The Uni-Level Description: A Uniform Framework for Representing Information in Multiple Data Models[*]

Shawn Bowers and Lois Delcambre

OGI School of Science and Engineering at OHSU
Beaverton OR 97006, USA
{shawn, lmd}@cse.ogi.edu

Abstract. One advantage of having several different representation schemes and data models is that users can select the right representation and associated tools for their particular need. However, multiple representations introduce structural, model-based heterogeneity, making it difficult to combine information from different sources and exploit information using generic tools (e.g., for querying or browsing). In this work, we define a uniform representation based on a meta-data-model called the *Uni-Level Description* (ULD) that can accommodate and accurately store information in a broad range of data models. The ULD defines three distinct *instance-of* relationships plus a relationship for modeling *conformance*, which is used to connect (data) constructs to other (schema) constructs and can be constrained to reflect the requirements of the data model. The ULD has been shown to enable powerful, generic transformation rules and simple generic browsing capability over information represented in diverse data models and representation schemes.

1 Introduction

This work is motivated by a simple observation—that most data models and representation schemes use a small set of basic structures such as scalar data, tuple, and collection constructs, composed in various ways, to store information. The basic idea of our work is to describe the constructs of a data model using these basic structures and then instantiate them to describe the schema and data that is present in an information source. The goal of our work is to develop a generic representation that:

- can enable generic tools (i.e., tools that work over multiple data models);
- can describe a variety of data models;
- includes the data, schema(s) if present, and the data model of sources;
- is flat, much like RDF, so that data model, schema, and data can be accessed at the same time, e.g., in the same query;

Relational Schema

Movies

mid	title	runtime	genre	company

Cast

mid	actor	character

Review

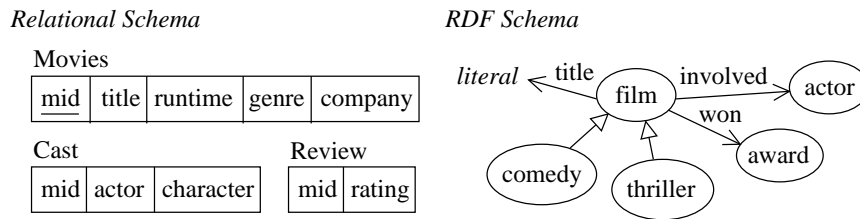| mid	rating

RDF Schema

Fig. 1. Similar schemas expressed in the relational and RDF data models.

- permits various descriptions of any particular data model;
- can describe information where the schema is missing, where it is partial (e.g., like an "open" DTD for XML), and where there are multiple levels of schema (e.g., where the type of one topic is another topic, and the type of that topic is yet another topic in a Topic Map);
- permits the definition and use of new, special-purpose data models.

Our representation is called the *Uni-Level Description* (ULD). One of the key characteristics of the ULD is the use of three, distinct *instance-of* relationships plus a *conformance* relationship. That is, the traditional *instance-of* or *type* relationship is not overloaded in the ULD (unlike in knowledge representation models such as RDF). The conformance relationship allows the configurer of a data model to specify the connection between a data construct definition (like the entity construct in the E-R model) with the corresponding schema construct definition (like the entity type construct in the E-R model). More than that, providing constraints on the conformance relation permits accurate description of a wide range of data models.

We focus on structural heterogeneity [11] of data models, as shown in Figure 1 and schemas, as shown in Figure 2. We envision an environment where arbitrary data sources are easily described using the ULD, permiting the use of generic, ULD-based tools that can work over arbitrary information sources, as shown in Figure 3. We have implemented a transformation facility [7] where powerful transformation rules expressed in Datalog can easily convert information from one source to another, in the presence of different schemas or data models. We have also implemented a browser [6], supported by a navigational API against the ULD, that permits both naive users (e.g., users that don't know what an XML element or attribute is) and sophisticated users (e.g., agents or crawlers) to access information sources in a generic manner.

