Incremental Navigation: Providing Simple and Generic Access to Heterogeneous Structures*

Shawn Bowers[1] and Lois Delcambre[2]

[1] San Diego Supercomputer Center at UCSD, La Jolla CA 92093, USA,
[2] OGI School of Science and Engineering at OHSU, Beaverton OR 97006, USA,
bowers@sdsc.edu lmd@cse.ogi.edu

Abstract. We present an approach to support *incremental navigation* of structured information, where the structure is introduced by the data model and schema (if present) of a data source. Simple browsing through data values and their connections is an effective way for a user or an automated system to access and explore information. We use our previously defined Uni-Level Description (ULD) to represent an information source explicitly by capturing the source's data model, schema (if present), and data values. We define generic operators for incremental navigation that use the ULD directly along with techniques for specifying how a given representation scheme can be navigated. Because our navigation is based on the ULD, the operations can easily move from data to schema to data model and back, supporting a wide range of applications for exploring and integrating data. Further, because the ULD can express a broad range of data models, our navigation operators are applicable, without modification, across the corresponding model or schema. In general, we believe that information sources may usefully support various styles of navigation, depending on the type of user and the user's desired task.

1 Introduction

With the WWW at our fingertips, we have grown accustomed to easily using unstructured and loosely-structured information of various kinds, from all over the world. With a web browser it is very easy to: (1) view information (typically presented in HTML), and (2) download information for viewing or manipulating in tools available on our desktops (e.g., Word, PowerPoint, or Adobe Acrobat files). In our work, we are focused on providing similar access to structured (and semi-structured) information, in which data conforms to the structures of a representation scheme or data model.

There is a large and growing number of structural representation schemes being used today including the relational, E-R, object-oriented, XML, RDF, and Topic Map models along with special-purpose representations, e.g., for exchanging scientific data. Each representation scheme is typically characterized by its choice of constructs for representing data and schema, allowing data engineers to select the representation best suited for their needs. However, there are few tools that allow data stored in different representations to be viewed and accessed in a standard way, with a consistent interface.

* This work supported in part by NSF grants EIA 9983518 and ITR 0225674.

The goal of this work is to provide generic access to structured information, much like a web browser provides generic access to viewable information. We are particularly interested in browsing a data source where a user can select an individual item, select a path that leads from the item, follow the path to a new item, and so on, *incrementally* through the source.

The need for incremental navigation is motivated by the following uses. First, we believe that simple browsing tools provide people with a powerful and easy way to access data in a structured information source. Second, generic access to heterogeneous information sources supports tools that can be broadly used in the process of data integration [8, 10]. Once an information source has been identified, its contents can be examined (by a person or an agent) to determine if and how it should be combined (or integrated) with other sources.

In this paper, we describe a generic set of incremental-navigation operators that are implemented against our Uni-Level Description (ULD) framework [4, 6]. We consider both a low-level approach for creating detailed and complete specifications as well as a simple, high-level approach for defining specifications. The high-level approach exploits the rich structural descriptions offered by the ULD to automatically generate the corresponding detailed specifications for navigating information sources. Thus, our high-level specification language allows a user to easily define and experiment with various navigation styles for a given data model or representation scheme. The rest of this paper is organized as follows. In Section 2 we describe motivating examples and Section 3 briefly presents the Uni-Level Description. In Section 4, we define the incremental navigation operators and discuss approaches to specifying their implementation. Related work is presented in Section 5 and in Section 6 we discuss future work.

2 Motivating Examples

When an information agent discovers a new source (e.g., see Figure 1) it may wish to know: (1) what data model is used (is it an RDF, XML, Topic Map, or relational source?), (2) (assuming RDF) whether any classes are defined for the source (what is the source schema?), (3) which properties are defined for a given class (what properties does the *film* class have?), (4) which objects exist for the class (what are the instances of the *film* class?) and (5) what kinds of values exist for a given property of a particular object of the class (what *actor* objects are involved in this *film* object?).

This example assumes the agent (or user) understands the data model of the source. For example, if the data model used was XML (e.g., see Figure 2) instead of RDF, the agent could have started navigation by asking for all of the available element types (rather than RDF classes). We call this approach *data-model-aware navigation*, in which the constructs of the data model can be used to guide navigation.

In contrast, we also propose a form of browsing where the user or agent need not have any awareness of the data-model structures used in a data source. The user or agent is able to navigate through the data and schema directly. As an example (again using Figure 1), the user or agent might ask for: (1) the kind of information the source contains, which in our example would include “films,” “actors,” and “awards,” etc., (2) (assuming the crawler is interested in films) the things that describe films, which

