

# A Generic Representation for Exploiting Model-Based Information

Shawn Bowers and Lois Delcambre

OGI School of Science and Engineering  
Oregon Health and Science University  
Beaverton, Oregon

Linköping University Electronic Press  
Linköping, Sweden

<http://www.ep.liu.se/ea/cis/2001/???/>

*Published on July 25, 2001 by  
Linköping University Electronic Press  
581 83 Linköping, Sweden*

**Linköping Electronic Articles in  
Computer and Information Science**  
*ISSN 1401-9841  
Series editor: Erik Sandewall*

©2001 Shawn Bowers and Lois Delcambre  
*Typeset by the author using L<sup>A</sup>T<sub>E</sub>X  
Formatted using étendu style*

**Recommended citation:**

*<Author>. <Title>. Linköping Electronic Articles in  
Computer and Information Science, Vol. ?(2001): nr ?.  
<http://www.ep.liu.se/ea/cis/2001/???/>. July 25, 2001.*

*This URL will also contain a link to the author's home page.*

*The publishers will keep this article on-line on the Internet  
(or its possible replacement network in the future)  
for a period of 25 years from the date of publication,  
barring exceptional circumstances as described separately.*

*The on-line availability of the article implies  
a permanent permission for anyone to read the article on-line,  
to print out single copies of it, and to use it unchanged  
for any non-commercial research and educational purpose,  
including making copies for classroom use.*

*This permission can not be revoked by subsequent  
transfers of copyright. All other uses of the article are  
conditional on the consent of the copyright owner.*

*The publication of the article on the date stated above  
included also the production of a limited number of copies  
on paper, which were archived in Swedish university libraries  
like all other written works published in Sweden.  
The publisher has taken technical and administrative measures  
to assure that the on-line version of the article will be  
permanently accessible using the URL stated above,  
unchanged, and permanently equal to the archived printed copies  
at least until the expiration of the publication period.*

*For additional information about the Linköping University  
Electronic Press and its procedures for publication and for  
assurance of document integrity, please refer to  
its WWW home page: <http://www.ep.liu.se/>  
or by conventional mail to the address stated above.*

## Abstract

There are a variety of ways to represent information and each representation scheme typically has associated tools to manipulate it. In this paper, we present a single, generic representation that can accommodate a broad range of information representation schemes (i.e., structural models), such as XML, RDF, Topic Maps, and various database models.

We focus here on model-based information where the information representation scheme prescribes structural modeling constructs. For example, the XML model includes elements, attributes, and permits elements to be nested, RDF models information through resources and properties, and the relational model provides (among other constructs) tables.

Having a generic representation for a broad range of structural models provides an opportunity to build generic technology to manage and store information. Additionally, we can exploit well understood database query languages, e.g., non-recursive Datalog and SQL, to transform information from one scheme to another. In this paper, we present the generic representation and demonstrate the associated mapping mechanism to transform information and discuss some of the opportunities and challenges presented by this work.

## 1 Introduction

In this research, we recognize that many distinct, yet highly useful representation schemes have been proposed, such as XML [8], RDF [19] and Topic Maps [7], and each representation scheme has various associated tools. Rather than promote the use of a single representation for information (to the exclusion of others), we propose a method to explicitly represent the structural model of the representation scheme along with its corresponding data. Doing so provides a number of advantages, including easily converting information from one representation scheme to another when needed. Transforming between schemes enables useful tools to be exploited against existing information simply by converting it from its original form to the form required by the tool of interest.

We focus on information representation schemes that are model-based. For example, the XML model includes elements with optional attributes and a relational database model represents information in tables. Some models require the use of a schema to set the organization for data. For others such as RDF, Topic Maps, and XML, the use of schema is optional. For example, a particular RDF representation may conform to an RDF Schema [9], a Topic Map may conform to a Topic Map definition, and an XML document may be valid for a document type definition (DTD).

In this paper, we present a generic representation scheme for model-based information, called the *uni-level description*. The uni-level description allows model, schema, and instance information to be represented explicitly for model-based information. The uni-level description leverages a metamodel (the *uni-level description metamodel*), which is comprised of basic structures that can be instantiated (via the uni-level description) to describe various models or representation schemes of interest. Additionally, we show how RDF and RDF Schema can be used to express the uni-level description.

This work is part of our general investigation into what we call *superimposed information* [14, 15, 20]. We briefly present superimposed information in Section 2. Section 3 describes the uni-level description metamodel, which provides a formalism to define various data models. The uni-level description is described in Section 4. Section 5 shows how Datalog over the uni-level description can formally specify and implement mappings from one model-based representation to another. Sections 6 and 7 describe our current work with the uni-level description and opportunities for future work. We discuss related work in Section 8 and offer conclusions in Section 9.

## 2 Superimposed Information

Suppose you're planning a vacation using the Web and you are able to drag and drop selected information (from Web sites) into a scratchpad tool that allows you to group information and annotate it. The scratchpad keeps links to the original information, allowing you to navigate back to it whenever you need to (e.g., you may have selected a regular hotel rate, but wonder if there is a weekend special).

You may also wish to use additional tools over the information you collect. For example, to help you manage the cost of the trip, including exchange rates, lodging, and transportation expenses, you use a spreadsheet application to process the appropriate data from the scratch pad. In developing an itinerary, you select information from the scratchpad and place it into a calendar application, which also allows you to browse back over the

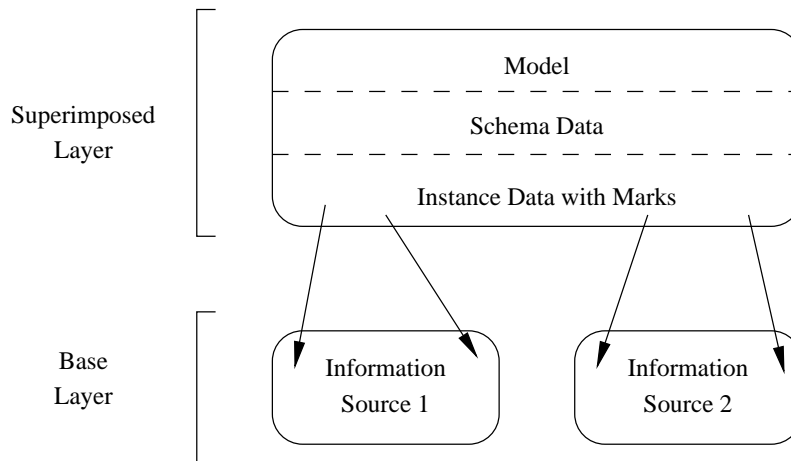


Figure 1: The superimposed and base layers.

original documents. You also decide to lift addresses and phone numbers relevant to the trip into your address book (e.g., to load into your PDA) and integrate the payment information you've collected into your checkbook software to develop a budget for your vacation.

This scenario leverages *superimposed information* to help manage and organize existing data. Superimposed information is an independent layer (the superimposed layer) of data used to provide additional links among selected elements from underlying information sources (called the base layer). The basic arrangement of the superimposed and base layers is shown in Figure 1. The base layer consists of information sources that are referenced by marks in the superimposed layer. Marks reference information at various granularities within a source, as desired by the user and as permitted by the addressing scheme. Information sources can be of many different types including HTML pages, XML documents, PDF files, Microsoft PowerPoint presentations, Excel spreadsheets, databases, and so forth.

In general, superimposed information has the following characteristics:

- It can contain marks that connect the superimposed layer to elements within the existing information sources.
- It can contain additional information about elements in the base layer (e.g., to organize, annotate, or highlight elements).
- It can contain new information, beyond what appears in the base layer.
- It can have varying degrees of structure and can employ various data models and schemas.
- It does not modify the base layer.

Our work on superimposed information has contributed a software architecture for building superimposed applications [15]. An overview of the architecture is shown in Figure 2. The Mark Manager is responsible for creating and resolving marks, where the marks can reference a broad range of information sources at various granularities. A superimposed application provides useful capabilities to the user that views and manipulates superimposed information. Our first superimposed application is called *SLIMP*ad,

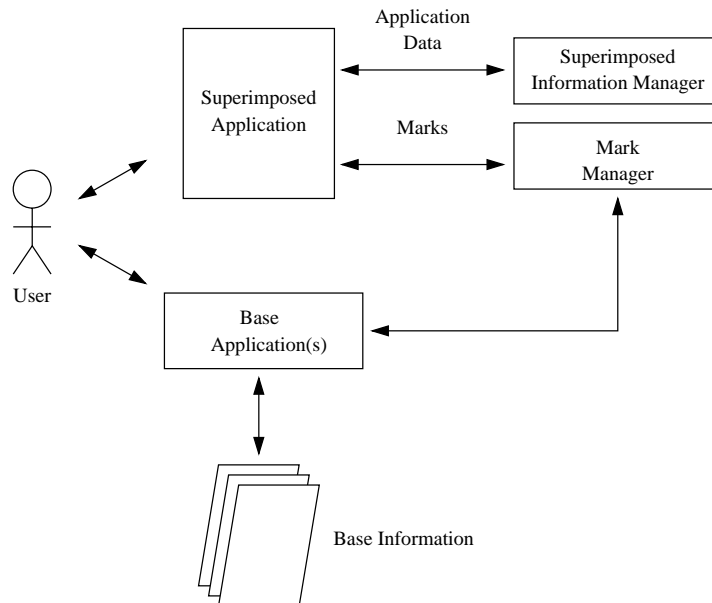


Figure 2: Overview of the superimposed application architecture.

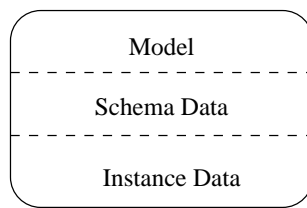


Figure 3: The three (potential) layers of information.

the Superimposed Layer Information Manager scratch Pad. The work described here was inspired by this architecture; we wanted to build a generic storage manager for superimposed information. The uni-level description presented here meets our original goal and has been implemented in our architecture (the Superimposed Information Manager in Figure 2). But, the uni-level description is broadly applicable. In particular, it enables the use of a database query language to provide a very powerful and flexible form of mapping from one representation scheme to another.

### 3 The Uni-Level Description Metamodel for Representing Model-Based Information

Information of interest in this research consists of three levels: model, schema, and instance data as shown in Figure 3. Instance data may not have a schema and it may or may not have a model, although we focus in this research on information representation schemes that exploit a model. It is also possible to describe a model, or a model and a schema, without having instance data present.

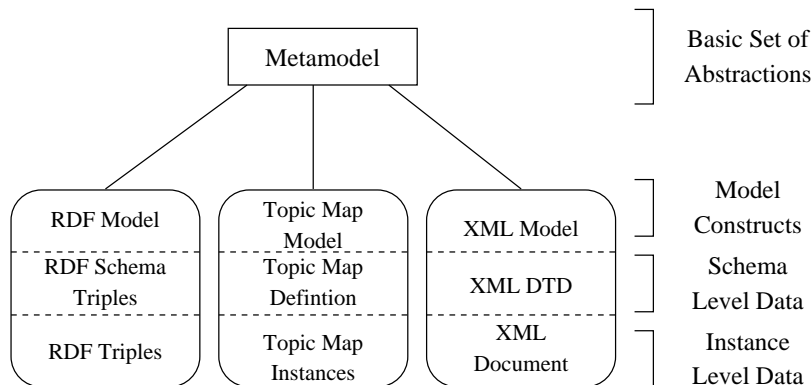


Figure 4: The RDF, Topic Map, and XML model, schema, and data.

