Xenon: An RDF Stylesheet Ontology

Dennis Quan
IBM T. J. Watson Research Center
1 Rogers Street
Cambridge, MA 02142 USA
+1 617 693 4612
dennisq@us.ibm.com

David R. Karger
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139 USA
+1 617 258 6167
karger@mit.edu

ABSTRACT
Bringing information to bear from diverse sources to automate tedious processes for the user is a key tenet of the Semantic Web vision. In previous papers we argued that visualizing semantically-connected corpora is not only a prime example of one of these processes but is also a critical problem that must be solved to gain acceptance from the broader community as to the benefits of RDF. In this paper we elaborate on the specific topic of ontologies for describing how resources should be presented to the user. We propose the creation of an RDF stylesheet language, reusing many of the key ideas of the XSL Transformations language (XSLT), but incorporating the requirement that when multiple ontologies are used in the description of a resource to be presented, multiple stylesheets, potentially from different authors, will need to be composed. This notion of enabling *heterogeneous composition* motivates the definition of our basic building block concepts of *view* and *lens*. In addition to abstractly characterizing our notion of an RDF stylesheet, we also describe our concrete instantiation of these ideas in the Xenon ontology. Finally, we give a concrete example illustrating how our stylesheet mechanisms enable developers to easily produce both HTML-based and custom rich client-based browsers for heterogeneous datasets.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Human Factors, Languages.

Keywords
Stylesheet, XSLT, RDF, Semantic Web.

1. INTRODUCTION
The Semantic Web vision proposes enabling radically new forms of automation from multiple information sources being made compatible with one another through a common representation—RDF. RDF Schema and OWL provide ways of describing how the concepts in different corpora with different schemas relate to one another. When a Semantic Web agent encounters data that invokes terms from multiple ontologies, it can use OWL definitions to intelligently decide how to deal with this data. The key concept from OWL we wish to highlight is *heterogeneous composition*—the notion that the knowledge embedded in OWL ontologies written by multiple parties can be readily combined to facilitate making inferences from across information corpora.

We argued in a previous paper [1] that providing a "user agent" is a fundamental prerequisite to the proliferation of the Semantic Web. By user agent we are referring to a Semantic Web browser, which, like previous user agents (e.g., the Web browser), is responsible for allowing users to navigate information corpora and consequently instilling in users the ability to experience the power of information integration first hand.

Like other Semantic Web agents, the Semantic Web browser must deal with the metadata heterogeneity that is both a strength and a source of complexity for the Semantic Web. In our previous paper, we defined the concept of *lens*—a component that extracts and displays a set of related information about a resource. Building browsing interfaces around lenses enables a limited form of heterogeneous composition because lenses can be defined to work across multiple classes and lenses from multiple authors can be combined at runtime to highlight information from different ontologies.

In this paper we take a step back from lenses to describe a more general mechanism for enabling heterogeneous composition for user agents. We term this mechanism a "stylesheet ontology" in analogy to the use of stylesheets on the Web and for XML as a way to abstract presentation from content. If we assume that stylesheets were deemed useful in the HTML and XML contexts, we claim that RDF possesses an even stronger need for stylesheets. HTML (and to some degree XML as well, when the schema in play is simple) is designed to yield human-readable content in a browser, whereas datasets that utilize the expressive power of RDF are rarely human-readable—regardless of the syntax used. Even when RDF is expressed in Notation3 syntax [7], one finds it challenging to discern the connected graph of relationships embedded in RDF content to the user through a straightforward textual syntax.

Our goal is to allow the creation of "intuitive" user interfaces that are specialized to specific kinds of resources and to specific situations. For example, resources shown on the screen should be properly identified by some human-readable appellation and almost never by a URI. Rarely are all the properties of a resource relevant in a given context; they should be properly filtered so as to not overload the user. The properties that are relevant need to be laid out and ordered in a sensible fashion or according to an accepted convention. Additionally, the user should not be impeded by the fact that two or more concepts regarded as equivalent to the user in some context (for example, "title of a

WWW 2005, May 10--14, 2005, Chiba, Japan.

book" and "subject of an e-mail" are both names) are represented differently by the system. Our stylesheet ontology aims to promote these goals within the context of a heterogeneous metadata space.

