A Generic Representation for Exploiting Model-Based Information*

Shawn Bowers and Lois Delcambre
Computer Science and Engineering Department
Oregon Graduate Institute
{shawn, lmd}@cse.ogi.edu

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 (analogous to a data model in a database). For example, the XML model includes elements, attributes, and permits elements to be nested. Similarly, RDF models information through resources and properties.

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 use the generic representation to exploit a formally defined mapping language to transform information, e.g., from one scheme to another. In this paper, we present the generic representation and the associated mapping formalism 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 the XML [8], RDF [18] and Topic Maps [7], and each representation schema 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,

* This work supported in part through NSF grants IIS-98-17492, CDA-97-03218, and EIA-99-83518, and DARPA grant N66001-00-8032.

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).

We present here a generic representation scheme for model-based information that allows the model, the schema, and the instance information to be represented explicitly. Our approach is to use a metamodel with an associated, underlying representation expressed in RDF and RDF Schema. The metamodel is comprised of basic structures that can be instantiated to describe various models or representation schemes of interest.

This work is part of our general investigation into what we call *superimposed information* [15, 16, 19]. We briefly present superimposed information in Section 2. Section 3 describes the metamodel that provides a formalism to define various data models. Section 4 describes the generic representation. Section 5 shows how logical rules over the generic representation can formally specify and implement mappings from one model-based representation to another. Section 6 discusses some of the challenges and opportunities presented by this work. We discuss related work in Section 7 and offer conclusions in Section 8.

2. Superimposed Information

Suppose you are planning a vacation using the Web. Imagine that 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).

To help you manage the cost of the trip, including exchange rates, lodging, and transportation expenses, you use a spreadsheet application, which automatically lifts the appropriate data from the scratchpad. 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 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 Palm) 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 addi-

tional 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.

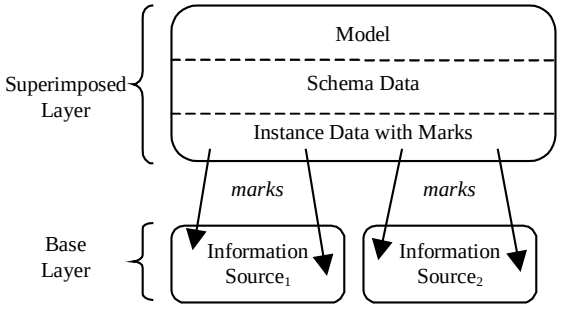


Figure 1. The superimposed and base layers.

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., by organizing, annotating, or highlighting 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 [16]. 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. The superimposed application provides useful capabilities to the user that views and manipulates superimposed information. Our first superimposed application is called *SLIMPad*, the superimposed layer information manager scratch pad. This paper describes our approach to building a generic storage manager for superimposed applications (the Superimposed Information Manager in Figure 2), where the data is stored using our generic representation. Both the generic representa-

tion scheme and the mapping formalism are applicable to any type of structured (or model-based) information in the superimposed or base layers.

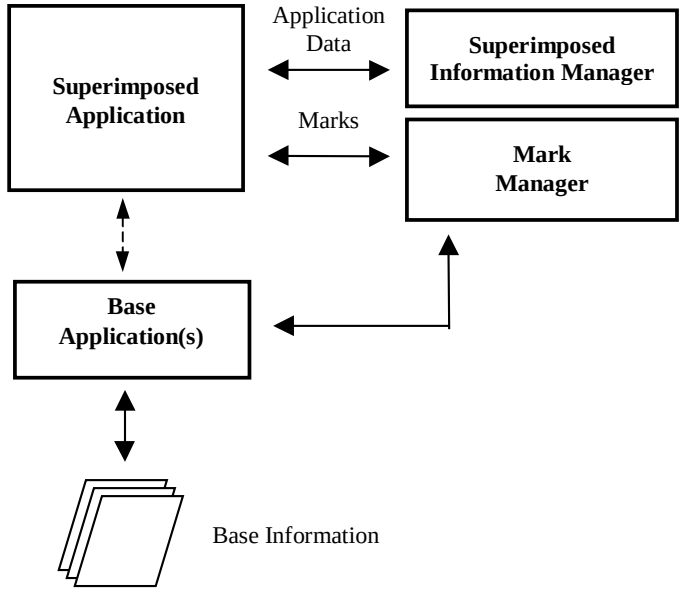


Figure 2. Overview of the superimposed application architecture.

3. The Metamodel for Representing Model-Based Information

Information of interest in this research consists of three levels: model, schema, and instances 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 and a schema, without having instance data present.

