Using the Uni-Level Description (ULD) to Support Data-Model Interoperability

Shawn Bowers [a,1,*] and Lois Delcambre [b,2]

[a] *UC Davis Genome Center, University of California, Davis 451 East Health Sciences Drive, Davis, CA, 95616, USA*

[b] *Department of Computer Science, Portland State University 1900 SW 4th Avenue, Portland, OR, 97201, USA*

Abstract

We describe a framework called the Uni-Level Description (ULD) for accurately representing information from a broad range of data models. The ULD extends previous meta-data-model approaches by: (a) providing uniform representation and access to data model, schema, and data, and (b) supporting data models with non-traditional schema arrangements, including those that allow optional and multiple levels of schema. Because the ULD is a flat, first-order representation, we show how Datalog over the ULD can provide a flexible mechanism to query, extract, and transform information from data sources that exhibit various types of structural heterogeneity.

Key words: Data Models, Information Integration, Meta Models, Schema Constraints, Conceptual Modeling

1 Introduction

Data and knowledge management systems are typically based on a *data model* or *representation* that sets the basic structures for organizing and storing data. A number of data models are in common use and each data model provides slightly different modeling structures. For instance, information is represented as *tables* in the relational data model, as *ordered trees* in semi-structured data models such as

* Corresponding Author. Tel: 530 754-5098, Fax: 530 754-9658

[1] Supported in part by NSF grants EIA 99-83518, IIS 98-17492, and DBI 0533368.

[2] Supported in part by NSF grants EIA 99-83518, IIS 98-17492, IIS 0514238, and DUE 0511050.

XML, and as *directed graphs* in semantic networks such as RDF. One important advantage of having multiple data models is that developers can select the data model that offers the most convenient representation for their particular needs, motivated in part by their desire to use tools associated with the data model. However, the use of multiple data models introduces the possibility of many kinds of structural heterogeneity.

While data models can differ in their modeling constructs, they can also significantly differ in the way schemas are used. A typical database data model *requires* the use of a schema, e.g., to define the tables of interest in a relational database. Then, quite naturally, all instance data in the database conforms to the schema. The purpose of the schema in a database system is to establish the basic structure for instance data (e.g., by describing the attributes present in a table) and establish constraints (e.g., such as key, foreign key, and domain constraints) that the instance data must satisfy.

However, newer data models and representation schemes, such as XML [1], RDF [2], and Topic Maps [3] have a less strict notion of schema. First, the use of a schema is optional; XML information need not have an XML schema or a DTD and RDF data need not have an RDF Schema. Thus we might be confronted with data that conforms to one (or more) schemas intermixed with data that is *not* associated with a corresponding schema. Second, the details of the conformance relationship may differ from one model to the next. In Topic Maps [3], one topic ("John Doe") may have another topic ("Employee") as its type but the Employee topic does not place any structural constraints on the "John Doe" topic. Third, some models, such as RDF and Topic Maps, allow multiple levels of type or schema-instance links. For example, the "Employee" topic might have a type of "Person" in addition to "John Doe" having "Employee" as a type.

One particularly challenging form of structural heterogeneity, sometimes called *schematic heterogeneity* [4–6], is when data of interest appears as instance data in one information source and as part of the schema in another source. For example, an Employee table in a relational database might have a state attribute as part of an address with values such as "OR" or "CA," whereas another information source might have multiple employee tables named (simply) "Oregon" and "California" and so on. The indication of the state for a given employee appears as data in one case and as the table name (in the schema) in the other case; this makes it difficult to access the state associated with an employee across these data sources in a uniform manner. Note that the flexible use of schemas in newer data models, as described above, admits several additional kinds of model-schema-instance heterogeneity.

In this work, we embrace the use of such a wide range of data models. And we are interested in supporting *data model interoperability*, where a user can easily access information from multiple sources, in spite of any structural heterogeneity that might be present. This paper presents a generic representation language that can be used to describe a broad range of data models, including models with flexibility in the use of schema. Our representation language is called the Uni-Level Description (ULD) because it represents the complete description of a data source, including an explicit description of the data model, any schema(s) if present, and all instance data, in a flat representation.

1.1 Introduction to the ULD

Consider a typical object-oriented model where the primary data model construct is called class and each instance of a class is an object. Figure 1 shows an excerpt from this model in the ULD on the left, and a traditional meta-data-model-based description on the right [7–14]. In a traditional framework, the class is described as a data model construct (as an instance of a construct in the meta-model), the actor class would be part of the schema (as an instance of the class construct), and individual objects, such as *deNiro*, comprise the data (as instances of the appropriate schema construct). A data-model definition in the ULD, on the other hand, establishes the data structures that will be used to hold schema information (*class*) as well as the data structures that will be used to hold data information (*object*). These are, in turn, instantiated with schema (*actor*) and data (*deNiro*) instances.

The ULD offers a number of advantages over the traditional framework:

- Data constructs and their corresponding instances can exist independently from any schema constructs or instances. That is, there is no requirement to have a schema construct first. In contrast, in the traditional framework, data can only be represented as an instance of an existing schema construct.
- *Conformance* relationships, where a particular data construct can or must conform to a particular schema construct, are modeled explicitly using the *conformance* (or simply *conf*) predicate, as shown in the middle portion of the ULD framework in Figure 1. The ULD allows the semantics of each conformance relationship to be specified separately and explicitly.
- At the instance level, *data-instance-of* relationships, where a particular data instance conforms to a particular schema instance, are modeled explicitly using the *d-inst* predicate, as shown in the bottom portion of the ULD framework in Figure 1. The *d-inst* predicate represents the traditional notion of schema-data instantiation. Explicitly specifying *d-inst* relationships supports the use of optional schemas; one data instance might be a *d-inst* of a schema object whereas another data instance might not be.
- Multiple levels of conformance and multiple levels of data-instance-of relationships can be represented directly using multiple *conf* and *d-inst* links, respectively.
- The construct type (i.e., data structure) used to represent schema constructs (such as a class) and the construct type (data structure) used to represent data constructs

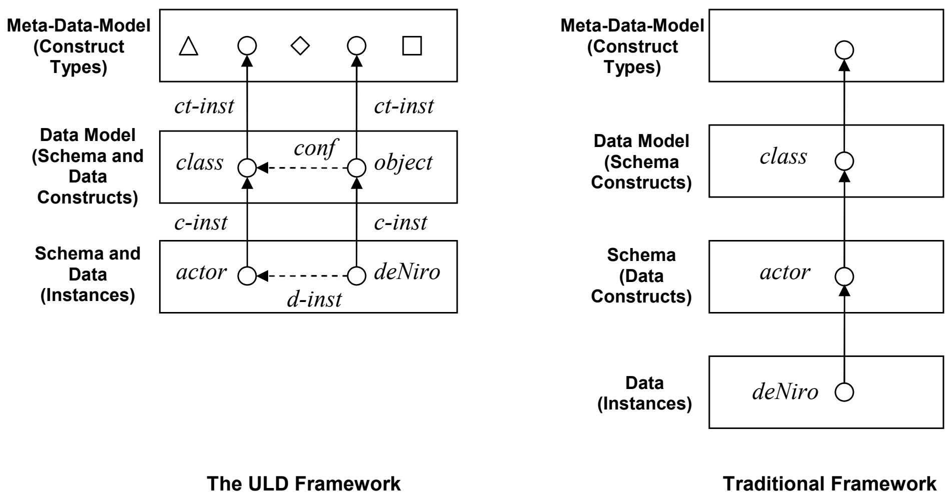
Meta-Data-Model (Construct Types)
ct-inst
ct-inst
Data Model (Schema and Data Constructs)
class
conf
object
c-inst
c-inst
Schema and Data (Instances)
actor
deNiro
d-inst
The ULD Framework
Meta-Data-Model (Construct Types)
Data Model (Schema Constructs)
class
Schema (Data Constructs)
actor
Data (Instances)
deNiro
Traditional Framework

Fig. 1. The ULD Framework (left) and a traditional meta-data-model framework (right). Each distinct shape denotes a different data structure.

(such as an object) can be different in the ULD. In contrast, in traditional frameworks the construct type of the data (implicitly) *must* follow the structure of the schema, further limiting the types of data models that can be captured.

1.2 The ULD as a Flat Representation

Although we typically draw a ULD configuration (e.g., a ULD representation of a particular data source) using the three levels shown in Figure 1, these configurations are "flat" in the ULD. That is, the construct types, the schema constructs, the data constructs, the schema objects, and the data objects are all part of a single first-order interpretation of a generic ULD theory, using the predicates shown as labels on the arrows in Figure 1. Data model constructs are introduced using the *ct-inst* relationships (i.e., *construct-type-instance-of*). Schema and data constructs introduced in a particular data model are instantiated using *c-inst* relationships (i.e., *construct-instance-of*). The *d-inst* relationship indicates that a data instance (e.g, the deNiro object) conforms to a schema instance (e.g., the actor class).

The following facts are contained in the flat, ULD representation of the example shown in Figure 1.

```
ct-inst(class, struct-ct)
ct-inst(object, struct-ct)
conf(object, class, 0:n, 1:1)
c-inst(deNiro, object)
d-inst(deNiro, actor)
```

The identifiers class, object, actor, and deNiro are defined and related using the generic ULD predicates ct-inst, conf, c-inst, and d-inst. The conf predicate states that a given object must be an instance of one class (via the 1:1 conformance constraint) and a class can have zero or more objects as instances (via the 0:n conformance constraint). As we discuss further in Section 2, additional ULD predicates are used to define the structures of these identifiers (e.g., that a class is composed of a set of attribute definitions, and so on). We treat the facts given above as the low-level ULD language, and introduce syntactic conventions in Section 2 for defining ULD configurations.