To describe multiple models, we define a level of abstraction above the model, the metamodel, which is used to define the model of interest to a superimposed application. Figure 4 shows the role of the metamodel for three information representations: RDF, Topic Maps, and XML.

The metamodel describes the basic structures used to define model constructs and their relationships. A model consists of schema and instance constructs that are used to define data. Additionally, the model describes the conformance relationships between instance-level data and schema-level data. Each level of the architecture can loosely be viewed as an instantiation of the level above it. Specifically, the model constructs are particular instantiations of the structures defined by the metamodel. The schema-level data are particular instantiations of the model’s schema constructs. And finally, the instance-level data are instantiations of the model’s instance constructs and can conform to the schema-level data (as opposed to being instances of the schema constructs themselves).

To illustrate the three levels, Figure 5 shows an informal example of model, schema, and instance data for XML. Notice we use an “open” DTD, which means that elements and attributes not defined in the DTD can be included in XML documents.

The definition of our metamodel is similar to other metamodel approaches in the object [18, 22] and database [1, 2, 3, 4, 10, 12, 13, 21] communities for describing structural models such as the entity-relationship model, the relational model, the hierarchical model, and the various semantic data models including the UML. However, the metamodel proposed here has a number of unique features. We relax the requirement of schema-first definitions, which require schema to be created prior to instances as required in most database systems. The metamodel also allows for data that is not explicitly typed. For example, a topic can exist in a Topic Map without being associated to a type and the metamodel can accommodate this directly.

The metamodel also allows multiple levels of schema-instance relationships. In a Topic Map, topics can have a type that is itself a topic, and so it too can have a type, resulting in two levels of schema and instance definition.

Figure 6 shows the basic structures of the metamodel. The *construct*, which defines a basic structure within a model, and the *structural connector*, which defines a relationship between constructs, are the essential primitives.

As shown, a mark and a lexical are defined as two special kinds of

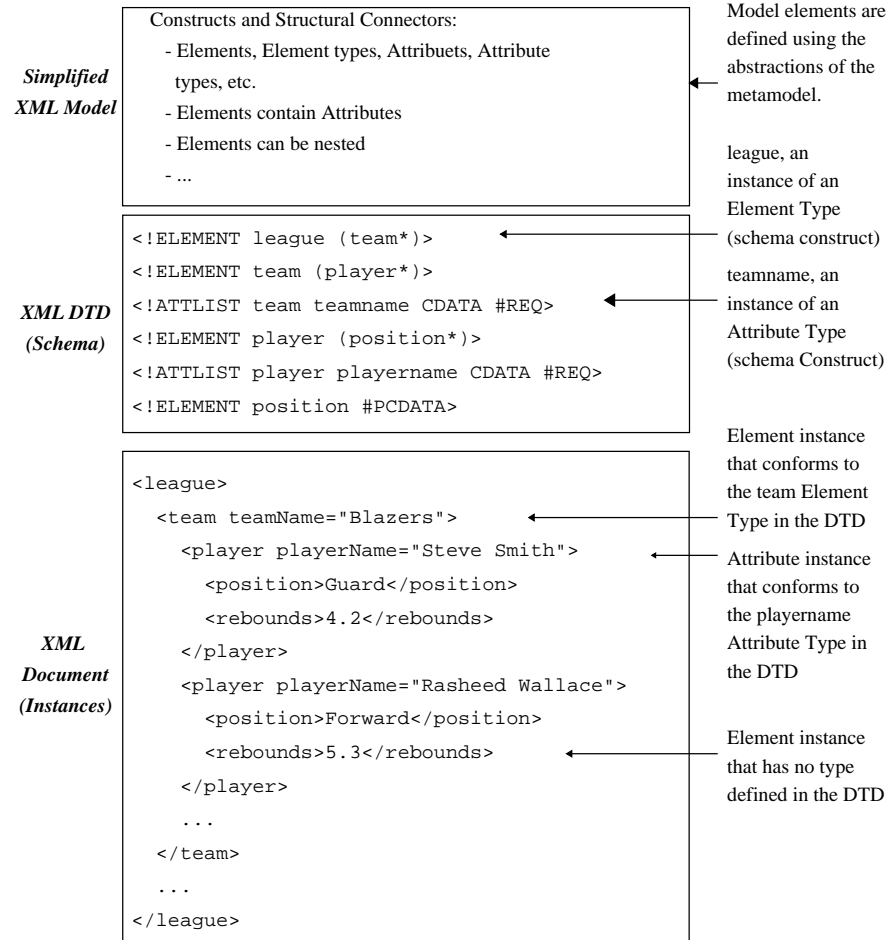


Figure 5: An example of each of the 3 levels (model, schema, and instance) for XML.

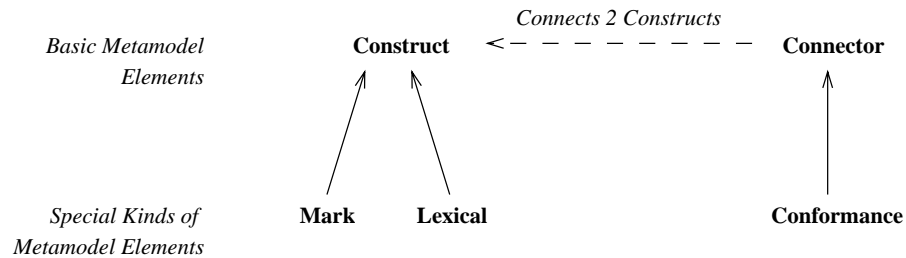


Figure 6: The elements defined by the metamodel.

<i>Metamodel Items</i>	<i>XML Model Items</i>
<b>Lexicals</b>	Primitive Content (PC)
<b>Constructs</b>	Element Type (ET) Attribute Type (AT) Element (E) Attribute (A) Primitive Content Type (PCT) Primitive Content (PC)
<b>Connectors</b>	Nested Element Type: <i>Connects 2 ETs</i> Nested Element: <i>Connects 2 Es</i> Element Content: <i>Connects Es to PCs</i> Element Content Type: <i>Connects ETs to PCTs</i> Element Attribute: <i>Connects Es to As</i> Attribute Element Type: <i>Connects ETs to ATs</i>
<b>Conformance</b>	Element Instance Of: <i>Connects Es to ETs</i> Attribute Instance Of: <i>Connects As to ATs</i> Content Instance Of: <i>Connects PCs to PCTs</i>

Table 1: Description of the XML model where items (right) are instances of the corresponding metamodel items (left).

constructs. A *mark* describes a model construct whose instances represent connection-points to other information sources. The metamodel allows marks to appear at various places within a superimposed model [14]. A *lexical* describes a model construct whose instances contain primitive-value types (like strings, integers, floats, and so on). There is also a special structural connector called a *conformance* connector, which specifies a schema-instance relationship between constructs.

Table 1 shows an example of the XML model defined in terms of our metamodel. The XML model has been simplified to consist of Element Types, Elements, Attribute Types, Attributes, Primitive Content Type (e.g., PCDATA), and Primitive Content along with a minimal set of relationships between them.

Element constructs in XML form a hierarchy, which is represented by the model connector Nested Element. Multiplicity constraints can be applied to connectors for further specification. For the Nested Element connector, we use multiplicity constraints to specify that an Element is either not nested or nested within one parent Element, and can have many Elements nested within it.

In Table 1, conformance connectors are used to specify schema-instance relationships between Attribute and Attribute Type, Element and Element Type, and Primitive Content and Primitive Content Type. We can also apply multiplicity constraints to conformance connectors. For example, by assigning the appropriate multiplicity constraints, we can specify whether an instance construct must be created prior to a schema construct.

The metamodel does not restrict model constructs to be at the instance- or schema-level. This allows models to have multiple levels of schema and instance definition. For example, in the Topic Map model we would define a Topic construct that has a conformance connector to itself.

Finally, in Table 1 we allow an Element Type construct to contain a Primitive Content Type construct. We define Primitive Content Type as a lexical construct, which means instances of the Primitive Content Type can

be primitive types such as string, integer, or more specialized types such as PCDATA for XML. Elements that conform to the Element Type must then contain Primitive Content with the type specified by the Primitive Content Type.

## 4 Representing Models, Schemas, and Instances

Models defined using the metamodel are stored using a representation scheme based on RDF. Although model engineers can specify models directly using the RDF representation, we believe it is more convenient to define models visually. Therefore, we also provide a visual representation of models using a subset of the UML.

### 4.1 The Resource Description Framework

RDF is a graph-based model for attaching metadata to information sources on the web (and can be itself considered a superimposed information model). It consists of a set of statements that are represented as triples. A triple denotes an edge between two nodes and has a property name (the edge label), a resource (a node), and a value (a node). A value can be either a resource or a literal. Resources can represent anything from web pages to abstract concepts. A literal is of a primitive type such as an integer or string. For example, the RDF triple (creator, "index.html", "Ora Lassila") can be read as "the creator of index.html is Ora Lassila" where "creator" is a property name, "index.html" a resource, and "Ora Lassila" a string literal [19].

RDF Schema is a type system for RDF. It provides a mechanism to define classes of resources and property types, which restrict the domain and range of a property. The resource *Class* is used to type resources and the resource *Property* is used to type properties. Each Property has a *domain* and *range* constraint. In addition, RDF Schema defines the property *subClassOf* to represent a subset-superset relationship between classes, *subPropertyOf* for a specialization relationship between properties, and *type* to specify resource creation. The RDF and RDF Schema specifications use XML as an interchange format to exchange RDF and RDF Schema triples.

### 4.2 The Metamodel Represented in RDF Schema

Figure 7 shows the definition of the metamodel using both the RDF XML syntax and RDF triples (for readability, the namespaces *rdf* and *rdfs* are not included). We represent *construct*, *mark*, and *lexical* as RDF classes, where *mark* and *lexical* are sub-classes of *construct*. Similarly, we represent *connector* and *conformance* as properties, each with a *construct* as domain. *Conformance* is a sub-property of *connector*.

Additionally, *domainMult* and *rangeMult*, which are used to specify the multiplicity of a connector end, are defined as RDF *constraint properties*. Constraint property is a higher-order property that can be used to add constraints to properties (domain and range are considered default constraint properties in RDF Schema). We define a multiplicity as one of *optional*, *required*, *multiple*, or *any* represented as 0..1, 1..1, 1..n, or 0..n, respectively. The *instanceOf* property is used to create model constructs and connectors (as instances of the metamodel constructs) as well as schema and instance

*The Metamodel represented in RDF XML*

```

<RDF>
  <Class ID="Construct"/>
  <Class ID="Mark">
    <subClassOf resource="#Construct"/>
  </Class>
  <Class ID="Lexical">
    <subClassOf resource="#Construct"/>
  </Class>
  <Property ID="Connector">
    <domain resource="#Construct"/>
  </Property>
  <Property ID="Conformance">
    <subPropertyOf resource="#Connector"/>
  </Property>
  <ConstraintProperty ID="domainMult">
    <domain resource="#Connector"/>
    <range resource="#String"/>
  </ConstraintProperty>
  <ConstraintProperty ID="rangeMult">
    <domain resource="#Connector"/>
    <range resource="#String"/>
  </ConstraintProperty>
  <Property ID="instanceOf"/>
  <Lexical ID="String"/>
  <Lexical ID="Integer"/>
</RDF>

```

*The Metamodel represented in RDF Triples*

```

(type, "Construct", Class)
(type, "Mark", Class)
(subClassOf, "Mark", Construct)
(type, "Lexical", Class)
(subClassOf, "Lexical", Construct)
(type, "Connector", Property)
(domain, "Connector", Construct)
(type, "Conformance", Property)
(subPropertyOf, "Conformance", Connector)
(type, "domainMult", ConstraintProperty)
(domain, "domainMult", Connector)
(range, "domainMult", String)
(type, "rangeMult", ConstraintProperty)
(domain, "rangeMult", Connector)
(range, "rangeMult", String)
(type, "instanceOf", Property)
(instanceOf, "String", Lexical)
(instanceOf, "Integer", Lexical)

```

Figure 7: The Metamodel represented in RDF XML and RDF triples.