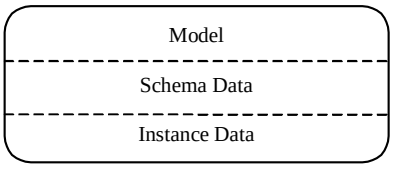


Figure 3. The three layers of information.

To describe multiple models, we define a level of abstraction above the model, called a 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.

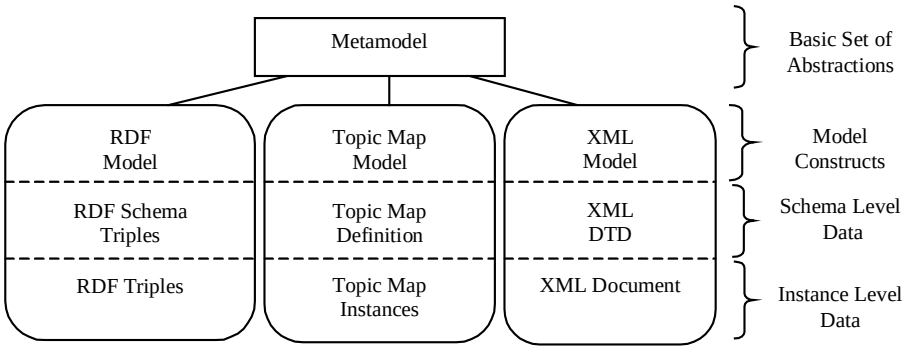


Figure 4. The RDF, Topic Map, and XML models within a superimposed layer.

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. More 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.

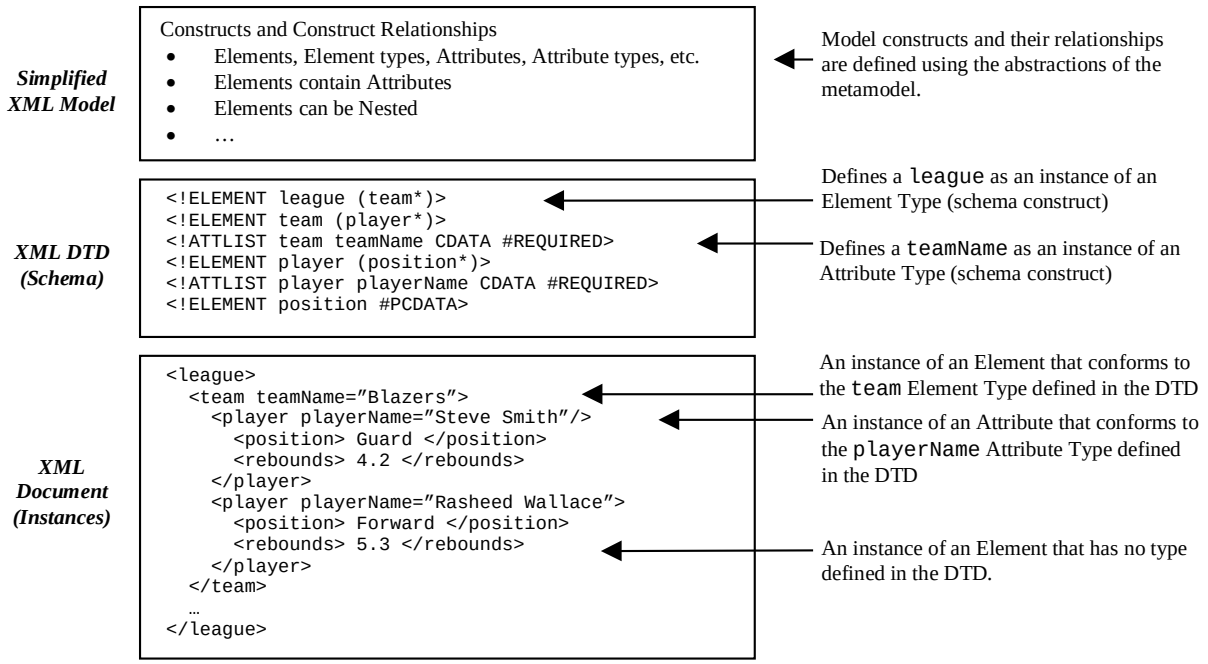


Figure 5. An example of each of the three levels (model, schema, and instance) for XML.

The definition of our metamodel is similar to other metamodel approaches in the object [21, 22] and database communities [1-4, 17, 10, 12, 13] 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 any type even if there is a Topic Map definition.