The ULD approach differs from data models that overload the instance-of or type relationship such as RDF, Telos [15], or other semantic-network-based data models. The flat representation in the ULD enables an ordinary query language to easily access the data model, schema(s), and instance data, in contrast to most formalizations of database and other schema-based data models which treat schema as an interpretation of the data model and data instances as an interpretation of the schema. Thus for the data shown in Figure 1, data model (class and object), schema (actor), and data (the deNiro object) are equally accessible and can be directly queried via the ULD predicates. We use Datalog rules as a query language for the ULD and we have investigated using Datalog to perform a wide variety of model-based transformations [16–19]. The rules are particularly powerful because model-to-model transformations as well as various kinds of transformations handling structural heterogeneity can be expressed directly.

One would expect that implementing data management functions directly using the ULD would be inefficient because conformance relationships would need to be interpreted and enforced. We do not necessarily advocate a direct ULD implementation. Rather, we are exploring the use of the ULD as a *supplemental view mechanism* to support data-model interoperability where generic, ULD-based tools can easily select and traverse information of interest, across a wide spectrum of information sources. We are interested in light-weight, *ad-hoc* uses of information, e.g., much like a web browser allows the user to browse through HTML. The ULD provides a simple mechanism to expose structural information (when needed) across diverse data models and schemas.

1.3 Organization of the Paper

In Section 2, we describe in detail the ULD and the constraints inherent in ULD theories, also called ULD configurations. We show examples of relational, XML, and RDF and RDF Schema (hereafter, simply denoted RDF) data sources expressed as ULD configurations. In Section 3, we show how Datalog can be used to access information at various levels within ULD configurations. In this way, we can select information from multiple sources using a single, ULD-based query language. In Section 4, we use Datalog with the ULD to provide a powerful transformation lan-

Movie

mid : *integer*	title : *string*	genre : *string*	company : *string*
1	The usual suspects	Thriller	Gramercy
2	Meet the parents	Comedy	Universal

Cast

mid : *integer*	character : *string*	actor : *string*
1	Roger Verbal Kint	Kevin Spacey
2	Jack Byrnes	Robert de Niro

Fig. 2. An example relational schema and instance

```
<!ELEMENT moviedb (movie*)>
<!ELEMENT movie (title, studio, genre*, review*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT studio (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ELEMENT review (#PCDATA)>
<!ATTLIST review rating #CDATA REQUIRED
                 source #CDATA REQUIRED>
```

XML DTD

```
<moviedb>
  <movie>
    <title>The Usual Suspects</title>
    <studio>Gramercy</studio>
    <genre>Thriller</genre>
    <review rating="8.7" source="IMDB"/>
  </movie>
</moviedb>
```

XML Instance Document

Fig. 3. An example XML DTD (top) and instance document (bottom)

guage that can easily accommodate various kinds of structural heterogeneity across data sources. In Section 5, we compare the ULD to other meta-data-model approaches and languages for resolving heterogeneity. Finally, in Section 6 we summarize our contributions and describe future work.

2 The Uni-Level Description

We first introduce three example data sources. Figures 2, 3, and 4 show similar schema and instance data represented in the relational, XML, and RDF data models. The examples not only use different data models, but they also exhibit schematic heterogeneity (e.g., with “Thriller” as data in the relational database but as a class in RDF), and data differences (with “Spacey, Kevin” in RDF and “Kevin Spacey” in XML). We note that for the RDF example, not all instance data in the source has a corresponding schema definition.

As previously mentioned, an information source is represented in the ULD as a configuration consisting of data model, schema, and instance representations. This section describes a high-level ULD syntax for defining configurations, demonstrates the language using the above example data sources, and shows how configurations

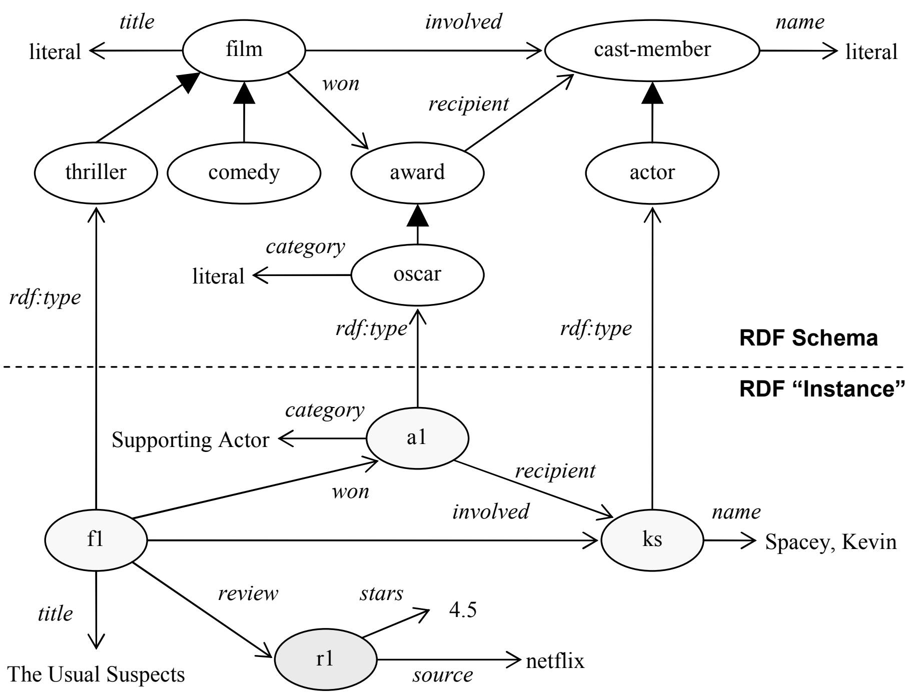

Fig. 4. An example RDF schema (top) and instance (bottom)

can be modeled as ULD first-order logic theories.

2.1 The ULD Representation Language

The ULD representation language is a high-level syntax for defining ULD configurations. Informally, a ULD configuration consists of a set of identifiers representing construct types, (schema and data) constructs, and (schema and data) construct instances related through ct-inst, conf, c-inst, and d-inst predicates. In addition, constructs and construct instances each have a corresponding structural *value*. In the case of constructs this value is a structure definition (such as a record or set type) and in the case of construct instances this value is an instantiated structure definition (a value of the corresponding type). Construct types define the allowable structural definitions. Here, we consider the struct-ct (i.e., for record-like structures), set-ct (for collections), union-ct (for union types), and atomic-ct (for basic types like string and integer) construct types. [3]

The ULD high-level syntax enables more concise descriptions of ULD configurations, compared to using the low-level ULD predicates. Figures 5, 6, and 7 use the ULD high-level syntax to describe (simple) versions of the relational, XML, and RDF data-model constructs, respectively. Constructs beginning with "uld" are predefined types, e.g., uld-string represents a default string atomic type. We also use

[3] In [17], we include bag and list collection types in addition to the set type described here.

uld-value and uld-valuetype as special constructs to denote the set of basic values (of any type) and value types (both default such as uld-string but also configuration-defined, such as pcdata), respectively. We discuss the use of these constraints in more detail below (see also Figure 8). Note that there are potentially many ways to describe a data model in the ULD, and these examples show only one choice of representation (e.g., see [17] for more detailed ULD descriptions of these and other data models.)

A construct definition in the high-level syntax can take one of the following forms.

- construct c := $[a_1$=>c_1, a_2=>$c_2, \ldots, a_n$=>$c_n]$ :: c_0 cardinality $[x_d{:}y_d, x_r{:}y_r]$

  This expression defines a construct c as a ct-inst of construct type struct-ct, where a_1 to a_n are distinct labels, $n \geq 1$, and c_0 to c_n are construct identifiers. Each expression a_i => c_i is the *component* of the construct c where a_i is the *component selector*. The symbol "::" here denotes a conformance definition, which is an optional part of the expression. When a conformance definition is present, construct instances of c must conform (i.e., be connected by a d-inst relationship) to construct instances of c_0 according to the domain ($x_d{:}y_d$) and range ($x_r{:}y_r$) cardinality constraints. If the conformance expression is not present, conformance is not permitted for the construct, i.e., there cannot be a d-inst relationship for the construct's instances. The cardinality constraints on conformance restrict the participation of associated instances in d-inst relationships, for both the domain and range of the relationship, to either: exactly-one, denoted 1:1; zero-or-one, denoted 0:1; zero-or-more, denoted 0:n; or one-or-more, denoted 1:n.

- construct c := set-of c_1 :: c_0 cardinality $[x_d{:}y_d, x_r{:}y_r]$

  This expression defines a collection construct c as a ct-inst of construct type set-ct. The definition restricts the values of instances of construct c to have as members only instances of the construct c_1. Each such member must have a distinct identifier. This construct definition can also contain an optional conformance definition with associated domain and range cardinality constraints. Note that every component of an instance of a struct-ct and set-ct construct is either an identifier or an atomic value. Thus, arbitrarily nested structures can be supported by nesting identifiers whose values contain nested identifiers, and so on.

- construct c := $c_1 \mid c_2 \mid \ldots \mid c_n$

  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 $n \geq 2$. The definition states that each instance of c_1 to c_n is also an instance of c. A union construct provides a simple mechanism to group heterogeneous structures (e.g., atomic and record types) as opposed to using a more formal *is-a* type of relationship (like in description logic), which requires strict inheritance semantics and only permits the grouping of common structures (e.g., classes). Here we do not consider conformance definitions for union types.

- construct c := atomic :: c_0 cardinality $[x_d{:}y_d,\ x_r{:}y_r]$

  This expression defines an atomic construct c as a ct-inst of construct type atomic-ct. Examples of atomic constructs include PCDATA and CDATA for XML and the data type *literal* for RDF. This construct can also contain a conformance definition.

For our definition of the relational data model shown in Figure 5, tables and relation types are one-to-one: each table conforms to exactly one relation type and vice versa. Similarly, each tuple in a table must conform, as shown by the range restriction of 1:1, to a relation type, and values of the tuple to attributes of a relation type. We assume each relation type can have at most one primary key. Note that we do not define, other than cardinality restrictions, the various constraints implicit in the data model definition. Data model constraints can be given using the ULD constraint language, described elsewhere [17].