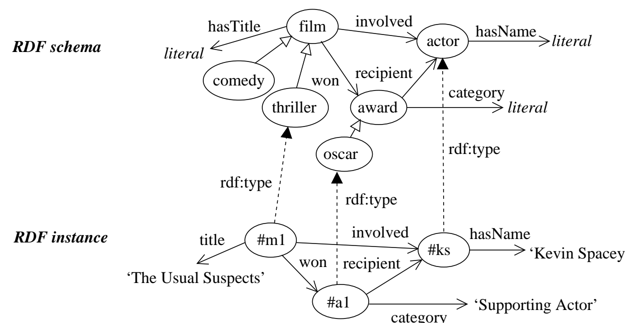
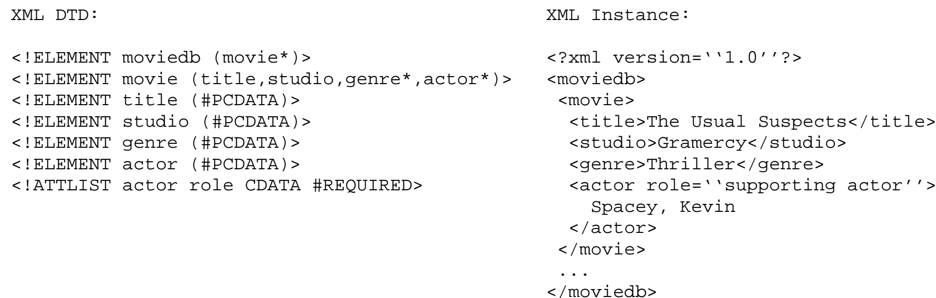


Fig. 1. An example of an RDF schema and instance.



```
XML DTD:

<!ELEMENT moviedb (movie*)>
<!ELEMENT movie (title,studio,genre*,actor*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT studio (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT actor (#PCDATA)>
<!ATTLIST actor role CDATA #REQUIRED>
```

```
XML Instance:

<?xml version=``1.0''?>
<moviedb>
 <movie>
  <title>The Usual Suspects</title>
  <studio>Gramercy</studio>
  <genre>Thriller</genre>
  <actor role=``supporting actor''>
    Spacey, Kevin
  </actor>
 </movie>
 ...
</moviedb>
```

Fig. 2. An example XML DTD (left) and instance document (right).

would include "titles" and relationships to awards and actors, (3) the available films in the source, and (4) the actors of a particular film, which is obtained by stepping across the "involved" link for the film in question. We call this form of browsing *simple navigation*.

3 The Uni-Level Description

The Uni-Level Description (ULD) is both a *meta-data-model* (i.e., capable of describing data models) and a distinct representation scheme: it can directly represent both schema and instance information expressed in terms of data-model constructs. Figure 3 shows how the ULD represents information, where a portion of an object-oriented data model is described. The ULD is a flat representation in that all information stored in the ULD is uniformly accessible (e.g., within a single query) using the logic-based operations described in Table 1.

Information stored in the ULD is logically divided into three layers, denoted *meta-data-model*, *data model*, and *schema and data instances*. The ULD meta-data-model, shown as the top level in Figure 3, consists of *construct types* that denote structural primitives. The middle level uses the structural primitives to define both data and schema

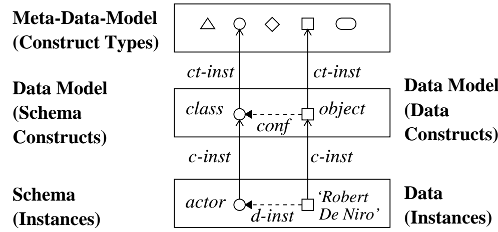


Fig. 3. The ULD meta-data-model architecture.

Operations for defining instance-of relationships	
ct-inst(c, ct)	Construct c is a *ct-inst* of construct-type ct.
c-inst(d, c)	Construct instance d is a *c-inst* of construct c.
d-inst(d_1, d_2)	Construct instance d_1 is a *d-inst* of instance d_2.
conf(d, r, c_1, c_2)	Instances of construct c_1 can conform to instances of construct c_2 as required by domain d and range r cardinality constraints (exactly one 1, zero or one ?, zero or many *, or one or many +).
Operations for restricting construct instances	
$c_1.s$->c_2	Instances of construct c_1 have selectors s with instances of construct c_2 as values.
setof(c_1, c_2)	Instances of construct c_1 are sets whose members are construct c_2 instances.
bagof(c_1, c_2)	Instances of construct c_1 are bags whose members are construct c_2 instances.
listof(c_1, c_2)	Instances of construct c_1 are lists whose members are construct c_2 instances.
unionof(c_1, c_2)	Instances of construct c_2 are also construct c_1 instances.
Operations for accessing instance structures	
$d_1.s$:d_2	Construct instance d_1 has value d_1 for selector s.
$d_1 \in d_2$	Construct instance d_1 is a member of collection d_2.
$d_1[i]$=d_2	Construct instance d_2 is at the i-th position in the list d_1.
$\|d_1\|$=n	The length of the collection construct instance d_1 is n.
$d_1 \in\in d_2$=n	Construct instance d_1 is a member of the bag d_2 n times.

Table 1. The ULD operations expressed as logical formulas.

constructs, possibly with *conformance relationships* between them. Constructs are necessarily instances of construct types, represented with ct-inst instance-of links. Similarly, every item introduced in the bottom layer, denoting actual data or schema items, is necessarily an instance of a construct in the middle layer, represented with c-inst instance-of links. An item in the bottom layer can be an instance of another item in the bottom layer, represented with *d-inst* instance-of links, as allowed by the conformance relationships specified in the middle layer. For example, in Figure 3, the *class* and *object* constructs are related through a conformance link, labeled *conf*, and their corresponding construct instances in the bottom layer, i.e., *actor* and the object with the name 'Robert De Niro' are related through a data instance-of link, labeled *d-inst*.