Additionally, the metamodel accommodates multiple levels of schema-instance relationships. In a Topic Map, topics can have a type that is also a topic, and so it too can have a type, resulting in two levels of schema and instance definition. The metamodel includes the notion of a mark, which signals a reference point to some other source of information (presumably with a different model). The metamodel allows marks to appear at various places within a superimposed model [15].

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.

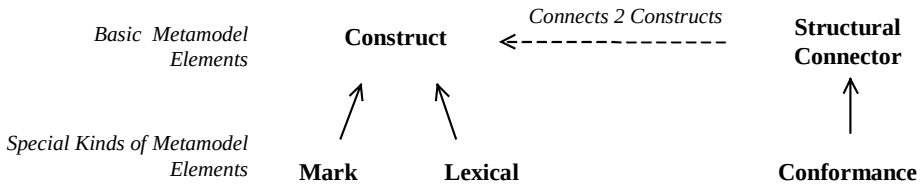


Figure 6. The elements defined by the superimposed-information metamodel.

As shown, a mark and a lexical are defined as two special kinds of constructs. A *mark* describes a model construct whose instances represent connection-points to other information sources. 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.

Table 1. The XML model described in terms of the superimposed-information metamodel. The elements of the XML model (bottom) are instances of the corresponding Metamodel element definitions (top).

	Constructs	**Lexicals**	**Connectors**	**Conformance Connectors**
XML Model	Element Type Attribute Type Element Attribute Primitive Content Type Primitive Content	Primitive-Content	Nested Element Type *Connects Two Element Types* Nested Element *Connects Two Elements* Element Content *Connects an Element to Primitive Content* Element Content Type *Connects an Element Type to Prim. Content Type* Element Attribute *Connects an Element to an Attribute* Attribute Element Type *Connects an Element Type to an Attribute Type*	Element Instance Of *Connects an Element to its Element Type* Attribute Instance Of *Connects an Attribute to its Attribute Type* Content Instance Of *Connects Prim. Content to its Primitive Content Type*

Element constructs in XML form a hierarchy, which is represented by the model connector Nested Element. By using multiplicity constraints, we can specify that an Element is either not nested or nested within one parent Element, and can have many Elements nested within it.

We use conformance connectors to specify schema-instance relationships between Attribute and Attribute Type, Element and Element Type, and Primitive Content and Primitive Content Type. We are also able to apply constraints to the relationship. 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, 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 Superimposed 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 provide a visual representation of models using a subset of the UML [22].

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 (an edge), 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 a primitive type such as an integer or string. For example, the RDF triple (creator, “index.html”, “Ora Lassilla”) can be read as “the creator of index.html is Ora Lassilla” where “creator” is a property name, “index.html” a resource, and “Ora Lassilla” a string [18].

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* con-

straint. 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 Defined using 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.

