

On Modeling Conformance for Flexible Transformation over Data Models

Shawn Bowers and Lois Delcambre
OGI School of Science & Engineering, OHSU
Beaverton, Oregon, 97006, USA
{shawn, lmd}@cse.ogi.edu

Abstract. Data models use a limited number of *basic structures* to represent data such as collections, attribute-value pairs, and scalars. They differ, however, in how structures are composed and whether they *permit* the specification of schema, permit *more than one schema*, or *require* schema. Support for transforming between heterogeneous representation schemes remains a significant challenge. To this aim, we extend our work on generic representation and transformation of model-based information by introducing a richer metamodel and abstract framework for representation. We also introduce several steps toward our vision of high-level transformations through mapping patterns and by exploiting inherent constraints.

1 Introduction

Taking information in one representation scheme (such as XML) and extracting some or all of it for use in another scheme (such as a Topic Map, a Relational database, or as RDF triples) is a surprisingly difficult task. In fact, few tools exist to help perform such transformations, which means that (potentially) complex, special purpose programs must be written for even very simple mappings. One reason such conversions are difficult is that representation schemes differ in the basic structural constructs and schema constraints they provide for organizing information. As a consequence, straightforward, one-to-one mappings between schemes rarely exist. In addition, the absence of high-level languages for expressing transformations between schemes places the burden of conversion on these special-purpose programs, making it difficult to define, maintain, and reason about transformation.

We observe, however, that structured information is typically based on a small set of structural constructs, composed in various ways. When described explicitly, these structural constructs can be exploited directly for transformation. For example, the constructs offered by the Relational model can be viewed as a set of tables, each consisting of a bag of rows, where each row has a set of named atomic or scalar attribute values. Similarly, an XML document can be viewed as a set of elements, possibly with attributes, where each element has a list of sub-elements or scalar values (PCDATA). We propose that a wide range of useful transformations can be performed through mappings between representation scheme constructs, *e.g.*, elements to tuples in tables, and so on.

In this chapter, we present a uniform framework that allows the description of arbitrary representation schemes along with a transformation language that can easily convert information within and across schemes. The goals of our research are to:

1. Capture the representation scheme or data model explicitly—by instantiating and composing the basic structures (or construct types) of a metamodel,
2. Support representation schemes with varying types of conformance, *i.e.*, we model schema-instance relationships for representation schemes where schema (or type information) can be required, optional, or can occur at multiple levels, and
3. Express transformation rules declaratively, *e.g.*, by allowing users to indicate simple correspondences between data model constructs and converting the associated information automatically.

The rest of this chapter is organized as follows. Section 2 describes various data models and representation schemes, with emphasis on how each differs in their use of conformance relationships. Section 3 presents the uniform framework for representing structured information, along with the four relationships of interest. Section 4 introduces transformation rules that exploit this representation and also discusses the possibility of higher-level transformation rules, making rule specification easier. Section 5 discusses related work and we conclude in Section 6 with a brief discussion of work in progress.

2 Data Models and Conformance

Database systems are *schema-first* in that a schema must be defined before any data can be placed in the database. The role of the schema is to introduce and name application-specific structures (such as the “Employee” table with “SSN” and “Name” as attributes for Employee) that will be used to hold application data. A schema imposes uniform structure and constraints on application data. Various tools such as a query processor, in turn, can exploit this uniform structure.

A number of data models exists that are not schema-first, including XML and other semi-structured models, RDF, the Topic Map Model, and various hypertext data models [1]. Both RDF and XML are models where the schema, *i.e.*, the RDF schema or DTD, respectively, is optional. More than that, even with a schema, the Topic Map model, RDF, and XML (with an “open” DTD) permit additional, schema-free structures to be freely mixed with structures that conform to a schema.

We model *conformance relationships* explicitly, as appropriate, for each data model of interest. Table 1 provides example data models with some of their associated structures that participate in conformance. For example, in object-oriented models, objects conform to classes, whereas in XML, elements conform to element types, and so on.

Table 2 describes some of the properties of conformance that are present in the data models of Table 1. First, we consider how the conformance relationship is established with three possibilities: implicitly, upon request, and explicitly as shown in the first section of Table 2. In the next section of Table 2, we see that the schema-first models also require conformance for their data values whereas the other models do not. And the models with optional conformance are also open: they permit conforming and non-conforming data to be freely mixed, as shown in the third section of Table 2. Some models allow a construct to conform to several structures, namely all but the relational model (fourth section of Table 2). Finally, we see that all of the non-schema-first models except XML permit multiple levels of conformance.

To represent a wide range of data models explicitly, we must support variations in the conformance relationship as shown by Table 2. We discuss the details of our representation

Table 1: Example data model structures involved in conformance

<i>Data Model</i>	<i>Selected structures, related by the conformance relationship</i>
Object-Oriented	Objects conform to Classes.
Relational	A Table of Tuples conforms to a Relation Scheme.
XML w/ DTD	Elements conform to Element Types and Attributes to Attribute Types.
RDF w/ RDFS	Objects (resources) conform to Classes and Properties to Property Types.
XTM	Topics conform to Topic Types (Topics), Associations conform to Association Types (Topics), and Occurrences conform to Occurrence Types (Topics).
E-R	Entities conform to Entity Types, Relationships conform to Relationship Types, and Values conform to Value Types.