This paper is organized as follows. The ULD is presented in Section 2 with a detailed description of the meta-data-model architecture and *conformance* and *instance-of* relationships. Section 3 describes a language for representing and accessing information sources using the ULD. Related work is presented in Section 4 and conclusions and future work are presented in Section 5.

```
Schema 1:
  <!ELEMENT movie (title+,studio,genre*,actor*)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT studio (#PCDATA)>
  <!ELEMENT genre (#PCDATA)>
  <!ELEMENT actor (#PCDATA)>
  <!ATTLIST actor role CDATA #REQUIRED>
Schema 2:
  <!ELEMENT movie (thriller|comedy)>
  <!ATTLIST movie name CDATA #REQUIRED>
  <!ELEMENT thriller (actor*,crew*)>
  <!ELEMENT comedy (actor*,crew*)>
  <!ELEMENT actor (#PCDATA)>
  <!ATTLIST actor role CDATA #REQUIRED>
  <!ELEMENT crew (#PCDATA)>
  <!ATTLIST crew role CDATA #REQUIRED>
```

Fig. 2. Heterogeneous XML DTDs that: (a) overlap, (b) are structurally different, and (c) are schematically different.
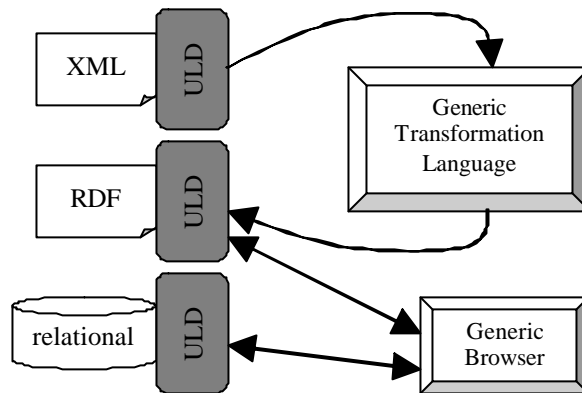

Fig. 3. ULD-enabled environment—generic tools can access information in various data models.

2   The Uni-Level Description Framework

The architecture used by most meta-data-model approaches [1–3, 8, 13, 9, 14] is shown in Figure 4. The architecture consists of four levels representing the meta-data-model, the meta-schema (i.e., the data model), the schema, and source data. A meta-schema represents the set of structural *constructs* provided by a data model; and schemas are represented as instantiations of the meta-schema constructs. For example, the meta-schema to describe the relational model would
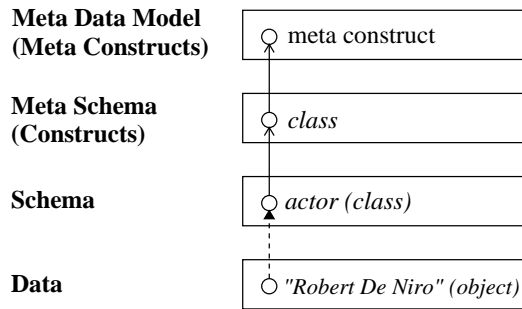
Fig. 4. The typical meta-data-model architecture.

consist of various data structures for representing relation types, their attribute types, primary and foreign keys, and so on.

A number of architectures use E-R, entity and relationship structures to define meta-schemas and their corresponding schemas [1–3, 8, 13, 14]. In this case, meta-schema constructs are defined as patterns, or compositions of entity and relationship types. These types are used to define entities and relationships representing corresponding schema items. With this approach, a meta-schema for the object-oriented data model would contain a class construct represented as an entity type (called *class*) with a name attribute and relationships to other constructs (e.g., that represent class attributes). A particular object-oriented schema is then represented as a set of entities and relationships that instantiate these meta-schema types. We note that in most meta-data-model architectures, data (the bottom level of Figure 4) is not explicitly represented—data is assumed to be stored outside of the system and is not directly represented in the architecture.

Traditional database systems generally require *complete schema* in which all data must satisfy all of the constraints imposed by the schema. Typical meta-data-model approaches assume data models require complete schema, and is reflected in the architecture of Figure 4, which assumes data follows from (only) the schema level.

2.1   The ULD Architecture

In contrast to Figure 4, the ULD uses the three-level architecture shown in Figure 5. The ULD meta-data-model (shown as the top level) consists of *construct types* (i.e., meta-constructs) that denote structural primitives. The middle level uses the structural primitives to define both data and schema constructs as well as possible conformance relationships among them.