The ULD offers flexibility through the conf and d-inst relationships. For example, an XML element that does not have an associated element type can be represented in the ULD; the element would simply not have a d-inst link to any XML element type.

The ULD represents an information source as a *configuration* containing the constructs of a data model, the construct instances (both schema and data) of a source, and the associated conformance and instance-of relationships. A configuration can be viewed as an instantiation of Figure 3. Each configuration uses a finite set of identifiers to denote construct types, constructs, and construct instances as well as a finite set of

schema constructs:		
ct-inst(elemType, struct-ct)	ct-inst(pcdata, atomic-ct)	elemType.hasModel->contentDef
ct-inst(attDefSet, set-ct)	ct-inst(cdata, atomic-ct)	setof(attDefSet, attDef)
ct-inst(attDef, struct-ct)	elemType.hasName->uldString	attDef.hasName->uldString
ct-inst(contentDef, set-ct)	elemType.hasAtts->attDefSet	setof(contentDef, elemType)
data constructs:		
ct-inst(element, struct-ct)	conf(*, ?, element, elemType)	element.hasChildren->content
ct-inst(attSet, set-ct)	conf(*, ?, attribute, attDef)	setof(attSet, attribute)
ct-inst(attribute, struct-ct)	unionof(node, element)	attribute.hasName->uldString
ct-inst(content, list-ct)	element.hasTag->uldString	attribute.hasVal->cdata
ct-inst(node, union-ct)	element.hasAtts->attSet	listof(content, node)
unionof(node, pcdata)		

Fig. 4. The XML with DTD data model.

ct-inst, c-inst, conf, and d-inst facts. We note that a configuration can be implemented as a logical view over an information source, and is not necessarily “materialized.”

The ULD meta-data-model contains primitive structures for tuples, i.e., sets of name-value pairs; set, list, and bag collections; atomics, for scalar values such as strings and integers; and unions, for representing non-structural, generalization relationships among constructs. The construct-type identifiers for these structures are denoted *struct-ct*, *set-ct*, *list-ct*, *bag-ct*, *atomic-ct*, and *union-ct*, respectively.

Figures 4, 5, and 6 give example descriptions of simplified versions of XML with DTDs, RDF with RDF Schema, and sample schema and data (from Figure 1) for the RDF model, respectively.[3] We note that there are potentially many ways to describe a data model in the ULD, and these examples show only one choice of representation.

The XML data model shown in Figure 4 includes constructs for element types, attribute types, elements, attributes, content models, and content, where element types contain attribute types and content specifications, elements can optionally conform to element types, and attributes can optionally conform to attribute types. We simplify content models to sets of element types for which a conforming element must have at least one subelement for each corresponding type.

The RDF data model with RDF Schema (RDFS) of Figure 5 includes constructs for classes, properties, resources, and triples. A triple in RDF contains a subject, predicate, and object, where a predicate can be an arbitrary resource, including a defined property. In RDFS, `rdf:type`, `rdfs:subClassOf`, and `rdfs:subPropertyOf` are considered special RDF properties for denoting instance and specialization relationships. However, we model these properties using conformance and explicit constructs. For example, a subclass relationship is represented by instantiating a `subClassOf` construct as opposed to using the special `rdfs:subClassOf` RDF property.[4]

A ULD query is expressed as a Datalog program [1] and is executed against a configuration. As an example, the first query below finds all available class names within an

---

[3] We use `uldValue` and `uldValuetype` as special constructs to denote scalar values and value types [4, 6]. Also, `uldString` and `uldURI` are default atomic constructs provided by the ULD.

[4] This ULD representation of RDF allows properties and isa relationships to be decoupled (compared with RDF itself). This approach does not limit the expressibility of RDF: partial, optional, and multiple levels of schema are still possible.

schema constructs:		
ct-inst(resource, union-ct)	conf(*, *, class, class)	simpleRes.hasURI->uldURI
ct-inst(rdfType, union-ct)	conf(*, *, prop, rdfType)	class.hasURI->uldURI
ct-inst(simpleRes, struct-ct)	unionof(rangeVal, class)	class.hasLabel->uldString
ct-inst(class, struct-ct)	unionof(rangeVal, uldValueType)	prop.hasURI->uldURI
ct-inst(prop, struct-ct)	subClass.hasSub->class	prop.hasLabel->uldString
ct-inst(rangeVal, union-ct)	unionof(resource, rdfType)	prop.hasDomain->class
ct-inst(subClass, struct-ct)	unionof(resource, simpleRes)	prop.hasRange->rangeVal
ct-inst(subProp, struct-ct)	unionof(rdfType, class)	subClass.hasSuper->class
conf(*, *, simpleRes, class)	unionof(rdfType, prop)	subProp.hasSub->prop
subProp.hasSuper->prop		
data constructs:		
ct-inst(triple, struct-ct)	unionof(objVal, resource)	triple.hasObj->objVal
ct-inst(objVal, union-ct)	triple.hasPred->resource	unionof(objVal, literal)
ct-inst(literal, atomic-ct)	triple.hasSubj->resource	