We briefly illustrate here, however, how comformance relationships can be further specified using first-order logic constraints (in particular full and embedded dependencies [20], a fragment of the ULD constraint language). The following constraints apply to the relational model of Figure 5, and are expressed over the underlying ULD predicates. Note that the member-of and struct-of predicates (defined below in Section 2.2.1) provide access to set and record (struct) instance values, respectively. The first constraint requires every tuple in a table to conform to the table's corresponding relation.

$$(\forall x,y,u)\ \text{c-inst}(x,\text{table}) \wedge \text{member-of}(y,x) \wedge \text{d-inst}(x,u) \rightarrow \text{d-inst}(y,u)$$

The next constraint requires every tuple-value in a table to conform to an attribute of the table's corresponding relation.

$$\begin{aligned}(\forall y,z,u,v,w)\ &\text{c-inst}(y,\text{tuple}) \wedge \text{member-of}(z,y) \wedge \text{d-inst}(z,w) \wedge \text{d-inst}(y,u) \wedge \\ &\quad \text{struct-of}(u,\text{atts},v) \rightarrow \text{member-of}(z,v)\end{aligned}$$

The next constraint requires every attribute in a relation to have a conforming tuple-value in a corresponding tuple.

$$\begin{aligned}(\forall x,y,z,u,v,w)\ &\text{c-inst}(x,\text{table}) \wedge \text{d-inst}(x,u) \wedge \text{struct-of}(u,\text{atts},v) \wedge \\ &\quad \text{member-of}(w,v) \wedge \text{member-of}(y,x) \rightarrow (\exists z)\ \text{member-of}(z,y) \wedge \text{d-inst}(z,w)\end{aligned}$$

The last constraint restricts tuples to at most one tuple-value for every attribute.

$$\begin{aligned}(\forall y,z_1,z_2,w)\ &\text{c-inst}(y,\text{tuple}) \wedge \text{member-of}(z_1,y) \wedge \text{member-of}(z_2,y) \wedge \\ &\quad \text{d-inst}(z_1,w) \wedge \text{d-inst}(z_2,w) \rightarrow z_1 = z_2\end{aligned}$$

```
% schema constructs
construct relation       : = [name=>uld-string, atts=>att-set]
construct att-set        : = set-of attribute
construct attribute      : = [name=>uld-string, domain=>uld-value-type]
construct pkey           : = [for-rel=>relation, key-atts=>pkey-att-set]
construct fkey           : = [for-rel=>relation, to-rel=>relation, fkey-atts=>fkey-att-set]
construct pkey-att-set   : = set-of attribute
construct fkey-att-set   : = set-of attribute
% data constructs
construct table          : = set-of tuple :: relation cardinality [1:1, 1:1]
construct tuple          : = set-of tuple-value :: relation cardinality [0:n, 1:1]
construct tuple-value    : = [value=>uld-value] :: attribute cardinality [0:n,1:1]
```

Fig. 5. A simple relational data model representation in the ULD

```
% schema constructs
construct pcdata         : = atomic
construct cdata          : = atomic
construct elem-type      : = [name=>uld-string, atts=>att-def-set, model=>content-def]
construct att-def-set    : = set-of att-def
construct att-def        : = [name=>uld-string]
construct content-def    : = set-of content-type
construct content-type   : = elem-type | uld-value-type
% data constructs
construct element        : = [tag=>uld-string, atts=>att-set, children=>content] ::
                               elem-type cardinality [0:n,0:1]
construct att-set        : = set-of attribute
construct attribute      : = [name=>uld-string, value=>cdata] :: att-def cardinality [0:n,0:1]
construct content        : = set-of node
construct node           : = element | pcdata
```

Fig. 6. A simple XML (with DTD) data model representation in the ULD

The XML data model shown in Figure 6 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. Although not shown here, the XML description can be extended (e.g., see [17]) to include richer content models including the various DTD regular-expression constraints.

Figure 7 shows a version of the RDF data model expressed in the ULD with 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 RDF, the properties rdf:type, rdfs:subClassOf, and rdfs:subPropertyOf are considered special and are used to denote instance and specialization relationships. In the description of Figure 7, however, we model rdf:type

```
% schema constructs
construct resource       := class | property | uri-ref
construct class          := [uri-val=>uld-uri, label=>uld-string] ::
                               class cardinality [0:n,0:n]
construct property       := [uri-val=>uld-uri, label=>uld-string, domain=>class,
                               range=>range-value]
construct uri-ref        := [uri-val=>uld-uri] :: class cardinality [0:n,0:n]
construct range-value    := resource | uld-value-type
construct sub-class      := [sub=>class, super=>class]
construct sub-property   := [sub =>property, super =>property]
% data constructs
construct triple         := [pred=>resource, subj=>resource, obj=>obj-value]
construct obj-value      := resource | literal
construct literal        := atomic
```

Fig. 7. A simple RDF data model representation in the ULD

using conformance and introduce the constructs sub-class and sub-property to model rdfs:subClassOf and rdfs:subPropertyOf, respectively. For example, a sub-class relationship is represented by instantiating (using c-inst) a sub-class construct as opposed to using the special rdfs:subClassOf RDF property. This approach allows RDF properties and structural relationships to be decoupled in the ULD representation and does not limit the expressiveness of RDF in that partial, optional, and multiple levels of schema are still possible.

Every ULD configuration can be initialized with default constructs that represent typical primitive value types, such as string, Boolean, integer, URI, and so on, as instances of atomic-ct. In addition, uld-value and uld-value-type are special constructs that work together to provide a mechanism for describing data models that permit user-defined primitive types (e.g., to support relational domains or XML Schema data type definitions). Figure 8 shows how these constructs are defined within the ULD framework. The uld-value construct is defined as a union-ct construct, representing the union of all defined atomic-ct constructs. Thus, when a new atomic-ct construct is created, it is added to the definition of uld-value.[4] In addition, every atomic-ct construct is assumed to conform to the uld-value-type construct. Instances of the uld-value-type construct serve as placeholders for atomic-ct constructs at the instance level. Each atomic-ct construct has a uld-value-type instance whose value is the name of the corresponding atomic-ct construct (see Figure 8). For example, if a new atomic-ct date type is created, the system assigns the construct as a union member of uld-value and creates a new uld-value-type instance with the value ‘date’. Individual dates (shown at the bottom right corner of Figure 8) are construct-instances of (as indicated by the c-inst relationship) the date construct in the middle of Figure 8 as well as of uld-value (by virtue of the fact that value is a

[4] We assume that the construct is added by the system managing the ULD configuration, and not by the user.

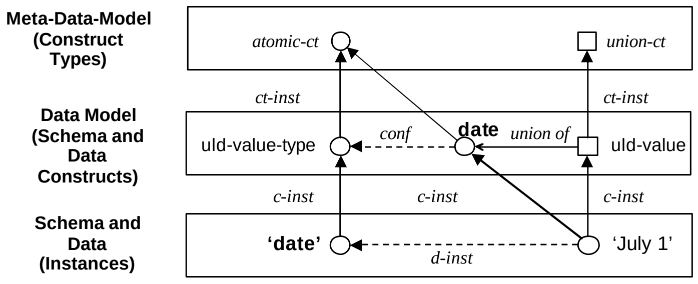

Fig. 8. The default uld-value and uld-value-type constructs for representing scalar types where date is a member of the union defining uld-value (i.e., date is a sub-construct of uld-union)

union construct). The system assigns each date value to be a d-inst of the 'date' uld-value-type instance. Thus, e.g., using these constructs, we can define a conformance constraint for the relational data model requiring that each tuple-value's component for the selector value is a d-inst of its corresponding attribute's component for the selector domain.

Sample schema and data (the bottom level in Figure 5) for the relational, XML, and RDF data models are shown in Figures 9, 10, and 11, respectively. Expressions for defining construct instances take one of the following forms, for instance identifiers d, d_1 to d_n, and construct identifier c

- instance d := c:[a_1:d_1, ..., a_i: d_i] :: [d_j , ..., d_n]
- instance d := c:{d_1, ..., d_i} :: [d_j, ..., d_n]
- instance d := c:v :: [d_1, ..., d_n]

These expressions define d as a c-inst of construct c. In the first case, the instance d is assigned the given record (struct) value. In the second case, d is assigned the given set value. And in the last expression, d is assigned the atomic value v. (Note that v here is optional, and if it is missing, we assume the identifier itself is the desired atomic value.) If c participates in a union construct c_0, d is also assigned as a c-inst of both c and c_0. The symbol "::" here is used to denote d-inst relationships, and is optional. When the identifier participates in (one or more) d-inst relationships, d is assigned as a d-inst of each construct instance d_j to d_n, each of which must be a c-inst of the construct that c conforms to.

Figure 9 defines the movie table from Figure 2 using the ULD relational model description of Figure 5. As shown, a movie is defined as a relation having four attributes and a single primary key. To keep the description more concise, we use the ULD versions of atomic types (i.e., uld-integer and uld-string), however, one could easily define the desired domain types for the example. In Figure 9, we also create a movie table as a d-inst of the movie relation. The d-inst relationship signifies that the table conforms to (is an instance of) the movie relation schema. As described earlier, a table must satisfy certain constraints to be considered a valid d-inst of a relation, namely, that each tuple it contains conforms to the relation schema and

```
% schema instances
instance movie        := relation:[name:'movie', atts:movie-atts]
instance movie-atts   := att-set:{a1, a2, a3, a4}
instance a1           := attribute:[name:'mid', domain:'uld-integer']
instance a2           := attribute:[name:'title', domain:'uld-string']
instance a3           := attribute:[name:'genre', domain:'uld-string']
instance a4           := attribute:[name:'company', domain:'uld-string']
instance movie-key    := pkey:[for-rel:movie, key-atts:movie-keys]
instance movie-keys   := pkey-att-set:{a1}
% data instances
instance movie-table  := table:{t1} :: [movie]
instance t1           := tuple:{v1, v2, v3, v4} :: [movie]
instance v1           := tuple-value:[value:1] :: [a1]
instance v2           := tuple-value:[value:'The Usual Suspects'] :: [a2]
instance v3           := tuple-value:[value:'Thriller'] :: [a3]
instance v4           := tuple-value:[value:'Gramercy'] :: [a4]
```

Fig. 9. Sample relational schema and data

all primary and foreign key constraints are satisfied.[5] In Figure 9 we also insert a tuple, corresponding to the first row of Figure 2, into the table where the tuple is assigned as a d-inst of the movie relation.