The paper is organized as follows. We begin with a discussion of the basic notions of the role heterogeneous composition plays in the visualization of RDF metadata. Next, we review the key concepts of the XSLT language and describe how they apply to RDF. We present our prototype Xenon stylesheet ontology that instantiates these concepts and describe a pattern matching mechanism that generalizes the corresponding notion from XSLT. Then, we apply the Xenon stylesheet ontology to define the concepts of view and lens and describe an example stylesheet that incorporates these two concepts. Finally, we discuss how our approach compares to and complements other systems for RDF presentation.

2. A MODEL FOR COMPOSING RDF PRESENTATION KNOWLEDGE

As a semantic network representation, RDF makes it easy for metadata fragments created by independent parties to be combined together into a cohesive whole. During this process, resources whose classes were designed separately may become connected by RDF statements, and similarly the aggregated list of properties for any given resource may include predicates coined by many different people. Thus, in creating a presentation system for RDF, a challenge exists as to how to visualize an arbitrary resource given the heterogeneity of its description.

There are two extremes to consider. One extreme is that any possible combination of classes and properties must have associated with it a user interface that was specifically designed for it. This setup is common in desktop applications today, such as in address book systems, because the schema is fixed at the time of development. It is especially useful when the user interface must possess a high level of refinement. The flip side is that, given the distributed nature of the Semantic Web, the space of possible combinations of classes and properties may be exponentially prohibitive, and only specific combinations that are frequently used will likely have specialized interfaces designed for them.

On the other extreme is a componentized model whereby presentation knowledge for each class and property is provided separately and can be assembled incrementally to produce a presentation for any given combination. This model certainly scales better given the setup of the Semantic Web, but the challenge becomes one of ensuring that the components will "snap together" cleanly when they must be brought to bear to present any given resource.

We argue for a paradigm that allows both extremes to coexist. Contributors of presentation knowledge should be permitted to express user interface constraints at any level of granularity and to whatever the extent known by the contributor. Such a system is consistent with RDF's fine level of granularity, where authors can assert knowledge about a resource as fine as one statement.

We will refer to such a contribution as a *template*. Templates specify how to show some piece of information related to a resource and consist of two main components: the presentation content to be displayed, and metadata describing the conditions under which the template is valid. Presentation content can incorporate text and graphics, as well as *placeholders* for parameterized content such as properties of the displayed resource (e.g., name, creator, etc.).

When these properties-to-be-displayed point to other resources, templates can contain placeholders for other templates to supply that fragment of the presentation. In this way, authors can reuse presentation knowledge not necessarily created by them. The use of placeholders and embedded templates also permits a reusable piece of presentation knowledge to be consolidated in specific templates and to not be duplicated across multiple templates. Of course, placeholder usage is not mandatory; it is possible under this scheme to accommodate completely monolithic templates that do not change when new presentation information is added.

Placeholders need not be specific about which template to embed. Instead, templates include metadata that constrain their usage, such as "made to fit in a small space" or "only applies to resources of type foo:Foo", and the placeholder can be set up to specify templates with certain criteria.

Part of the constraint definition for a template can indicate what role the template is serving. Some templates might show a summary of the given resource, while others may only provide some portion, such as the title or the date (where the definition of what date to use would depend on the kind of resource). Roles are important because they free the developer to specify as little or as much presentation knowledge as desired or available.

This arrangement brings about a sort of template "marketplace", whereby a template can advertise its need for another template that can play a specific role. As a result, multiple templates may exist for any given set of constraints. This is a good thing—multiple authors should be allowed to contribute their ideas. Some templates will be very specific, and others will be more general, allowing other templates to add in their contributions. Some templates will present semantically-differing views of the resource, and others will only change the presentation superficially like MP3 player skins. Furthermore, we believe that users should be given the freedom to choose amongst the different possibilities whenever a placeholder is involved. It is in this flexibility that true end user customizability of the interface can be achieved. This idea is discussed further in Section 7.

3. APPLYING THE CORE CONCEPTS OF XSLT TO RDF

So far we have described a framework for characterizing presentation knowledge that encourages reuse when applied to the visualization of heterogeneous datasets. What we have not described are the nuts-and-bolts details, such as how templates are implemented and how expressive the template definition language is. To actually instantiate this framework as a concrete language that can be used, we look to systems that have been developed that espouse the core concepts involved. Here we examine XSLT—a W3C-standardized language for describing transformations from one XML document to another. Although the structure of an XSLT program's input is an XML file and hence tree-shaped, we have found that the XSLT paradigm for doing transformation to be applicable to general RDF graphs. In this section we analyze XSLT to guide the development of a complete language based on our framework.