Table 2: Examples properties of the conformance relationship

<i>Conformance Properties</i>	<i>Data Model</i>
How is the conformance relationship established?	
when data is created (implicitly)	Object-Oriented, Relational, and E-R
when computed (upon request)	XML w/ DTD (documents must be validated)
when conformance is declared (explicitly)	RDF w/ RDFS (via <code>rdf:type</code> property) and XTM (via class-instance association)
Is conformance required?	
required	Object-Oriented, Relational, and E-R
optional	RDF (RDF Schema is not required), XTM (type information not required), and XML (DTD is not required)
Is conformance open?	
no	Object-Oriented, Relational, and E-R
yes	RDF w/ RDFS, XTM, and XML
Is multiple conformance allowed?	
no	Relational
yes	Object-Oriented (inheritance), RDF w/ RDFS (Property/Class with multiple inheritance), XTM (Topics can be instances of multiple Topic Types), and E-R
How many levels of conformance are permitted?	
one	Relational, E-R, and Object-Oriented (when not in the next entry)
two (sometimes)	Object-Oriented (some models support classes as objects)
zero to any number	XML (zero or one), XTM, and RDF

in the next section, discuss transformation in Section 4, and further consider related work in Section 5.

3 The Uniform Framework

The main component of the uniform framework is a three-level, metamodel architecture that incorporates conformance relationships. In this section, we describe our metamodel architecture, introduce a concrete metamodel (*i.e.*, a specific set of basic structures), and present a logic-based description language that enables uniform data model, schema, and instance description.

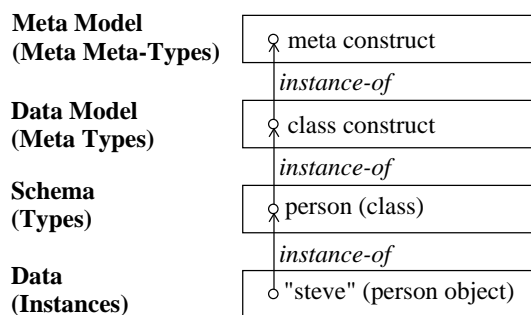


Figure 1: A typical four-level metamodel architecture.

3.1 The Generic Metamodel Architecture

Figure 1 exemplifies a typical metamodel architecture. As shown, the architecture has four levels: the metamodel, data model, schema, and data levels, where each item in a level is considered an instance of an item in the level above it. As shown, typical metamodel architectures are characterized by their assumption that data models are schema-first, *i.e.*, schema items must be created prior to data items.

In contrast, Figure 2 shows the three-level metamodel architecture we employ. The top level (denoted as the metamodel) consists of *construct types*, which represent the basic structures used within data models for defining schema and data. Examples include types such as collection, name-value pair, atomic, and so on. The middle layer defines data model and schema *constructs* along with their composition. For example, both a `class` and `object` construct along with an explicit conformance definition between them (represented by the relationship type labeled `conf`) are defined in the middle layer. In this way, the metamodel is used to define all the constructs of the data model, not just those for creating schema.

Finally, the bottom level represents *instances* and (data-level) *conformance relationships*. For example, an object (with the value “steve”) would be connected with a particular class (with name “person”) using the `d-inst` relationship.

The architecture distinguishes three kinds of instance-of relationships, in addition to the conformance relationship, as shown in Figure 2. Constructs introduced in the middle layer are necessarily an instance (`ct-inst`, read as “construct-type instance”) of a metamodel construct type. (Note that the metamodel currently has a range of basic structures as construct types and the set of construct types can be easily extended if needed.) Similarly, any value introduced in the bottom layer is necessarily an instance (`c-inst`, read as “construct instance”) of the constructs introduced in the middle layer. The flexibility of the framework is due to the conformance relationships. Conformance relationships are expressed within the middle layer (*e.g.*, to indicate that an XML element can optionally conform to an XML element type) and the corresponding data-level instance-of relationships (`d-inst`, read as “data instance”) expressed within the bottom layer (*e.g.*, to indicate that a particular element is an instance of another element type).

A *configuration* consists of a particular choice of metamodel (*i.e.*, set of construct types), a particular data model (*i.e.*, constructs) defined using the construct types, and a collection of instances defined using the constructs. A configuration serves as input to a transformation. Thus, a transformation takes one (or possibly more) configurations and generates a new

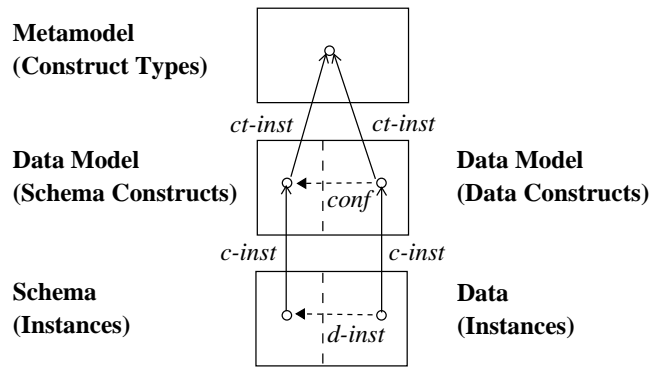


Figure 2: The three-level metamodel architecture.

configuration.

A configuration \mathcal{C} is a tuple (M, D, I, T) whose components are defined as follows.