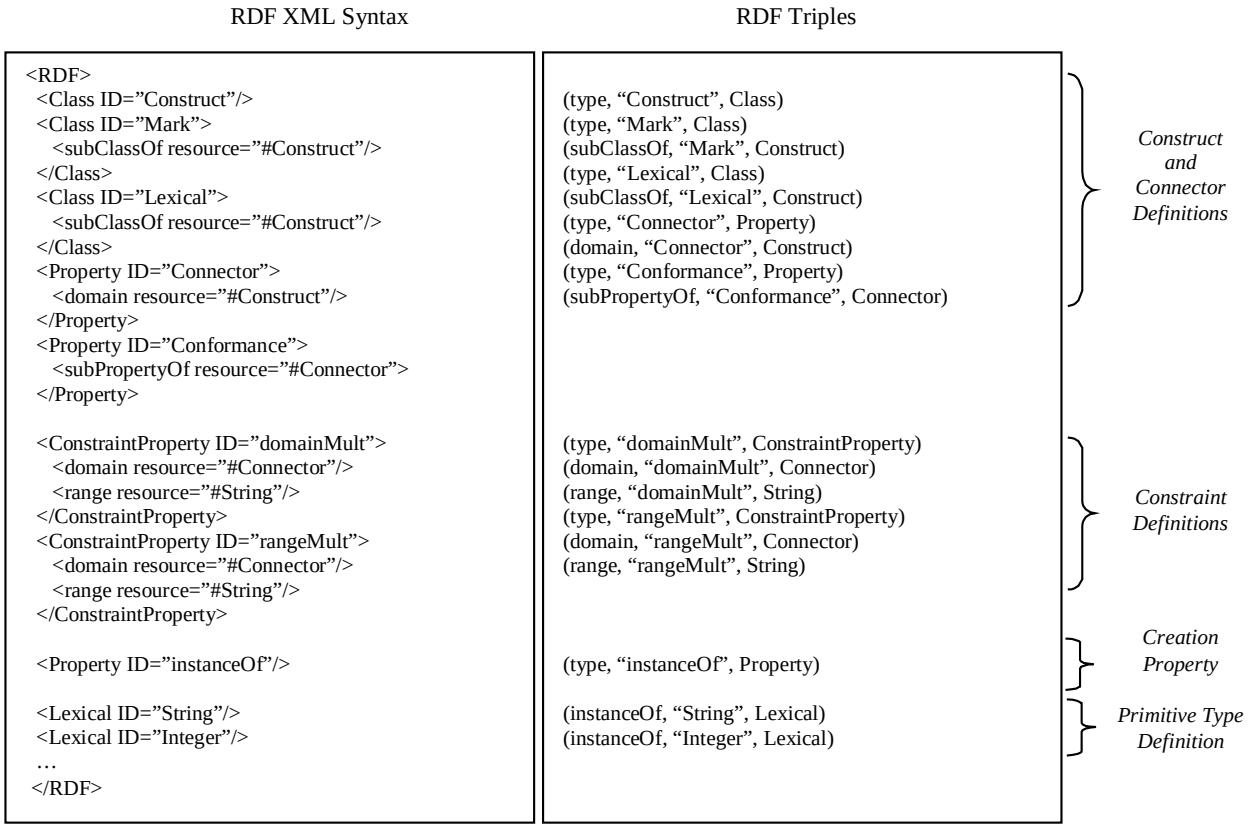


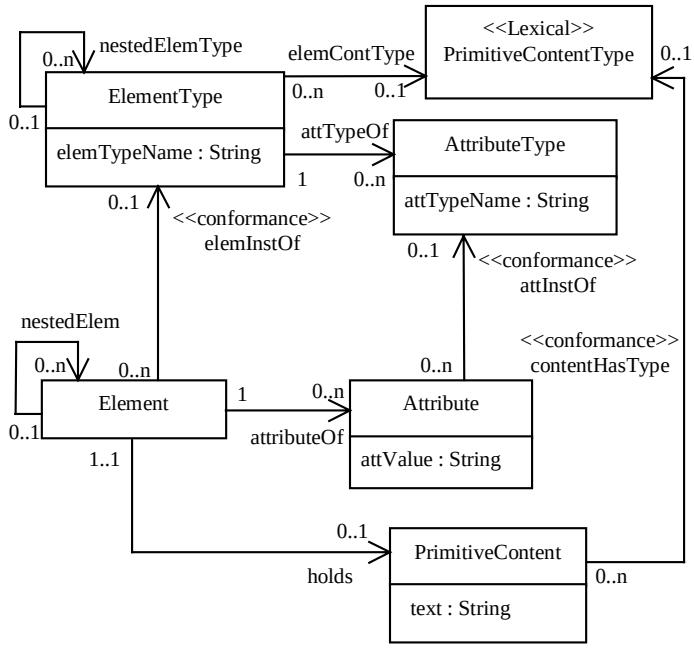
Figure 7. The metamodel represented in RDF XML and as RDF Triples.

Additionally, *domainMult* and *rangeMult*, which are used to specify the multiplicity of a connector end, are defined as RDF *constraint properties*. A constraint property is a higher-order property that can be used to add constraints to a property (domain and range are considered default constraints 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 well as schema and instance data. 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. We use this same approach for defining connector so that we may connect not only to constructs but also to metamodel structures themselves such as Lexical. 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) using similar definitions.

4.3. The RDF and Visual Model Representation

Figure 8 depicts both the RDF and visual representations of the simplified XML model of Section 3 using the metamodel. 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.




```
<RDF>
  <Construct ID="ElementType"/>
  <Construct ID="AttributeType"/>
  <Construct ID="Element"/>
  <Construct ID="Attribute"/>
  <Construct ID="Content"/>
  <Lexical ID="ContentType"/>

  <Connector ID="elemTypeName">
    <domain resource="#ElementType"/>
    <range resource="#String"/>
    <domainMult value="0..n"/>
    <rangeMult value="1..1"/>
  </Connector>
  <Connector ID="nestedElemType">
    <domain resource="#ElementType"/>
    <range resource="#ElementType"/>
    <domainMult value="0..1"/>
    <rangeMult value="0..n"/>
  </Connector>
  <Conformance ID="elemInstOf">
    <domain resource="#Element"/>
    <range resource="#ElementType"/>
    <domainMult value="0..n"/>
    <rangeMult value="0..1"/>
  </Conformance>
  …
</RDF>
```

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

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.