Figure 10 defines the movie database XML DTD from Figure 3 using the ULD data model description of Figure 6. As shown, moviedb, movie, title, studio, genre, review, rating, and source are defined as element types, nested through content models (content-defs). The title, studio, and genre element types have pcdata as their content models. Figure 10 also defines the instance document shown at the bottom of Figure 3. We define one instance each for the moviedb, movie, title, studio, genre, and review element, which are properly nested using ULD structures.

Figure 11 defines a portion of the RDF schema and instance from Figure 4 using the ULD data model description of Figure 7. The film, thriller, cast-member, and actor classes are defined along with their associated properties and sub-class definitions. Also shown is part of the instance from the bottom of Figure 4, including the resource without an associated class having the (undefined) stars and source properties.

We have implemented the ULD high-level syntax shown here within Prolog using straightforward operator declarations (with associated functions). Thus, the syntax above (with only slight modifications, e.g., replacing "-" with "_") is an abstract and high-level language that can be used directly (within Prolog) to manipulate and further process ULD expressions. We can also use Prolog directly to convert un-

[5] Note that the ULD is typically populated with data that exists "natively" (e.g., from a DBMS for relational data) and thus is assumed to satisfy conformance constraints.

```
% schema instances
instance moviedb          := elem-type:[name:'moviedb', model:moviedb-def]
instance moviedb-def      := content-def:{movie}
instance movie            := elem-type:[name:'movie', model:movie-def]
instance movie-def        := content-def:{title, studio, genre, review}
instance title            := elem-type:[name:'title', model: text-content]
instance text-content     := content-def:{'pcdata'}
instance studio           := elem-type:[name:'studio', model:text-content]
instance genre            := elem-type:[name:'genre', model:text-content]
instance review           := elem-type:[name:'review', atts:review-att-defs]
instance review-att-defs  := att-def-set:{rating, source}
instance rating           := att-def:[name:'rating']
instance source           := att-def:[name:'source']
% data instances
instance elem1            := element:[tag:'moviedb', children:elem1-children] :: [moviedb]
instance elem1-children   := content:{elem2}
instance elem2            := element:[tag:'movie', children:elem2-children] :: [movie]
instance elem2-children   := content:{elem3, elem4, elem5, elem6}
instance elem3            := element:[tag:'title', children:elem3-children] :: [title]
instance elem3-children   := content:{'The Usual Suspects'}
instance elem4            := element:[tag:'studio', children:elem4-children] :: [studio]
instance elem4-children   := content:{'Gramercy'}
instance elem5            := element:[tag:'genre', children:elem5-children] :: [genre]
instance elem5-children   := content:{'Thriller'}
instance elem6            := element:[tag:'review', atts:elem6-atts] :: [review]
instance elem6-atts       := att-set:{a1, a2}
instance a1               := attribute:[name:'rating', value:'8.7'] :: [rating]
instance a2               := attribute:[name:'source', value:'IMDB'] :: [source]
```

Fig. 10. Sample XML (with DTD) data and schema

derlying data sources, represented in a relational, XML, or RDF data model, into corresponding ULD configurations. In particular, we use the various SWI-Prolog [21] modules for accessing these sources (within Prolog) and define rules that map data sources into their corresponding ULD representation. Observe that for each underlying data model, there is a single set of Prolog rules to execute the conversion, i.e., the Prolog rules for a particular data model are written once, and can be applied to all sources of that model. We describe additional approaches for querying and transforming ULD configurations via Datalog in more detail in Sections 3 and 4.

2.2 The ULD as a First-Order Representation

A significant advantage of the ULD compared to other meta-data-model approaches is that, similar to F-Logic [22], ULD expressions are "syntactic sugar" for first-

```
% schema instances
instance film          := class:[uri-val:‘#film’, label:‘film’]
instance thriller      := class:[uri-val:‘#thriller’, label:‘thriller’]
instance member        := class:[uri-val:‘#cast-member’, label:‘cast-member’]
instance actor         := class:[uri-val:‘#actor’, label:‘actor’]
instance title         := property:[uri-val:‘#title’, label:‘title’, domain:film, range:‘literal’]
instance involved      := property:[uri-val:‘#involved’, label:‘involved’, domain:film,
                               range:member]
instance name          := property:[uri-val:‘#name’, label:‘name’, domain:member,
                               range:‘literal’]
instance thriller-film := sub-class:[sub:thriller, super:film]
% data instances
instance f1            := uri-ref:[uri-val:‘#f1’] :: [thriller]
instance ks            := uri-ref:[uri-val:‘#ks’] :: [actor]
instance r1            := uri-ref:[uri-val:‘#r1’]
instance t1            := triple:[pred:title, subj:f1, obj:‘The Usual Suspects’]
instance t2            := triple:[pred:review, subj:f1, obj:r1]
instance t3            := triple:[pred:involved, subj:f1, obj:ks]
instance t7            := triple:[pred:name, subj:ks, obj:‘Spacey, Kevin’]
instance t8            := triple:[pred:stars, subj:r1, obj:‘4.5’]
instance t9            := triple:[pred:source, subj:r1, obj:‘netflix’]
instance stars         := uri-ref:[uri-val:‘#stars’]
instance source        := uri-ref:[uri-val:‘#source’]
```

Fig. 11. Sample RDF(S) data and schema

order representations. Here we describe the underlying first-order representation of the ULD.

Note that there are two fundamental types of information given in ULD configurations: the set of identifiers that denote construct types, constructs, and construct instances, and the values associated with these identifiers. Thus, in the first-order representation, we provide predicates for accessing the various instance-of relationships between identifiers (i.e., ct-inst, c-inst, d-inst, and conf) as well as predicates for accessing the structural values associated with identifiers (i.e., to access primitive, struct, and set values).

In Section 2.2.1 we first define a formal and abstract data structure for representing ULD configurations. This data structure uses a natural representation of the structured values of configuration identifiers. For example, we represent the value of an n-place struct-ct construct as an n-tuple of ordered pairs (s_i, c_i), where s_i is a selector and c_i a construct identifier. Similarly, we represent set-ct construct instances directly using sets. Then, in Section 2.2.2, we define a ULD theory consisting of ULD predicate symbols. We also define a mapping that converts a given configuration expressed in the abstract model to its corresponding interpretation of the ULD theory. In Section 2.2.3, we further define the ULD theory by specifiying axioms that all such interpretations satisfy. These axioms can be used to reason over ULD configurations, e.g., to infer certain additional c-inst relationships.

2.2.1 An Abstract Model of Configurations

The abstract model of a configuration $\mathcal{F}$ is a tuple of the form

$$(O_T, O_C, O_D, \mathcal{V}, val, ext_{ct}, ext_c, ext_d, conf)$$

consisting of identifiers, values, identifer-to-value mappings, instance-of mappings, and conformance relationships. We define each of these structures below.

Identifiers and Value Mappings. The sets O_T, O_C, and O_D consist, respectively, of construct-type identifiers, construct identifiers, and construct-instance identifiers. We require O_T, O_C, and O_D to be pairwise disjoint sets for a configuration $\mathcal{F}$. The set $\mathcal{V}$ contains the structural values of $\mathcal{F}$. The function $val : O_C \cup O_D \to \mathcal{V}$ maps each construct in O_C and instance in O_D to its value in $\mathcal{V}$. The sets of identifiers and values represent the domain of the configuration.

Instance/Extension Mappings. The functions ext_{ct}, ext_c, and ext_d represent the three types of ULD instance-of relationships ct-inst, c-inst, and d-inst, respectively. The function $ext_{ct} : O_T \to 2^{O_C}$ maps each construct-type identifier ct in $\mathcal{F}$ to its "extent," consisting of each construct identifier c that is defined as a construct-type instance of ct (where 2^{O_C} denotes the powerset of O_C). The function $ext_c : O_C \to 2^{O_D}$ maps each construct identifier c in $\mathcal{F}$ to each of its instance identifiers d that are defined as a construct instance of c. And, the function $ext_d : O_D \to 2^{O_D}$ maps each instance identifier d in $\mathcal{F}$ to each of its instance identifiers d_0 that are defined as data instances of d. As shown below, these three functions trivially map to the ct-inst, c-inst, and d-inst ULD first-order predicates.

Conformance. Conformance is represented using the relation

$$conf \subseteq O_C \times O_C \times \{0{:}1, 1{:}1, 0{:}n, 1{:}n\} \times \{0{:}1, 1{:}1, 0{:}n, 1{:}n\}$$

that contains only those elements (c_1, c_2, d, r) in which instances of c_1 are allowed to be data instances of instances of c_2, according to the domain d and range r constraints. Like with the instance-of relationships, the conformance relation trivially maps to the conf predicate in the ULD first-order representation.

Construct Values. The construct values of $\mathcal{F}$ consist of struct-ct, set-ct, and union-ct construct definitions represented as follows.[6] We note that here we do not consider bag and list structures, as in [17]. Every construct identifier $c \in ext_{ct}(\text{struct-ct})$ has a value of the form

$$val(c) = ((s_1, c_1), (s_2, c_2), \ldots, (s_n, c_n)) \in \mathcal{V}$$

such that each (s_i, c_i) is an ordered pair with $s_i \in \mathcal{V}$ as the component selector and $c_i \in O_C$ as the construct identifier of the i-th component of c for $1 \leq i \leq n$. Every construct identifier $c \in ext_{ct}(\text{set-ct})$ has a singleton value

$$val(c) = c_0 \in \mathcal{V}$$

such that construct instances of c are defined to only contain construct instances of $c_0 \in \mathcal{C}_T$. And, every construct identifier $c \in ext_{ct}(\text{union-ct})$ has a set value of the form

$$val(c) = \{c_1, c_2, \ldots, c_n\} \in \mathcal{V}$$

such that c is defined as the union of construct identifiers c_1 to c_n in O_C.