- M is a set of named construct types in the top layer. (*The Metamodel*)
- D is a tuple (C, R) where C is a set of named constructs defined using the types of M , and R is a set of ordered pairs (c_1, c_2) for $c_1, c_2 \in C$ denoting a conformance definition and is read as “instances of c_1 can conform to instances of c_2 .” (*The Middle Layer*)
- I is a set of named instances. The name of an instance acts as its unique identifier. (*The Bottom Layer*)
- T is a tuple (T_{ct}, T_c, T_d) in which $T_{ct} \cup T_c \cup T_d$ is the set of type-instance relationships of the configuration where T_{ct} , T_c , and T_d are disjoint sets of ordered pairs (i_1, i_2) such that for $t \in T_{ct}$ we require $i_1 \in C$ and $i_2 \in M$ (i.e., *ct-inst*), for $t \in T_c$ we require $i_1 \in I$ and $i_2 \in C$ (i.e., *c-inst*), and for $t \in T_d$ we require $i_1, i_2 \in I$ (i.e., *d-inst*) such that $(i_1, c_1), (i_2, c_2) \in T_C$ and $(c_1, c_2) \in R$. (*Cross-Layer Connections and Data Instances*)

We informally define construct type, construct, and instance as follows.

- A construct type ct is a tuple (adt, P) such that adt represents an abstract (or parameterized) data type and P is a set of restriction properties.
- A construct c is a tuple (ct, ρ) such that $ct \in M$ is a construct type for which c is an instance (and hence $(c, ct) \in T_{ct}$) and ρ is a set of restrictions that obey P and serve to restrict the instances of the construct and define its compositional aspects.
- An instance i is a tuple (c, val) such that $c \in C$ and val is a valid instance of its associated construct type’s adt and obeys the restrictions ρ of c .

Intuitively, a construct type specifies a basic data structure. Thus, the value of a construct instance is a value of its construct type’s basic data structure. A description of composition constraints on the basic structures (i.e., constructs) of a model is also given. For example, a particular construct might represent a collection of name-value pairs.

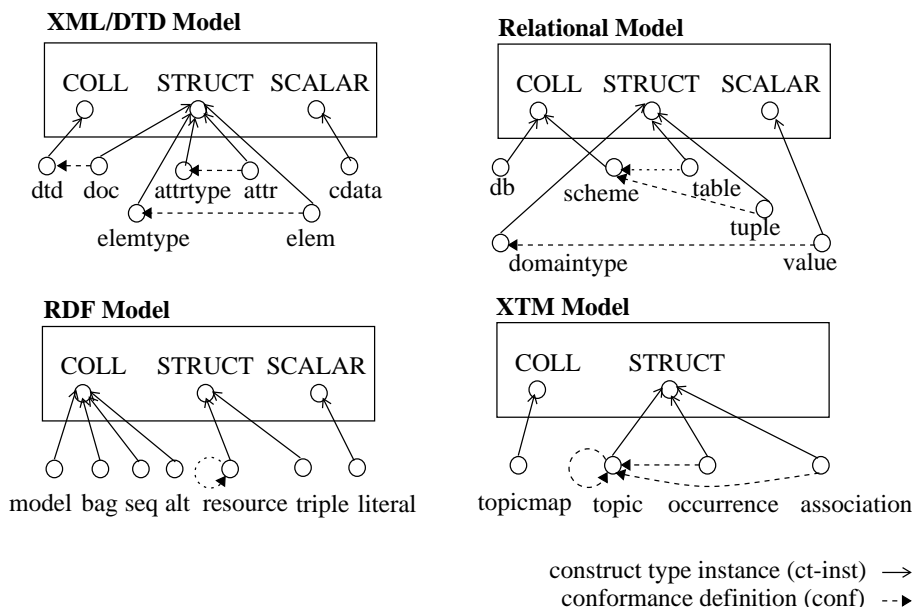


Figure 3: Relational, RDF, and XTM data models described within the generic framework.

3.2 Example Description Language and Basic Structures

Here we introduce a concrete set of metamodel construct types, and demonstrate their use for describing data models. We also present a logic-based language for representing constructs and instances.

The construct types in the metamodel are collection, (specifically, set_{ct} , $list_{ct}$, bag_{ct}), $struct_{ct}$, and $scalar_{ct}$, representing collections, name-value pairs, and atomic data (like strings and integers), respectively. We should note that this choice of construct types is only one possible set of structures that are sufficient for the data models we describe in this chapter. Figure 3 shows the XML, Relational, RDF, and Topic Map data model constructs expressed using our construct types (note that we use COLL to represent the collection construct types). Conformance is shown explicitly, e.g., XML elements can conform to element types, Relational tables to schemes, RDF resources to resources, and Topic Map associations to topics.

We use the following predicates for representing the instance and conformance relationships of configurations. The $ct-inst(c, ct)$ predicate relates a construct c to a construct type ct , which must be either set_{ct} , $list_{ct}$, bag_{ct} , $struct_{ct}$, or $scalar_{ct}$. The $conf(c_1, c_2)$ predicate specifies that a construct c_1 can conform to another construct c_2 . The $c-inst(d, c)$ predicate relates an instance d with a construct c . Finally, the $d-inst(d_1, d_2)$ predicate defines a conformance relationship between instances d_1 and d_2 .

We introduce the $comp$ predicate for describing composition constraints over model constructs. The predicate can take the following forms.

$$\begin{aligned}
 &comp(c_1, setof(c_1, \dots, c_n)) \\
 &comp(c_2, listof(c_1, \dots, c_n)) \\
 &comp(c_3, bagof(c_1, \dots, c_n)) \\
 &comp(c_4, structof(a_1 \rightarrow c_1, \dots, a_n \rightarrow c_n)) \\
 &comp(c_5, union(c_1, \dots, c_n))
 \end{aligned}$$