RDF Syntax for 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="rebounds1">
 </Element>
</RDF>
```

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", e5)

(instanceOf, "playerName1", Attribute)
(attInstOf, "playerName1", playerName_attr)
(attValue, "playerName1", "Steve Smith")

(instanceOf, "position1", Element)
(elemInstOf, "position1", position_type)

(instanceOf, "rebounds1", Element)
```

Figure 9. Schema and instance data of the XML model.

Figure 9 shows example schema- and instance-level data that represent the XML document excerpt of Figure 5. Notice that RDF 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.

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 [14], 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.

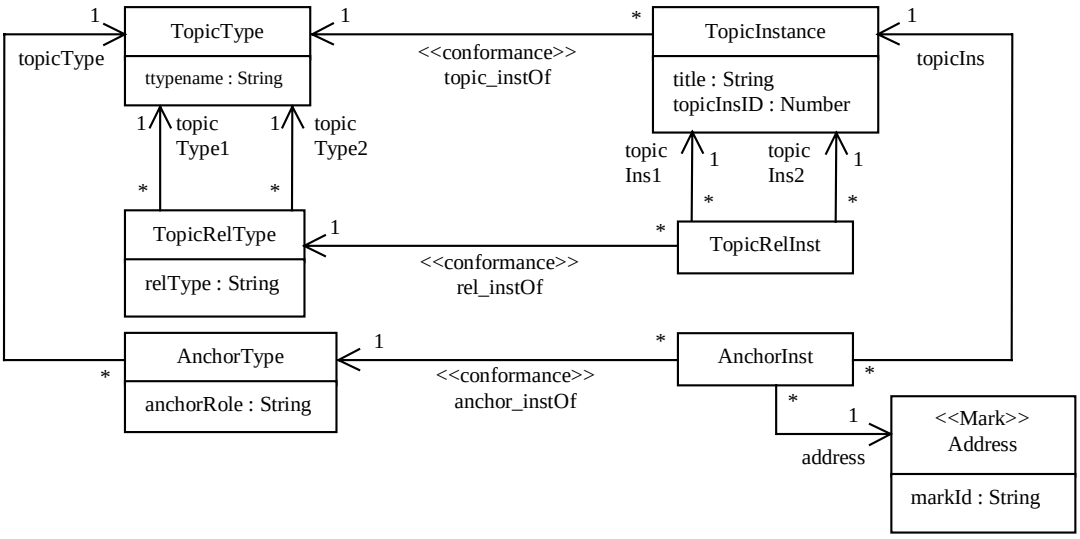


Figure 10. CARTE's Structured Map model.

SLIMPad uses the Bundle-Scrap model and is designed to be a superimposed scratchpad application [16]. 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, and dragging it into the scratch pad. Once content is placed into SLIMPad, a *scrap* is created that contains a mark with a reference back to the content. 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.

Figure 11 shows a portion of the Bundle-Scrap data model with schema-level constructs, which we are incorporating into SLIMPad that allows 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 optional multiplicity constraints, 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 for making notes to describe the patients in an intensive care unit and for developing hypothesis about individual patients. Each template has an associated graphical user interface component for entering data (some of which are fairly unstructured while others are strict in the types of data that can be entered, much like a traditional form).