Fig. 5. The RDF with RDF Schema data model.

schema:		
c-inst(film, class)	thriller.hasURI:'#thriller'	prop.hasLabel:'#hasTitle'
c-inst(title, prop)	filmthril.hasSub:thriller	prop.hasRange:'literal'
c-inst(thriller, class)	film.hasURI:'#film'	thriller.hasLabel:'thriller'
c-inst(filmthril, subclass)	film.hasLabel:'film'	filmthril.hasSuper:film
prop.hasDomain:film	prop.hasURI:'#title'	
data:		
c-inst(m1, simpleRes)	d-inst(m1, film)	t1.hasSubj:m1
c-inst(t1, triple)	m1.hasURI:'#m1'	t1.hasObj:'The Usual Suspects'
d-inst(m1, thriller)	t1.hasPred:title	

Fig. 6. Portion of schema and data for RDF(S).

RDF configuration. Note that upper-case terms denote variables and lower-case terms denote constants. The rule is read as "If C is an RDF class and the label of C is X, then X is a classname." The second query returns the property names of all classes in an RDF configuration. This query, like the first, is expressed solely against the schema of the source. The third query below is expressed directly against data, and returns the URI of all RDF resources used as a property in at least one triple, where the resource may or may not be associated with schema.

classname(X) ← c-inst(C, class), C.hasLabel:X.
hasProp(X, Y) ← c-inst(C, class), c-inst(P, prop), P.hasDomain:C, C.hasLabel:X, P.hasLabel:Y.
dataprop(X) ← c-inst(T, triple), T.hasPred:P, P.hasURI:X.

The following three queries are similar to the previous three, but are expressed against an XML configuration. The first query finds the names of all available element types in the source, the second finds, for each element-type name, its corresponding attribute-definition names, and the last finds all available attribute names as a data query.

elemtype(X) ← c-inst(E, elemType), E.hasName:X.
atttype(X,Y) ← c-inst(E, elemType), E.hasName:X, E.hasAtts:AL, AT∈AL, AT.hasName:Y.
atts(X) ← c-inst(A, attribute), A.hasName:X.

Finally, the following query returns all constructs that serve as *struct-ct* schema constructs and their component selectors. This query is solely expressed against the data-model constructs.

schemastruct(SC, P) $\leftarrow$ ct-inst(SC, struct-ct), conf(DC, SC, X, Y), SC.P->C.

4 Navigation Operators

The ULD presents a complete, highly detailed description of a data source, with interconnected model, schema, and data information. In the ULD, each construct type, construct, and instance is represented by an id and every id, in turn, has an associated value. The value can be either an atomic value (such as a literal in RDF) or a structured value (such as a set or bag of ids).

We view navigation as a process of traversing a graph consisting of *locations* (nodes) and *links* (bi-directional edges), superimposed over a ULD source file. A location is either a construct type, construct, or instance in the ULD; thus a location is anything with an id. A link is a (simple or compound) path, from one location to another, through the connections in the ULD.

A navigation binding consists of an implementation for the following functions. For a binding, we assume a finite set of location names $\mathcal{L}$ and a finite set of link names $\mathcal{N}$, where both $\mathcal{L}$ and $\mathcal{N}$ consist of atomic string values. Navigation consists of moving from one location name to another. The binding should include only those locations that are meaningful to the intended user community, with appropriate links.

Starting Points. The operator *sloc* : $\mathcal{P}(\mathcal{L})$ returns all available entry points into an information source. We require the result of *sloc* to be a set of locations (as opposed to links). Note that $\mathcal{P}(\mathcal{L})$ stands for the power set of $\mathcal{L}$.

Links. The operator *links* : $\mathcal{L} \rightarrow \mathcal{P}(\mathcal{N})$ returns all out-bound links available from a particular location. For some locations, there may not be any links available, i.e., the *links* operator may return the empty set.

Following Links. The operator *follow* : $\mathcal{L} \times \mathcal{N} \rightarrow \mathcal{P}(\mathcal{L})$ returns the set of locations that are at the end of a given link. We use the *follow* operator to prepare to move to a new location from our current location. Given the set of locations returned by the *follow* operator, the user or agent directing the navigation can choose one as the new location.

Types. The operator *types* : $\mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$ returns the (possibly empty) set of types for a given location. We use the *types* operator to obtain locations that represent the schema for a data item. A particular location may have zero, one, or many associated types.

Extents. The operator *extent* : $\mathcal{L} \rightarrow \mathcal{P}(\mathcal{L})$ returns the (possibly empty) set of instances for a given location. The *extent* operator computes the inverse of the *types* operator.