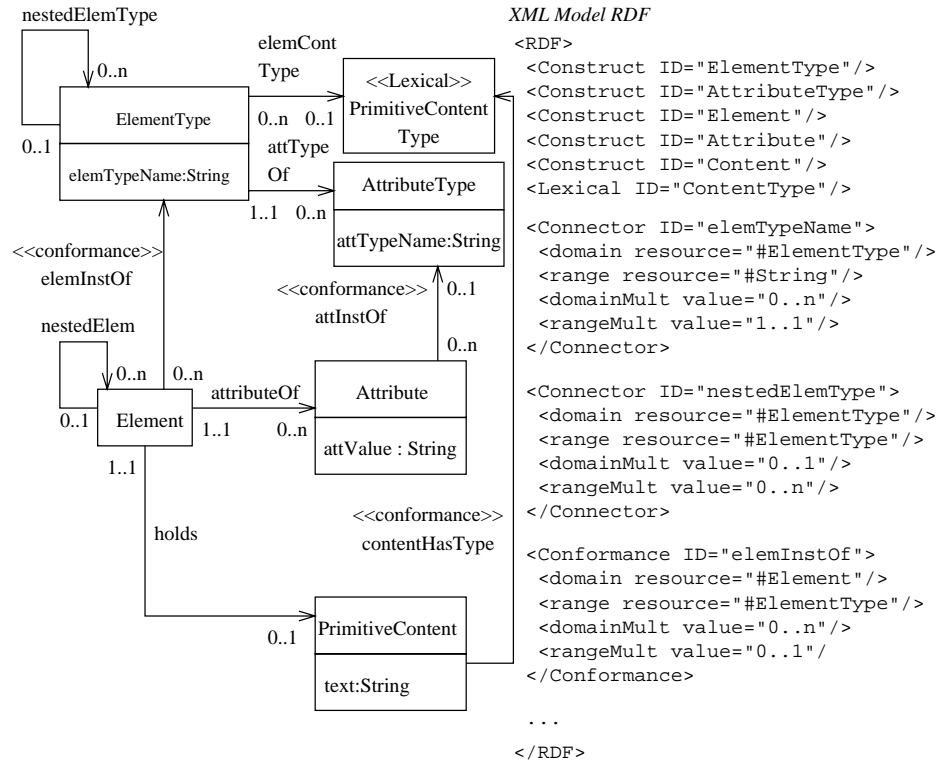


Figure 8: The XML model represented using UML, with a sample of the RDF representation.

data (as instances of the model constructs). Notice that the domain and range of `instanceOf` are not defined, which means that `instanceOf` can be used over any resource and contain any value. Finally, in Figure 7 we create two lexical constructs: string and integer. If desired, a model engineer can specify new primitive types (e.g., PCDATA for XML) using similar definitions.

### 4.3 The RDF and Visual Model Representation

Using the metamodel elements, Figure 8 depicts both the RDF and visual representation of the simplified XML model of Section 3. In the visual description, UML classes are mapped to constructs; relationships and class attributes are mapped to connectors. Attributes must have a lexical construct as a range (e.g., string or integer) and implicitly have a domain multiplicity of optional and a range multiplicity of required. UML stereotypes are used to distinguish marks and lexicals from constructs, and conformance connectors from regular connectors.

The schema-level constructs of Figure 8 are Element Type, Attribute Type, and Primitive Content Type. The instance-level constructs are Element, Attribute, and Primitive Content. The conformance connector between Element and Element Type, Attribute and Attribute Type, and Primitive Content and Primitive Content Type specify the schema-instance relationships.

Figure 9 shows example schema- and instance-level data that represent

*Excerpt of schema- and instance-level data (as RDF XML)*

```

<RDF>
  ...
  <ElementType ID="player_type">
    <elemTypeName value="player"/>
    <nestedElemType resource="#position_type"/>
    <attTypeOf resource="#playerName_attr"/>
  </ElementType>
  <ElementType ID="position_type">
    <elemTypeName value="position"/>
  </ElementType>
  <AttributeType ID="playerName_attr">
    <attName value="playerName"/>
  </AttributeType>
  <Element ID="player1">
    <elemInstOf resource="#player_type"/>
    <attributeOf resource="#playerName1"/>
    <nestedElem resource="#position1"/>
    <nestedElem resource="#rebounds1"/>
  </Element>
  <Attribute ID="playerName1">
    <attInstOf resource="#playerName_attr"/>
    <attValue value="Steve Smith"/>
  </Attribute>
  <Element ID="position1">
    <elemInstOf resource="#position_type"/>
  </Element>
  <Element ID="rebounds"/>
</RDF>

```

*Excerpt of schema- and instance-level data (as RDF Triples)*

```

...
(instanceOf, "player_type", ElementType)
(elemTypeName, "player_type", "player")
(nestedElemType, "player_type", position_type)
(attTypeOf, "player_type", playerName_attr)
(instanceOf, "position_type", ElementType)
(elemTypeName, "position_type", "position")
(instanceOf, "playerName_attr", AttributeType)
(instanceOf, "player1", Element)
(elemInstOf, "player1", player_type)
(attributeOf, "player1", playerName1)
(nestedElem, "player1", position1)
(nestedElem, "player1", rebounds1)
(instanceOf, "playerName1", Attribute)
(attInstOf, "playerName1", playerName_attr)
(attValue, "playerName1", "Steve Smith")
(instanceOf, "position1", Element)
(elemInstOf, "position1", position_type)
(instanceOf, "rebounds1", Element)

```

Figure 9: Excerpt of the XML schema- and instance-level data represented in RDF XML and RDF triples.

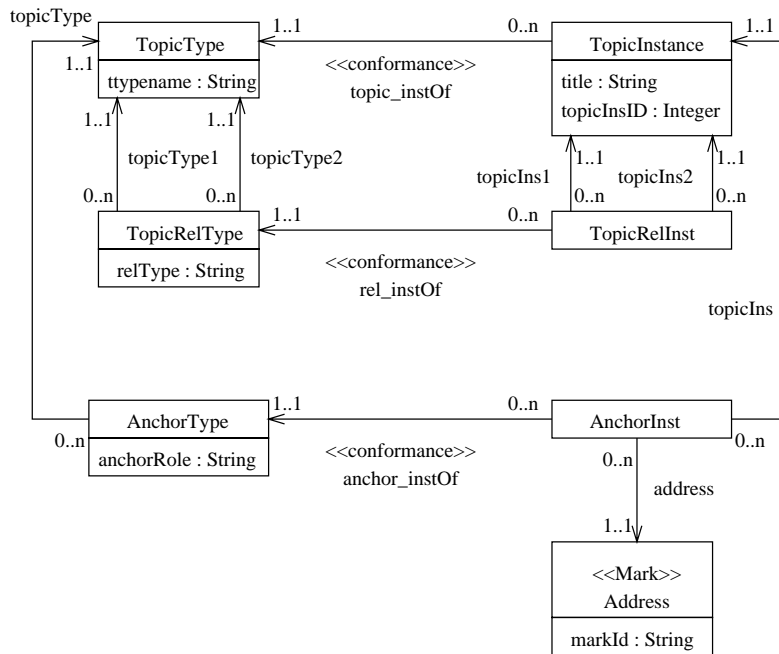


Figure 10: CARTE’s Structured-Map data model.

the XML document excerpt of Figure 5. Notice that our approach provides a uniform (or flat) representation of model, schema, and instance by allowing the model, schema, and instance data to be described using only RDF triples.

As another example, Figures 10 and 11 describe the Structured-Map and Bundle-Scrap models using the UML visual representation. The Structured-Map model is a simplified version of the Topic Map model, which uses a single level of schema and instance definition to allow Structured-Map data to be easily stored in a relational database. The model is designed for CARTE [16], which is a program that dynamically creates Web pages to navigate Structured-Map data. In CARTE, marks are represented as URLs. The user navigates through Topic Types, Topic Instances, and Topic Relations to reach Anchors, which contain marks. When a mark is selected, the referenced URL is displayed in a new Web browser window.

TopicType, TopicRelType, and AnchorType represent the schema constructs of the model. TopicInstance, TopicRelInst, AnchorInst, and Address represent the instance constructs of the model. CARTE requires that a schema be defined before instance-level data can be created. For example, a TopicType (perhaps named “painter”) must exist prior to creating a conforming TopicInstance (e.g., with the name “Van Gogh”). We express this requirement through the use of range multiplicity constraints on each conformance relationship.

SLIMPad uses the Bundle-Scrap model and is designed to be a super-imposed scratchpad application [15]. Users interact with SLIMPad by selecting content from any of a number of information sources including XML documents, Microsoft PowerPoint slides, Excel spreadsheets, and PDF files. Once selected, the content can be dragged into the scratch pad. Information selections within SLIMPad are represented as scraps, each of which contain

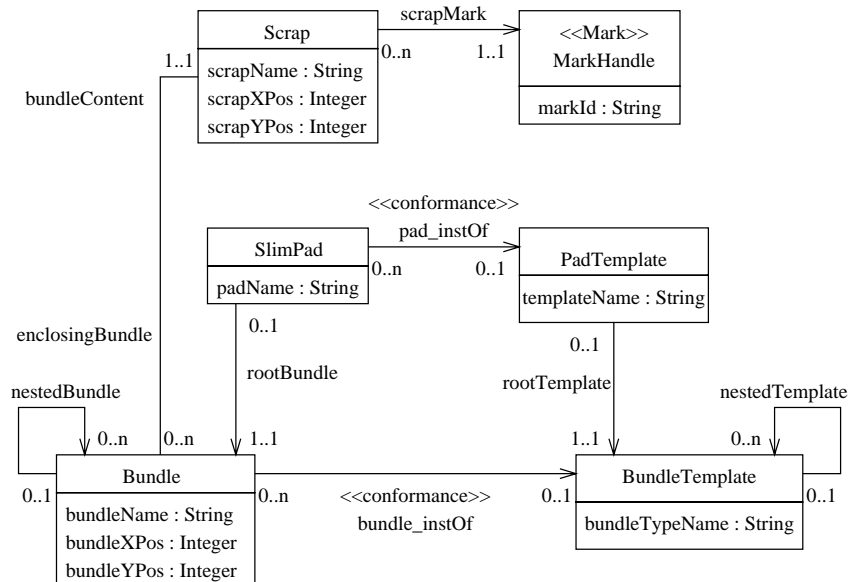


Figure 11: SLIMPad’s Bundle-Scrap model.

a mark that serves as a reference back to the content’s information source. Scraps can be organized into bundles and bundles can be nested. By selecting a scrap, the content referenced by the corresponding mark (inside the scrap) is displayed and highlighted at the information source.