The ULD architecture distinguishes three kinds of *instance-of* relationships. Constructs introduced in the middle layer are necessarily an instance (*ct-inst*) of a construct type in the meta-data-model. Similarly, every item introduced in
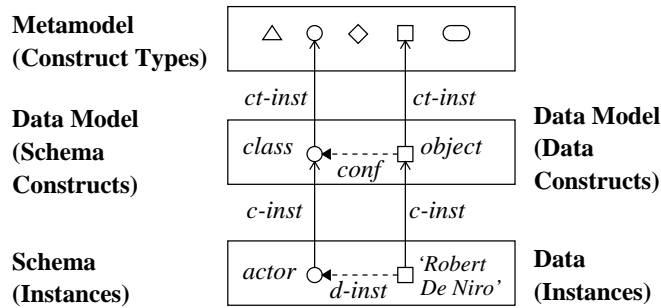
Fig. 5. The ULD meta-data-model architecture.

the bottom layer is necessarily an instance (*c-inst*) of a construct in the middle layer. Finally, a data item in the bottom layer can be an instance (*d-inst*) of another data item, as allowed by *conformance* relationships in the middle layer. The ULD is able to represent a wide range of data models using the flexibility offered by the *conf* and *d-inst* relationships. For example, in XML, conformance between elements and element types is optional. Thus, an XML element without a corresponding element type would not have a *d-inst* link to an XML element type as part of a partial or optional schema.

The ULD represents an information source as a *configuration*, which contains the construct types of the meta-data-model, the constructs of a data model, the instances (both schema and data) of a source, and the associated instance-of relationships. Thus, a configuration can be viewed as an instantiation of Figure 5. Each configuration uses a finite set of identifiers for denoting construct types, constructs, and instances as well as a finite set of *ct-inst*, *c-inst*, *conf*, and *d-inst* relationships. Together, the use of identifiers with explicit instance-of relationships allows all three levels of Figure 5 to be stored in a single configuration (i.e., as a single level)—enabling direct and uniform access to all information in a source.

In a configuration, all construct and instance identifiers have exactly one associated value, where a value is a particular instantiation of a structure (e.g., a tuple or collection value). Primitive values such as strings, integers, and Booleans are treated as subsets of the data-item identifiers in a configuration.[1] The primitive structures defined in the ULD meta-data-model include tuples (lists of name-value pairs), collections (set, list, and bag), atomics (for primitive values like strings and integers), and unions (for representing union types, i.e., a non-structural, generalization relationship among types). The construct type identifiers are denoted *set-ct*, *list-ct*, *bag-ct*, *struct-ct*, *atomic-ct*, and *union-ct* representing collection, tuple, atomic, and union structures, respectively.

---

[1] This choice is primarily for enabling a more concise representation—primitive values could also be represented as distinct sets, disjoint from identifiers.

3 The ULD Representation Language

3.1 Representing Data Models in the ULD

Figures 6, 7, and 8 use the language of the ULD to describe the relational, XML, and RDF data models, respectively. Note that there are potentially many ways to describe a data model in the ULD, and these examples show only one choice of representation. As shown, a construct definition can take one of the following forms.

`construct` c = $\{a_1\text{->}c_1, a_2\text{->}c_2, \ldots, a_n\text{->}c_n\}$ `conf(domain=`x`,range=`y`):`c'
  This expression defines a tuple-construct c as a *ct-inst* of construct type *struct-ct*, where a_1 to a_n are unique strings, $n \geq 1$, and c_1 to c_n and c' are construct identifiers. Each expression $a_i = c_i$ is called a *component* of the construct c where a_i is called the component *selector*. If the *conf* expression is present, instances of c may conform (i.e., be connected by a *d-inst* relationship) to instances of c' according to the domain and range constraints on the expression. If the conformance expression is not present, conformance is not permitted for the construct (i.e., there cannot be a *d-inst* relationship between the construct instances). The cardinality constraints on conformance (x and y above) restrict the participation of associated instances in *d-inst* relationships to either exactly one (denoted as `1`), zero or one (denoted as `?`), zero or more (denoted as `*`), or one or more (denoted as `+`) for both the domain and range of the relationship.