As a simple example, the following (partial) navigation binding can be defined for the data shown in Figure 6

sloc = {‘class’, ‘property’, ‘subClass’, ‘resource’, ‘triple’, ...}
extent(‘class’) = {‘film’, ‘comedy’, ‘thriller’, ...}
links(‘thriller’) = {‘title’}
extent(‘thriller’) = {‘#m1’, ...}
follow(‘#m1’, ‘title’) = {‘The Usual Suspects’}

We express the navigation functions in Datalog using the predicates described below. An operator binding is a set of ULD queries, where the head of each query is a navigation operation (expressed as a predicate). Thus, operator bindings are defined as global-as-view mappings from the ULD (typically, over any configuration of a data model) to the navigation operations. We propose two ways to specify a navigation binding: as a set of low-level ULD queries and as a high-level specification that is used to automatically generate the appropriate navigation bindings.

- *sloc*(r), where r represents a starting location.
- *links*(l, k), where k is a link from location l.
- *follow*(l, k, r), where r is a location that is found by following link k from some location l.
- *types*(l, r), where l is a location of type r.
- *extent*(l, r), where r is in the extent of l.

To illustrate, the following low-level binding queries present a view of an RDF configuration, where we only allow navigation from data items with corresponding schema. Thus, this example supports simple browsing. The starting locations are classes and the available links from a class are its associated properties and its associated instances. The definition uses an additional intensional predicate *subClassClosure* for computing the transitive closure of the RDF subclass relationship.

```
sloc(L)           ← c-inst(C, class), C.hasLabel:L.
links(L, K)       ← c-inst(C, class), C.hasLabel:L, c-inst(P, property),
                    P.hasLabel:K, P.hasDomain:C′, subClassClosure(C, C′).
links(L, K)       ← c-inst(C, class), d-inst(O, C), O.hasURI:L, c-inst(T, triple), T.hasPred:X,
                    T.hasSubj:O, c-inst(X, property), X.hasLabel:K, X.hasDomain:C′,
                    subClassClosure(C, C′).
follow(L, K, R) ← c-inst(C, class), C.hasLabel:L, c-inst(T, property), T.hasLabel:K,
                    T.hasDomain:C′, T.hasRange:C″, C″.hasLabel:R, subClassClosure(C, C′).
follow(L, K, R) ← c-inst(C, class), C.hasLabel:L, c-inst(T, property), T.hasLabel:K,
                    T.hasDomain:C′, T.hasRange:R, R=‘literal’, subClassClosure(C, C′).
follow(L, K, R) ← c-inst(C, class), d-inst(O, C), O.hasURI:L, c-inst(T, triple), T.hasPred:X,
                    T.hasSubj:O, T.hasObj:R, c-inst(R, literal), c-inst(X, property), X.hasLabel:K,
                    X.hasDomain:C′, subClassClosure(C, C′).
follow(L, K, R) ← c-inst(C1, class), d-inst(O1, C1), O1.hasURI:L, c-inst(T, triple), T.hasPred:X,
                    T.hasSubj:O1, T.hasObj:O2, O2.hasURI:R, c-inst(X, property), X.hasLabel:K,
                    X.hasDomain:C′, subClassClosure(C, C′).
extent(L, R)      ← c-inst(C, class), C.hasLabel:L, d-inst(O, C), O.hasURI:R.
type(L, R)        ← c-inst(C, class), C.hasLabel=R, d-inst(O, C), O.hasURI:L.

subClassClosure(C, C)     ← c-inst(C, class).
subClassClosure(C1, C3) ← c-inst(S, subClass), S.hasSub:C1, S.hasSuper:C2,
                              subClassClosure(C2, C3).
```

In general, with low-level binding queries a user can specify detailed and exact descriptions of the navigation operations for data sources. To specify higher-level bindings, a user selects certain constructs as locations and certain other constructs as links. Using this specification, the navigation operators are automatically computed by traversing the appropriate instances of locations and links in the configuration. Figure 7 shows

RDF Binding:

$B = (L, N, S, F)$
$L =$ class, resource, literal
$N =$ property, triple
$S =$ class
$F =$ triple : resource [triple/hasSubj] ⇒ literal [triple/hasObj],
triple : resource [triple/hasSubj] ⇒ resource [triple/hasObj],
property : class [property/hasDomain] ⇒ class [property/hasRange],
property : class [property/hasDomain] ⇒ literal [property/hasRange]

name(X, N) ← c-inst(X, class), X.hasLabel:N.
name(X, N) ← c-inst(X, resource), X.hasURI:N.
name(N, N) ← c-inst(N, literal).
name(X, N) ← c-inst(X, property), X.hasLabel:N.
name(X, N) ← c-inst(X, triple), X.hasPred:R, c-inst(R, property), R.hasLabel:N.

Fig. 7. A high-level binding for simple navigation of RDF.

an example of a high-level binding definition for RDF, where RDF classes, resources, and literals are considered sources for locations and RDF properties and triples are considered sources for links (Figure 10 shows a similar binding for XML, which we discuss later).

We define a high-level binding specification as a tuple (L, N, S, F). The disjoint sets L and N consist of construct identifiers such that L is the set of constructs used as locations and N is the set of constructs used as links. The set $S \subseteq L$ gives the entry points of the binding. Finally, the set F contains *link definitions* (described below).