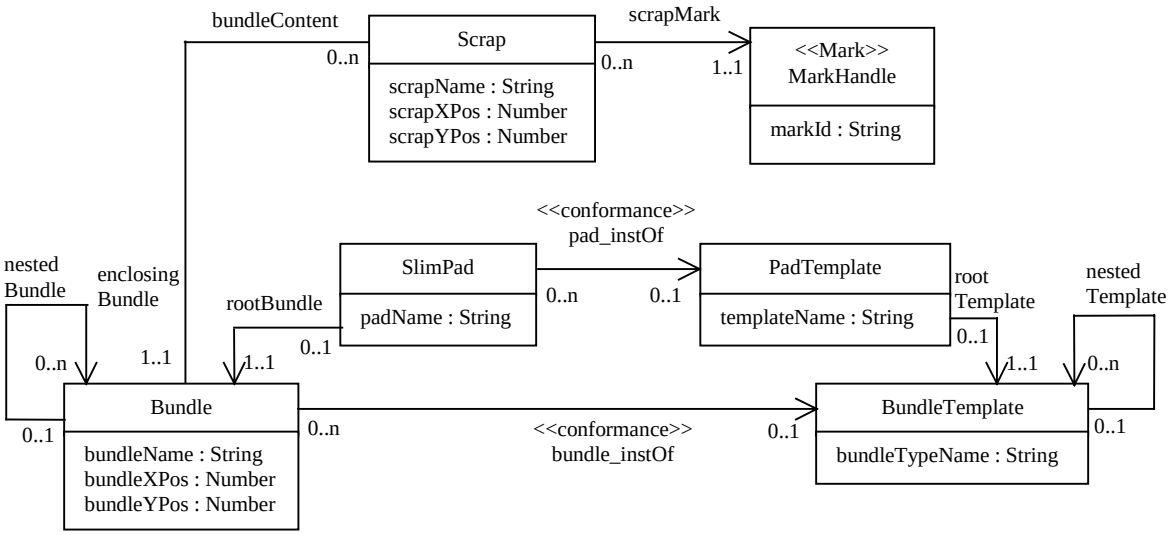


Figure 11. SLIMPad's Bundle-Scrap Model.

In the metamodel, every connector is uni-directional. 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, 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 structurally transform 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 target information into triples. Included in the triples is a description of the data model used by the application. These source triples are transformed through mapping rules into target triples, which are then injected into the desired tool (again either through a programming interface or stored external to the application). The remainder of this section describes the mapping formalism that can be used to transform triples.

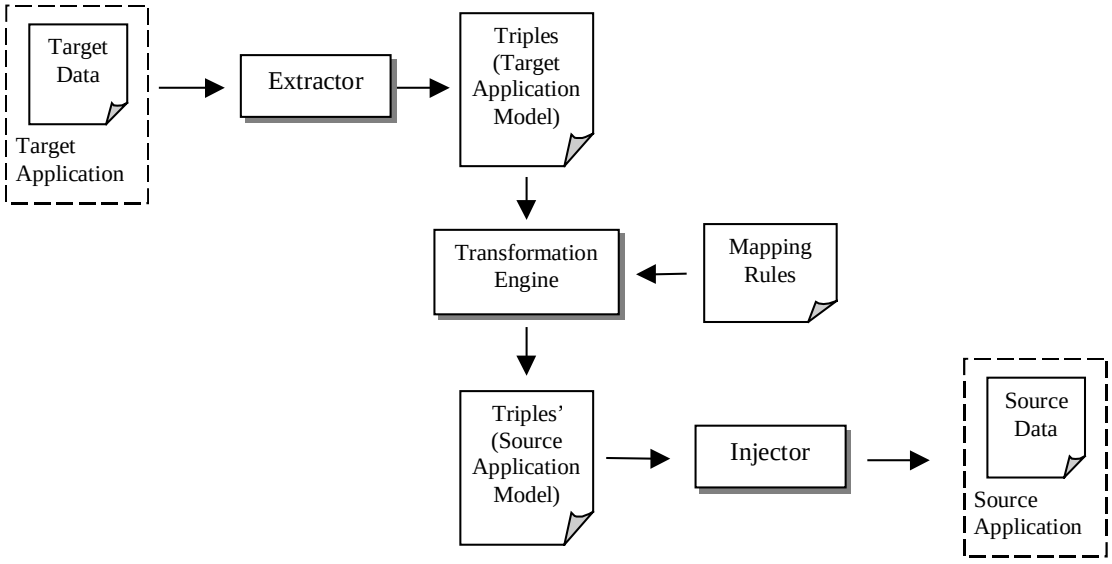


Figure 12. The conceptual architecture in which mappings are performed.

5.1. Mapping Rules

Mappings are specified using production rules defined over triples of the RDF representation. Since a triple can be viewed as a simple predicate (e.g., "triple(creator, index.html, Ora Lassilla)"), we can specify mapping rules using a logic-based language such as Prolog, which allows us to both specify and implement the mappings. Note that the mappings can be implemented using a number of languages; we use Prolog as just one example. We do not require mappings to be complete since only part of a model or schema may be needed while using a specific tool.

Table 2 contains the basic definitions used to specify mappings. RDF triples are represented with the predicate τ. Quotes are used to denote constants and upper-case letters denote variables. For example,

the predicate τ('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). The predicate *S* is true if its τ-predicate is in a set of triples named *L*. For example, *S*('xml', τ('instanceOf', 'Element', 'Construct')) would be true if there were an "Element" construct defined in a set of triples denoted "xml." Typically, *L* will represent either the source or target triples. Finally, we define a mapping rule as a production in which the left- and right-hand sides consist of *S*-predicates. The left-hand side *S*-predicates must be true in order to generate the right-hand side *S*-predicates (i.e., the right-hand side *S*-predicates are produced by the rule). For example, the mapping rule *S*('source', τ('creator', X, Y)) ⇒ *S*('target', τ('owner', X, Y)) would add a triple τ('owner', X, Y) to the target set for every triple that matched τ('creator', X, Y) in the source set. Therefore, if τ('creator', 'index.html', 'Ora Lassilla') is a triple in the source, then the triple τ('owner', 'index.html', 'Ora lassilla') will be added to the target.

