning) {DayOfWeek})
```

When this expression is applied to a text, the macro reference first expands into the substitution value, and the resulting regular expression is matched against the text.

Macro references are fully recursive in our reference implementation, but cyclical references are forced to terminate at the first recurrence of a previously expanded macro, so that infinite recursion does not occur. Lexicon substitution is similar.

OSML data frames combined certain aspects of the concepts of constant and context. OSMX data frames fully separate the two. Value phrases in OSMX have one set of regular expressions to describe constants to be extracted, and another set of expressions to describe contextual clues (characters that must appear or that may appear near the constant to be extracted). OSMX also gives better control over keyword phrases, allowing them to be associated with individual value phrases or the entire data frame.

4 OntosEngine: an OSMX-based implementation

We have created a working data-extraction system in Java that adheres to the architecture defined by the data-extraction framework. We intend for this new system to serve as a point of reference for future implementations or enhancements, so we refer to the system as the *reference implementation* of the framework. This term should not be confused with the term *framework definition*, which is the set of classes, interfaces, and other architectural components that establish the parameters of the framework itself, without respect to any particular implementation.

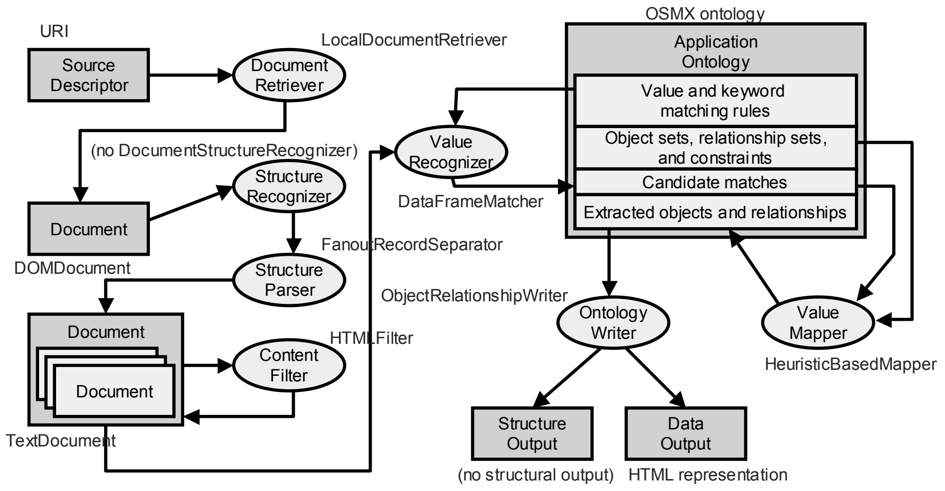


Figure 2. Architecture of the new Ontos system under the framework.

We start with the highest level of functionality, the `DataExtractionEngine`. Figure 2 shows a high-level flow diagram of the reference implementation. We extend this abstract class, creating the `OntosEngine` class, in order to implement the `doExtraction()` method. We also implement initialization code that allows command-line parameters or entries in a configuration file to specify the implementation modules' subclasses at runtime.

After loading all necessary modules, the `doExtraction()` method instantiates an `OntosExtractionPlan` and invokes its `execute()` method. As its name suggests, `OntosExtractionPlan` extends the `ExtractionPlan` abstract class. For our implementation of `ExtractionPlan`, we define a straightforward algorithm for performing the extraction. Iterating over each `Document` returned by the `DocumentRetriever` module, we obtain a document tree from the available `DocumentStructureParser` or retain the original `Document` if no parser module was specified in the initialization phase. For each `Document` in the document tree, we remove markup with the available `ContentFilter` and then perform value recognition and mapping with the appropriate components. Finally, we write the results to a human-readable HTML file. Table 1 lists the correspondences between framework classes and their respective reifications in the reference implementation.

Framework Class	**Reference-Implementation Class**
`DataExtractionEngine`	`OntosEngine`
`ExtractionPlan`	`OntosExtractionPlan`
`DocumentRetriever`	`LocalDocumentRetriever`
`DocumentStructureRecognizer`	not needed in reference implementation
`DocumentStructureParser`	`FanoutRecordSeparator`
`Document`	`DOMDocument, TextDocument`
`ContentFilter`	`HTMLFilter`
`ValueRecognizer`	`DataFrameMatcher`
`ValueMapper`	`HeuristicBasedMapper`
`OntologyWriter`	`ObjectRelationshipWriter`

Table 1. Correspondences between framework and reference-implementation classes.

`LocalDocumentRetriever` is a `DocumentRetriever` that locates and retrieves documents from a specified directory in the local file system. Our reference implementation does not attempt any retrieval from online sources such as Web search engines, but since the built-in Java API makes retrieving URI's straightforward, this would not be difficult to implement.