Each construct in L and N has an associated naming definition that describes how to compute the name of an instance of the construct. The name would typically be viewed by the user during navigation. The naming definitions serve to map location and link instances to appropriate string values. For example, in Figure 7, RDF classes and properties are named by their labels, resources are named by their URI values, a literal value is used directly as its name, and the name of a triple is the name of its associated predicate value.

The incremental operators in a high-level binding specification are computed automatically by traversing connected instances. We define the following generic rules to compute when two instances are connected. (Note that these connected rules only perform single-step traversal, and can be extended to allow an arbitrary number of steps, which we discuss at the end of this section.)

connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, struct-ct), X_1.S:X_2.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, bag-ct), $X_2 \in X_1$.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, list-ct), $X_2 \in X_1$.
connected(X_1, X_2) ← c-inst(X_1, C), ct-inst(C, set-ct), $X_2 \in X_1$.
connected(X_2, X_1) ← connected(X_1, X_2).

A connected formula is true when there is a structural connection between two instances. Note that the rules above do not consider the case when two items are linked by a $d\text{-}inst$ relationship. Instead, the $d\text{-}inst$ relationship is directly used by the *types* and *extent* operators, whereas connections are used by the *links* and *follow* operators.

sloc(R)	$\leftarrow$ B_S(S), c-inst(X, S), name(X, R).
links(L, K)	$\leftarrow$ name(X, L), connected(X, Y), B_F(F), linkSource(F, X, Y), name(Y, K).
follow(L, K, R)	$\leftarrow$ name(X, L), connected(X, Y), B_F(F), linkSource(F, X, Y), name(Y, K), connected(Y, Z), linkTarget(F, Y, Z), name(Z, R).
type(L, R)	$\leftarrow$ name(X, L), c-inst(X, P), B_L(P) d-inst(X, Y), c-inst(Y, Q), B_L(Q), name(Y, R).
extent(L, R)	$\leftarrow$ name(Y, L), c-inst(Y, Q), B_L(Q), d-inst(X, Y), c-inst(X, P), $B_L(P)$, name(X, R).

Fig. 8. Datalog rules to compute links and locations.

The link definitions of F have the form $c_k : c_1[p_1] \Rightarrow c_2[p_2]$ where:

– The behavior of the link construct $c_k \in N$ is being described by the rest of the expression. For example, in Figure 7, the first link definition is for triple constructs.
– For $c_1, c_2 \in L$, the construct c_k can serve to link an instance of c_1 to an instance of c_2. Thus, we can traverse from instances of c_1 to instances of c_2 via an instance of c_k. For example, in Figure 7, the first link definition says that we can follow a resource instance to a literal instance if they are connected by a triple.
– The expressions p_1 and p_2 further restrict how instances of c_k can be used to link c_1 and c_2 instances, respectively. For example, in Figure 7, the first link definition states that triples link resources and literals through the triple's hasSubj and hasObj selector, respectively.

We define the *linkSource*(f, i_1, i_k) and *linkTarget*(f, i_k, i_2) clauses as follows. Given a link definition $f \in F$ and a connection from i_1 to i_k such that connected(i_1, i_k) is true (where i_k is the link instance), *linkSource*(f, i_1, i_k) is true if $f = c_k : c_1[p_1] \Rightarrow c_2[p_2]$ such that i_1 is an instance of c_1 (that is, c-inst(i_1, c_1) is true), i_k is an instance of c_k (that is, c-inst(i_k, c_k) is true where c_k is a link construct), and i_1 and i_k are connected according to the expression p_1. Similarly, for an $f \in F$, *linkTarget*(f, i_k, i_2) is true if $f = c_k : c_1[p_1] \Rightarrow c_2[p_2]$ such that i_k is an instance of c_k (a link construct), i_2 is an instance of c_2, and i_k and i_2 are connected according to the expression p_2.

Given the above definitions, we automatically compute each navigation operator using the Datalog queries in Figure 8. We assume each operator is represented as an intensional predicate (as before) and the binding specification $B = (L, N, S, F)$ is stored as a set of unary extensional predicates B_L, B_N, B_S, and B_F. For example, the expression $B_L(X)$ binds X to a location in L for binding B. We also assume that the *name* predicate is stored as an intensional formula (as defined in the binding).

The first rule in Figure 8 finds the set of entry points: It obtains a construct in the set of starting locations, finds an instance of the construct, and then computes the name of the instance. The second rule finds the locations with links. For each named instance in the configuration that is connected to another instance, we use the *linkSource* predicate to check if it is a valid connection, we check to make sure that the link instance (represented as the variable Y) is valid, and then compute the name of the instance. The third rule is similar to the second, except it additionally uses the *linkTarget* to determine the new location. Finally, the last two rules use the *d-inst* relationship to find types and extents, respectively.

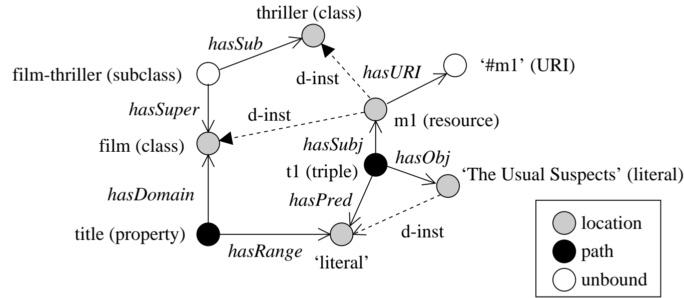


Fig. 9. An example of labeled instances for RDF.