XSLT is a Turing universal language with functions, static and dynamic[1] scoping, and functional constructs for implementing conditional logic and loops. These features let programmers package commonly-used code for reuse, do complex data manipulations, and implement other functionality commonly needed when developing a user interface. Certainly, Turing universality is a double-edged sword: the downside of many declarative user interface toolkits is that they are not expressive enough and must be abandoned in favor of a "full" programming language for many frequent use cases; yet, when languages are too expressive, automated analysis for use by WYSIWYG form designers or optimizing compilers becomes difficult. However, because XSLT is a functional language, many forms of static analysis remain possible, and the lack of support for side-effects and restricted data access mechanisms ensure that XSLT programs are generally verifiably safe to run, as are most programs that run within a sandboxed virtual machine[2].

XSLT's support for functions manifests itself in the form of templates. Like the templates notion discussed in Section 2, XSLT templates describe a portion of the resulting presentation that is parameterized by some input node (in the case of XSLT, XML DOM node, and in RDF's case, RDF graph node) and can be restricted for use on specific kinds of nodes. XSLT templates can also invoke other templates, both directly and by use of a kind of placeholder known in XSLT as "apply-templates". Finally, XSLT templates can also take an arbitrary number of additional parameters.

Data access in XSLT is achieved through XPath expressions that allow an XSLT program to navigate the input document and to extract information from it. What ties XSLT's notion of data access to the XML DOM tree model is the fact that XPath is being used. When XPath is replaced by a graph query language, such as SPARQL [8] or RDQL [10], we find that the XSLT template paradigm maps nicely onto the framework delineated in Section 2: templates produce output fragments with placeholders that extract data from the RDF graph instead of from an XML input document.

When a placeholder calls for another template to be embedded to present a given node, the template whose match pattern most closely fits the given node is used. Match patterns in XSLT are described using XPath syntax, but as before, when XPath patterns are replaced with graph matching patterns, we find that the XSLT model remains applicable. The main challenge is in defining a graph matching pattern language whereby patterns can be ordered by match specificity so that placeholders can identify the "best" match. We address this problem in the next section.

Templates in XSLT can be assigned to a *mode*; modes are used to partition the set of templates defined by an XSLT program and can be specified in an apply-templates call to filter the set of templates from which a match can be made. We feel that mode is just one possible attribute that can be used to restrict the set of templates, and given the expressive power of RDF, modes can be replaced by general RDF queries for isolating a group of templates.

4. XENON PROTOTYPE IMPLEMENTATION

In the previous section we asserted that the XSLT model can be effectively reapplied to RDF to implement the stylesheet framework described in Section 2. The language we have built to implement this modified XSLT model is called Xenon and is described in this section. In addition to replacing XPath with RDF analogs, we have also made some changes to the language that help it to better act as a medium for recording presentation knowledge in a Semantic Web environment.

The Xenon stylesheet language is specified as an RDF ontology. In other words, the role the abstract syntax tree (AST) usually plays in functional languages such as XSLT or Lisp is played by an RDF fragment. The Xenon RDF-based AST is actually a directed acyclic graph (DAG). By representing an AST in RDF, we have therefore drawn a correspondence between the terms "language" and "ontology": a Xenon stylesheet is RDF written with respect to the Xenon ontology.

There are several benefits to using RDF to represent a Xenon stylesheet. First, stylesheet information should be as easily distributable and universal as schema or ontology information, and using an RDF representation removes most of the syntactic barriers to achieving these goals. Second, there is no need to force users to adopt a specialized syntax; existing syntaxes such as Notation3 and RDF/XML can be chosen from. Third, RDF has a built-in notion of what it means to merge two fragments of RDF, and so the semantics of combining two Xenon stylesheets is therefore clearly defined.

Xenon, like XSLT[3], is a pure functional language with no side effects. The basic element in Xenon code therefore is the expression, and as is typical in functional languages, expressions are designed to be arbitrarily nested. Expressions are manifested in the form of resources, whose rdf:type assertion dictates the type of expression being recorded. Some of an expression's properties point to other expressions, just as an AST node has children nodes that represent its arguments. There are three kinds of expressions: built-in expressions (analogous to special forms in Scheme or Lisp), native expressions, and user-defined template calls.