Because our reference implementation mimics legacy Ontos in its focus on extraction from single- and multiple-record documents, we only depend on one `DocumentStructureParser` implementation, so no `DocumentStructureRecognizer` is necessary.

Our `DocumentStructureParser` reference implementation, `FanoutRecordSeparator`, anticipates that the document has a shallow multi-record structure, and divides the document into sub-documents accordingly. It is possible for the `FanoutRecordSeparator` to return a single-node document tree, in which case the engine treats the tree as a single-record document.

Our present focus for extraction is on HTML documents, which contain a considerable amount of markup extraneous to the data-extraction process. We therefore provide an `HTMLFilter` implementation of `ContentFilter`. It essentially serves as an **`Adapter`** [Ga95] for a preexisting utility that strips out unwanted HTML. This exemplifies how we can incorporate existing code into the framework in addition to writing original implementations.

In total, these modules (`LocalDocumentRetriever`, `FanoutRecordSeparator`, and `HTMLFilter`) prepare a document for value recognition and mapping to the ontology. At this point in processing, the framework requires an ontology, which for testing and comparison purposes is the obituary ontology introduced in [Em99].

Our reference implementation of `ValueRecognizer` is called `DataFrameMatcher`, which uses OSMX with its data frames as the ontology language. This module locates the recognition rules specified by each data frame and applies them to the input text. The matcher identifies all substrings in the text that match the recognition rules and, for each such substring, constructs a `MatchedText` object that records the matched substring, its starting and ending character positions (in the context of the filtered document), and the URI of the `Document` from which it was extracted. A `MatchedText` object also maintains a status attribute, indicating whether the match has been accepted, rejected, or not yet processed by the `ValueMapper`.

We provide an `OntologyWriter` subclass called `ObjectRelationshipWriter` that produces a human-readable hierarchical list of objects and relationships in each data instance stored with the input ontology. The output format is HTML, which suits our present purposes since we intend humans and not computers to process the results in our reference implementation.

An aspect of the reference implementation that deserves further explanation is the `ValueMapper`, which is the most complex part of the extraction system (and a primary reason for creating this new framework, due to the difficulty of extending the legacy version of Ontos). The module's task is to infer a set of mappings between extracted values (the `MatchedText` objects) and object sets in the ontology. Each mapping is realized as a lexical object. `ValueMapper` must also infer the existence of nonlexical objects and relationships between objects.

Not every `MatchedText` object will necessarily become a lexical object; nor will the `ValueMapper` implementation infer an object or relationship wherever an object set or relationship set exists. In fact, a key problem for the `ValueMapper` to solve is how to decide when to generate an object or relationship. Our implementation employs various heuristics to solve this problem (see [Em99]). Full details are found in [We05].

5 Evaluation of OntosEngine

We have claimed that our framework provides a flexible approach to building a data-extraction system. While such a claim is not fully testable within the scope of this paper, we can assess whether the framework is sufficient for supporting data extraction. Past experience with other frameworks gives us a high degree of confidence in our approach. Thus, we have provided a reference implementation of the framework to demonstrate that it is sufficiently complete. We must now show the framework does not negatively impact extraction results available from the legacy Ontos system. And indeed, our reference implementation of the framework does achieve comparable, if not better, results.

We demonstrate this by performing extraction using the same set of documents and the same ontology on both our new implementation and the legacy system, then comparing the results. The subject domain for this experiment is obituary listings such as those commonly found in local newspapers. Our corpus is a set of recent obituaries from two different newspapers—the *Salt Lake Tribune* and the *Arizona Daily Star*—containing a total of 25 individual obituaries. The ontology used for both experiments is the same, with the exception that for our new implementation we represent the ontology with OSMX, and for the legacy system we represent it with the OSML language. The representational differences between these languages do not influence extraction accuracy.

In our experiment, we measure precision and recall with respect to the lexical object sets from the ontology. Our basis for comparison is a set of manually derived extraction results based on the judgment of the human experimenter. We score correct (exact and partial) matches, false positives (incorrect mappings), and false negatives (missed mappings). Exact matches are defined as identical values appearing in the same mapping for both automatic and manual results; for partial matches, the automatically extracted value may be a substring of the manual result, or it could subsume or overlap the manual result. For example, a partial match might contain only "July 21," where the human would extract "July 21, 1993," or vice versa. If the values are entirely different, such as "August 23, 1979" and "July 21, 1993," we consider them to be different mappings and record a false positive. We record as a false negative the failure to extract any sort of mapping (exact, partial, or incorrect) to correspond with one extracted manually.