As shown in Figure 11, the structure of the Bundle-Scrap model is quite simple. The only information contained in a scrap is its name (to be displayed on the SLIMPad screen), its x-y location (on the screen), and its referenced mark. A bundle is a named collection represented as a rectangle on the screen, with an x-y location. Users may optionally introduce and nest bundles as desired. There is one bundle, the `rootBundle`, to contain the content of one SLIMPad, by default.

Also shown in Figure 11 are schema-level constructs (`BundleTemplate` and `PadTemplate`), which we are incorporating into SLIMPad. The schema-level constructs allow users to create and use templates to construct Bundles. By instantiating a template, the user is given a set of default bundles organized hierarchically within the scratch pad (somewhat similar to Microsoft PowerPoint Templates). Additionally, we may add templates for scraps (not shown in the Figure 11). By using the multiplicity constraint *optional*, bundles can be created without an associated template (similar to an open DTD for XML).

SLIMPad was originally developed to help medical experts in decision-critical tasks [17]. Within the medical domain, we see a number of different templates being used, including templates to help manage important patient statistics in an intensive care unit and for developing hypothesis about individual patient problems and treatments. Templates typically have associated graphical user interface components for entering data (some of which are unstructured while others are strict in the types of data that can be entered, much like a traditional form).

In the metamodel, every connector is uni-directional (i.e., directed). However, in Figure 11 arrows are not included in the association between

a Scrap and a Bundle. This signifies that there are two uni-directional connectors, namely bundle content and enclosing bundle, between a Scrap and a Bundle. The bundle content and enclosing bundle connectors are treated as inverse connectors, in the underlying, generic representation.

## 5 Transforming Model Based Information Through Mappings

In this section, we describe our approach to transform structural information from one representation to another. The approach allows information to be transformed and delivered to tools that use different models and schemas. In general, there will be various ways to map between information representations, depending on the user’s desires and the applications of interest. Therefore, instead of trying to automatically generate mappings, we provide a technique for mappings to be specified by the mapping architect.

Figure 12 shows the conceptual architecture in which mapping rules are applied to transform between representation schemes. The information to be transformed (the source) is extracted from its native format, which might be within an application accessible through a programming interface or located externally (e.g., an XML file). The extraction step converts the source information into triples. Included in the triples is a description of the data model used by the application. The transformation is specified as a set of Datalog rules with the collection of source triples as input. The mapping process computes the fixed point of the Datalog program. In the normal case, only target triples are created by mapping rules<sup>1</sup>. The remainder of this section describes the mapping rules that can be used to transform triples.

### 5.1 Mapping Rules

Mappings are specified using production rules defined over the triples used to express the uni-level description. We do not require mappings to be complete since only part of a model, schema, or instance may be needed while using a specific tool.

Table 2 contains the basic definitions used to specify mappings. Triples are represented with the predicate  $\tau$ . Terms surrounded by single-quotes denote constants and terms starting with upper-case letters denote variables. For example, the formula  $\tau(\text{‘creator’}, X, Y)$  is used in a mapping rule to match all triples that are related through the property “creator” (since  $X$  and  $Y$  are variables). Additionally, we require each formula of predicate  $\tau$  to be contained in a collection of triples, which is denoted using a subscript on  $\tau$ . For example, the ground formula  $\tau_{xml}(\text{‘instanceOf’}, \text{‘Element’}, \text{‘Construct’})$  states that the triple  $\tau(\text{‘instanceOf’}, \text{‘Element’}, \text{‘Construct’})$  is in the triple collection  $xml$ . Note that we can represent the subscripted version of  $\tau$  using a standard, first order predicate by adding the triple collection name as an argument, e.g.,  $\tau'(\text{‘xml’}, \text{‘instanceOf’}, \text{‘Element’}, \text{‘Construct’})$ . For convenience, We use the subscripted form when we present example mappings.

We define a mapping rule as a production in which the left- and right-hand sides consist of a conjunction of atoms of  $\tau$ . Although we use non-

---

<sup>1</sup>One could imagine using Datalog rules against the source collection of triples for other purposes than mapping. This might result in creating new *source* triples.

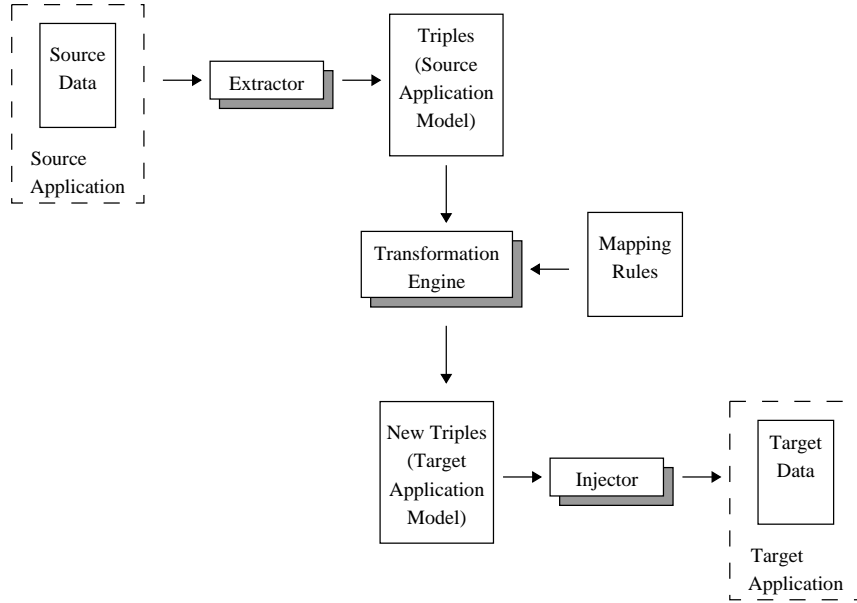


Figure 12: The conceptual architecture in which mappings are performed.

traditional syntax, the mapping rule shown in Table 2 is short-hand for the following set of Datalog rules:

$$\begin{aligned} \tau_t(d_1, e_1, f_1) &:- \tau_{l_1}(a_1, b_1, c_1), \dots, \tau_{l_n}(a_n, b_n, c_n) \\ &\vdots \\ \tau_t(d_m, e_m, f_m) &:- \tau_{l_1}(a_1, b_1, c_1), \dots, \tau_{l_n}(a_n, b_n, c_n), \end{aligned}$$

or, more precisely:

$$\begin{aligned} \tau'(t, d_1, e_1, f_1) &:- \tau'(l_1, a_1, b_1, c_1), \dots, \tau'(l_n, a_n, b_n, c_n) \\ &\vdots \\ \tau'(t, d_m, e_m, f_m) &:- \tau'(l_1, a_1, b_1, c_1), \dots, \tau'(l_n, a_n, b_n, c_n). \end{aligned}$$

We prefer the syntax in Table 2 because it emphasizes the fact that the presence of certain triples in the source or target collection (i.e., the set of all  $\tau_{l_i}(a_i, b_i, c_i)$  for  $1 \leq i \leq n$ ) cause triples (i.e., the set of triples  $\tau_t(d_j, e_j, f_j)$  for  $i \leq j \leq m$ ) to be created in the target collection.

As an example of a mapping rule, consider the rule  $\tau_{source}(\text{'creator'}, X, Y) \Rightarrow \tau_{target}(\text{'owner'}, X, Y)$ . This rule adds a triple  $\tau_{target}(\text{'owner'}, X, Y)$  to the target collection for every triple that matched  $\tau_{source}(\text{'creator'}, X, Y)$  in the collection *source*. Therefore, if  $\tau_{source}(\text{'creator'}, \text{'index.html'}, \text{'Ora Lassila'})$  is a triple in the source, the triple  $\tau_{target}(\text{'owner'}, \text{'index.html'}, \text{'Ora Lassila'})$  will be added to the target collection.

Table 3 describes several functions including the conversion function, which is used to perform mappings. The conversion function applies a set of mapping rules to the source collection and the emerging target collection of triples. The conversion function computes the least fixed point of the set of mapping rules and source triples. We have implemented the conversion

Symbol	Definition
$\tau_c$	A predicate that represents a triple in a collection $c$ , for example, $\tau_{source}(\text{'instanceOf'}, \text{'Element'}, \text{'Construct'})$ states that the triple $\tau(\text{'instanceOf'}, \text{'Element'}, \text{'Construct'})$ is in the triple collection named <i>source</i> . A triple collection is simply a named collection of triples.
$M$	A mapping that consists of a set of mapping rules.
$m$	A mapping rule (for the simple case of a single source, $s$ , and single target, $t$ ) with the form: $\tau_{l_1}(a_1, b_1, c_1), \dots, \tau_{l_n}(a_n, b_n, c_n) \Rightarrow$ $\tau_t(d_1, e_1, f_1), \dots, \tau_t(d_m, e_m, f_m)$ <p>where for <math>1 \leq i \leq n</math>, <math>l_i \in \{s, t\}</math>, <math>a_i</math>, <math>b_i</math>, and <math>c_i</math> are either constants or variables, and <math>n \geq 0</math>; for <math>1 \leq j \leq m</math>, <math>d_j</math>, <math>e_j</math>, and <math>f_j</math> are either constants or variables and <math>m \geq 1</math>; and for all variables <math>v \in \{d_1, \dots, d_m, e_1, \dots, e_m, f_1, \dots, f_m\}</math>, <math>v</math> must appear in the left side of this rule.</p>

Table 2: Predicate and mapping rule definitions.

function using Prolog, by using Prolog’s built in predicate *findAll* to perform the fixed point of the mapping rules. For each mapping rule, we place the triples in the head into another predicate (named  $S$ ) since Prolog allows only a single formula in the head of a rule. The ground  $S$  formulas returned by *findall* are then “unwrapped” and stored as the resulting triples (i.e., the target collection).

We have also implemented the conversion function using embedded SQL queries to express the body of the rules and multiple `INSERT` statements to express the atoms present in the head of the rule. The mappings that we have used so far in our experiments can be fully expressed in non-recursive datalog without negation, although, it is possible that someone may wish to define mapping rules using a more powerful rule language such as Datalog with recursion. Since we have not yet needed mapping rules with recursion, and since SQL finds all triple combinations that satisfy the body of the rules, for our purposes we are able to fire each rule just once in the SQL implementation.