We describe six basic expressions built into Xenon. First, the xe:Let expression binds a variable name to another expression; this variable name is statically scoped (cf. xsl:variable). The corresponding xe:Identifier expression evaluates to the value named by the given variable name. xe:With is like xe:Let, except that the variable name is dynamically scoped (dynamic scoping is used to support cascading properties, such as font or background color). xe:If evaluates to the first consequent if the condition evaluates to true; otherwise, it evaluates to the second consequent (cf. xsl:choose). xe:ForEach iterates over a list of objects and concatenates the values that result from evaluating the body expression with the loop variable bound to the current object (cf. xsl:for-each). Finally, xe:Resource allows a URI immediate value to be expressed.

[1] Only a fixed set of variables such as the current node and current position are dynamic in XSLT 1.0; in the upcoming XSLT 2.0 specification, the user can define an unlimited number of dynamically-scoped variables called "tunnel parameters".

[2] Obviously, one thing that cannot be detected, as with Java, JavaScript, and other Turing universal languages, is whether the code will terminate.

[3] XSLT 1.0 is a pure functional language when the generate-id() function is never used.

Native expressions provide access to functionality that is most easily implemented by the underlying system but does not require new variable bindings to be introduced. We highlight two such expressions here. xe:Select encapsulates a SPARQL query and returns a list of results. xe:ApplyTemplates takes a resource and a SPARQL query constraining the list of templates over which to be searched and returns the content created by that template.

In addition, native expressions perform another basic function: the instantiation of content. Unlike XSLT, where arbitrary XML result tree fragments can be instantiated, Xenon abstracts the production of result content into a library of native expressions. Another way to think about this is that Xenon requires the output content to conform to a schema that is known in advance. In our initial implementation, we have chosen to use a slightly-modified version of the XHTML tag set to name these native expressions.

There are two reasons for our decision to abstract content generation in terms of native templates. First, it is not necessary to define a general RDF-to-RDF transformation language: that is the whole point of an inference engine. Second, our representation allows us to create both markup-based (e.g., XHTML) and widget-based versions from the same stylesheet. This is because the Xenon representation allows a stylesheet processor to instantiate widgets straight from the native expressions, much as is done in systems such as Glade for GTK [12]. Work on such an implementation is in progress.

Finally, a user-defined template call expression is a resource whose type is a class of type xe:Template. This is the mechanism by which new abstractions can be created in a stylesheet. We have chosen to derive xe:Template from rdfs:Class in order to take advantage of RDF Schema in specifying the parameters that a template can accept: the parameters to an instance of an xe:Template are simply rdf:Property's whose domain is the xe:Template. In other words, to instantiate a template *x*, where *x* has rdf:type xe:Template, create a resource whose rdf:type is *x*.

Templates, being defined in RDF, can have additional metadata associated with them. We have predefined the xe:role property as one derivative of the original xsl:mode attribute (other derivative properties will be introduced later). Other properties, such as those from the Dublin Core ontology [11], can be used for documentation purposes.

Additionally, match patterns for templates are defined using the xe:matchPattern property, which points to an rdf:List of xe:MatchPattern resources. An xe:MatchPattern has two elements: a SPARQL query, and an integer score value (cf. xsl:priority). The Xenon engine, when evaluating a xe:ApplyTemplates expression, will walk the list of match patterns for a candidate template and determine whether the target resource, when substituted into the SPARQL query, produces a graph fragment that is a subgraph of the RDF store in use. For those match patterns that are satisfied, the corresponding score values are added up, and a template whose cumulative score is the highest is selected.

5. HIGHER-LEVEL ABSTRACTIONS: LENSES AND VIEWS

The Xenon ontology provides a generic framework for describing how a resource may be transformed into a presentation as well as a template matching system for supporting heterogeneous composition. Our development of the Haystack system has elucidated some higher-level abstractions built on top of Haystack's custom template matching system that we have found to be useful for building Semantic Web user interfaces. The two key concepts from this work are *lenses* and *views*. In this section, we elucidate these concepts in terms of the Xenon framework and describe how they can be applied to build real-world user interfaces.

5.1 Lenses

A lens is a component that shows information from a (possibly singleton) set of properties of a resource that makes sense being shown together [1]. Examples of lenses include a name lens, which shows the human-readable name of a resource, and a summary lens, which shows the set of key properties for a given resource. Lenses are often used to abstract a general concept across multiple classes. For example, the notion of "name" makes sense for both books and people, although the precise predicate or other means of representation used might be radically different. Using match patterns, we can define two lens templates that play the same xe:role—xe:nameLens.