Our results appear in Table 2. The first column lists the object sets of the ontology. Other columns list the total correct matches (with partial matches in parentheses), the total false positives, total false negatives, precision, and recall for each of the two newspapers from which the input obituaries came. The rightmost columns give the overall precision and recall numbers. Each row of the table gives the results for the legacy Ontos system (top numbers) and the new Ontos system (bottom numbers, in boldface).

	Salt Lake Tribune 9-Oct-2004					*Arizona Daily Star 9-Oct-2004*					*Overall*	
Object Set	*C(P)*	*FP*	*FN*	*R%*	*P%*	*C(P)*	*FP*	*FN*	*R%*	*P%*	*R%*	*P%*
Deceased	15	2	0	100	88	9	0	1	90	100	96	92
Person	**15**	**2**	**0**	**100**	**88**	**10**	**0**	**0**	**100**	**100**	**100**	**93**
Deceased	8 (7)	0	0	100	100	5 (4)	0	0	100	100	100	100
Name	**8 (7)**	**0**	**0**	**100**	**100**	**6 (4)**	**0**	**0**	**100**	**100**	**100**	**100**
Age	5	7	0	100	42	3	6	6	33	33	57	38
	5	**4**	**0**	**100**	**56**	**4**	**5**	**6**	**40**	**44**	**60**	**50**
Death	12	3	3	80	80	8	1	1	89	89	83	83
Date	**11**	**4**	**4**	**73**	**73**	**9**	**1**	**1**	**90**	**90**	**80**	**80**
Birth	13	2	2	87	87	2	5	1	67	29	83	68
Date	**11**	**4**	**4**	**73**	**73**	**2**	**6**	**1**	**67**	**25**	**72**	**57**
Funeral	11	4	0	100	73	8	2	0	100	80	100	76
	11	**4**	**0**	**100**	**73**	**8**	**2**	**0**	**100**	**80**	**100**	**76**
Funeral	8	3	3	73	73	4	1	4	50	80	63	75
Date	**7**	**4**	**4**	**64**	**64**	**6**	**0**	**2**	**75**	**100**	**68**	**76**
Funeral	7	3	4	64	70	4	3	4	50	57	58	65
Time	**5**	**5**	**6**	**46**	**50**	**3**	**5**	**5**	**38**	**38**	**42**	**44**
Funeral	5 (2)	4	3	70	64	2	5	2	50	29	64	50
Address	**6 (2)**	**3**	**2**	**80**	**73**	**1**	**7**	**3**	**25**	**13**	**64**	**47**

Interment	1	12	0	100	8	0	7	0	100	0	100	5
	1	**10**	**0**	**100**	**9**	**0**	**9**	**0**	**100**	**0**	**100**	**5**
Interment	1	0	0	100	100	-	-	-	-	-	100	100
Date	**1**	**0**	**0**	**100**	**100**	-	-	-	-	-	**100**	**100**
Interment	1	0	0	100	100	-	-	-	-	-	100	100
Address	**1**	**0**	**0**	**100**	**100**	-	-	-	-	-	**100**	**100**
Viewing	10	47	0	100	18	4	18	1	80	18	93	18
	7	**5**	**3**	**70**	**58**	**5**	**2**	**0**	**100**	**71**	**80**	**63**
Viewing	1	4	0	100	20	0	0	5	0	0	17	25
Date	**0**	**6**	**1**	**0**	**0**	**0**	**1**	**5**	**0**	**0**	**0**	**0**
Beginning	5	1	5	50	83	4	0	1	80	100	60	90
Time	**4**	**2**	**6**	**40**	**67**	**4**	**1**	**1**	**80**	**80**	**53**	**73**
Ending	6	0	4	60	100	4	0	1	80	100	67	100
Time	**2**	**2**	**8**	**20**	**50**	**3**	**1**	**2**	**60**	**75**	**33**	**63**
Viewing	2	1	2	50	67	0	3	4	0	0	25	33
Address	**0**	**4**	**4**	**0**	**0**	**1**	**3**	**3**	**25**	**25**	**14**	**14**
Relative	75 (32)	196	28	77	35	58 (35)	94	45	67	50	73	41
Name	**67 (31)**	**191**	**38**	**72**	**34**	**69 (37)**	**78**	**32**	**77**	**58**	**74**	**43**

Table 2. Results of extraction of obituaries from two newspapers.
(Boldface numbers give results from the new Ontos system; normal font indicates legacy Ontos results. Columns are: C(P)=Correct (Partial); FP=False Positive; FN=False Negative; R%=Recall; P%=Precision.)