Instance Values. The instance values of $\mathcal{F}$ consist of record, set, and atomic structures represented as follows. Every instance identifier $d \in ext_c(c)$ where $c \in ext_{ct}(\text{struct-ct})$ has a value of the form

$$val(d) = ((s_1, d_1), (s_2, d_2), \ldots, (s_n, d_n)) \in \mathcal{V}$$

such that each (s_i, d_i) is an ordered pair having s_i as the component selector and $d_i \in O_D$ as the instance identifier of the i-th component of d for $1 \leq i \leq n$. (Note that each s_i for d must be in the corresponding value of c.) Every instance identifier $d \in ext_c(c)$ where $c \in ext_c(\text{set-ct})$ has a set value of the form

$$val(d) = \{d_1, d_2, \ldots, d_n\} \in \mathcal{V},$$

which denotes the (possibly empty) set value of d for $d_i \in O_D$ and $1 \leq i \leq n$. And, every instance identifier $d \in ext_c(c)$ where $c \in ext_c(\text{atomic-ct})$ has a value

$$val(d) = v \in \mathcal{V},$$

capturing the singleton atomic value of d.

[6] The atomic-ct constructs in our framework do not have additional structure other than an identifier, thus the value of an atomic construct such as pcdata is null.

2.2.2 *Mapping Configurations to First-Order Representations*

Given a configuration $\mathcal{F}$, we define an interpretation $I_{\mathcal{F}}$ as a mapping from $\mathcal{F}$ to the ULD first-order representation. The domain of the interpretation $I_{\mathcal{F}}$ is the set of construct type, construct, and instance identifiers as well as the set of atomic and structural values of $\mathcal{F}$.

For interpretations we also assume the standard functions and relations for accessing structural values within a configuration. In particular, the expression $v.s$ returns the construct or construct-instance identifier given a (tuple) value v and a selector s (where s is a selector for v), and "$\in$" is the standard set membership relation. Given these assumptions, for a configuration $\mathcal{F}$, the interpretation $I_{\mathcal{F}}$ is defined as

$$\begin{aligned}
\textsf{ct-inst}(x,y)^I &\equiv x \in ext_{ct}(y) \\
\textsf{c-inst}(x,y)^I &\equiv x \in ext_{c}(y) \\
\textsf{d-inst}(x,y)^I &\equiv x \in ext_{d}(y) \\
\textsf{conf}(x,y,d,r)^I &\equiv (x,y,d,r) \in conf \\
\textsf{struct-type-of}(x,s,y)^I &\equiv x \in ext_{ct}(\textsf{struct-ct}) \wedge val(x).s = y \\
\textsf{set-of}(x,y)^I &\equiv x \in ext_{ct}(\textsf{set-ct}) \wedge val(x) = y \\
\textsf{union-of}(x,y)^I &\equiv x \in ext_{ct}(\textsf{union-ct}) \wedge y \in val(x) \\
\textsf{struct-of}(x,s,y)^I &\equiv \exists c\; x \in ext_{c}(c) \wedge c \in ext_{ct}(\textsf{struct-ct}) \wedge y = val(x).s \\
\textsf{member-of}(x,y)^I &\equiv \exists c\; y \in ext_{c}(c) \wedge c \in ext_{ct}(\textsf{set-ct}) \wedge x \in val(y)
\end{aligned}$$

Each expression above defines a mapping where the left-hand expression (the atom) is true if and only if the right-hand expression is true. The ULD first-order representation consists of exactly the predicates used in the left-hand expressions above.

2.2.3 *ULD Axioms*

We take advantage of the first-order ULD representation to capture implicit ULD axioms. For example, the following two axioms are expressed in first-order logic and constrain ULD struct-ct constructs and instances. The first axiom states that if s is a selector for a struct-ct construct instance, then s must be defined in the corresponding construct. The second states that every instance of a struct-ct construct c_1 has a value for each selector defined by c_1.

$$(\forall x,y,s,c_1)\ \textsf{struct-of}(x,s,y) \wedge \textsf{c-inst}(x,c_1) \rightarrow (\exists c_2)\ \textsf{struct-type-of}(c_1,s,c_2)$$
$$(\forall x,s,c_1,c_2)\ \textsf{struct-type-of}(c_1,s,c_2) \wedge \textsf{c-inst}(x,c_1) \rightarrow (\exists y)\ \textsf{struct-of}(x,s,y)$$

As another example, the following axiom defines the required relationship between set-ct constructs and their instances.

$$(\forall x, y, c_1, c_2)\ \mathsf{member\text{-}of}(x, y) \wedge \mathsf{c\text{-}inst}(y, c_1) \wedge \mathsf{set\text{-}of}(c_1, c_2) \rightarrow \mathsf{c\text{-}inst}(x, c_2)$$

The next axiom gives the ULD constraint for union-ct construct instances.

$$(\forall x, c_1, c_2)\ \mathsf{c\text{-}inst}(x, c_2) \wedge \mathsf{union\text{-}of}(c_1, c_2) \rightarrow \mathsf{c\text{-}inst}(x, c_1)$$

Finally, the following two axioms restrict conformance: the first requires that the corresponding constructs of data-instances related by the d-inst relationship have conformance definitions, and the second requires that constructs conform to at most one other construct.

$$(\forall x, c_1, c_2)\ \mathsf{d\text{-}inst}(x, y) \wedge \mathsf{c\text{-}inst}(x, c_1) \wedge \mathsf{c\text{-}inst}(y, c_2) \rightarrow (\exists d, r)\ \mathsf{conf}(c_1, c_2, d, r)$$
$$(\forall c_1, c_2, c_3, d_1, r_1, d_2, r_2)\ \mathsf{conf}(c_1, c_2, d_1, r_1) \wedge \mathsf{conf}(c_1, c_3, d_2, r_2) \rightarrow c_2 = c_3$$

From these two axioms we can derive the following axiom, which can be used to uniquely infer the construct of an instance participating in a d-inst relationship.

$$(\forall x, c_1, c_2)\ \mathsf{d\text{-}inst}(x, y) \wedge \mathsf{c\text{-}inst}(x, c_1) \wedge \mathsf{conf}(c_1, c_2, d, r) \rightarrow \mathsf{c\text{-}inst}(y, c_2)$$

A number of additional axioms are given in [17]. These axioms define the various constraints on valid ULD configurations, and in certain cases can be used to ensure that query expressions and transformations result in well-formed configurations (as shown in [17]).

In addition to supporting the high-level ULD syntax, the Prolog implementation described previously also includes rules to convert a configuration into a corresponding ULD first-order representation. This first-order representation is used directly for querying and transforming configurations via Datalog rules, as described in the following two sections.

3 Using Datalog as a ULD Query Langauge

The two primary motivations for a ULD query language are: (a) providing users with the ability to easily discover and navigate information within unfamiliar data sources, and (b) allowing multiple, structurally heterogeneous data sources to be easily accessed and combined. In previous work [19] we defined a separate language on top of the ULD for incrementally navigating and browsing data sources. Here, we give examples of using Datalog directly as the ULD query language to access and navigate data sources. An advantage of the ULD is that it provides a uniform language to access various levels of data sources expressed in different data models. For example, the ULD can support queries posed directly against the data model (e.g., to determine the constructs used), the schema, and instance data of a source, along with queries that access two or more of these levels at once.

This section describes the use of Datalog as a ULD query language. We give a number of example ULD queries against the data sources of Figures 9, 10, and 11. We also demonstrate how ULD queries can be used to access multiple data sources simultaneously. In this way, the ULD can provide integrated views over data sources with distinct data models, without requiring a user to perform additional transformations, e.g., by converting each source to a common data model.

3.1 Datalog and the ULD

A ULD query is expressed as a Datalog [23,20] program, consisting of a set of restricted horn-clause rules that can be executed against ULD configurations. For example, the following query finds all available class names within a ULD configuration that describes an RDF data source. We use upper-case terms to denote variables and lower-case terms to denote constants.

```
classname(X)  :- c-inst(C, class), struct-of(C, label, X)
```

Similarly, the following query returns the property names of all classes in an RDF configuration.

```
propname(X, Y)  :-  c-inst(C, class), c-inst(P, property),
                    struct-of(P, domain, C), struct-of(C, label, X),
                    struct-of(P, label, Y)
```

Note that these two queries use only schema definitions within the data source (i.e., classes and properties). The following query accesses both schema and data information to return all title values in the configuration.

```
filmtitle(X)  :-  c-inst(P, property), struct-of(P, label, 'title'), c-inst(T, triple),
                  struct-of(T, pred, P), struct-of(T, obj, X)
```

It is also possible to query data instances without accessing schema definitions. For example, the following query returns the URI of all RDF resources used as a property in at least one triple. The returned resource may or may not be associated with schema. For example, this query would return, in addition to the defined properties, the '#stars' and '#source' properties of Figure 11.

```
dataprop(X)  :-  c-inst(T, triple), struct-of(T, pred, P), struct-of(P, uri-val, X)
```

Once the above query is executed, a user may wish to find additional information about particular properties. For example, the following query returns all values of the '#source' property, where '#source' was returned by the previous query. Note that this is an example of navigation, in which the user has found an item of interest (here, the ‘#source’ property), and is now “traversing” the property, finding additional information (e.g., the value ‘netflix’).

```
propval(X)  :-  c-inst(T, triple), struct-of(T, pred, P), struct-of(P, uri-val, ‘#source’),
                struct-of(T, obj, X)
```