In Xenon, we can model this phenomenon by creating *lens templates*. Lens templates have type xe:LensTemplate (which derives from xe:Template) and are assumed to only take one parameter: xe:target (whereas general templates can take an arbitrary number of parameters[4]). Properties, such as the size class of the output of the lens template, can be specified in RDF; in particular, we predefine the xe:size property to allow a lens template to be associated with a resource that identifies a size class, such as xe:fullScreen, xe:oneLine, xe:thumbnail, etc. As with many Semantic Web concepts, these size class resources (or other possible property values of lens templates) then act as a contract between the lens author and lens consumer, whereby reuse of a resource such as xe:oneLine can be relied upon to convey the semantics intended by the author of the resource, and new resources referring to different semantics can be coined at will.

5.2 Views

Lens can be used to show selected pieces of information regarding a resource. However, if one wants to simply "show" a resource and delegate the responsibility of deciding which selected pieces to show, a developer can embed a special kind of lens called a view. In Haystack, a view is defined to be a component that generates a region on the screen that *represents* a given resource [9]. To motivate our definition of view, consider the interface exposed by a typical GUI e-mail client. Person resources are found throughout such an application: in the From field in an e-mail editor, under the From column in the inbox listing, as a row in the address book listing, as a standalone window showing an address book entry, etc. Similarly, when a calendar resource is shown as a monthly calendar instead of as a daily calendar, the user is invoking different views to visualize that calendar resource. In the Haystack paradigm, these widgets that display representations of resources are all considered views of those resources.

It is useful to embed views whenever the decision of what is a "representative" presentation should be delegated to others. For example, when a list of resources need to be displayed, instead of

[4] Because Xenon templates play the same role as functions in other functional languages, a utility function that takes multiple parameters would be an example of a template that might not take a single parameter.

having the designer of the list template decide the best lens for or best way to present each element of the list, he or she can insert placeholders for views of a specific size for each resource on the list to be inserted. Views are also often embedded within other views or lenses. If a lens needs to show a list of recipients for an e-mail message, it can embed icon-sized views of people, mailing list, and group resources, thereby shifting the responsibility of deciding the "canonical" visualization of these resources, given the size constraints, to others.

Developers who are designing lenses can designate a lens template as being a *view template* if he or she believes, given the size and/or other constraint metadata defined for the lens template, that the lens produces a presentation that is likely to be representative, in the user's eyes, of the resource being shown. For example, if one creates an icon tile-sized lens of a calendar that only shows the name of the owner of the calendar, this lens would make a poor icon tile-sized view, because if a user is looking at a folder of resources as a series of icon tiles, each resource in the folder would be represented by an icon tile-sized view, and it is unlikely that the user would recognize a line of text with only a person's name on it as being a calendar.

One common way to build a view template is to consider what the key aspects of a resource are and to embed lenses to display them. The default icon tile-sized view template may simply embed placeholders for an icon lens and a name lens, as well as possibly some generally-useful summary information such as a date lens or a lens that indicates the resource's type. (This is the arrangement typically seen in a file system browser such as Windows Explorer.)

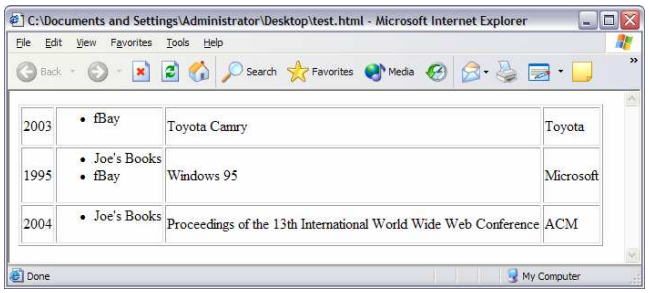
In Xenon, we can model view templates as templates with type xe:ViewTemplate (which derives from xe:LensTemplate) and role xe:view and that are assumed to only take one xe:target parameter. All of the other properties used to characterize lens templates apply equally to view templates.

6. EXAMPLE: A CATALOG BROWSER

To illustrate how an application can be built using lenses and views, we have constructed an example that enables a user to browse a heterogeneous product catalog listing. Our example is based on the following sample data, given in Notation3 format:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
schema#> .
@prefix xe:
<http://haystack.lcs.mit.edu/schemata/xenon#> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix sample: <http://tempuri.org/sampledata#> .
@prefix schema: <http://tempuri.org/sampleschema#>
.
@prefix : <http://tempuri.org/samplestylesheet#> .

sample:collection
   a                    schema:Collection ;
   schema:member        sample:camry ;
   schema:member        sample:windows95 ;
   schema:member        sample:www2004 .