$ct-inst(dtd, set_{ct})$	$comp(dtd, setof(elem-t))$
$ct-inst(elem-t, struct_{ct})$	$comp(elem-t, structof(tag \rightarrow string, attrtypes \rightarrow attr-tset))$
$ct-inst(attr-t, struct_{ct})$	$comp(attr-t, structof(name \rightarrow string))$
$ct-inst(subelem-t, struct_{ct})$	$comp(subelem-t, structof(parent \rightarrow elem-t, child \rightarrow elem-t))$
$ct-inst(attr-tset, set_{ct})$	$comp(attr-tset, setof(attr-t))$
$ct-inst(doc, struct_{ct})$	$comp(doc, structof(root \rightarrow elem))$
$ct-inst(elem, struct_{ct})$	$comp(elem, structof(tag \rightarrow string, attrs \rightarrow attrset))$
$ct-inst(attr, struct_{ct})$	$comp(attr, structof(name \rightarrow string, val \rightarrow cdata))$
$ct-inst(attrset, set_{ct})$	$comp(attrset, setof(attr))$
$ct-inst(nestedelem, struct_{ct})$	$comp(nestedelem, structof(parent \rightarrow elem, child \rightarrow node))$
$ct-inst(pdata, scalar_{ct})$	$ct-inst(cdata, scalar_{ct})$
$comp(node, union(elem, pdata))$	$conf(doc, dtd)$
$conf(elem, elem-t)$	$conf(attr, attr-t)$

Figure 4: Simplified description of the XML with DTD data model.

$ct-inst(db, set_{ct})$	$comp(db, setof(scheme, table))$
$ct-inst(scheme, set_{ct})$	$comp(scheme, setof(fieldtype))$
$ct-inst(table, struct_{ct})$	$comp(table, structof(name \rightarrow string, rows \rightarrow tuples))$
$ct-inst(tuples, bag_{ct})$	$comp(tuples, bagof(tuple))$
$ct-inst(tuple, set_{ct})$	$comp(tuple, setof(field))$
$ct-inst(fieldtype, struct_{ct})$	$comp(fieldtype, structof(name \rightarrow string, dom \rightarrow domaintype))$
$ct-inst(field, struct_{ct})$	$comp(field, structof(name \rightarrow string, val \rightarrow value))$
$ct-inst(domaintype, struct_{ct})$	$comp(domaintype, structof(name \rightarrow string))$
$ct-inst(value, scalar_{ct})$	$conf(table, scheme)$
$conf(tuple, scheme)$	$conf(value, domaintype)$

Figure 5: Simplified description of the Relational data model.

As shown, each kind of *comp* formula uses an additional formula to specify composition constraints, namely, *setof*, *listof*, *bagof*, or *structof*. For example, c_1 is assumed to be defined as $ct-inst(c_1, set_{ct})$ whose instances are sets that can contain instances of constructs c_1, c_2, \dots, c_n (i.e., the *setof* predicate defines the homogeneity of the construct's instances). We also permit the definition of convenience-constructs (as shown by c_5 above), which do not have corresponding *ct-inst* formulas and must be associated with a *union* predicate. Such constructs serve as a place-holder for the union of the corresponding constructs.

The last predicate we define is *val*, which gives the value of an instance. Each ground *val* formula must have an associated *c-inst* formula (which is also ground) of the appropriate form, as shown below.

$val(d_1, set(v_1, \dots, v_n))$
$val(d_2, list(v_1, \dots, v_n))$
$val(d_3, bag(v_1, \dots, v_n))$
$val(d_4, struct(a_1 = v_1, \dots, a_n = v_n))$

To illustrate the definition language, Figures 4, 5, 6, and 7 define (simplified versions of) the XML with DTD, Relational, Topic Map, and RDF data models, respectively. Note that we assume *string* is a default scalar construct for each model and that the juxtaposition of entries is for exposition only. Finally, Figure 8 shows example instances for the XML model.

<i>ct-inst(tm, set_{ct})</i>	<i>comp(tm, setof(topic, assoc))</i>
<i>ct-inst(assoc, struct_{ct})</i>	<i>comp(assoc, structof())</i>
<i>ct-inst(topic, struct_{ct})</i>	<i>comp(topic, structof(name → string))</i>
<i>ct-inst(assoc-mem, struct_{ct})</i>	<i>comp(assoc-mem, structof(ac → assoc, mem → member))</i>
<i>ct-inst(topic-occ, struct_{ct})</i>	<i>comp(topic-occ, structof(top → topic, occ → occurrence))</i>
<i>ct-inst(occurrence, struct_{ct})</i>	<i>comp(occurrence, structof(val → occ – val))</i>
<i>ct-inst(member, struct_{ct})</i>	<i>comp(member, structof(role → string))</i>
<i>ct-inst(mem-topic, struct_{ct})</i>	<i>comp(mem-topic, structof(mem → member, top → topic))</i>
<i>ct-inst(resource, struct_{ct})</i>	<i>comp(resource, structof(uri → string))</i>
<i>comp(occ-val, union(resource, string))</i>	<i>conf(topic, topic)</i>
<i>conf(assoc, topic)</i>	<i>conf(occurrence, topic)</i>

Figure 6: Simplified description of the Topic Map (XTM) data model.

<i>ct-inst(rdfmodel, set_{ct})</i>	<i>comp(rdfmodel, setof(literal, blank, triple, coll))</i>
<i>ct-inst(uri, struct_{ct})</i>	<i>comp(uri, structof(ref → string))</i>
<i>ct-inst(blank, struct_{ct})</i>	<i>comp(blank, structof())</i>
<i>ct-inst(alt, set_{ct})</i>	<i>comp(alt, setof(node))</i>
<i>ct-inst(seq, list_{ct})</i>	<i>comp(seq, listof(node))</i>
<i>ct-inst(bag, bag_{ct})</i>	<i>comp(bag, bagoof(node))</i>
<i>ct-inst(triple, struct_{ct})</i>	<i>comp(triple, structof(pred → uri, subj → resource, obj → node))</i>
<i>ct-inst(literal, scalar_{ct})</i>	<i>comp(node, union(resource, literal))</i>
<i>comp(coll, union(set, bag, alt))</i>	<i>comp(resource, union(uri, blank, coll, triple))</i>
<i>conf(resource, resource)</i>	

Figure 7: Simplified description of the RDF data model.