Also shown in Table 3 are the functions *extract-model* and *extract-schema*. The *extract-model* function can be used to extract model information from a set of triples (i.e., the constructs, connectors, constraints, and so forth). Similarly, the *extract-schema* function gathers schema information from a set of triples. *Extract-schema* returns the construct instances that are at the schema-ends of conformance connectors along with the connections between the schema construct instances. (Note that in the uni-level description, a particular construct instance is at the schema level if another construct instance has a conformance connector to it.) Finally, Table 3 defines the *guid* skolem function, which is used to generate unique identifiers. The use of a skolem function, to generate a unique id to be placed in multiple triples in the target collection as shown in the second rule of Figure 20, is our only departure from pure Datalog. Our use of *guid* is similar to the use

Function	Definition
Conversion: $M \times s \rightarrow t$	Conversion takes a mapping $M$ and applies the mapping rules of $M$ to a triple collection $s$ (the source) and returns a new collection $t$ . Conversion implements a rule-based algorithm that computes the fixed-point of applying mapping rules to the source and target collections.
Extract model: $c \rightarrow c'$	$c$ and $c'$ are triple collections such that $c' \subseteq c$ and $c' = c_1 \cup c_2$ where: $c_1 = \{t \mid \exists x \exists y (t = \tau_c(\text{'instanceOf'}, x, y) \wedge y \in \{\text{'Construct'}, \text{'Lexical'}, \text{'Connector'}, \text{'Conformance'}\})\}$ $c_2 = \{t \mid \exists p \exists u \exists v (u = \tau_{c_1}(\text{'instanceOf'}, x, y) \wedge t = \tau_c(p, x, v) \wedge y \in \{\text{'Connector'}, \text{'Conformance'}\} \wedge p \in \{\text{'domain'}, \text{'range'}, \text{'domainMult'}, \text{'rangeMult'}, \text{'conformsTo'}\})\}$
Extract schema: $c \rightarrow c'$	$c$ and $c'$ are triple collections such that $c' \subseteq c$ and $c' = c_1 \cup c_2$ where: $c_1 = \{t \mid \exists u \exists v \exists x (u = \tau_c(\text{'instanceOf'}, p, \text{'Conformance'}) \wedge v = \tau_c(p, y, z) \wedge t = \tau_c(\text{'instanceOf'}, z, x))\}$ $c_2 = \{t \mid \exists p \exists u \exists v (u = \tau_{c_1}(\text{'instanceOf'}, r, x) \wedge v = \tau_{c_1}(\text{'instanceOf'}, s, y) \wedge t = \tau_c(p, r, s))\}$
$guid \rightarrow x$	A 0-ary Skolem function that returns a unique identifier $x$ .

Table 3: Functions used with mappings.

of a skolem function to generate unique object identifiers in object-oriented query languages, for example.

## 5.2 Inter-Model, Inter-Schema, and Model-to-Schema Mappings

Figure 13 illustrates three types of mappings. Each example shows information from a source collection of triples being mapped to a target collection to convert data from the source into data that conforms to the target. Although we focus on conversion, it is also possible to perform integration between collections, where integration goes a step further by combining the source and target data. The mapping rules can be used to provide mapping or integration at both the schema and instance levels.

Figure 13(a) is an inter-model mapping in which the schema- and instance-level data of the source are converted to valid schema- and instance-level data of the target. Figure 13(b) is an inter-schema mapping, which is similar to an inter-model mapping except the source schema is being

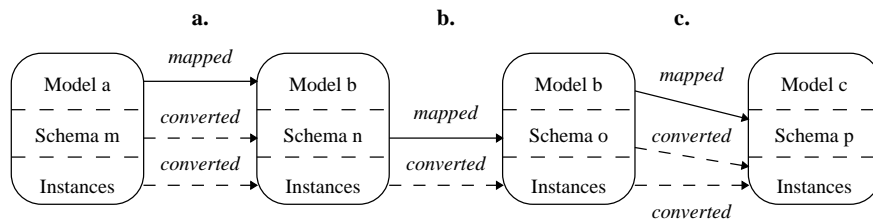


Figure 13: Three mappings: (a) inter-model, (b) inter-schema, and (c) model-to-schema

mapped, not the model. Figure 13(c) is one example of a mapping between levels. In this case, it is a model-to-schema mapping where the model of the source is mapped to schema in the target. Both the schema and instances of the source are converted to *instances* in the target that conform to the mapped target schema.

Figure 14 shows an inter-model mapping between the Bundle-Scrap model and the Structured-Map model. The goal of the mapping is to allow SLIMPad data to be used with CARTE.

Figure 15 illustrates four mapping rules between the Bundle-Scrap model (shown as the source) and the Structured-Map model (shown as the target) using the UML model representation. The first mapping rule, Figure 15(a), specifies a mapping between the Bundle Template schema construct and the Topic Type schema construct. That is, all Bundle Templates in the source will be converted to Topic Types in the target. Figure 15(b) is a mapping between Bundles and Topic Instances, which are both at the instance-level. Figure 15(c) is a mapping between two conformance connectors: `bundle_instOf` and `topic_instOf`. The mapping states that each `bundle_instOf` relationship in the source should be converted to a `topic_instOf` relationship in the target. Finally, the mapping in Figure 15(d) shows the `nestedTemplate` connector being mapped to a Topic Relationship Type. As the schema-level data is converted, a new Topic Relationship Type instance will be created with its `relType` attribute set to the constant value “`nested_template`.” Additionally, the instance from which the `nestedTemplate` connector originates (a Bundle Template instance) is converted to a Topic Type instance and assigned to the end of the corresponding `topicType1` connector. Similarly, the instance at the end (the range side) of the `NestedTemplate` connector is converted to a Topic Type instance and assigned to the end of the corresponding `topicType2` connector.

Figure 16 shows mapping rules that correspond to the mappings of Figure 15. We use the constant ‘source’ to represent the Bundle-Scrap triple collection and the constant ‘target’ to represent the new collection of triples created from the mappings.

An inter-schema mapping is shown in Figure 13(b). In an inter-schema mapping the source and target models are the same, but two distinct schemas are mapped so that the source instance-level data can be converted to data that conforms to the target schema. Figure 17 illustrates an inter-schema mapping using the XML model. The source is an animal taxonomy DTD containing Element Types such as *genus* and *species*. The target is a bookmark list DTD with Element Types such as *folder* and *bookmark*. One reason to perform this type of mapping might be to reuse existing tools for browsing bookmark lists on taxonomies.

Figure 18 demonstrates three rules that may be used to perform part

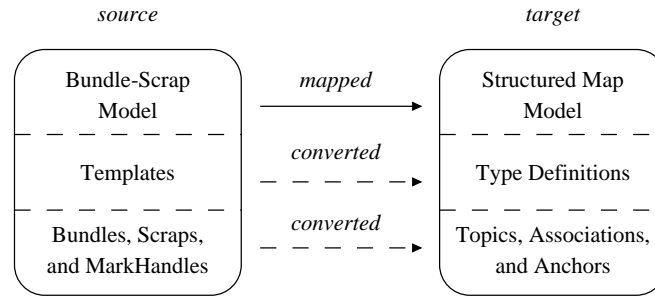


Figure 14: An inter-model mapping between the Bundle-Scrap model and Structured-Map model.

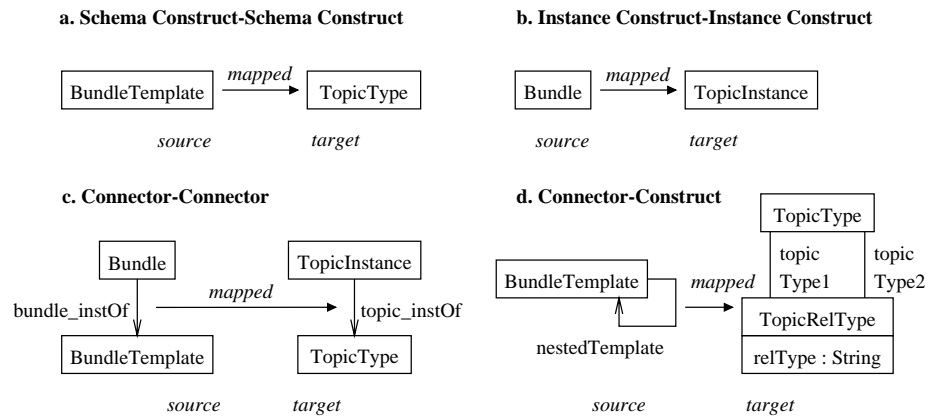


Figure 15: Informal description of inter-model mappings from Bundle-Scrap to Structured Maps.

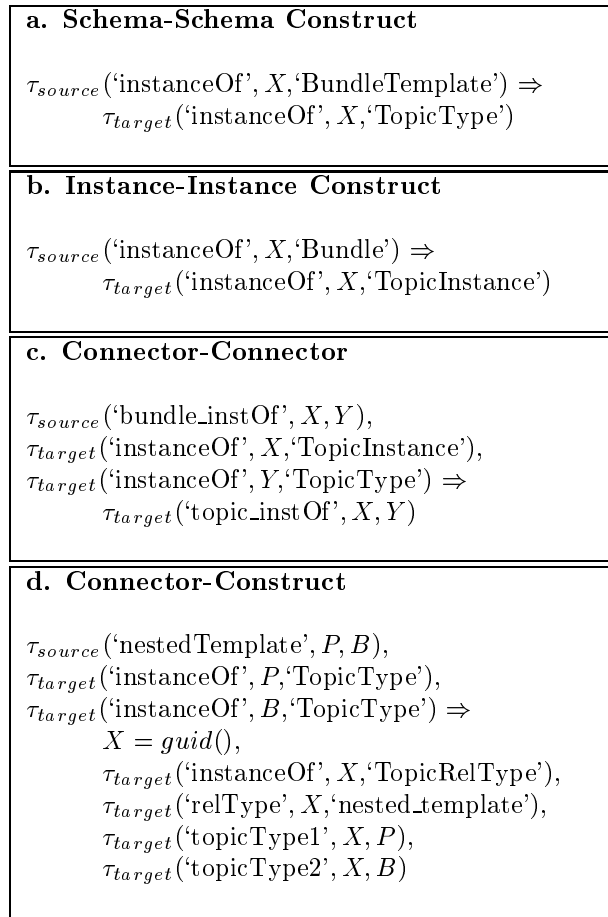


Figure 16: Inter-model mappings from Bundle-scrap to Structured Maps.

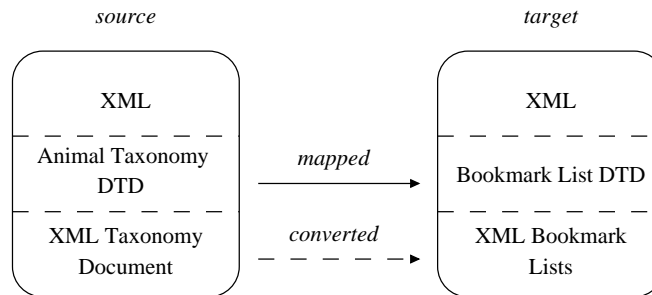


Figure 17: An example inter-schema mapping between two XML DTDs.

of the mapping of Figure 17. The first two mapping rules select genus and species elements along with their name attributes, and maps them to a folder with the appropriate title (e.g., title “homos” or “sapiens”). The last rule maintains the source document’s nested structure, so that species elements (e.g., the sapiens elements) are mapped as nested folders within their appropriate genus folders. The source and result of the conversion is also shown in Figure 18.

The last mapping of Figure 13, shown as Figure 13(c), is called a model-to-schema mapping. Here, the model of the source is mapped to schema-level data in the target, which allows the schema- and instance-level data of the source to be converted to valid instance-level data in the target. Figure 19 shows a model-to-schema mapping in which the Structured-Map model is mapped to an XML DTD. The Structured-Map schema-level and instance-level data are converted to an XML document. The benefit of this mapping would be to use XML as the interchange format for Structured-Maps.