sample:camry
   a                    schema:Car ;
   schema:name          "Toyota Camry" ;
   schema:manufacturer  "Toyota" ;
   schema:model         "2003" .

sample:windows95
   a                    schema:Software ;
   schema:name          "Windows 95" ;
   schema:creator       "Microsoft" ;
   schema:version       "1995" .

sample:www2004
   a                    schema:Book ;
   dc:title             "Proceedings of the 13th
International World Wide Web Conference" ;
   dc:creator           "ACM" ;
   dc:date              "2004" .

sample:joesBooks
   a                    schema:BookStore ;
   rdfs:label           "Joe's Books" ;
   schema:sells         sample:windows95 ;
   schema:sells         sample:www2004 .

sample:fBay
   a                    schema:AuctionSite ;
   rdfs:label           "fBay" ;
   schema:sells         sample:windows95 ;
   schema:sells         sample:camry .
```

The presentation we are aiming to produce from this data looks like the following (given a target representation of HTML to be shown in a Web browser):



We have intentionally left out column headers, excessive formatting, and other extraneous elements from our example to better illustrate the core concepts at play. The columns here list the year a product was made available, a list of the stores at which the product can be bought, the name of the product, and the creator of the product. Note that the schemas used in the sample data for representing cars uses different RDF properties for naming similar attributes than does the schema for books.

6.1.1 *Creating a Simple View Template*

The second column shows a listing of the stores that carry the product named in that row. Therefore, it is likely that the decision of how to represent the store resources in those listings should be delegated to a view. We use the xe:inline size class to refer to the fact that the view should be a piece of text suitable for insertion into a paragraph. Given the size constraints, a reasonable presentation is one in which only the title is shown. The following code defines such an inline view template that embeds the name lens:

```
xe:target
   rdfs:domain :DefaultInlineView

:DefaultInlineView
   a           xe:ViewTemplate ;
   xe:role     xe:view ;
   xe:size     xe:inline ;
   xe:match    () ;        # Matches everything
   xe:body [
      a  xe:ApplyTemplates ;
      xe:select [
         a        xe:Identifier ;
         xe:name  xe:target
```

```
      ] ;
      xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
SELECT ?TARGET WHERE (?TARGET xe:role
xe:nameLens)"
   ]
```

Two observations should be pointed out at this point. First, the match pattern is empty, so the default match score of 1 will apply. If other, more specific templates are created in this size class, then they must produce match scores greater than 1. As is the case in many domains, we believe conventions will develop to specify the scales of match scores for specific roles.

Additionally, our SPARQL queries assume that only one variable, ?TARGET, is being queried for.[5]

6.1.2 Creating a Simple Lens Template

To illustrate the process of defining a lens template, we will create a default name lens that uses the rdfs:label property by default to extract the name of a resource:

```
xe:target
   rdfs:domain :DefaultNameLens

:DefaultNameLens
   a        xe:LensTemplate ;
   xe:role  xe:nameLens ;
   xe:size  xe:inline ;
   xe:match () ;        # Matches everything
   xe:body [
      a  xhtml:Text ;
      xe:body [
         a                 xe:Select ;
         xe:singleResult   "true" ;
         xe:filter         "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?TARGET WHERE ($xe:target dc:title
?TARGET)"
      ]
   }
}
```

In this example, one notes that the SPARQL query can use a $ prefix to refer to variables currently in scope, and SPARQL queries can be annotated with the xe:singleResult parameter to instruct the query engine to only return one result.

6.1.3 Creating a Specialized Lens Template

The columns in the listing are all showing different properties of the resources represented by the rows of the table, but as we pointed out earlier, the resources do not all share exactly the same schematic representation for these properties. We have therefore constructed lenses to abstract away these differences in underlying representation.

To define the "year lens", we need to create lens templates for each of the possible underlying representations of "year". These templates are distinguished by their match patterns. Here is the definition for the year lens for the schema:Car class:

```
xe:target
   rdfs:domain :CarYearLens
```

[5] Because the SPARQL specification is currently a draft, we have adopted a syntactic variation on the SPARQL syntax with our internal implementation and will update it to conform to the final syntax when it is released. The latest SPARQL syntax has been incorporated into this paper for ease of readability.