(a).	(b).
<!ELEMENT activity (profession hobbyist ...)*>	<activity>
<!ELEMENT profession (surgeon professor ...)*>	<profession>
<!ELEMENT professor (#PCDATA)>	<professor>Maier</professor>
	</profession>
	</activity>
(c).	
<i>c-inst(t, dtd)</i>	<i>val(t, set(et1, et2, et3))</i>
<i>c-inst(t1, elem-t)</i>	<i>val(t1, struct(name = “activity”))</i>
<i>c-inst(t2, elem-t)</i>	<i>val(t2, struct(name = “profession”))</i>
<i>c-inst(t3, elem-t)</i>	<i>val(t3, struct(name = “professor”))</i>
<i>c-inst(s1, subelem-t)</i>	<i>val(s1, struct(parent = t1, child = t2))</i>
<i>c-inst(s2, subelem-t)</i>	<i>val(s2, struct(parent = t2, child = t3))</i>
<i>c-inst(d, doc)</i>	<i>val(d, struct(root = e1))</i>
<i>c-inst(e1, elem)</i>	<i>val(e1, struct(tag = “activity”))</i>
<i>c-inst(e2, elem)</i>	<i>val(e2, struct(tag = “profession”))</i>
<i>c-inst(e3, elem)</i>	<i>val(e3, struct(tag = “professor”))</i>
<i>c-inst(n1, nestedelem)</i>	<i>val(n1, struct(parent = e1, child = e2))</i>
<i>c-inst(n2, nestedelem)</i>	<i>val(n2, struct(parent = e2, child = e3))</i>
<i>c-inst(n3, nestedelem)</i>	<i>val(n3, struct(parent = e3, child = “Maier”))</i>
<i>c-inst(d, t)</i>	<i>c-inst(“Maier”, pcddata)</i>
<i>d-inst(e1, t1)</i>	<i>d-inst(e2, t2)</i>
<i>d-inst(e3, t3)</i>	

Figure 8: An example (a) XML DTD, (b) conforming document, and (c) both represented in the description language.

3.3 Specifying Constraints

In addition to describing composition of basic structures, we can also provide a mechanism for defining the constraints of a data model. Namely, we leverage the logic-based description language for constraints by allowing constraint expressions (*i.e.*, rules). We denote constraints using the annotation *constrain*, *e.g.*, the following rule defines a conformance constraint over the Relational model.

$$\begin{aligned} \text{constrain} : d\text{-inst}(x, t) \leftarrow & \\ & c\text{-inst}(x, \text{table}), c\text{-inst}(y, \text{scheme}), \\ & \neg d\text{-inst}(x, x_s), \neg d\text{-inst}(y_t, y), \\ & \neg \text{non-conform}(x, y). \\ \text{non-conform}(x, y) \leftarrow & \\ & \text{val}(x, b), \text{member}(r, b), \neg d\text{-inst}(r, y). \end{aligned}$$

The constraint is read as: “ x can conform to y if x is a Table, y is a Scheme, x doesn’t conform to any other Schemes, y doesn’t have any other conforming Tables, and every Row in x conforms to y .” Note that specifying constraints in this way provides a powerful mechanism for describing complex data model descriptions [2].

4 Transforming Representation Schemes

In this section, we present a language for transforming representation schemes based on horn-clause logic rules, provide example mappings, and discuss higher-level transformations for simplifying the specification and implementation of complex transformations.

4.1 A Transformation Language

A transformation is applied to one or possibly more configurations in which the result is a new configuration. Here, we only consider the case where a single configuration is transformed. We define a transformation as a function $\sigma : M \times C \rightarrow C'$ where M is a set of *mapping rules*, C is the source configuration, and C' is the new configuration called the “destination” of the transformation. The transformation function σ computes the fix-point of the mapping rules applied to the source and destination configurations.

Mapping rules can take the following form:

$$dp_1, \dots, dp_n \leftarrow p_1, \dots, p_m,$$

where both dp and p are (a) formulas (using the predicates of Section 3) with ground literals or variables as arguments or (b) the formula $id(I, id)$. The id predicate takes a list I of input constants and provides a constant (*i.e.*, id) that serves as a unique identifier for the input list. The id predicate is implemented as a skolem function.¹ The syntax we use for mapping rules is shorthand for rules having a single, outer formula in the head of the program clause (via the *head* formula):