Table 2. Predicate and mapping rule definitions.

Symbol	Definition
τ	A predicate that represents an RDF triple, for example τ('creator', 'url', 'person').
L	Denotes a set (that is, database or file) of triple predicates (which usually is the source or target set).
S	A predicate of the form *S*(*L*, τ) that is true if τ ∈ *L*.
M	A mapping that consists of a set of mapping rules.
m	A mapping rule with the form: *T* ⇒ *T*′, where *T*, *T*′ are sets of *S* predicates. The rule can be read as follows: if the left hand side matches (i.e., each *S* ∈ *T* is true) then for each *S*(*L*, τ) ∈ T′ add τ to *L*.

Table 3 describes the conversion function, which is used to perform mappings. The conversion function applies a set of mapping rules to a source and target set of triples, and is implemented by the transformation engine in Figure 12. Conversion applies a set of mapping rules to the source and (possibly) target triple sets, and adds the resulting triples to the target triple set. In one of our current implementations, a Prolog program performs the conversion function.

Also shown in Table 3 are the convenience functions *extract-model* and *extract-schema*. The extract-model function can be used to extract model information from a set of triples. Similarly, the extract-schema function gathers schema information from a set of triples. It returns the construct instances that are at the schema-ends of conformance connectors along with the connections between the schema construct instances. (Note that this approach works for cases where there is only one level of schema and instance.) Finally, Table 3 consists of a skolem function called *guid*, which is used to generate unique identifiers.

Table 3. Functions used to provide mappings.

Function	Definition
Conversion : $M \times L_s \times L_t \rightarrow L_t'$	Conversion takes a mapping M and applies the mapping rules of M to a source set L_s and a target set L_t, and returns the updated target set L_r'. Conversion implements a rule-based algorithm that computes the fixed-point of applying mapping rules to the source and target triple sets.
ExtractModel : $L \rightarrow L'$	$L' \subseteq L$ and $L' = L_1 \cup L_2$ where: L_1 = {t | ∃x,y and t∈L and t=τ('instanceOf',x, y) and y='Construct', 'Mark', 'Lexical', 'Connector', or 'Conformance' } L_2 = {t | ∃p,u,v,x,y and u∈L_1 and u=τ('instanceOf', x, y) and t∈L and t=τ(p,x,v) and y='Connector' or 'Conformance' and p='domain', 'range', 'domainMult', or 'rangeMult'}.
ExtractSchema : $L \rightarrow L'$	$L' \subseteq L$ and $L' = L_1 \cup L_2$ where: L_1 = {t | ∃p,u,v,x,y,z and u,v∈L and u=τ('instanceOf', p, 'Conformance') and v=τ(p, y, z) and t=τ('instanceOf', z, x)} L_2 = {t | ∃p,r,s,u,v,x,y and t∈L and u,v∈L_1 and u=τ('instanceOf', r, x) and v=τ('instanceOf', s, y) and t=τ(p, r, s)}.
guid → x	A 0-ary Skolem function that returns a unique identifier x.

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 set of triples being mapped to a target set 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 triple sets, 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.

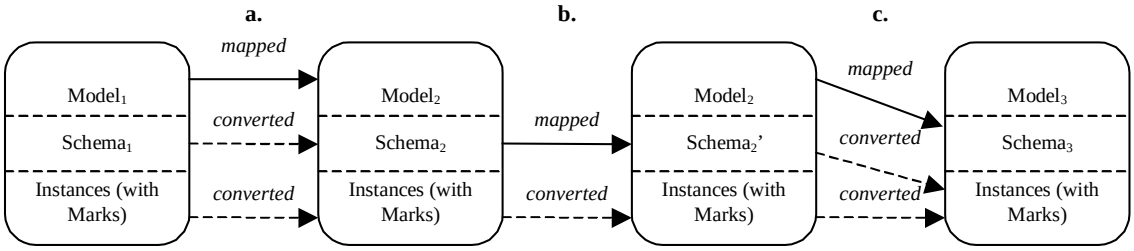


Figure 13. Three mappings: (a) inter-model, (b) inter-schema, and (c) model-to-schema.

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 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.