To demonstrate the approach, we use the binding definition of Figure 7 and the sample configuration of Figure 9. This configuration shows part of Figure 1 as a graph whose nodes are construct instances and edges are either connections between structures or $d\text{-}inst$ links. Consider the following series of invocations.

1. *sloc* = {'film','thriller'}. According to the binding definition, the *sloc* operator returns all the labels of *class* construct instances. As shown in Figure 9, the only *class* construct instances are *thriller* and *film*.
2. *links*('film') = {'title'}. The *links* operator is computed by considering each connected construct instance of *film* until it finds a construct instance whose associated construct is in N. As shown, the only such instance is *title*, which is an RDF property.
3. *extent*('film') = {'#m1'}. The *extent* operator looks for the *d-inst* links of the given instance. As shown, the only such link for *film* is to *m1*.
4. *follow*('#m1', 'title') = {'The Usual Suspects'}. The *follow* operator starts in the same way as the *links* operator by finding instances (whose constructs are in N) that are connected to the given instance. For the given item, the only such instance in Figure 9 is *t1*. The *follow* operator then returns the *hasObj* component of *t1*, according to the link definition for *triple*.

Note that for this example, the rules for computing the *follow* and *links* operators only consider instances that are directly connected to each other (through the *connected* predicate). Thus, invoking the operator links('thriller') will not return a result (for our example) because there are no link construct instances directly connected to the associated RDF class. However, the properties of a class in RDF also include the properties of its superclasses. We can include such information by expanding the set of connection rules. One approach is to allow the navigation specifier to add a connected rule specifically for the subclass case. Alternatively, we can extend the connection definition to compute the transitive closure (of connections) using the following rule.

connected(X_1, X_3) $\leftarrow$ connected(X_1, X_2), connected(X_2, X_3).

We also allow binding specifications to include multiple-step path expressions in F. For example, we add the following link definitions to the RDF binding specification to correctly support subclasses.

XML Binding:

$B = (L, N, S, F)$
$L =$ elemType, element, cdata, pcdata
$N =$ attDef, attribute, contentDef, content
$S =$ elemType
$F =$ attDef : elemType [elemType/hasAtts/] $\Rightarrow \{\}$,
attribute : element [element/hasAtts/] $\Rightarrow$ cdata [attribute/hasVal],
contentDef : elemType [elemType/hasModel] $\Rightarrow$ elemType [contentDef/],
content : element [element/hasChildren] $\Rightarrow$ element [content/],
content : element [element/hasChildren] $\Rightarrow$ pcdata [content/]

name(X, N) $\longleftarrow$ c-inst(X, elemType), X.hasName:N.
name(X, N) $\longleftarrow$ c-inst(X, element).
name(X, X) $\longleftarrow$ c-inst(X, cdata).
name(X, X) $\longleftarrow$ c-inst(X, pcdata).
name(X, N) $\longleftarrow$ c-inst(X, attDef), X.hasName:N.
name(X, N) $\longleftarrow$ c-inst(X, attribute), X.hasName:N.
name(X, N) $\longleftarrow$ c-inst(X, contentDef), N='hasChildType'.
name(X, N) $\longleftarrow$ c-inst(X, content), N='hasChild'.

Fig. 10. A high-level binding for direct navigation of XML.

property : class [property/hasDomain/hasSuper/hasSub] $\Rightarrow$ class [property/hasRange]
property : class [property/hasDomain/hasSuper/hasSub] $\Rightarrow$ literal [property/hasRange]

Finally, Figure 10 shows a high-level binding specification for the XML data model of Figure 4. The binding specification assumes the transitive connection relation defined above. Element types, elements, and atomic data serve as locations, with element types as starting locations. Attribute definitions, attributes, content definitions, and content serve as links. We use 'hasChildType' and 'hasChild' strings as the names of the links for content definitions and element content, respectively. Note the attDef link definition is a special case in which attDef links always lead to an empty set of locations (denoted using the empty set in Figure 10). Also, the ending '/' in a link-definition path denotes traversal into the elements of a collection structure (as opposed to denoting the collection structure itself).

5 Related Work

A number of approaches provide browsing capability for traditional databases. Motro [14] seeks to enable users who are (1) not familiar with the data model of the system, (2) not familiar with the organization of the database (i.e., the schema), (3) not proficient with the use of the system (i.e., the query language), (4) not sure what data they are looking for (but are looking for something interesting or suitable), and/or (5) not clear how to construct the desired query. As more structured information finds its way on the Web, we believe these issues become more pressing for users as well as for software agents wishing to exploit structured information.

Database browsing typically assumes a fixed data model [2, 14, 12, 17, 3, 8, 18] (either relational, E-R, or object-oriented). Only a few systems allow browsing schema and data in isolation [3, 12, 14], where most support browsing data only through schema (i.e., navigating data using items of the schema). Hypertext systems, including those

with structured data models, also use browser-based interfaces [15, 13]. These systems, as in database approaches, are developed for a single data model, and support limited browsing styles.