`construct` c = `set of` c_1 `conf(domain=`x`,range=`y`):`c'
  This expression defines a set-construct c as a *ct-inst* of construct type *set-ct*. The definition restricts instances of c to sets whose members must be instances of construct c_1. In addition, for instances of c, each member must have a unique identifier. Bag types and list types are defined similarly. These constructs can also contain conformance definitions.

`construct` c = c_1 `|` c_2 `|` `...` `|` c_n
  This expression defines a union-construct c as a *ct-inst* of construct type *union-ct*, where c_1 to c_n are distinct construct identifiers for $n \geq 2$ such that for all i from 1 to n, $c \neq c_i$. All instances of c_1 to c_n are considered instances of c, however, we do not allow instances of only c directly. A union construct provides a simple mechanism to group heterogeneous structures (e.g., atomics and structs) into a single, union type, as opposed to *isa* relationships, which offer inheritance semantics, and group like structures (e.g., class-like structures).

`construct` c = `atomic conf(domain=`x`,range=`y`):`c'
  This expression defines an atomic type c as a *ct-inst* of construct type *atomic-ct*. These constructs can also contain a conformance definition.

  As shown in Figure 6, tables and relation types are one-to-one such that each table conforms to exactly one relation type. Each tuple in a table must conform (as shown by the range restriction 1) to the table's associated relation type. We assume each relation type can have at most one primary key.

```
% schema constructs
construct relation  = {relname->string, atts->attlist}
construct attlist   = list of attribute
construct attribute = {aname->string, attof->relation, domain->valuetype}
construct pkey      = {pkeyof->relation, keyatts->attlist}
construct fkey      = {fkeyof->relation, ref->relation, srcatts->attlist}
% instance constructs
construct table     = bag of tuple conf(domain=1,range=1):relation
construct tuple     = list of value conf(domain=*,range=1):relation
```

Fig. 6. Description of the relational data model in the ULD.

```
% schema constructs
construct pcdata     = atomic
construct cdata      = atomic
construct elemtype   = {name->string, atts->attdefset, cmodel->contentdef}
construct attdefset  = set of attdef
construct attdef     = {name->string, attof->elemtype}
construct contentdef = set of elemtype
% data constructs
construct element    = {tag->string, atts->attset, children->content}
                         conf(domain=*,range=?):elemtype
construct attset     = list of attribute
construct attribute  = {name->string, attof->element, val->cdata}
                         conf(domain=*,range=?):attdef
construct content    = list of node
construct node       = element | pcdata
```

Fig. 7. Simplifed ULD description of the XML/DTD data model.

The XML data model shown in Figure 7 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).

Finally, Figure 8 shows the RDF(S) data model expressed in the ULD, and includes constructs for classes, properties, resources, and triples. In RDF, `rdf:type`, `rdf:subClassOf`, and `rdf:subPropertyOf` are considered special RDF properties for denoting instance and specialization relationships. However, we model these properties using conformance and explicit constructs (i.e., with subclass and subprop). Therefore, RDF properties in the ULD represent regular relationships; we do not overload them for type and subclass/subproperty def-

```
construct literal  = atomic
construct resource = {val->uri} conf(domain=*,range=*):class
construct class    = {rid->resource, label->string}
construct prop     = {rid->resource, label->string, domain->class,
                        range->rangeval}
construct rangeval = class | valuetype
construct subclass = {super->class, sub->class}
construct subprop  = {super->prop, sub->prop}
construct triple   = {pred->resource, subj->resource, obj->objval}
                      conf(domain=*,range=*):prop
construct objval   = resource | literal
```

Fig. 8. (Simplified ULD description of the RDF(S) data model.

initions. This approach does not limit the definition of RDF; partial, optional, and multiple levels of schema are still supported.