```
:CarYearLens
   a        xe:LensTemplate ;
   xe:role  :yearLens ;
   xe:size  xe:inline ;
   xe:match (
      [  a           xe:MatchPattern ;
         xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> SELECT ?TARGET WHERE ($xe:target
rdf:type schema:Car)" ;
         xe:score    "2"
      ]
   ) ;
   xe:body [
      a  xhtml:Text ;
      xe:body [
         a                 xe:Select ;
         xe:singleResult   "true" ;
         xe:filter         "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?TARGET WHERE ($xe:target schema:model
?TARGET)"
      ]
   }
}
```

6.1.4 The Sold-by Lens

The lens that shows the list of stores that sell the given resource is different from the ones we have seen so far because it embeds a list of resources. The sold-by lens template uses an xe:ForEach to iterate over the stores that carry the product, wraps each in an <LI> tag, and uses an xe:ApplyTemplates to embed the appropriate inline view template:

```
xe:target
   rdfs:domain :DefaultSoldByLens

:DefaultSoldByLens
   xe:role     sample:soldByLens ;
   xe:match    () ;     # Matches everything
   xe:body [
      a  xhtml:UL ;
      xe:body [
         a           xe:ForEach ;
         xe:select [
            a           xe:Select ;
            xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
SELECT ?TARGET WHERE (?TARGET schema:sells
?xe:target)"
         ] ;
         xe:loopVar  :x ;
         xe:body [
            a  xhtml:LI ;
            xe:body [
               a  xe:ApplyTemplates ;
               xe:select [
                  a        xe:Identifier ;
                  xe:name  :x
               ] ;
               xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
SELECT ?TARGET WHERE (?TARGET xe:size xe:inline)
(?TARGET xe:role xe:view)"
            ]
         ]
      ]
   ]
```

6.1.5 *Putting it all together*

Finally we can create a view template for the sample:collection resource. First, here is a top level template for driving the example:

```
sample:CollectionExampleDriver
   a  xe:Template ;
   xe:body [
      a  xhtml:P ;
      xe:body [
         a           xe:With ;
         xe:name     :lensList ;
         xe:select [
            a           xe:Select ;
            xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX : <http://tempuri.org/samplestylesheet#>
SELECT ?TARGET WHERE (:lenses schema:member
?TARGET)"
         ] ;
         xe:body [
            a  :CollectionView ;
            xe:target [
               a           xe:Resource ;
               xe:resource sample:collection
            ]
         ]
      ]
   ]

sample:lenses
   :member  xe:nameLens ;
   :member  xe:creatorLens ;
   :member  sample:yearLens ;
   :member  sample:soldByLens
```

The template iterates over the members of the collection and embeds a second, helper template that in turn generates a table row and embeds the appropriate lenses within <TD> tags. Note the use of the dynamically-scoped variable, defined in the driver code, for passing in the list of lenses. Because view templates are normally invoked via xe:ApplyTemplates, which only accepts one parameter, using dynamic scoping permits other parameters to be passed through (cf. xsl:with-param in xsl:apply-templates).

```
xe:target
   rdfs:domain :CollectionView

:CollectionView
   a  xe:ViewTemplate ;
   xe:match (
      [  a           xe:MatchPattern ;
         xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> SELECT ?TARGET WHERE ($xe:target
rdf:type schema:Collection)" ;
         xe:score    "2"
      ]
   ) ;
   xe:body [
      a  xhtml:TABLE ;
      xe:body [
         a  xe:ForEach ;
         xe:select [
            a  xe:Select ;
            xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
SELECT ?TARGET WHERE ($xe:target schema:member
?TARGET)"
         } ;
         xe:loopVar  :x ;
         xe:body [
            a  xhtml:TR ;
            xe:body [
               a  :Helper ;
               xe:target [
                  a        xe:Identifier ;
                  xe:name  :x
               ]
            ]
         ]
      ]
   ]