The following queries are similar to the previous examples, 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; the third finds the set of movie titles; the fourth finds all available attribute names (a data-only query); and the last query finds the set of attribute values for review ‘source’ attributes in the configuration.

```
elemtype(X)     :- c-inst(E, elem-type), struct-of(E, name,X)
atttype(X,Y)    :- c-inst(E, elem-type), struct-of(E, name, X), struct-of(E, atts, AS),
                   member-of(A, AS), struct-of(A, name, Y)
title(X)        :- c-inst(E, elem-type), struct-of(E, name, ‘title’), d-inst(T, E),
                   struct-of(T, children, C), member-of(X, C)
atts(X)         :- c-inst(A, attribute), struct-of(A, name, X)
titleattval(X)  :- c-inst(A, attribute), struct-of(A, name, ‘source’), struct-of(A, value, X)
```

The next three queries are similar to the first three above, but are expressed against a relational database configuration.

```
relation(X)     :- c-inst(R, relation), struct-of(R, name, X)
atttype(X,Y)    :- c-inst(R, relation), struct-of(R, name, X), structof(R, atts, AS),
                   member-of(A, AS), struct-of(A, name, Y)
title(X)        :- c-inst(A, attribute), struct-of(A, name, ‘title’), d-inst(V, A),
                   struct-of(V, value, X)
```

Finally, queries can be posed against data-model constructs directly. The following query returns all schema constructs within any given configuration. By schema construct, we mean a construct that appears as the range of a conformance definition.

```
schemastruct(S)  :- conf(C, S, D, R)
```

Simlarly, the following query returns all constructs that serve as struct-ct schema constructs along with their component selectors.

```
schemastruct(S, P)  :- ct-inst(S, struct-ct), conf(D, S, X, Y), struct-type-of(S, P, C)
```

It is also possible to access data values directly within a configuration, without having prior knowledge of the data model used. As a simple example, the following query returns all atomic values of struct-ct construct components that serve as data for a corresponding schema item within a configuration.

```
dataval(V)  :- ct-inst(S, struct-ct), d-inst(D, S), struct-type-of(S, P, C),
               ct-inst(C, atomic-ct), struct-of(D, P, V)
```

3.2 Accessing Multiple Data Sources

Using the ULD query language, it is possible to access multiple data sources at once. For this purpose, we permit ULD query atoms to be prefixed with a data-source identifier. Intuitively, prefixed atoms are true if and only if they are true in the configuration specified by the prefix.

To illustrate, the following two queries combine the XML and RDF sources given in Figures 10 and 11 into a single "view" that provides the name, genre, and review information for all films in the two sources. The following rule represents the XML mapping, where "xml" is used as the prefix identifier of the XML data source.

```
review(F, G, R, S)  :- xml:d-inst(M, movie), xml:struct-of(M, children, C),
                       xml:member-of(T, C), xml:d-inst(T, title),
                       xml:struct-of(T, children, TC), xml:member-of(F, TC),
                       xml:member-of(GE, C), xml:d-inst(GE, genre),
                       xml:struct-of(GE, children, GC), xml:member-of(G, GC),
                       xml:member-of(RE, C), xml:d-inst(RE, review),
                       xml:struct-of(RE, atts, AS), xml:member-of(A1, AS),
                       xml:d-inst(A1, rating), xml:struct-of(A1, value, R),
                       xml:member-of(A2, AS), xml:d-inst(A2, source),
                       xml:struct-of(A2, value, S).
```

The following rule represents the RDF mapping, where "rdf" is used as the prefix identifier of the RDF data source.

```
review(F, G, R, S)  :- rdf:d-inst(M, G), rdf:c-inst(SC, sub-class),
                       struct-of(SC, sub, G), struct-of(SC, super, film),
                       rdf:c-inst(T1, triple), rdf:struct-of(T1, subj, M),
                       rdf:c-inst(T1, pred, title), rdf:c-inst(T1, obj, F),
                       rdf:c-inst(T2, triple), rdf:struct-of(T1, subj, M),
                       rdf:struct-of(T1, pred, review), rdf:struct-of(T1, obj, RO),
                       rdf:c-inst(T3, triple), rdf:struct-of(T3, subj, RO),
                       rdf:struct-of(T3, pred, stars), rdf:struct-of(T3, val, R),
                       rdf:c-inst(T4, triple), rdf:struct-of(T4, subj, RO),
                       rdf:struct-of(T4, pred, source), rdf:struct-of(T4, val, S)
```

Note that these queries are "global-as-view" definitions [24] over the two sources that reconciles both their data-model and schematic heterogeneities (with respect to the corresponding film information).

4 The ULD Transformation Language

One of the main applications of meta-data-models is data-model transformation. A number of approaches [9,10,25] attempt to automatically convert schemas in one data model to equivalent schemas in another data model according to fixed sets of model-to-model conversion rules. However, this approach (similar to the problem of schema mapping) is only applicable for certain data models and under restrictive assumptions. In practice, user-defined programs are typically used to convert between data sources having distinct data models. One reason for the use of these special-purpose programs is that data-model constructs are typically not one-to-one. For example, there are various ways to transform an XML data source into a relational data source, and each transformation can have benefit in different situations [26].

The goal of the ULD transformation language is to provide a declarative and formal language for specifying transformation rules across structurally heterogeneous data sources. Such languages can help users to more easily define data-model transformations when compared with the current use of *ad-hoc* and special-purpose programs. As with the ULD query language, the ULD transformation language facilitates conversions at various levels across data sources. The rest of this section describes the basic ULD transformation language, and demonstrates different types of transformations.

4.1 ULD Logic-Based Transformation

The ULD transformation language is similar in spirit to the Well-founded Object Language (WOL) [27]. Both are declarative, logic-based languages for expressing mappings. WOL is specifically meant for a restricted object-oriented data model and for expressing schema mappings. We extend this approach to express a wide range of transformations, including schema mappings for schemas represented in different data models and generic data-model transformations.

A ULD mapping rule is an extended Datalog rule expressed against source and target configurations. For example, the following simple rule converts RDF classes to relations in the relational data model.

```
rel:c-inst(C, relation) ⇐ rdf:c-inst(C, class)
```

The rule is read “if C is a class identifier in the “rdf” configuration, then C is a relation identifier in the “rel” configuration,” where “rdf” is the source configuration identifier and “rel” the target configuration identifier. The mapping above states only that RDF identifiers in a source configuration are mapped to relation identifiers in a target configuration. Additional rules are needed to transform class names to relation names, class properties to relation attributes, and so on.

More formally, a *transformation* consists of one or more mapping rules and an optional set of rules that define intensional predicates (used by the mapping rules). We assume that a target configuration contains a description of its data model prior to the execution of a transformation. Target configurations may also contain additional schema and instance data.

ULD mapping rules extend Datalog by allowing Skolem functions as well as multiple atoms in a rule head. Note that this treatment of rules is similar to second-order tuple-generating dependencies used for a similar purpose in schema mappings [28], as well the rule language used in WOL [27], among others [25,29,30]. ULD mapping rules take the general form

$$\psi(\mathbf{x}) \Leftarrow \varphi(\mathbf{x})$$

where the head of the rule is a conjunction $\psi(\mathbf{x})$ of one or more (prefixed) ULD atoms expressed over a target configuration (for $\mathbf{x}$ a vector of terms, i.e., variables, constants, or Skolem terms), and the body of the rule is a conjunction of zero or more (prefixed) atoms $\varphi(\mathbf{x})$ expressed against a source configuration. We note that φ can also contain atoms expressed against a target configuration, intensional atoms defined in the transformation, or equations assigning variables to Skolem terms.

When a mapping rule is executed, each resulting atom in the head of the rule is added to (or asserted in) a target configuration. In other words, a mapping rule specifies one or more updates to target configurations. The output of a transformation is the updated target configuration that results from computing the fixpoint of the corresponding mapping and intensional rules.

In the previous example, a new relation is constructed in the target configuration for every class in the source configuration. As another example, the following mapping rule adds the name to every generated relation in the target configuration.

```
rel:struct-of(C, name, N), rel:c-inst(N, uld-string) ⇐ rdf:c-inst(C, class),
    rdf:struct-of(C, label, N)
```

Each atom in the head of the rule is asserted in the target configuration such that every variable is replaced with its corresponding variable assignment. (A new variable assignment is obtained in each evaluation of the body of the rule.)

In certain cases, identifiers in the source configuration can conflict with identifiers in the target configuration. The following rule uses a Skolem function to generate non-conflicting identifiers in the target configuration.

rel:c-inst(R, relation) ⇐ rdf:c-inst(C, class), R = f(C)

For convenience, we provide a default Skolem function *id* for defining explicit mappings between identifiers. The function *id* is special syntax that denotes a unique variable-arity Skolem function that generates new identifiers using existing source-configuration identifiers and optional labels (constants).

The following mapping uses *id* to define the explicit connection between source and target variables. The rule generates a relation identifier and an attribute-set identifier for each class identifier in the source.

```
rel:c-inst(R, relation), rel:struct-of(R, name, N), rel:struct-of(R, atts, A),
    rel:c-inst(A, att-set) ⇐ rdf:c-inst(C, class), rdf:struct-of(C, label, N),
    R = id(C, rel), A = id(C, att)
```

In this example, given the same value for C (where "rel" is a constant), R = *id*(C, rel) always unifies R with the same identifier. Similarly, A = *id*(C, att) always unifies A with the same identifier, different from R (because of the different constant att as opposed to rel). Other mapping rules can now obtain the relation (or attribute set) identifier generated from the original class (or attribute set), as shown in the following mapping rule, which converts properties to relation attributes.

```
rel:c-inst(T, attribute), rel:struct-of(T, name, N), rel:member-of(T, A) ⇐
    rdf:c-inst(P, prop), rdf:struct-of(P, domain, C), rdf:struct-of(P, label,N),
    A = id(C, att), T = id(P).
```

In general, *id* takes the form V = *id*(V_1, V_2, ..., V_n) for $n \geq 1$, where V_1 to V_n are (ground) variables representing source-configuration identifiers or constants and V is a variable bound with the created identifier. If V is already bound, the equation $V = id(V_1, \ldots, V_n)$ evaluates to true if V can be unified with the value of the generated identifier, and false otherwise.