$$\text{head}(dp_1, \dots, dp_n) \leftarrow p_1, p_2, \dots, p_m.$$

¹We use the abbreviation $id(v_1, v_2, \dots, id)$ to denote the list $[v_1, v_2, \dots, v_n]$.

```

% element types become schemes
add : c-inst(S, scheme) ← c-inst(ET, elem-t), id(ET, S).
% attribute types become field types
add : c-inst(FT, fieldtype) ← c-inst(AT, attr-t), id(AT, FT).
% elements become tuples
add : c-inst(T, tuple) ← c-inst(E, elem), id(E, T).
% attribute become fields
add : c-inst(F, field) ← c-inst(A, attr), id(A, F).
% cdata to values
add : c-inst(V2, value) ← c-inst(V1, cdata), id(V1, V2).
% sub element types to schemes
add : c-inst(S, scheme) ← c-inst(ST, subelem-t), id(ST, S).
% field types for sub element type schemes
add : c-inst(FT1, fieldtype), c-inst(FT2, fieldtype) ←
    c-inst(ST, subelem-t), val(ST, struct(parent → P, child → C), id(ST, P, FT1), id(ST, C, FT2).
% nested elements become tuples
add : c-inst(T, tuple) ← c-inst(NE, nestedelem), id(NE, T).
% each scheme has a table
add : c-inst(T, table) ← dest : c-inst(S, scheme), id(S, T).
% tables conform to their schemes
add : d-inst(T, S) ← dest : d-inst(S, scheme), id(S, T).
% values conform to the datatype domain type
add : d-inst(V, datatype) ← dest : c-inst(V, value).
% tuples conform to their schemes
add : d-inst(TP, S) ← c-inst(E, elem), id(E, TP), d-inst(E, ET), id(ET, S).

```

Figure 9: Rules for a model-to-model transformation.

Mapping rules are read in the normal way, *i.e.*, if each p_i for $1 \leq i \leq n$ is true, then each dp_j for $1 \leq j \leq m$ is true. Additionally, we permit two annotations on mapping rule formulas. The *add* annotation indicates that the formula be added to the destination configuration. The *dest* annotation matches formulas against those in the destination configuration (the source configuration is the default). We do not allow transformations to modify the source configuration.

4.2 An Example Model-to-Model Mapping

Figures 9 and 10 show a portion of a simple model-to-model mapping between XML and the Relational model using their descriptions from Section 3. Figure 9 shows instance and conformance rules and Figure 10 shows basic structure conversions. This mapping converts XML element types to Relational schemes and converts the rest of the data accordingly. Specifically, tables contain elements with their associated attribute values and a new table is created for each subelement-type relationship, which is where nested elements are stored. Figure 11 shows the tables that would result from example XML data. Note that we choose to skolemize each identifier to assure uniqueness in the destination (*e.g.*, the destination may contain other values prior to the transformation).

While model-to-model transformations are useful, there are a number of other transformations that can be supported in this framework, including schema-to-schema, model-to-schema (also called “modeling the model” [3]), and various mixtures of each [4, 5].

```

% create an empty scheme for each element type
add : val(S, set()) ← c-inst(ET, elem-t), id(ET, S).
% add a field type for each attribute type
add : val(FT, struct(name = N, dom = cdatatype)) ←
  c-inst(AT, attr-t), val(AT, struct(name=N)), id(AT, FT).
% add a field type for element ids
add : val(FT, struct(name = "id", dom = stringtype)) ← c-inst(ET, elemtype), id(ET, "id", FT).
% add field types to associated element type schemes
add : member(FT, FTSet) ←
  c-inst(ET, elem-t), val(ET, struct(tag = _, attrtypes = ATSet)), val(ATSet, Set),
  member(AT, Set), id(ET, S), ID(AT, FT), dest : val(S, FTSet).
% create an empty scheme for each sub-element type
add : val(S, set()) ← c-inst(ST, subelem-t), id(ET, S).
% create two field types for sub element type scheme
add : val(FT1, struct(name = PN, dom = stringtype)),
  val(FT2, struct(name = CN, dom = stringtype)) ←
  c-inst(ST, subelem-t), val(ST, struct(parrent = P, child = C)),
  val(P, struct(tag = PN, attrtypes = _)), val(C, struct(tag = CN, attrtypes = _)),
  id(P, FT1), id(C, FT2).
% create a table element type
add : c-inst(R, tuples), val(R, bag()), val(T, struct(name = N, rows = R)) ←
  c-inst(ET, elem-t), val(ET, struct(tag = N, attrtypes = _)), id(ET, S), id(S, T), id(T, R).
% create a field for each attribute
add : val(F, struct(name = N, val = V)) ←
  c-inst(A, attr), val(A, struct(name = N, val = C)), id(A, F), id(C, V).
% create a field for each element as an id
add : val(F, struct(name = "id", val = V)) ←
  c-inst(E, elem), id(E, "id", F), toString(F, V).
...

```

Figure 10: Structural rules for a model-to-model transformation.

4.3 Higher-Level Transformations

While mapping rules are expressed using a declarative language, they specify the low-level details of transformations. Such an approach is attractive since it enables extremely flexible transformations (*e.g.*, model-to-model, schema-to-schema, and arbitrary combinations), however, specifying mapping rules is not trivial. In the remainder of this section, we describe two approaches for higher-level rule specification (each of which builds on the current mapping rule approach), which we believe can make writing rules simpler. We briefly describe *transformation patterns* for capturing and re-using common transformation conventions and discuss semi-automatic *rule derivation*, which is enabled through our generic framework.

We view a transformation pattern as a mapping rule abstraction, *i.e.*, a transformation pattern is a parameterized specification taking mapping rules as input to create a more complex transformation. In this way, patterns are analogous to higher-order functions like `map` and `fold` in functional programming languages. In general, a pattern can be used as a runtime operation where a set of arguments is provided to perform the concrete transformation, or, as a mapping rule generator, *i.e.*, given a set of arguments, the pattern is used to generate (one or more) concrete mapping rules.

For example, there are often many ways to transform between models allowing multiple

```

<ELEMENT professor (person)>
<ATTLIST professor dept CDATA #REQUIRED>
<ELEMENT person EMPTY>
<ATTLIST person name CDATA #REQUIRED>

```

```

<professor dept="CSE">
<person name="Maier"/>
</professor>

```

(c).