ULD configurations are populated with default constructs representing typical primitive value types, such as *string*, *Boolean*, *integer*, *url*, etc., as instances of *atomic-ct*. The *valuetype* and *value* constructs are special constructs that work together to provide a mechanism for data models to permit user-defined primitive types (e.g., to support XML Schema basic types or relational domains). The *value* construct is defined as the union (i.e., a union construct) of all defined *atomic-ct* constructs. Thus, when a new *atomic-ct* construct is created, it is automatically added to *value*'s definition. The *valuetype* construct has an instance with the same name (represented as a string value) as each construct of *value*. To add a new primitive type (e.g., a date type), we create a new construct (with the identifier *date*), add it as a member of *value* (recall *value* is a union construct), and create a new *valuetype* instance 'date,' connected by *c-inst*.

3.2 Representing Instances in the ULD

Examples of schema and data expressed in the ULD for XML and RDF are shown in Figures 9 and 10, respectively. Figure 9 gives (a portion of) the first XML DTD of Figure 2 and Figure 10 gives (a portion of) the RDF schema of Figure 1. As shown, expressions for defining instances take the form: i = c v d-inst:$i_1, i_2, \ldots, i_l$, where c is a construct, v is a valid value for construct c, and i_1 to i_l are instance identifiers for $l \geq 0$. The expression defines i as a construct instance (*c-inst*) of c with value v. Further, i is a data instance (*d-inst*) of i_1 to i_l, which must be instances of the construct(s) c conforms to. The d-inst expression is not present if i is not a data instance of another instance (i.e, $l = 0$).

Given the construct c, we say $c \in ct\text{-}inst(ct)$ is true if and only if c is defined as an instance (*ct-inst*) of construct type ct. Similarly, the expressions $i \in c\text{-}inst(c)$ and $i \in d\text{-}inst(i')$ are true if and only if i is defined as an instance (*c-inst*) of construct c and a data instance (*d-inst*) of i'. Finally, the relation $conf(c, c', x, y)$

is true if and only if a conformance relationship is defined from c to c' (i.e., instances of c are allowed to be data instances of c'). The domain constraint of the conformance relationship is x, and the range constraint is y. An instance is *valid* for a configuration if it is both well-formed and satisfies the constraints of its associated construct. We note that each data-level identifier can only be defined once in a configuration, and must have exactly one associated construct (i.e., must be a *c-inst* of exactly one construct). The following constraints are imposed by constructs on their instances.

Value Definition. If i is an instance of construct c (i.e., $i \in$ *c-inst*(c)) and c is a *struct-ct* construct (i.e., $c \in$ *ct-inst*(*struct-ct*)) then the value of i must contain the same number of components as c, each component selector of i must be a selector of c, and each component value of i must be an instance of the associated component construct value for c. Alternatively, if c is a collection construct, then the value of i must be a collection whose members are instances of the construct that c is a collection of, and if c is a set, i's elements must have unique identifiers.

Conformance Definition. If i is an instance of construct c (i.e., $i \in$ *c-inst*(c)), i' is an instance of construct c' (i.e., $i' \in$ *c-inst*(c')), and c can conform to c' (i.e., *conf*(c, c', x, y) is true for some x, y), then i is allowed to be a data instance (*d-inst*) of i'. Further, domain and range cardinality constraints on conformance can restrict allowable data-instance relationships. Namely, if the domain constraint on the conformance relationship is + (i.e., $x =$+), i must participate in at least one data-instance relationship (i.e., there must exist an i' where $i \in$ *d-inst*(i')). For a domain constraint of 1, i must be a data instance of exactly one i'. Similarly, if the range constraint on the conformance relationship is + (i.e., $y =$+), every i' must have at least one associated i (i.e., for each i' there must exist an i where $i \in$ *d-inst*(i')). For a range constraint of 1, exactly one i must be a data instance of i'.

3.3 Querying ULD Configurations

Information can be accessed in the ULD through queries against configurations. A query is expressed as a range-restricted Datalog program (i.e., a set of Datalog rules). A rule body consists of a set of conjoined ULD expressions possibly containing variables. For example, the following query finds all available class names within an RDF configuration. (Note that upper-case terms denote variables and lower-case terms denote constants.)

classes(X) $\leftarrow$ C $\in$ *c-inst*(class), C.label=X.