We permit other special-purpose functions in addition to *id* such as functions to concatenate strings. User-defined functions are also permitted, which can also help resolve specific differences in data values. For example, a user-defined function can be used to convert full names in last-name-first format to full names in first-name-first format. As in Datalog, we require function parameters to be ground when a function is called, i.e., every parameter is either a constant or a variable unified with a constant through a variable binding [23].

4.2 General Data-Model Transformations

A generic data-model transformation converts any configuration of a particular data model into a corresponding configuration within a different data model. The following examples demonstrate part of a standard XML to relational transformation. In this generic transformation, each XML element type is mapped to a relation, and each element is mapped to a tuple with a unique identifier. The attribute set of the corresponding element type's relation also contain the attributes of the element type as well as an attribute named "content" for element types containing only pcdata. For element types with complex content models, "edge" relations are created (i.e., representing an XML tree edge). This transformation is often referred to as the "attribute" approach [26]. For a more detailed version of this transformation, see [17]. The result of the mapping for the data source of Figure 10 is shown in Figure 12. Below we describe the part of the transformation that constructs the relations of Figure 12.

The first mapping rule below converts every XML element type into a relational table. Note that for simplicity we drop the prefixes in the following rules, but assume that each formula in the body selects information from the XML source and each formula in the head inserts information into the relational source.

```
c-inst(E, relation), struct-of(E, name, N), c-inst(N, uld-string) ⇐ c-inst(E, elem-type),
        struct-of(E, name, N)
```

The following rule creates an attribute set for each corresponding relation of the element type, and adds a new attribute to store the unique identifier of each corresponding element.

```
c-inst(E, relation), c-inst(AS, att-set), c-inst(A, attribute), struct-of(A, name, 'id'),
        struct-of(A, domain, 'uld-string'), member-of(A, AS) ⇐ c-inst(E, elem-type),
        AS = id(E, atts), A = id(E, id)
```

The next rule creates a content attribute for element types having pcdata content models.

```
c-inst(C, attribute), struct-of(C, name, 'content'), struct-of(C, domain, 'uld-string'),
        member-of(C, AS) ⇐ c-inst(E, elem-type), struct-of(E, model, M),
        member-of(pcdata, M), AS = id(E, atts), C = id(E, content)
```

The following rule creates a relational attribute for each element-type attribute.

```
c-inst(A, attribute), struct-of(A, name, T), struct-of(A, domain, 'uld-string'),
        member-of(A, AS) ⇐ c-inst(E, elem-type), struct-of(E, atts, AD),
        member-of(A, AD), struct-of(A, name, T), AS = id(E, atts)
```

Finally, the next rule creates an edge relation (i.e., storing the edges from an XML document tree; see [26]) from each complex content model. We use the *concat* function to combine the two names of the element types for the generated relation.

```
c-inst(R, relation), struct-of(R, name, N3), c-inst(A1, attribute), c-inst(A2, attribute),
        struct-of(A1, domain, 'uld-string'), struct-of(A2, domain, 'uld-string'),
        struct-of(A1, name, E1), struct-of(A2, name, E2), member-of(A1, AS),
        member-of(A2, AS) ⇐ c-inst(E1, elem-type), struct-of(E1,model, M),
        member-of(E2, M), struct-of(E1, name, N1), struct-of(E2, name, N2),
        N3 = concat(N1, N2), R = id(E1, E2, rel), AS = id(E1, E2, atts),
        A1 = id(E1, E2, att1), A2 = id(E1, E2, att2)
```

The data-level, element-to-tuple mappings can similarly be defined as ULD transformation rules. In addition to this mapping, other standard XML to relational mappings can be constructed using the ULD transformation language.

4.3 Specialized Transformations

In addition to generic data-model transformations, it is also possible to define more specialized mappings using the ULD. An example is a schema-to-schema mapping, where instances of a particular schema in one data model are converted to corresponding instances of a schema in another model.

The following transformation rule is a simple schema-to-schema mapping, in which instances of the DTD of Figure 10 are converted to instance data of the RDF data source of Figure 11.

```
c-inst(M, uri-ref), struct-of(M, uri-val, U), d-inst(M, comedy), c-inst(T, triple),
    struct-of(T, subj, M), struct-of(T, pred, title), struct-of(T, obj, V) ⇐
    d-inst(M, movie), struct-of(M, children, MC), member-of(G, MC),
    d-inst(G, genre), struct-of(G, children, GC), member-of('comedy', GC),
    member-of(E, MC), d-inst(E, title), struct-of(E, children, EC),
    member-of(V, EC), U = id(M, uri), T = id(M, E)
```

This mapping converts each movie element whose genre is "comedy" to a corresponding comedy instance in the RDF source. Note that in this example, the mapping is not strictly schema-to-schema: we promote the genre value in the source to schema in the target. This flexibility of the ULD allows for a wide range of specialized transformations, which can map between various data-model, schema, and instance levels in both source and target configurations.

As another example of a specialized transformation, the ULD can also be used to capture "model-to-data" conversions. The purpose of a model-to-data mapping is to specify a serialization (or encoding) of one data model using another data model. For example, the RDF data model defines a standard XML serialization in which any RDF source (both schema and data) can be represented as an XML document. An advantage of using the ULD transformation language for model-to-data mappings is that it provides a formal, explicit, and executable definition of the desired serialization.

5 Related Work

There has been a great deal of work over the past few decades on the problem of information integration, particularly in a database context. Most approaches for *moviedb*

id
id1

movie

id
id2

title

id	*content*
id3	'The Usual Suspects'

studio

id	*content*
id4	'Gramercy

genre

id	*content*
id5	'Thriller'

review

id	*rating*	*source*
id6	'8.7'	'IMDB'

moviedbmovie

moviedb	*movie*
id1	id2

movietitle

movie	*title*
id2	id3

moviestudio

movie	*title*
id2	id4

moviegenre

movie	*title*
id2	id5

moviereview

movie	*review*
id2	id6

Fig. 12. The result of applying the attribute transformation

integration focus on resolving differences in schema as opposed to differences in data models. Schema transformation approaches (e.g., [29,27,30]) convert valid instance data under one schema to valid instance data under another schema, where the source and target schemas are assumed to be heterogeneous. These approaches typically define a special-purpose, query-based transformation language to express schema mappings that, when executed, perform the desired instance-data conversion. In addition, a number of approaches attempt to determine schema mappings [31] automatically (or semi-automatically) by finding matching names and structures within the schemas. Schema matching is a central component of so-called model management [12,32,33] where schema mappings can be used to solve several metadata management problems.

Approaches for schema transformation and integration typically assume all schemas are represented in a single data model. When multiple data models are present, a common approach is to convert all data to a single data model. In general, developers use *ad hoc*, special-purpose programs for resolving structural heterogeneity [12]. Writing programs to convert information between data models is often a time-consuming and error-prone task that leads to complex programs that are difficult to maintain and reuse.

In this section, we first discuss related meta-data-model architectures and, second, we present several models with associated query languages that share some of the goals of the ULD.

5.1 Meta-Data-Model-Based Architectures

A self-describing data model [7] adds limited meta-data-model capabilities to an existing data model in which the meta-data-model and the data model are the same. That is, the same storage structures are used for explicitly representing schema as for representing data. For example, a self-describing representation of the relational data model would encode a relational schema—the set of relation names, their attributes, foreign and primary keys, and domains—using a fixed set of relations, called a catalog schema. The catalog instance can then be queried, e.g., to find the attributes of a relation. This approach has been adopted and is supported by most database management systems.

Meta-data-model architectures (see Figure 1) are often more general than self-describing data models in that they are designed to enable interoperability across heterogeneous data models. These approaches describe each heterogeneous source using the same meta-data-model, which provides a common representation for accessing schemas via the meta-schema structures. A number of approaches use E-R (entity and relationship) structures to define meta-schemas and their corresponding schemas [9,10,8,13]. In this case, meta-schema constructs are defined as patterns, or compositions of entity and relationship types. These patterns are then used to define entities and relationships representing corresponding schema items. Using this approach, a meta-schema for an 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 (with YAT [11] being an exception), data (e.g., the bottom level on the right side of Figure 1) is stored in the database system, and is not represented in the architecture explicitly.

An important assumption made by existing meta-data-model approaches is that data, the bottom level on the right side of Figure 1, must adhere to the schema, the third level down on the right side of Figure 1. That is, all data is considered to be an instance of existing schema definitions. This assumption stems from traditional database systems, which generally require complete schema in which all data must satisfy all of the constraints imposed by a schema. Optional- and partial-schema data models cannot be represented with the assumptions underlying such a traditional architecture. There is no way to instantiate meta-schema constructs to create data without first creating a schema construct; the meta-schema only defines structures for representing schema.

In the remainder of this section, we consider several specific meta-data-model-based systems.

Atzeni and Torlone's MDM [9,10] 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 [11], 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. YAT uses a declarative, Horn-clause language to define schema mappings for data conversion.

Barsalou and Gangopadhyay use metatypes [8], 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 is assumed to contain instances of the corresponding schema types. Metatypes are formalized as a second-order structure.

The Meta-Object Facility (MOF) [13] is a meta-data-model architecture proposed by the Object Management Group. The MOF contains a complex and large number of primitive structures (inspired by the UML meta-model) for defining data models. The main focus of the MOF is to enable information models, such as UML class diagrams, to be shared between applications.

Although in general it is possible to represent a data model and its associated schema(s) and data in other data models or representation schemes, the ULD is the only representation scheme that we are aware of that explicitly describes, retains, and exploits the description of heterogeneous data models and can also accommodate flexibility in the use of schemas.