xe:target
   rdfs:domain :Helper

:Helper
   a        xe:Template ;
   xe:body [
      a  xe:ForEach ;
      xe:select [
         a        xe:Identifier ;
         xe:name  :lensList
      } ;
      xe:loopVar  :x ;
      xe:body [
         a  xhtml:TD ;
         xe:body [
            a           xe:ApplyTemplates ;
            xe:select [
               a        xe:Identifier ;
               xe:name  xe:target
            ] ;
            xe:filter   "PREFIX xe:
<http://haystack.lcs.mit.edu/schemata/xenon#>
PREFIX : <http://tempuri.org/samplestylesheet#>
SELECT ?TARGET WHERE (?TARGET xe:role $:x)"
         ]
      ]
   ]
```

7. DISCUSSION AND RELATED WORK

There are a number of benefits that arise from the use of a functional Turing universal stylesheet ontology such as Xenon. First, it is easy to support abstractions that permit presentation knowledge to be described to the extent to which a developer, a designer, or some other contributor wishes to specify. We have described one possible abstraction, views and lenses, that has been shown to have broad applicability in Haystack across domains as varied as media players, e-mail, and bioinformatics [1], but other abstractions are of course possible.

One theme that is prevalent is the notion that code and data are not heavily distinguished, a notion reminiscent of languages such as Lisp. In the catalog browser example from Section 6, one sees that some pieces of presentation metadata seem more like JSP-style HTML generation code (e.g., :CollectionView), whereas others are more like fragments of configuration data meant to act as parameters to other templates (e.g., sample:lenses). Because templates use RDF Schema to describe their parameters, both “code” and “data” have the same form and differ only by the complexity of the domain they are describing.

As a result, the stylesheet designer, in creating a presentation system, is given a lot of freedom to split the user interface specification between the “static code library”—the portion that stays fixed—and “customizations”—the part that is provided by people contributing ontology-specific adjustments to the system. On one extreme, contributors provide only a “bare-bones specification” (e.g., a list of important properties as a property of the rdfs:Class) that is consumed by a default view template that matches against all resources and locates the important properties using a query against the class. On the other extreme, contributors

provide stylesheet templates that use the full expressive power of the functional language, including conditional code, loops, placeholders for other templates, etc. An example of a paradigm in the middle is one in which templates simply call other templates with static parameters such as the salient properties for each class:

```
:PersonViewTemplate
   a xe:ViewTemplate ;
   xe:body [
      a :DefaultViewTemplate ;
      :importantProperties ( :name :address ) ;

      # Pass the target resource through
      xe:target [
         a xe:Identifier ; xe:name xe:target
      ]
   ]
```

We believe this level of flexibility allows developers to produce reusable libraries of code that implement a variety of different interface styles. Furthermore, our stylesheet mechanism enables systems that rigidly support only one of the extremes given above to gradually move to a more general paradigm of template-based customization. For example, the Longwell faceted metadata browser [3] allows users to visualize a resource that is in focus by means of a customizable list of properties. Users can also pivot to other resources that are related to the current resource by a shared property (these pivot points are called facets). The specifications of which properties to display and what facets are available are defined against a custom RDF schema. Using the "bare-bones" approach, one could construct a stylesheet that read these specifications and produced screens similar to those of Longwell. Other automatically-generated, form-based RDF user interfaces, such as those produced by Protégé [13] and SEAL [2], could be reproduced with a similar approach. However, by adding view templates that overrode the default templates, one could begin to gradually introduce higher levels of customization than those implied by the custom RDF schema-driven configuration metadata data.

One difference between Xenon and systems like Protégé is that Xenon only attacks the problem of visualizing existing RDF content. We are also interested in the problem of interfaces that record RDF, as was discussed in a previous paper [9]. The Haystack browser supports RDF authoring, but we are looking for ways to add authoring support into Xenon.

8. REFERENCES

[1] Quan, D. and Karger, D. How to Make a Semantic Web Browser. Proceedings of WWW 2004.

[2] Stojanovic, N., Maedche, A., Staab, S., Studer, R., Sure, Y. SEAL: a framework for developing SEmantic PortALs. Proceedings of the International Conference on Knowledge Capture October 2001.

[3] The Simile Longwell Project. http://simile.mit.edu/longwell/.

[4] XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt.

[5] XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/.

[6] Quan, D., Huynh, D., and Karger, D. Haystack: A Platform for Authoring End User Semantic Web Applications. Proceedings of ISWC 2003.

[7] Berners-Lee, T. Primer: Getting into RDF & Semantic Web using N3. http://www.w3.org/2000/10/swap/Primer.html.

[8] SPARQL Query Language for RDF. http://www.w3.org/TR/2004/WD-rdf-sparql-query-20041012/.

[9] Quan, D., Karger, D., and Huynh, D. RDF Authoring Environments for End Users. Proceedings of Semantic Web Foundations and Application Technologies 2003.

[10] RDQL—A Query Language for RDF. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[11] Dublin Core Metadata Initiative. http://dublincore.org/.

[12] Glade GTK+ User Interface Builder. http://glade.gnome.org/.

[13] Noy, N., Sintek, M., Decker, S., Crubezy, M., Ferguson, R., and Musen, M. Creating Semantic Web Contents with Protege-2000. *IEEE Intelligent Systems* 16 (2), 2001, pp. 60-71.