The following formulas are allowed in the body of ULD rules. The membership operator $\in$ is used to access items in the sets *ct-inst*, *c-inst*, or *d-inst*. For example, the expression C $\in$ *c-inst*(*class*) above finds RDF class identifiers C in the given configuration. In addition, the membership operator can access elements in collection structures, where an expression of the form $v \in x$ is true if

```
movie   = elemtype {name:'movie', atts:nilad, cmodel:moviecm}
nilad   = attdefset []
moviecm = contentdef [title, studio, genre, actor]
title   = elemtype {name:'title', atts:nilad, cmodel:nilcm}
nilcm   = contentdef []
genre   = elemtype {name:'genre', atts:nilad, cmodel:nilcm}
actor   = elemtype {name:'actor', atts:actorat, cmodel:nilcm}
actorat = attdefset [role]
role    = attdef {name:'role', attof:actor}
e1      = element {tag:'movie', atts:nilas, children:e1cnt} d-inst:movie
nilas   = attset []
e1cnt   = content [e2,e3,e4]
e2      = element {tag:'title', atts:nilas, children:e2cnt} d-inst:title
e2cnt   = content ['Usual Suspects']
e3      = element {tag:'genre', atts:nilas, children:e3cnt} d-inst:genre
e3cnt   = content ['thriller']
e4      = element {tag:'actor', atts:e4as, children=e4cnt} d-inst:actor
e4as    = attset [a1]
e4cnt   = content ['Kevin Spacey']
a1      = attribute {name:'role', attof:e4, val:'supporting'} d-inst:role
...
```

Fig. 9. Sample XML data and schema (DTD) expressed in the ULD.

```
film    = resource {val:'#film'}
title   = resource {val:'#title'}
comedy  = resource {val:'#comedy'}
filmc   = class {rid:film, label:'film'}
titlep  = prop {rid:title, label:'title', domain:filmc, range:'literal'}
comedyc = class {rid:comedy, label:'comedy'}
fc      = subclass {super:filmc, sub:comedyc}
f1 = resource {val:'http://.../review.html'} d-inst:comedyc,filmc
t1 = triple {pred:title, subj:f1, obj:'Meet the Parents'} d-inst:titlep
...
```

Fig. 10. Sample RDF(S) data expressed in the ULD.

v is a member of x's value and x is an instance of a set, list, or bag construct. Both v and x can be variables (bound or unbound) or constants. Finally, the expression $x.y = v$ (alternatively, $x \rightarrow y = v$) can be used to access data components (component definitions), where x is a data identifier (construct identifier), y is a component selector of x (component selector definition of x), and v is the value of the component (construct of the component definition). The terms x, y, and v can be either variables or constants.

1. elemtypes(X)   ← E ∈ *c-inst*(elemtype), E.name=X.
2. atttypes(X,Y) ← A ∈ *c-inst*(attdef), A.name=Y, A.attof=E, E.name=X.
3. movies(X)   ← AT ∈ *c-inst*(attdef), AT.name='title', A ∈ *d-inst*(AT), A.val=X.
4. atts(X)   ← A ∈ *c-inst*(attribute), A.name=X.
5. attvals(X)   ← A ∈ *c-inst*(attribute), A.name='title', A.val=X.

Fig. 11. Example XML queries for schema directly, data through schema, and data directly.

The following query returns the property names of all classes in an RDF configuration. Note that this query, like the previous one, is expressed solely against the schema of the source.

properties(X,Y) ← C ∈ *c-inst*(*class*), P ∈ *c-inst*(*prop*), P.domain=C, C.label=X, P.label=Y.

After finding the available classes and properties of the schema, we can then use this information to find data instances. That is, a user could issue the previous query, see that the source contains title properties, and then construct the following query that returns all film titles in the configuration.

films(X) ← P ∈ *c-inst*(*prop*), P.label='title', T ∈ *d-inst*(P), T.obj=X.

ULD queries can access information at various levels of abstraction within an information source, including direct access to data (i.e., accessing data items without first accessing schema), direct access to schema (as shown in the previous queries), and direct access to data-model constructs. For example, the following query is expressed directly against data, and returns the *uri* of all resources used as a property in at least one triple (note that the resource may or may not be associated with schema).

allprops(X) ← T ∈ *c-inst*(*triple*), T.pred=R, R.val=X.