Professor Table		Person Table		Professor-Person Table	
<i>id</i>	<i>dept</i>	<i>id</i>	<i>name</i>	<i>professor</i>	<i>person</i>
e1	"CSE"	e2	"Maier"	e1	e2

Figure 11: Results from the XML to Relational transformation.

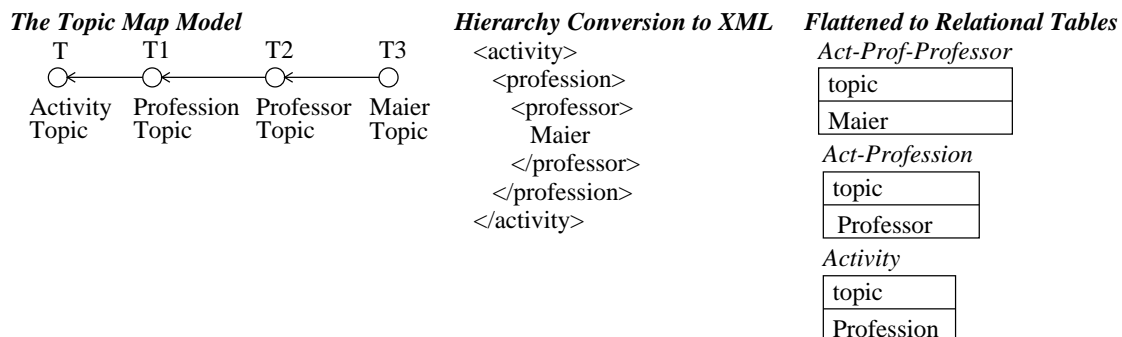


Figure 12: An example of flattening and hierarchy conversion of the Topic Map model.

levels of conformance to models permitting only a single level. One such convention is to flatten multiple levels of conformance, making the conformance implicit in the destination. A different convention is to represent source instance-of relationships as hierarchical relationships in the destination (again, conformance becomes implicit). Examples are shown in Figure 12 where the source is a Topic Map and the target a Relational database (for the flattening approach) and an XML document (for the hierarchy approach). The source has three levels of conformance: topic “Maier” is an instance of topic “Professor”; topic “Professor” is an instance of topic “Profession”; and topic “Profession” is an instance of topic “Activity.”² In the case of XML, the hierarchical relationship is modeled as nested elements, and for the Relational example, each table represents a path of the original conformance relationship.

The following rule shows an example transformation pattern called *hierarchy-convert* for generically applying the hierarchy conversion (the language shown for describing patterns is for exposition only; we are currently exploring appropriate languages):

```

pattern : hierarchy-convert(T, M1, M2, A1, A2) =
  % convert inter-nodes T and T1
  if (d-inst(T3, T2), d-inst(T1, T), c-inst(T, C), c-inst(T1, C), c-inst(T2, C)) then
    map (using rule M1) T → S and T1 → S1 and attach (using rule A1) S1 → S
  % convert last node
  if (d-inst(T3, T2), -d-inst(T4, T3), c-inst(T2, C), c-inst(T3, C), c-inst(T4, C)) then
    map (using M1) T2 → S2 and map (using M2) T3 → S3 and attach (using A2) S3 → S2.

```

The pattern takes five arguments: the construct instance T serving as the top item of the

²Notice that “Maier” is an instance of a “Professor,” but not an instance of a “Profession.”

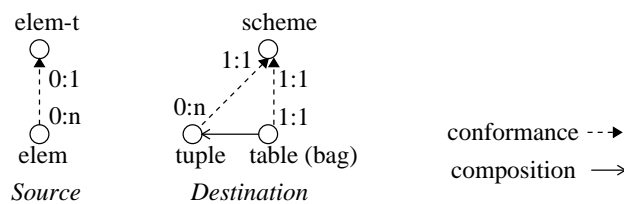


Figure 13: Source and destination constructs for rule derivation.

multi-level conformance relationship (e.g., “Activity” in Figure 12); a mapping rule M_1 to map source items to target items (e.g., topics to XML elements); a special mapping rule M_2 for converting items at the leaf level (e.g., instead of mapping topics to elements, we would map topics to XML element content); and mapping rules A_1 and A_2 for attaching the hierarchical relationships in the destination.

The *hierarchy-convert* pattern is broken into two sub-rules. The first rule identifies items above the leaf conformance items (e.g., the “Activity” and “Profession” topics) and converts them to target items (using M_1). The second rule is similar, but for leaf nodes. In our example, the second rule would convert the topic “Maier” into XML content (using M_2), and attach it to the “Professor” element (using A_2).

We see a number of potential patterns as useful, e.g., a simple class of patterns are those that convert between construct types. That is, patterns for converting lists to sets and bags, flattening nested sets, merging sets, lists, bags, and structs, and converting a group of structs into sets, and so on.

Patterns help make it easier to write rules by enabling the reuse of commonly used transformations. Another approach for simplifying rule writing is semi-automatic rule derivation. The goal is to generate the structural mapping rules (like those in Figure 10) whenever possible from the simpler construct level rules (such as those in Figure 9) by leveraging the composition constraints of constructs.

For example, consider the excerpt of XML and Relational models shown in Figure 13. The following mapping rule specifies a construct transformation between XML element types and Relational schemes:

$$c-inst(ET, scheme) \leftarrow c-inst(ET, elem-t).$$

Based on the description and constraints of the source model, we can conclude the following: (1) every *scheme* must have exactly one *table* (and vice versa), (2) every *table* is a bag of *tuple*’s, (3) every *tuple* must conform to its *table*’s conforming *scheme*, and (4) more than one *elem* can conform to an *elem-t*.