The links and locations abstraction used by incremental navigation is similar in spirit to the graph-based model of RDF and RDF Schema. The Object Exchange Model (OEM)—the semi-structured representation of TSIMMIS [11, 16]—is another simple abstraction. Both TSIMMIS and some database browsing systems [2, 9, 14, 8] support user navigation mixed with user queries, which we would like to explore as an extension to our current navigation operators. Finally, Clio [18] provides some support for navigation, specifically to help users build data-transformation queries. Clio supports "data walks," which display example data involved in each potential join path between two relations, and "data chases," which display all occurrences of a specific value within a database.

6 Conclusion and Future Work

We believe incremental navigation provides a simple, generic abstraction, consisting of *links* and *locations* (and *types* and *extents* when applicable), that can be applied over arbitrary data models. More than that, with the high-level binding approach, it becomes relatively straightforward to specify the links and locations for a data model, thus enabling generic and uniform access to information represented in any underlying data model (described in the ULD). We believe that this approach can be extended beyond navigation to include querying information sources (i.e., querying links and locations) and for specifying high-level mappings between data sources.

We have implemented a prototype browser [5] to demonstrate incremental navigation, both for data-model aware and simple navigation of RDF, XML, Topic Map, and relational sources. In addition, some of the ideas of incremental navigation appear in the Superimposed Schematics browser [7], which allows users to incrementally navigate an ER schema and data source. Based on these experiments, we believe incremental navigation is viable, and helps reduce the work required to develop such browsing tools.

For future work, we intend to investigate whether additional ULD information can be used, such as data-model constraints, to help validate and generate operator bindings. We are also interested in defining a language to express path-based queries over the links and locations abstraction offered by incremental navigation. One issue is to determine whether algorithms and optimizations can be defined to efficiently compute (i.e., unfold) the binding-specification rules to answer such path queries. Finally, we believe that the incremental-navigation operators can be easily expressed as a standard web-service interface (where information sources have corresponding web-service implementations), providing generic, web-based access to heterogeneous information.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Company, 1995.

2. B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, and P. Sudarshan. BANKS: Browsing and keyword searching in relational databases. In *Proceedings of the Twenty-Eighth Very Large Data Bases (VLDB) Conference*, 2002.
3. R. Agrawal, N. Gehani, and J. Srinivasan. OdeView: The graphical interface to Ode. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 34–43, 1990.
4. S. Bowers. *The Uni-Level Description: A Uniform Framework for Managing Structural Heterogeneity*. PhD thesis, OGI School of Science and Engineering, OHSU, December 2003.
5. S. Bowers and L. Delcambre. JustBrowsing: A generic API for exploring information. In *Demo Session at the 21st International Conference on Conceptual Modeling (ER)*, 2002.
6. S. Bowers and L. Delcambre. The uni-level description: A uniform framework for representing information in multiple data models. In *Proceedings of the 22nd International Conference on Conceptual Model (ER)*, volume 2813 of *Lecture Notes in Computer Science*, pages 45–58. Springer-Verlag, 2003.
7. S. Bowers, L. Delcambre, and D. Maier. Superimposed schematics: Introducing E-R structure for *in-situ* information selections. In *Proceedings of the 21st International Conference on Conceptual Model (ER)*, volume 2503 of *Lecture Notes in Computer Science*, pages 90–104. Springer-Verlag, 2002.
8. M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An integrated query/browser for object databases. In *Proceedings of 22nd International Conference on Very Large Data Bases (VLDB)*, pages 203–214. Morgan Kaufmann, 1996.
9. T. Catarci, G. Santucci, and J. Cardiff. Graphical interaction with heterogeneous databases. *The VLDB Journal*, 6(2):97–120, 1997.
10. W. W. Cohen. Some practical observations on integration of web information. In *Informal Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 55–60, 1999.
11. J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, page 483. ACM Press, 1995.
12. M. Kuntz and R. Melchart. Ergonomic schema design and browsing with more semantics in the Pasta-3 interface for E-R DBMSs. In *Proceedings of the Eight International Conference on Enity-Relationship Approach*, pages 419–433, 1989.
13. C. C. Marshall, F. M. Shipman III, and J. H. Coombs. VIKI: Spatial hypertext supporting emergent structure. In *European Conference on Hypertext Technology (ECHT)*, pages 13–23. ACM Press, 1994.
14. A. Motro. BAROQUE: A browser for relational databases. *ACM Transactions on Office Information Systems*, 4(2):164–181, 1986.
15. J. Nanard and M. Nanard. Should anchors be typed too? An experiment with macweb. In *Proceedings of Hypertext*, pages 51–62, 1993.
16. Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 251–260. IEEE Computer Society, 1995.
17. T. Rogers and R. Cattell. Entity-Relationship databases user interfaces. In *Sixth International Conference on Entity-Relationship Approach*, pages 353–365, 1997.
18. L. L. Yan, R. J. Miller, L. M. Haas, and R. Fagin. Data-driven understanding and refinement of schema mappings. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM Press, 2001.