To specify the mapping, we map a Topic Instance construct in the Structured-Map model to an Element Type in an XML DTD with an Attribute Type titled “name.” For the Topic Instance title “painter,” the conversion results in the XML tag `<topic name="painter">`. Figure 20 shows two rules to perform the mapping. Notice that the first rule has an empty left-hand side. Rules without left-hand sides automatically match, which means that the right-hand side triples are always added to the target.

A model-to-schema mapping is only one example of mapping between levels. In general, we impose no limitation on the types of mappings that can be specified within a set of mapping rules. That is, for any mapping between source and target, there may be mapping rules that are model-to-model, schema-to-schema, and so forth.

## 6 Applying and Enhancing the Uni-Level Description

In this section, we describe three separate directions we have been pursuing to extend the applicability of the uni-level description. First, we describe an approach to automatically generate an application programming interface that simplifies the work required for an application developer to use the uni-level description. We then briefly discuss an approach for enhancing the performance of mapping rules, which takes advantage of fixed models and possibly fixed schemas (i.e., for some applications the model and possibly the schema are static). Finally, we describe how we have extended the metamodel including the addition of collections (specifically, lists and sets).

### 6.1 The Application-Specific Data Manipulation Interface

To make it simpler for application developers to store data using the uni-level description (i.e., so that triples don’t have to be manipulated by hand), we have developed what we call the *Data Manipulation Interface* (DMI). The DMI provides the bridge between the application data and the uni-level description, which becomes the underlying data representation scheme for the application. The developer of an application is able to create and manipulate data of interest by invoking the DMI. The implementation of the DMI routines present objects of interest to the application but also maintains and stores the same information as triples. The DMI is specific to the

<p><b>a. Source Schema</b></p> $\tau_{source}(\text{'instanceOf'}, \text{'genus\_type'}, \text{'ElementType'})$ $\tau_{source}(\text{'elementTypename'}, \text{'genus\_type'}, \text{'genus'})$ $\tau_{source}(\text{'instanceOf'}, \text{'species\_type'}, \text{'ElementType'})$ $\tau_{source}(\text{'elementTypename'}, \text{'species\_type'}, \text{'species'})$ $\tau_{source}(\text{'nestedElementType'}, \text{'genus\_type'}, \text{'species\_type'})$ $\tau_{source}(\text{'instanceOf'}, \text{'name\_attType'}, \text{'AttributeType'})$ $\tau_{source}(\text{'attTypeName'}, \text{'name\_attType'}, \text{'name'})$ $\tau_{source}(\text{'attTypeOf'}, \text{'genus\_type'}, \text{'name\_attType'})$ $\tau_{source}(\text{'attTypeOf'}, \text{'species\_type'}, \text{'name\_attType'})$										
<p><b>b. Target Schema</b></p> $\tau_{source}(\text{'instanceOf'}, \text{'folder\_type'}, \text{'ElementType'})$ $\tau_{source}(\text{'elementTypename'}, \text{'folder\_type'}, \text{'folder'})$ $\tau_{source}(\text{'nestedElementType'}, \text{'folder\_type'}, \text{'folder\_type'})$ $\tau_{source}(\text{'instanceOf'}, \text{'title\_attType'}, \text{'AttributeType'})$ $\tau_{source}(\text{'attTypeName'}, \text{'title\_attType'}, \text{'title'})$ $\tau_{source}(\text{'attTypeOf'}, \text{'folder\_type'}, \text{'title\_attType'})$										
<p><b>c. Example Mapping Rules</b></p> <p>rule 1: <math>\tau_{source}(\text{'elemInstOf'}, X, \text{'genus\_type'})</math>,  <math>\tau_{source}(\text{'attributeOf'}, X, A)</math>,  <math>\tau_{source}(\text{'attInstOf'}, A, \text{'name\_attType'})</math>,  <math>\tau_{source}(\text{'attValue'}, A, T) \Rightarrow</math>  <math>\tau_{target}(\text{'elemInstOf'}, X, \text{'folder\_type'})</math>,  <math>\tau_{target}(\text{'attInstOf'}, A, \text{'title\_attType'})</math>,  <math>\tau_{target}(\text{'attributeOf'}, X, A)</math>,  <math>\tau_{target}(\text{'attValue'}, A, T)</math></p> <p>rule 2: <math>\tau_{source}(\text{'elemInstOf'}, X, \text{'species\_type'})</math>,  <math>\tau_{source}(\text{'attributeOf'}, X, A)</math>,  <math>\tau_{source}(\text{'attInstOf'}, A, \text{'name\_attType'})</math>,  <math>\tau_{source}(\text{'attValue'}, A, T) \Rightarrow</math>  <math>\tau_{target}(\text{'elemInstOf'}, X, \text{'folder\_type'})</math>,  <math>\tau_{target}(\text{'attInstOf'}, A, \text{'title\_attType'})</math>,  <math>\tau_{target}(\text{'attributeOf'}, X, A)</math>,  <math>\tau_{target}(\text{'attValue'}, A, T)</math></p> <p>rule 3: <math>\tau_{source}(\text{'nestedElemType'}, X, Y) \Rightarrow</math>  <math>\tau_{target}(\text{'nestedElemType'}, X, Y)</math></p>										
<p><b>d. Source Document before (left) and after (right) conversion</b></p> <table> <tr> <td><code>&lt;genus name="Homos"&gt;</code></td> <td><code>&lt;folder title="Homos"&gt;</code></td> </tr> <tr> <td><code>  &lt;species name="Sapiens"&gt;</code></td> <td><code>  &lt;folder title="Sapiens"&gt;</code></td> </tr> <tr> <td><code>    ...</code></td> <td><code>    ...</code></td> </tr> <tr> <td><code>  &lt;/species&gt;</code></td> <td><code>  &lt;/folder&gt;</code></td> </tr> <tr> <td><code>&lt;/genus&gt;</code></td> <td><code>&lt;/folder&gt;</code></td> </tr> </table>	<code>&lt;genus name="Homos"&gt;</code>	<code>&lt;folder title="Homos"&gt;</code>	<code>  &lt;species name="Sapiens"&gt;</code>	<code>  &lt;folder title="Sapiens"&gt;</code>	<code>    ...</code>	<code>    ...</code>	<code>  &lt;/species&gt;</code>	<code>  &lt;/folder&gt;</code>	<code>&lt;/genus&gt;</code>	<code>&lt;/folder&gt;</code>
<code>&lt;genus name="Homos"&gt;</code>	<code>&lt;folder title="Homos"&gt;</code>									
<code>  &lt;species name="Sapiens"&gt;</code>	<code>  &lt;folder title="Sapiens"&gt;</code>									
<code>    ...</code>	<code>    ...</code>									
<code>  &lt;/species&gt;</code>	<code>  &lt;/folder&gt;</code>									
<code>&lt;/genus&gt;</code>	<code>&lt;/folder&gt;</code>									

Figure 18: Example (a) source schema and (b) target schema, (c) inter-schema mapping rules, and (e) source document excerpt before and after conversion.

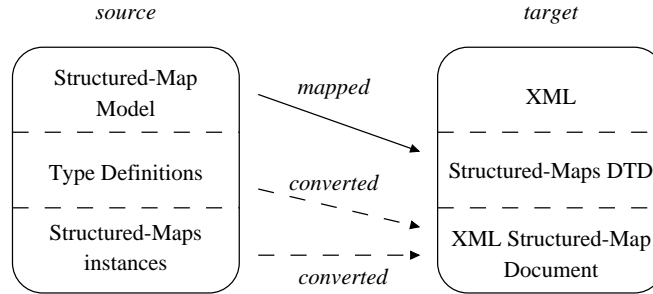


Figure 19: Example of a model-to-schema mapping.

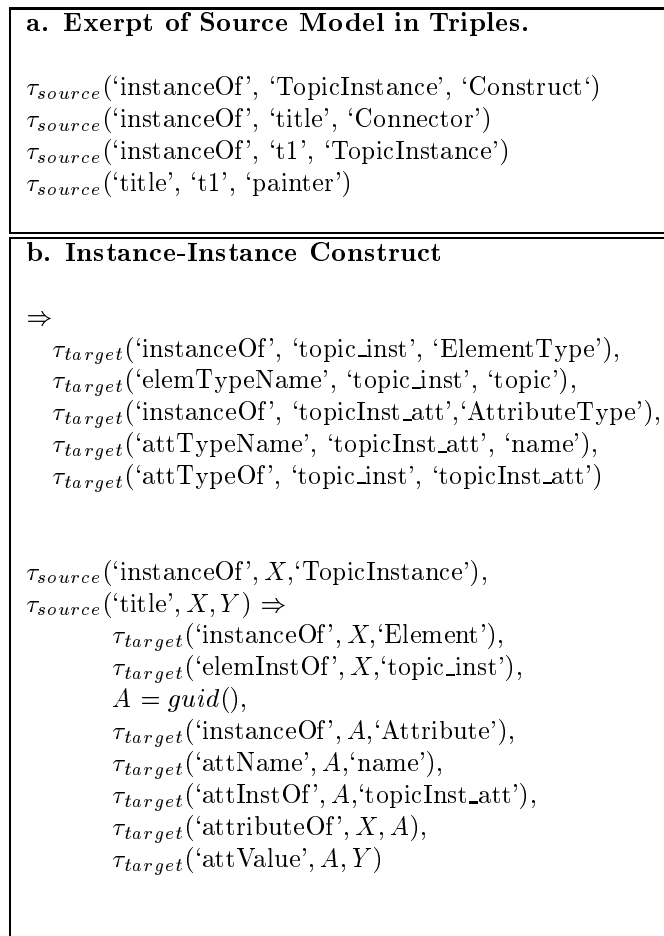


Figure 20: Example Mapping Rules.

data model for the application. As an example, the SLIMPad application is able to create and manipulate Bundle and Scrap objects through the DMI while the persistent version of the data is stored as triples.

A number of strategies can be used to implement a DMI. To date, we have built two DMIs, one for SLIMPad and one for a modified XML data model that can represent multiple sets of XML documents. Both SLIMPad and the XML application (which is an XML extractor) are written in object-oriented languages and this influenced our DMI implementations. Our strategy is to present the application with a set of classes that represent the data model constructs along with read-only operations (i.e., the objects of these classes can't be modified by the application) and provide a separate class that contains the operations to create and update the read-only objects.

Figure 21 shows the classes for SLIMPad that the DMI presents, where DMI read-only objects have the same name as the original constructs of the model and DMI update operations are in a class called *SLIMPadDMI*. Each construct in the Bundle-Scrap data model is represented by a read-only class. As shown in Figure 21, the DMI includes operations to create, delete, connect, detach, and retrieve construct instances, update required connectors, persist, and load triple data. When a set of triples are loaded through the DMI, it creates the appropriate set of read-only objects that represent the corresponding application data (the logical representation). Applications like SLIMPad can use the read-only objects for navigating the schema- and instance-level data.

Figure 22 shows the components used to manage triples through the DMI. Besides providing the appropriate manipulation operations, the DMI must keep the underlying triples consistent with the application data. To do this, the DMI uses the triple manager (TRIM), which has operations to create, remove, persist (e.g., through XML files), query, and materialize simple views over the underlying triples. The TRIM component sits on top of a storage component, which we call the TRIM Store. As the name implies, the TRIM Store stores the underlying triples. Currently, the TRIM Store is an XML file, however, we could employ a DBMS to store the underlying triples.