Once this query is issued, a user may wish to find additional information about a particular resource. For example, the following query returns all values of a resource used as a title property of a triple.

propvals(X) ← T ∈ *c-inst*(*triple*), T.pred='#title', T.obj=X.

Figure 11 shows a similar set of queries as those presented here, but for XML sources. The first query finds all available element types in the source, the second finds all available attribute types, the third finds the set of movie titles, the fourth finds the set of attributes (as a data query), and the last query finds the set of values for title attributes in the configuration.

Finally, a query can be posed against a data-model directly to determine the available constructs in a source. For example, the following query returns all constructs that serve as *struct-ct* schema constructs and their component selectors. (Note that conformance relationships are accessed through the *conf* relation defined previously.)

schemastructs(SC,P) ← SC ∈ *inst-ct*(*struct-ct*), *conf*(DC,SC,X,Y), SC→P=C.

3.4 Specifying Additional Rules for Conformance

The ability to query the ULD is not only important for providing uniform access to information sources, but can also be used to specify additional constraints on configurations (i.e., axioms). Here, we consider the use of queries for further defining conformance relationships, beyond cardinality restrictions.

A conformance definition consists of a restricted set of rules. Namely, the head of a conformance-definition is always of the form: $X \in d\text{-}inst(Y)$, which specifies that X can be a data-instance of Y when the conditions specified in the body of the rule are true. For example, according to the definition in Figure 6, tables can conform to relation types, however, the conformance specification given does not provide the necessary conditions to determine when conformance can occur (it states that tables and tuples must conform to relation types). The following two rules elaborate the conformance definitions for the relational model.

$X \in d\text{-}inst(Y) \leftarrow X \in c\text{-}inst(\text{table}),\ Y \in c\text{-}inst(\text{relation}),$
$\qquad\qquad forall_{T \in X}(T \in d\text{-}inst(Y)).$
$X \in d\text{-}inst(Y) \leftarrow X \in c\text{-}inst(\text{tuple}),\ Y \in c\text{-}inst(\text{relation}),\ Y.\text{atts=AS},$
$\qquad\qquad \text{length}(X,L),\ \text{length}(AS,L),$
$\qquad\qquad forall_{V \in X}(\text{memberAt}(V,X,I),\ \text{memberAt}(A,AS,I),$
$\qquad\qquad A.\text{domain=D},\ \text{d-inst}(V,D)).$

The first rule states that for a table to conform to a relation type, each tuple in the table must be a data instance of the relation type. Note that we use the shorthand notation *forall* to simulate a universal quantifier in Datalog. The expression can be replaced to create a Datalog rule by using the standard technique of introducing an intensional predicate and double negation.[2] The second rule states that for a tuple to conform to a relation type it must have the same number of values as attributes in the relation type, and each value must be a data instance of the corresponding attribute's domain

Conformance constraints can be used to determine whether the result of transforming one source into another creates a valid (target) configuration. We are interested in investigating how conformance constraints can be further exploited within the ULD and in identifying a general set of concise constraints (much like in description logics) for specifying additional conformance definitions.

4 Related Work

Most approaches for resolving structural heterogeneity focus on schema and assume a common data model (e.g., [4, 5, 10]). Here, we discuss alternative techniques that consider disparate data models and we discuss their limitations.

---

[2] For example, we can rewrite the first rule as:
$X \in d\text{-}inst(Y) \leftarrow X \in c\text{-}inst(\text{table}),\ Y \in c\text{-}inst(\text{relation}),\ \neg\ \text{not-conf}(X,Y).$
$\text{not-conf}(X,Y) \leftarrow T \in X,\ \neg\ T \in d\text{-}inst(Y).$

A *self-describing* data model [12] describes its own structure to integrate the representation of data, schema, and meta-schema (i.e., a description of the possible schema structurings). Self-describing data models use their own structuring capability to represent meta-schema and schema information. To access multiple data models, users must still use distinct languages and interfaces for each source. Self-describing data models require conventions to distinguish meta-schema from schema, and schema from data.

Atzeni and Torlone's MDM [1, 2, 14] uses primitives similar to E-R entity and relationship types and data-model constructs are defined as compositions of these structures. Schemas in MDM are instantiations of these data-model structures and MDM does not consider source data.