For half of the object sets, the new system performs as well as or better than the legacy system. Of the remaining nine object sets, the new system performs worse for seven, and for the other two it equals or excels the legacy system's performance in either recall or precision, but not both. Overall, the new system performs marginally better than the legacy system, with the major differences stemming from a few intentional departures by the new system from the rules followed by the legacy system. The logic in the new system is significantly cleaner than in the legacy system, and rather than rely on fortuitous quirks in the legacy system, we chose to implement a cleaner semantics and see how the two would compare. Generally, the performance between the two systems is remarkably similar, considering the very different algorithms at the heart of the systems. Where performance degrades in the new system, the code is modular enough to allow poorly performing code to be optimized or replaced without impacting the rest of the system. This bears out our claim that the framework is sufficient to support the task of data extraction at the same level as the legacy system, while providing a much more capable supporting code infrastructure.

6 Conclusion

We have proposed, designed, and provided a reference implementation for a framework for ontology-based data extraction. This framework offers improved modularity and extensibility to support further data-extraction research. We have demonstrated that the framework is sufficiently developed to support a re-implementation of BYU Ontos that preserves the quality of legacy Ontos while also using modular heuristics code.

Additionally, we have designed an XML Schema, OSMX, that provides an XML storage definition for OSM ontologies. Newly added features give OSMX greater capabilities for representing data-extraction ontologies and the instance data extracted for them. A library of JAXB-generated Java classes supports programmatic access to OSMX-compliant documents. This library allows Ontos, OntologyEditor, and future tools to exchange ontologies and data, expanding the research possibilities while minimizing the need for specialized information interchange protocols or file formats.

The products of these efforts provide a solid basis for continued research on ontology-based data extraction. Future researchers will be better able to focus on specific problems in the field while maintaining confidence that plugging their code into the existing system and comparing the results can rapidly validate their work. The value of this contribution is in future ontology-based data extraction research opportunities made possible or practical because of the framework.

Bibliography

[AK97] Ashish, N. and C. Knoblock. "Wrapper generation for semi-structured Internet sources," *SIGMOD Record*, Volume 26, Number 4, December 1997, pp. 8-15.

[CMM01] Crescenzi, V., G. Mecca, P. Merialdo. "RoadRunner: Towards automatic data extraction from large Web sites," In *Proceedings of the 27th International Conference on Very Large Data Bases*, Rome, Italy, 11-14 September 2001, pp. 109-118.

[DMR02] Davulcu, H., S. Mukherjee, I.V. Ramakrishnan. "Extraction techniques for mining services from Web sources," In *Proceedings of the IEEE International Conference on Data Mining* (ICDM), Maebashi, Japan, 9-12 December 2002, pp. 601-604.

[Em99] Embley, D.W., D.M. Campbell, Y.S. Jiang, S.W. Liddle, D.W. Lonsdale, Y.-K. Ng, R.D. Smith. "Conceptual-model-based data extraction from multiple-record Web pages," *Data & Knowledge Engineering* 31 (1999), pp. 227-251.

[EKW92] Embley, D.W., Barry D. Kurtz, Scott N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Yourdon Press, 1992.

[Emb80] Embley, D.W. "Programming with data frames for everyday data items," *AFIPS '80 Proceedings*, Anaheim, California, 19-22 May 1980, pp. 301-305.

[Eng02] Engels, R. *Del 7: CORPORUM OntoWrapper: Extraction of structured information from web based resources*. On-to-Knowledge Consortium, 2002. At http://www.ontoknowledge.org.

[Ga95] Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

[Ha97] Hammer, J., H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, V. Vassalos. "Template-based wrappers in the Tsimmis system," In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 13-15 May1997, pp. 532-535.

[KWD97] Kushmerick, N., D. Weld, R. Doorenbos. "Wrapper induction for information extraction," In *Proceedings of the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 23-29 August 1997, pp. 729-737.

[La02] Laender, A.H.F., B.A. Ribeiro-Neto, A.S. da Silva, J.S. Teixeira. “A brief survey of Web data extraction tools,” *SIGMOD Record*, Volume 31, Number 2, June 2002, pp. 84-93.

[LEW00] Liddle, S.W., D.W. Embley, S.N. Woodfield. “An active, object-oriented, model-equivalent programming language.” *Advances in Object-Oriented Data Modeling*, MIT Press, 2000, pp. 333-361.

[LHE03] Liddle, S.W., K.A. Hewett, D.W. Embley. “An Integrated Ontology Development Environment for Data Extraction,” In *Proceedings of the 2nd International Conference on Information System Technology and its Applications* (ISTA2003) , Kharkiv, Ukraine, 19-21 June 2003, *Lecture Notes in Informatics*, vol. P-30, pp. 21-33.

[SFM03] Shah, U., T. Finin, J. Mayfield. “Information retrieval on the Semantic Web.” In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, McLean, Virginia, 4-9 November 2002, pp. 461-468.

[We05] Wessman, A. “A Framework for Extraction Plans and Heuristics in an Ontology-Based Data-Extraction System”, *Masters Thesis*, Computer Science Department, Brigham Young University, 2005.