Our goal is to automatically generate the application-specific DMI for a data model expressed using the metamodel (either graphically or using triples). One of the main challenges in automatic DMI generation is to guarantee that DMI operations respect the data model. That is, we don't want to generate DMI operations that can be used to create data that does not conform to the data model. As a simple example, the Bundle-Scrap DMI shouldn't allow creation of SlimPad objects with more than one root bundle.

Our approach to guarantee consistent DMI operations is to, whenever possible, define them in such a way that it is unnecessary to verify that the operation's result is consistent with the constraints of the data model. We do this by providing DMI operations that are not atomic (i.e., they could be broken into smaller operations), but result in triples that do not conflict with the constraints of the data model. For example, to create a SLIMPad you must use the *create-SLIMPad-Bundle* operation, which takes arguments to create both a SLIMPad and a Bundle, and as a side effect, attaches the Bundle instance to the SLIMPad instance through the rootBundle connector. According to the Bundle-Scrap model, a SLIMPad must have a root Bundle (i.e., the range multiplicity of the rootBundle connector is 1..1) and given a particular Bundle, the Bundle is either the root Bundle of a SLIMPad or it isn't (as specified by the domain multiplicity

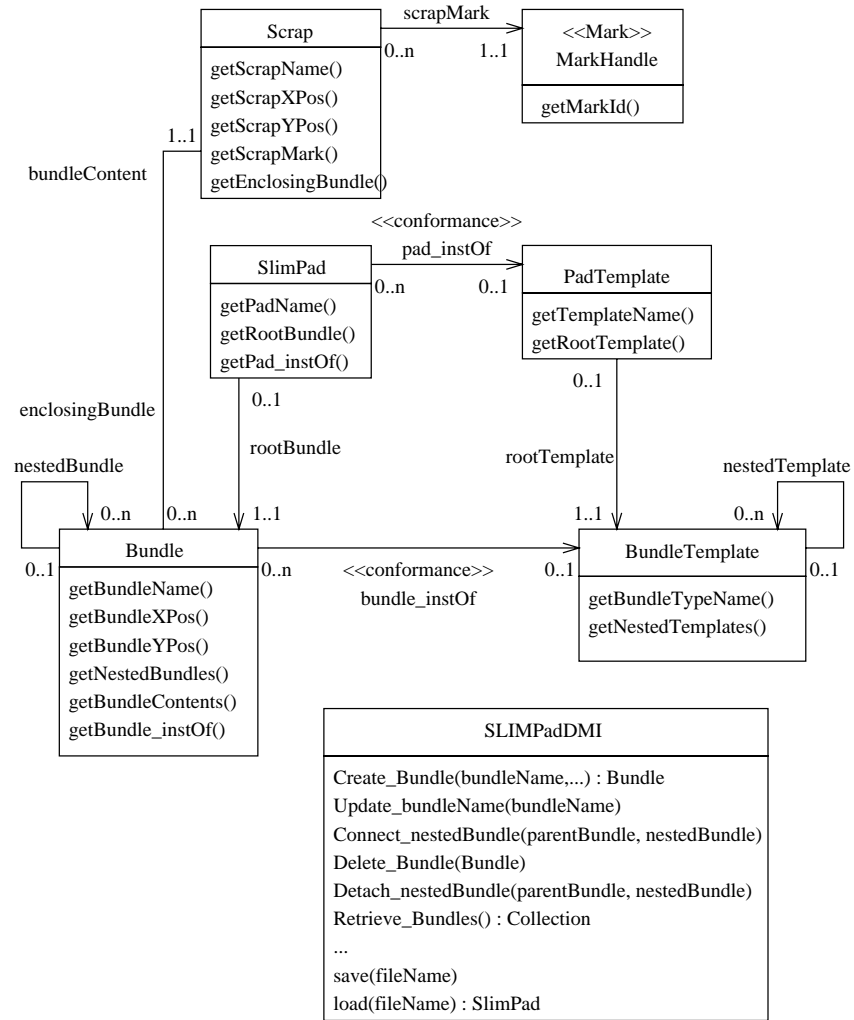


Figure 21: The DMI and corresponding classes for SLIMPad.

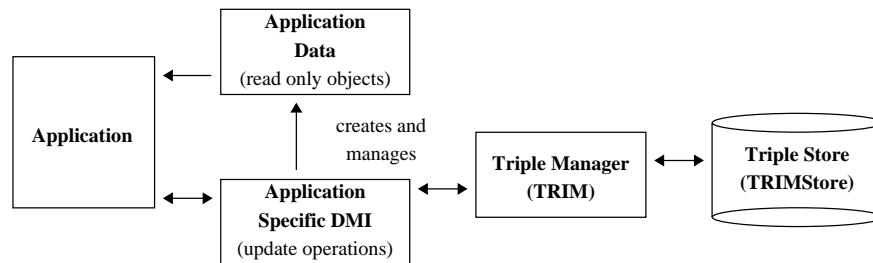


Figure 22: The DMI within the superimposed-information management architecture.

0.1 of the rootBundle connector). If instead we had a SLIMPad creation operation that took a Bundle instance as an argument (rather than creating the Bundle itself), it would be possible to violate the data model constraints by attaching the same Bundle instance to different SLIMPad instances. Note that we cannot use this approach for every potential DMI operation.

Additionally, we see a benefit to allowing programmers customize a DMI for their particular application needs. For example, a superimposed application (such as SLIMPad) may require complex data management capabilities such as being able to query application data whereas an extractor or injector (see Figure 12) will require only a minimal set of operations (e.g., extractors do not generally need delete capabilities). We are currently investigating ways to allow developers to specify such DMI constraints.

## 6.2 Exploiting Fixed Models and Schemas

For the application-specific DMI, we exploit the fact that applications generally use fixed data models. We can use this same approach of exploiting fixed data models when we transform information. For a particular mapping between triple collections, if we know the source or target has a fixed model, we can pre-generate specialized predicates to be used in the mapping rules. For example, instead of using a triple predicate to describe a Bundle instance with a number of related triples for the attributes of the Bundle, we can introduce a *bundle*-predicate with arguments for the bundle identifier, bundle name, position, and so on. This pre-generation step (or “early binding” of model information) not only makes it easier to write mapping rules (since the resulting predicates are more descriptive), but also provides performance benefits.

Performance of the mappings improves because fewer joins (or bindings, in Prolog) are required with the predicate that represents a Bundle, for example. That is, multiple triple formulas are required (and must be joined together) to gather all the information about a bundle (recall that there is a triple for each attribute). However, only one formula is required if we use the bundle predicate directly (since all of its attributes are represented by the predicate). This approach is similar to pointers in object oriented databases, which can be viewed as joins between classes where the join is performed by following pointers. Performance is also improved because we partition the facts into several predicates (or tables if we use SQL), each of which contain fewer entries than a single triple predicate (or single table in SQL), which is the case in both the Prolog and SQL implementations. Note that performance benefits are dependent on the complexity of the rules as well as the size of the data being transformed.

Beyond leveraging fixed data models, we are currently looking at ways to exploit fixed schemas. Our current approach is to promote a given schema to the model level (called *schema-lifting*). We treat the schema as if it were a fixed model and either generate a DMI specifically for the schema or pre-generate the appropriate schema-level predicates for use in mapping rules. Schema-lifting presents two challenges. First, we must have a way of deciding the correct names to use for the new, fixed-schema predicates. We must also determine which items are “lifted.”

One approach to finding names for the new predicates is to use construct identifier names, assuming they were chosen with this purpose in mind. For example, we might have the triple  $\tau(\text{'instanceOf'}, \text{'genus'}, \text{'ElementType'})$ , where “genus” is the ElementType instance identifier. Using the identifier, we can create a construct with the name “genus” (and a corresponding map-

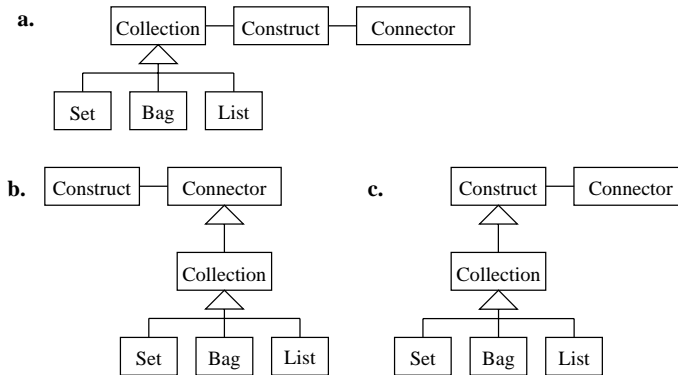


Figure 23: Three approaches to adding collections to the Metamodel: (a) as a new metamodel element, (b) as a special kind of connector, and (c) as a special kind of construct.

ping rule predicate could be generated with the same name). A more general approach it to use an external specification to define predicate names (i.e., something similar to a key specification). In the case of XML, we might have a specification that tells us to use the `elementType` property as the construct name for `ElementType` instances.

To solve the second challenge of determining the schema items to be lifted, we can use the `extract-schema` function from Section 6 to select the schema-level data for early-binding.

### 6.3 Metamodel Extensions

Our motivation for extending the metamodel is to accommodate a broader range of structural models (and modeling features). However, we do not want to clutter the metamodel with elements that can be represented using existing metamodel elements.

One item that is missing in the metamodel is collection. Currently, the metamodel implicitly provides the use of bags through multiplicity constraints on connectors (i.e., 0..n or 1..n range constraints), but lists and sets cannot be completely described. We believe collections should be first-class elements at the model-level. That is, it should be possible to explicitly define collection structures, including names, constraints, and the types of elements the collection can have, at the model-level. A number of models, including object-oriented database varieties, allow collections to be specified at the schema-level as well, which we would support. Collections are also important for abstract addressing (see Section 7). We have chosen to introduce sets, bags, and lists as the collection types.

Figure 23 shows three possible approaches for supporting collections in the metamodel. The first approach, shown as Figure 23(a), adds collection as a unique element within the metamodel, where a collection can be a set, bag, or list. To implement this approach, a mechanism to instantiate collections at the model-level would need to be introduced, which would involve creating an identifier for the collection and defining the types of constructs a collection can contain. The main drawback to this approach is that we cannot use connectors, since they connect constructs, with the collection (making it difficult to use the `instanceOf` connector, and so on).

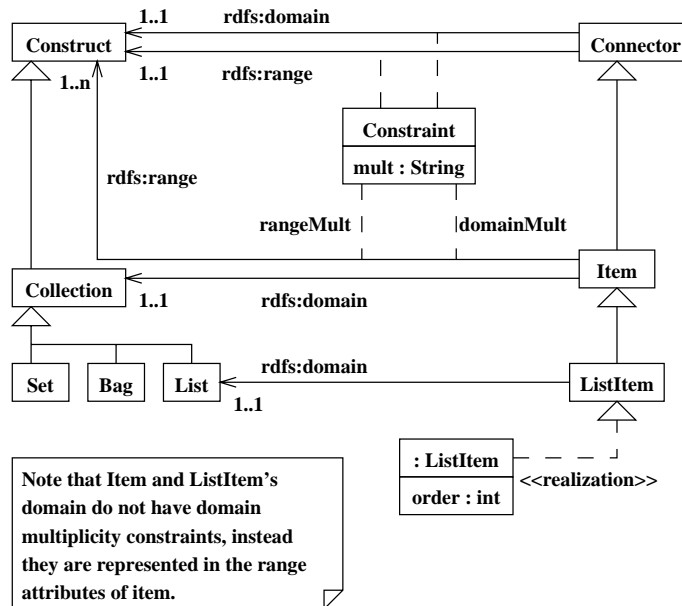


Figure 24: Representation of collection as a special Construct.