In YAT [9], a meta-data-model is used to define XML *tree patterns*, which are DTDs that permit variables. A tree pattern describes a meta-schema, a partially instantiated tree pattern denotes a schema, and a fully instantiated tree pattern represents the content of an information source. YAT's meta-data-model has limited structuring capabilities for representing data-model constructs and schemas, and instead, defines simple conventions to represent source data as hierarchies.

Gangopadhyay and Barsalou use *metatypes*, which are similar to MDM primitive structures, to define data models. A data model is represented as a collection of specialized metatypes, each serving as a specific data model construct. A schema instantiates the associated data-model metatypes, and a database in the federation is assumed to contain instances of the corresponding schema types (metatypes are considered second-order).

5   Conclusions and Future Work

The ULD extends existing meta-data-model approaches by providing richer modeling capabilities for defining data models. In particular, the ULD permits both data and schema constructs with their conformance relationships. The ULD also provides explicit instance-of relationships, which allow uniform representation of data-model constructs, schema, and data, making each level directly accessible and explicitly connected. The framework enables (accurate) representation of a broader range of data models than previous approaches by permitting zero, one, or more levels of schema. In addition, the ULD enables a transformation language [7], which can specify and mix data, schema, and data-model mappings. The transformation language is an extension of the query language presented in this paper. We also have a generic navigation language on top of the ULD to uniformly browse information represented in diverse data models and representation schemes [6].

Our current implementation of the ULD provides converters for RDF(S), XML, and relational sources into the ULD (i.e., into a Prolog knowledge base), where queries and transformations can be executed (via rules expressed in Prolog). We are also developing a Java-based API for the ULD that contains function

calls for accessing the various ULD predicates (*ct-inst*, *c-inst*, *d-inst*, and so on) to leave data "in place" (i.e., un-materialized).

References

1. P. Atzeni and R. Torlone. Schema translation between heterogeneous data models in a lattice framework. In *Proceedings of the 6th IFIP TC-2 Working Conference on Data Semantics (DS-6)*, pages 345–364. Chapman and Hall, 1995.
2. P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 1996.
3. T. Barsalou and D. Gangopadhyay. M(DM): An open framework for interoperation of multimodel multidatabase systems. In *Proceedings of the 8th International Conference on Data Engineering (ICDE'92)*, pages 218–227. IEEE Computer Society, 1992.
4. C. Batini, M. Lenzerini, and S.B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
5. P.A. Bernstein, A.Y. Halevy, and R. Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, December 2000.
6. S. Bowers and L. Delcambre. JustBrowsing: A generic API for exploring information. In *Demo Session at the 21st International Conference on Conceptual Modeling (ER'02)*, 2002.
7. S. Bowers and L. Delcambre. On modeling conformance for flexible transformation over data models. In *Proceedings of the ECAI Workshop on Knowledge Transformation for the Semantic Web*, pages 19–26, 2002.
8. K. Clapool and E. Rudensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *Proceedings of the 5th International Conference on Enterprise Information Systems*, 2003. To Appear.
9. S. Cluet, C. Delobel, Jérôme Siméon, and K. Smaga. Your mediators need data conversion! In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 177–188. ACM, 1998.
10. S. B. Davidson and A. Kosky. WOL: A language for database transformations and constraints. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'98)*, pages 55–65. IEEE Computer Society, 1997.
11. J. Hammer and D. McLeod. On the resolution of representational diversity in multidatabase systems. In *Management of Heterogeneous and Autonomous Database Systems*, pages 91–118. Morgan Kaufmann, 1998.
12. L. Mark and N. Roussopoulos. Integration of data, schema and meta-schema in the context of self-documenting data models. In *Proceedings of the 3rd International Conference on Entity-Relationship Approach (ER'83)*, pages 585–602. North-Holland, 1983.
13. OMG. *Meta Object Facility (MOF) Specification*, Sept. 1997. OMG Document ad/97-08-14.
14. R. Torlone and P. Atzeni. A unified framework for data translation over the web. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE'01)*, IEEE Computer Society, pages 350–358, 2001.