5.2 Related Data Models and Query Languages

The use of a flat model for the ULD, with support for optional and multiple levels of schema, was inspired by RDF as evidenced by an earlier version of the ULD [16] with only two structural primitives, named construct and connector, much like the RDF resource and property, respectively. The version of the ULD presented here extends our earlier ULD model by providing explicit constructs for records (structs) and collections (sets) and by allowing arbitrarily nested structures. These additional structural constructs permit a natural and direct description of a wider range of data models than the earlier, simpler version of the ULD. These additional structures also helped to solve performance problems associated with the earlier version of the ULD caused by the number of join operations required to reconstruct, e.g., a record and its constituent attributes.

There have been several RDF query languages proposed, including RQL [34], RDQL [35], and SPARQL [36], the most recent. These query languages differ in purpose, syntax, and expressive power as described in a recent survey paper [37]. But all of these languages, naturally, provide explicit mechanisms to match and extract information from RDF data sources. Because RDF is a flat representation, where triples that describe the schema can be intermixed with triples that describe instance data along with an explicit triple to connect the two, an RDF query language can solve the problem of schematic heterogeneity by providing direct access to schema and instance data in a single query. However, information in other data models must first be transformed or encoded in RDF before any of the RDF query languages can be used and RDF lacks specific constructs for describing other data model constructs.

F-Logic [22] also shares some of the same goals as the ULD in that F-Logic seeks to query both the schema and the data, in a single, first-order, logic-based language. F-Logic was designed to formalize object-oriented models. The approach used in F-Logic to solve the problem of schematic heterogeneity is to include data instances (i.e., objects) at the leaf level in the class hierarchy that represents the schema. F-Logic is limited to describing data models where the schema must be described before any data can be entered.

IQL [38] is a functional query language that provides very rich structuring capability, with arbitrary nesting of structures. The structuring capability of IQL is similar to the structuring capability of the ULD except that nested structures in IQL are directly nested whereas in the ULD each component of a record or element of a set construct can hold the id of another structure as opposed to the actual, nested structure.

In general, the ULD is the only meta-data-model approach we know of that explicitly models conformance (through the conf and d-inst predicates). We believe this is due, in part, to the assumption that schema is required in most of the related approaches. In addition, in most approaches, transformation is defined at the data model or schema level (but not both).

6 Concluding Remarks

This paper describes the Uni-Level Description (ULD): a high-level and declarative meta-data-model language for representing heterogeneous data sources. A ULD representation of a data source includes an explicit description of the source's data-model constructs, schema information, and instance data. The ULD is a flat representation that provides uniform access to all levels of a data source, facilitating the management of structural heterogeneity. The ULD is also a structural model in that it provides a structural type system (consisting of record, collection, and primitive types) for defining data-model constructs, schemas, and instance data. The structural values associated with constructs, schemas, and instance data are represented in the ULD using a small set of first-order formulas, which can be directly queried and transformed using Datalog.

Using the ULD, data-model constructs are explicitly described and instantiated to represent the schema and instance information of a source. The ULD offers a novel approach that permits both schema constructs and data constructs of a model to be described. The ULD also permits special relationships for defining the conformance between schema and data constructs. This approach can be used to describe a wider range of data models (including RDF, Topic Maps, and XML) and can more accurately capture the often subtle and complex schema and data relationships within a model.

We describe both a Datalog-based ULD query and transformation language, and demonstrate their use through a number of examples. The query language can be used against a single ULD configuration (i.e., data source described in the ULD) or multiple configurations for defining integrated views over sources. The transformation language can be used to define a wide range of conversions between data sources, including model-to-model, schema-to-schema, model-to-schema, and numerous variants.

In this paper, we also compare the ULD to other meta-data-model approaches and briefly describe our ongoing Prolog-based implementation. As future work, we intend to use the rich descriptions provided by the ULD to help verify and validate transformation rules and further develop generic, data-model independent operators, i.e., operations that can be generically applied across a variety of data models, such as those for enabling incremental navigation and browsing.

References

[1] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler (Eds.), Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, World Wide Web Consortium (W3C), 2000, http://www.w3.org/TR/2000/REC-xml-20001006.

[2] O. Lassila, R. R. Swick (Eds.), Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation, World Wide Web Consortium (W3C), 1999, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[3] M. Biezunski, M. Bryan, S. Newcomb, Topic maps, ISO/IEC 13250 (2000).

[4] J. Hammer, D. McLeod, On the resolution of representational diversity in multidatabase systems, in: Management of Heterogeneous and Autonomous Database Systems, Morgan Kaufmann, 1998, pp. 91–118.

[5] R. J. Miller, Using schematically heterogeneous structures, in: Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1998, pp. 189–200.

[6] L. V. Lakshmanan, F. Sadri, I. N. Subramanian, SchemaSQL - A language for interoperability in relational multi-database systems, in: Proceedings of the 22nd Conference on Very Large Databases (VLDB), 1996.

[7] L. Mark, 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), 1983, pp. 585–602.

[8] T. Barsalou, D. Gangopadhyay, M(DM): An open framework for inter-operation of multimodel multidatabase systems, in: Proceedings of the 8th International Conference on Data Engineering (ICDE), 1992, pp. 218–227.

[9] P. Atzeni, 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), 1995, pp. 345–364.

[10] P. Atzeni, R. Torlone, Management of multiple models in an extensible data-base design tool, in: Proceedings of the 5th International Conference on Extending Data-base Technology (EDBT), Vol. 1057 of LNCS, 1996, pp. 79–95.

[11] V. Christophides, S. Cluet, J. Siméon, On wrapping query languages and efficient xml integration, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2000, pp. 141–152.

[12] P. A. Bernstein, A. Y. Halevy, R. Pottinger, A vision of management of complex models, SIGMOD Record 29 (4) (2000) 55–63.

[13] OMG, Meta Object Facility (MOF) Specification, OMG Document ad/97-08-14 (September 1997).

[14] K. T. Claypool, E. A. Rudensteiner, Sangam: A framework for modeling heterogeneous database transformations, in: Proceedings of the 5th International Conference on Enterprise Information Systems, Vol. 1, 2003, pp. 219–224.

[15] M. Jarke, S. Eherer, R. Gallersdórfer, M. A. Jeusfeld, M. Staudt, ConceptBase – A deductive object base manager, Tech. rep., Aachener Informatik-Berichte 93-14 (1995).

[16] S. Bowers, L. Delcambre, Representing and transforming model-based information, in: Proceedings of the Workshop on the Semantic Web at ECDL, 2000.

[17] S. Bowers, The uni-level description: A uniform framework for managing structural heterogeneity, Ph.D. thesis, OGI School of Science and Engineering, OHSU (2003).

[18] S. Bowers, 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), Vol. 2813 of LNCS, 2003.

[19] S. Bowers, L. Delcambre, Incremental navigation: Providing simple and generic access to heterogeneous structures, in: Proceedings of the 23rd International Conference on Conceptual Model (ER), Vol. 3288 of LNCS, 2004, pp. 668–681.

[20] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley Publishing Company, 1995.

[21] University of Amsterdam, SWI-Prolog, http://www.swi-prolog.org/.

[22] M. Kifer, G. Lausen, J. Wu, Logical foundations of object-oriented and frame-based languages, Journal of the ACM 42 (4) (1995) 741–843.

[23] J. D. Ullman, Priniciples of Database and Knowledge-Base Systems, Volume II, Computer Science Press, 1989.

[24] M. Lenzerini, Data integration: A theoretical perspective, in: Proceedings of the Twenty-first ACM Symposium on Principles of Database Systems (PODS), 2002.

[25] P. Atzeni, P. Cappellari, P. A. Bernstein, ModelGen: Model independent schema translation, in: Proceedigns of the International Conference on Data Engineering (ICDE), 2005, pp. 1111–1112.

[26] F. Tian, D. J. DeWitt, J. Chen, C. Zhang, The design and performance evaluation of alternative XML storage strategies, SIGMOD Record 31 (1) (2002) 5–10.

[27] S. B. Davidson, A. Kosky, WOL: A language for database transformations and constraints, in: Proceedings of the 13th International Conference on Data Engineering (ICDE), 1997, pp. 55–65.

[28] R. Fagin, P. G. Kolaitis, L. Popa, W. C. Tan, Composing schema mappings: Second-order dependencies to the rescue, in: Proceedings of the Twenty-third ACM Symposium on Principles of Database Systems (PODS), 2004, pp. 83–94.

[29] S. Abiteboul, S. Cluet, T. Milo, Correspondence and translation for heterogeneous data, in: Proceedings of the 6th International Conference on Database Theory, Vol. 1186 of LNCS, 1997, pp. 351–363.

[30] L. Popa, Y. Velegrakis, R. J. Miller, M. Hernández, R. Fagin, Translating Web data, in: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), 2002, pp. 598–609.

[31] E. Rahm, P. A. Bernstein, A survey of approaches for automatic schema matching, The VLDB Journal 10 (4) (2001) 334–350.

[32] P. A. Bernstein, Applying model management to classical meta data problems, in: Proceedings of the Conference on Innovative Data Systems Research (CIDR), 2003, pp. 209–220.

[33] S. Melnik, P. A. Bernstein, A. Halevy, E. Rahm, Supporting executable mappings in model management, in: Proceedings of the ACM SIGMOD International Conference, 2005.

[34] G. Karvounarakis, S. Alexaki, D. P. Vasillis Christophides, M. Scholl, RQL: A declarative query language for RDF, in: Proceedings of the WWW 2002 Conference, 2002.

[35] A. Seaborne (Ed.), RDQL - A Query Language for RDF, W3C Submission, World Wide Web Consortium (W3C), 2004, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[36] E. Prud'hommeaux, A. Seaborne (Eds.), SPARQL Query Language for RDF, W3C Working Draft, World Wide Web Consortium (W3C), 2005, http://www.w3c.org/TR/2005/WD-rdf-sparql-query-20050419/.

[37] P. Haase, J. Broekstra, A. Eberhart, R. Volz, A comparison of rdf query languages, in: Proceedings of the Third International Semantic Web Conference, 2004.

[38] A. Poulovassilis, A tutorial on the iql query language, version 1.0, Tech. rep., AutoMed Technical Report No. 28 (2004).