From the above mapping rule requesting each *elem-t* be converted to a *scheme* and the above constraints, it is reasonable to consider the following mappings.

1. Each *elem* should be mapped to a *tuple* since many *elem*’s can be associated with each *elem-t*, and it is only through *tuple*’s that multiple items can be associated with a given *table*.
2. A *table* should be created for each converted *elem-t* (i.e., each new *scheme*).
3. The corresponding *table* (for the new *scheme*) should contain the converted *tuple* items.

Note that to generate these rules, we can employ structural transformation patterns, *e.g.*, to convert the source *d-inst* relations into a bag *table* of the appropriate *tuples*, and so on. Although a simple example, the derived rules are equivalent to those used to map element types to schemes in the transformation of Section 4.2. Thus, combining patterns and constraint-based rule derivations can ease the task of rule specification, both of which are enabled by the uniform framework. We intend to investigate such higher-level transformation rules further.

5 Related Work

This chapter extends our earlier work [4, 5], which introduced the use of a uniform, RDF-based representation scheme allowing model, schema, and instance data to be represented using triples. The metamodel introduced here is significantly more complex and has a relatively complete set of structural primitives, including typical database modeling constructs: *struct* (similar to tuple) and explicit collection primitives *set*, *bag*, and *list*. More importantly, these structures can be freely composed as needed for the data model being described. We also define a clear separation of basic structures (*i.e.*, construct types) from instance-of and conformance relationships—this separation differs from RDF, which has only a single version.

The metamodel architecture we propose is the only approach we know of that explicitly models conformance. Existing approaches [6, 7, 8, 9, 10] focus on solving database interoperability issues and assume (and exploit) schema-first models. Typically, transformation is defined at the data model or schema level (but not both). The approach we propose enables mappings at various levels of abstraction, *e.g.*, model-to-model, model-to-schema, and combinations of these. In addition, we permit partial mappings, *i.e.*, we do not require complete (or one-to-one) transformations.

The use of logic-based languages for transformation has been successfully applied [9, 2], *e.g.*, WOL [2] is a schema transformation language that uses first-order logic rules for both schema transformations and specifying (schema) constraints.

Finally, a number of recent, informal model-to-model and model-to-schema mappings have been defined between XML and Relational models [11] and between Topic Maps and RDF [3, 12]. We believe having a formal transformation language can benefit both defining and implementing such mappings.

6 Summary

The framework we propose has the following advantages. It allows a data model designer to explicitly specify when and how one construct conforms to another. It can be used to describe models that require schema, don't require schema, permit multiple schemas, allow multiple levels of schema-instance connections, and include any or all of these possibilities. It introduces four explicit relationships, *ct-inst*, *conformance*, *c-inst*, and *d-inst*, which can be exploited for transformation and by other tools such as a query language or a browser. Finally, it provides orthogonal, complementary specification of conformance and instance-of relationships versus composition and constraints for describing data structures that exist in the information representation. We believe patterns and constraint-based transformation also offer powerful mechanisms for transformation and intend to further investigate their use.

Acknowledgement

This work was supported in part by the National Science Foundation through grants EIA 9983518 and IIS 9817492.

References

- [1] Marshall, C., Shipman, F., Coombs, J.: VIKI: Spatial hypertext supporting emergent structure. In: Proceedings of the European Conference on Hypertext Technology (ECHT'94), Edinburgh, Scotland, ACM (1994) 13–23
- [2] Davidson, S., Kosky, A.: WOL: A language for database transformations and constraints. In: Proceedings of 13th International Conference on Data Engineering (ICDE'97), Birmingham, U.K. (1997) 55–65
- [3] Moore, G.: RDF and Topic Maps: An exercise in convergence. In: Proceedings of the XML Europe Conference, Berlin, Germany (2001)
- [4] Bowers, S., Delcambre, L.: Representing and transforming model-based information. In: Proceedings of the First International Workshop on the Semantic Web (SemWeb'00), held in conjunction with the 4th European Conference on Digital Libraries, Lisbon, Portugal (2000) 1–14
- [5] Bowers, S., Delcambre, L.: A generic representation for exploiting model-based information. In: The Electronic Transactions on Artificial Intelligence (ETAI Journal). Volume 5. (2001, Section D) 1–34
- [6] Atzeni, P., Torlone, R.: 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., Avignon, France (1996) 79–95
- [7] Barsalou, T., Gangopadhyay, D.: M(DM): An open framework for interoperation of multimodel multi-database systems. In: Proceedings of the 8th International Conference on Data Engineering (ICDE'92). IEEE Computer Society Press, Tempe, Arizona, USA (1992) 218–227
- [8] Christophides, V., Cluet, S., Siméon, J.: On wrapping query languages and efficient XML integration. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, Texas, USA (2000) 141–152
- [9] McBrien, P., Poulouvassilis, A.: A uniform approach to inter-model transformations. In: Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE'99). Volume 1626 of Lecture Notes in Computer Science., Heidelberg, Germany (1999) 333–348
- [10] Papazoglou, M., Russell, N.: A semantic meta-modeling approach to schema transformation. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), Baltimore, Maryland, USA, ACM (1995) 113–121
- [11] Bohannon, P., Freire, J., Roy, P., Siméon, J.: From XML schema to relations: A cost-based approach to XML storage. In: Proceedings of the 18th International Conference on Data Engineering (ICDE'02). IEEE Computer Society Press, San Jose, California, USA (2002) 64–75
- [12] Lacker, M., Decker, S.: On the integration of Topic Maps and RDF. In: Proceedings of the 1st International Semantic Web Working Symposium (SWWS '01), California, USA (2001) 331–344