Figure 23(b) shows a different approach, where a collection is a special type of connector. The approach is limited in that you cannot define multiple collection content types (i.e., the constructs that the collection can contain). Also, collections would not be first-class elements themselves and, for example, we could not attach additional attributes (i.e., connectors) to the collection or have multiple constructs reference the same collection.

The approach we are currently experimenting with is shown in Figure 23(c), where a collection is defined as a special type of construct. This approach allows collections to be first-class elements at the model-level. That is, collections can be used as if they were ordinary constructs and connectors can reference them. To implement this approach, we add a new type of connector to the metamodel called *item*, which specifies the allowable constructs within particular collections. Figure 24 shows a working draft of this approach. A particular collection can be the domain for at most one item connector, but item connectors are allowed to have a range consisting of multiple constructs (representing the allowable constructs of the collection). The item connector contains a domain constraint for each construct allowable in the collection, which describes the number of collection instances each allowable construct instance can participate in. Additionally for lists, we must include order, which is done using a *listItem* connector, a specialization of the item connector.

Another metamodel extension is the additions of two new constraints that can be placed on connectors, called “inverseOf” and “conformsTo.” Both allow two connectors to be associated. As their names imply, inverseOf constrains two connectors to be inverse relations and conformsTo constrains one connector to be at the instance-level and the other to be the corresponding connector at the schema-level.

Other additions we are exploring include adding choice connectors (for variant or union specification) and abstract addressing items such as hierarchy specification, entry point constraints, and key constraints (over col-

lections).

## 7 Future Work

We are interested in applying the uni-level description to the problem of information addressing. An important part of superimposed information is the idea of maintaining marks to a base layer of information. We envision extending our current superimposed-information architecture [15] to enable generic, logical addressing capabilities across a wide range of information sources to support marks. We believe we can define various addressing schemes against the basic structures of the uni-level description metamodel. These addressing schemes can then be used to generically address information sources represented using the uni-level description. As a simple example, the metamodel includes the connector (between constructs). An associated addressing mechanism would be a relative address that takes you from one construct to the connected construct. Such relative addressing is the basis for path expressions. Similarly, hierarchy of constructs is a structural feature that naturally suggests an addressing mechanism.

We recognize that an information source, such as a document, may have multiple manifestations. For example, a particular document might be manifest in Word, in HTML, and in PDF. By leveraging mappings between information sources (represented using the uni-level description), we believe we can support “canonical addressing,” i.e., an address in one document manifestation can be used to address the same information in another manifestation.

We see a number of benefits to abstract addressing including:

1. **The separation of physical from logical (or conceptual) addresses.** A number of base information sources provide very low-level addressing capabilities. Examples include text files and raw audio files. However, by providing a higher level, structural skeleton, over these low-level addressing mechanisms we can logically address these sources via the skeleton.
2. **The enrichment of existing addressing capabilities.** Here we are interested in exploiting structural features that exist in information sources but are not supported by the base application’s addressing capabilities. For example, the structural model may allow hierarchically structured data even though the base application does not support hierarchical addressing.
3. **The ability to map between addressing schemes.** Two different information sources may have slightly different hierarchical addressing models. By mapping between them, we can create canonical marks (i.e., a single address for multiple document manifestations).

Also included in our future plans is the development of a visual mapping language to simplify the creation of mapping rules between representation schemes. Based on the visual mapping, we want to generate the necessary underlying mapping rules that would be needed to perform the desired transformations. The main challenges in this work are to define an appropriate interface that enables complex transformations, while being easy to use (for example, circling or clicking source and target constructs to define mappings at the model-level).

## 8 Related Work

A number of metamodels have been developed (see Atzeni and Torlone [1, 2, 3], Barsalou and Gangopadhyay [4], McBrien and Poulouvasilis [21], Cluet et al. [12], and the Meta Object Facility [18]) with primary focus on supporting interoperability. Our metamodel is different from these approaches because we do not require model-first nor schema-first definitions. Rather, we support the independent specification of model, schema, and instance and we permit the application to explicitly specify the relationship between schema and instance. Metamodels for describing database data models [1, 2, 3, 4, 21, 10, 12, 13] and object-oriented models [18, 22] require that instances be the extension of schema in which schema must be defined first. By not enforcing schema-first definitions and allowing instances to be independent of schema, our metamodel is able to accurately define various models such as XML and Topic Maps. Additionally, by explicitly representing the relationship between schema and instance we can specify more complex situations such as multiple levels of schema-instance relationships.

Another difference between our approach and other metamodel approaches is that we employ a single, generic representation scheme for model, schema, and instance data. The representation scheme allows mappings to be defined in a uniform way between models (inter-model), schema (inter-schema), model and schema (model-to-schema), and any mixture of the three levels. In contrast, the Hypergraph Data Model (HDM) can store schemas defined in diverse models and be used to specify transformations from the extent of a schema in one model (e.g., the relational model) to the extent of a similar schema in a different model (e.g., the entity-relationship model). But, both types of transformations are considerably more difficult to specify when compared to our mapping rules, because they do not explicitly represent models or instances.

Atzeni and Torlone [1, 2, 3] rather than use a logic-based language, employ procedural inter-model mapping specifications. They also require complete mappings between models, whereas we allow partial mappings to provide a wider range of cases, and they only map between models (and not between schema or models and schema). Our approach of using a horn-clause logic language as a basis for specifying transformations is similar to the languages defined by HDM, YAT [10], and WOL [13] for transforming data.

The Meta Object Facility (MOF) defines an architecture that uses a metamodel to enable the sharing of information between object-oriented applications. Currently, the main application of the MOF architecture is to store and interchange UML class diagrams between analysis and design tools. The MOF uses the XML Metadata Interchange (XMI) as a representation scheme for exchange. XMI prescribes a method to generate an XML DTD to represent a model. (Note that the XML DTD is generated by hand and the UML DTD is the only version currently available.) XML documents that conform to the DTD represent schema-level data. Unlike our approach, there is no way to represent instance-level data. Also, MOF does not provide any support for mapping between models.

The Microsoft Repository [5, 6] is similar to the MOF, except it does not define a metamodel. Instead, a global model called the Open Information Model is used to define schemas. However, our approach provides a mechanism to represent various models precisely to leverage available tools that are based on a particular model.

## 9 Conclusion

We have presented a framework for representing and transforming various model-based representation schemes. The framework consists of a meta-model for describing data models, which uniquely allows for the explicit specification of the relationship between schema and instance, and provides a single representation scheme, based on RDF, to describe model, schema, and instance data that can be used generically by various applications. Based on the generic representation scheme, we provide a mapping formalism that can be used to transform data at various levels, including model and schema mappings. We present a number of ways to leverage the representation scheme such as the application-specific data manipulation interface, abstract addressing, and the uses of fixed model and schema. Finally, we discussed a number of potential metamodel extensions to allow for a wider range of modeling features.

## Acknowledgements

This work was supported in part through NSF grants IIS-98-17492, CDA-97-03218, and EIA-99-83518, and DARPA grant N66001-00-8032. We would also like to thank David Maier for his helpful discussions, contributions, and comments as well as Longxing Deng and Mathew Weaver for their contributions to the SLIMPad architecture.

## References

- [1] Paolo Atzeni and Riccardo Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Information Systems*, 18(6):349–362, September 1993.
- [2] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of the International Conference on Extending Database Technology (EDBT '96)*, volume 1057 of *Lecture Notes in Computer Science*, pages 79–95, March 1996.
- [3] Paolo Atzeni and Riccardo Torlone. MDM: A multiple-data-model tool for the management of heterogeneous database schemes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 528–531, May 1997.
- [4] Thierry Barsalou and Dipayan Gangopadhyay. M(DM): An open framework for interoperation of multimodel multidatabase systems. In *Proceedings of the 8th International Conference on Data Engineering (ICDE '92)*, pages 218–227, February 1992.
- [5] Philip Bernstein and Thomas Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, 1999.
- [6] Philip Bernstein, Brian Harry, Paul Sanders, David Shutt, and Jason Zander. The Microsoft repository. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, pages 3–12, August 1997.
- [7] Michel Biezunski, Martin Bryan, and Steve Newcomb. *Topic Maps*. <http://www.y12.doe.gov/sgml/sc34/document/0129.pdf>, April 1999. ISO/IEC 13250.

- [8] Tim Bray, Jean Paoli, and C.M. Sperger-McQueen. *Extensible Markup Language (XML) 1.0*. <http://www.w3.org/TR/2000/REC-xml-20001006>, October 2000. W3C Recommendation.
- [9] Dan Brickley and R.V. Guha. *Resource Description Framework (RDF) Schema Specification 1.0*. <http://www.w3.org/TR/2000/CR-rdf-schema-20000327/>, March 2000. W3C Candidate Recommendation.
- [10] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient XML integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 141–152, May 2000.
- [11] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. <http://www.w3.org/TR/xpath>, November 1999. W3C Recommendation.
- [12] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 177–188, June 1998.
- [13] Susan Davidson and Anthony Kosky. WOL: A language for database transformations and constraints. In *Proceedings of the 13th International Conference on Data Engineering (ICDE '97)*, pages 55–65, April 1997.
- [14] Lois Delcambre and David Maier. Models for superimposed information. In *Advances in Conceptual Modeling (ER '99)*, volume 1727 of *Lecture Notes in Computer Science*, pages 264–280, November 1999.
- [15] Lois Delcambre, David Maier, Shawn Bowers, Mathew Weaver, Longxing Deng, Paul Gorman, Joan Ash, Mary Lavelle, and Jason Lyman. Bundles in captivity: An application of superimposed information. In *Proceedings of the 17th International Conference on Data Engineering (ICDE '01)*, pages 111–120, April 2001.
- [16] Lois Delcambre, David Maier, Radhika Reddy, and Lougie Anderson. Structured maps: Modeling explicit semantics over a universe of information. *International Journal on Digital Libraries*, 1(1):20–35, 1997.
- [17] Paul Gorman, Joan Ash, Mary Lavelle, Jason Lyman, Lois Delcambre, David Maier, Mathew Weaver, and Shawn Bowers. Bundles in the wild: Managing information to solve problems and maintain situation awareness. *Library Trends: Accessing Digital Library Services*, 49(2), 2000.
- [18] The Object Management Group. *Meta Object Facility (MOF) Specification*. <http://www.omg.org/cgi-bin/doc?ad/99-09-04>, September 1999.
- [19] Ora Lassila and Ralph Swick. *Resource Description Framework (RDF) Model and Syntax Specification*. <http://www.w3.org/TR/REC-rdf-syntax/>, February 1999. W3C Recommendation.
- [20] David Maier and Lois Delcambre. Superimposed information for the internet. In *Proceedings of the ACM SIGMOD Workshop on The Web and Databases (WebDB '99)*, pages 1–9, June 1999.

- [21] Peter McBrien and Alexandra Poulovassilis. 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*, pages 333–348, June 1999.
- [22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.