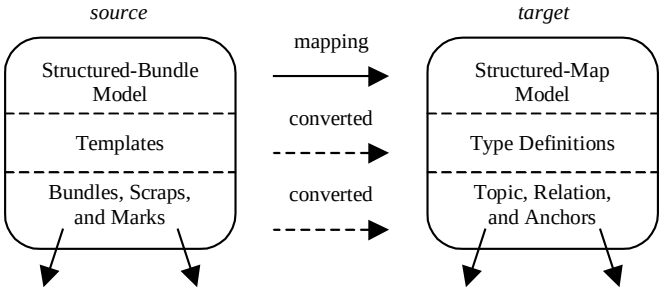


Figure 14. An inter-model mapping between the Bundle-Scrap Model and Structured Map Model.

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.

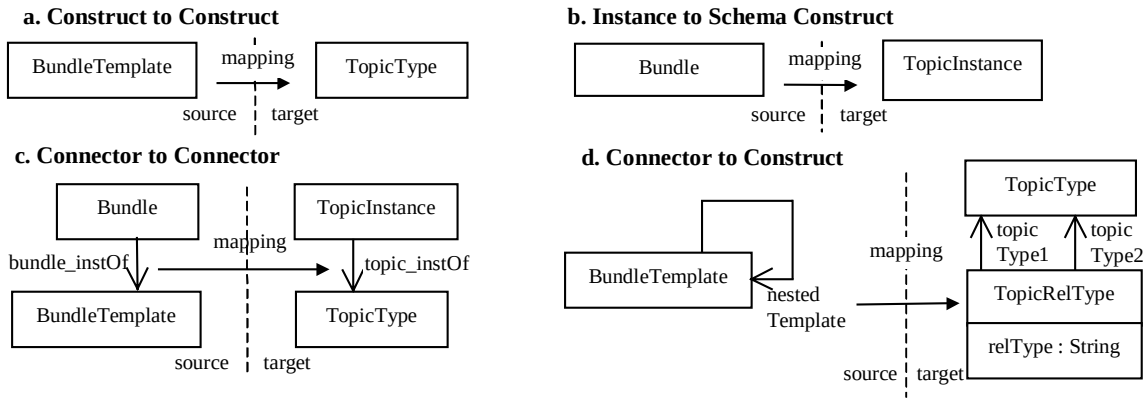


Figure 15. Inter-model mappings from Bundle-Scrap to Structured Maps represented visually.

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

a. Construct to Construct
S('source', τ('instanceOf', X, 'BundleTemplate'))
⇒
S('target', τ('instanceOf', X, 'TopicType'))

b. Instance to Schema Construct
S('source', τ('instanceOf', X, 'Bundle'))
⇒
S('target', τ('instanceOf', X, 'TopicType'))

c. Connector to Connector
S('source', τ('bundle_instOf', X, Y)),
S('target', τ('instanceOf', X, 'TopicInstance')),
S('target', τ('instanceOf', Y, 'TopicType'))
⇒
S('target', τ('topic_instOf', X, Y))

d. Connector to Construct
S('source', τ('nestedTemplate', P, B)),
S('target', τ('instanceOf', P, 'TopicType')),
S('target', τ('instanceOf', B, 'TopicType')),
X = guid()
⇒
S('target', τ('instanceOf', X, 'TopicRelType')),
S('target', τ('relType', X, 'nested_template')),
S('target', τ('topicType1', X, P)),
S('target', τ('topicType2', X, B))

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

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.

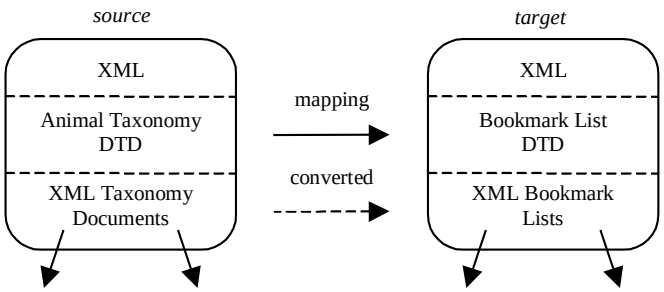


Figure 17. An example of an inter-schema mapping between two XML DTDs.

Figure 18 demonstrates three rules that are used to perform part of the mapping of Figure 17. The first rule takes content that is designated as genus (e.g., "homos") and maps it to a folder (titled "homos"). Similarly, species content (e.g., "sapiens") is mapped as a nested folder (titled "sapiens") within a genus folder.

Source Schema

τ('instanceOf', 'genus_type', 'ElementType')
τ('elementTypeName', 'genus_type', 'genus')
τ('instanceOf', 'species_type', 'ElementType')
τ('elementTypeName', 'species_type', 'species')
τ('nestedElement', 'genus_type', 'species_type')

Target Schema

τ('instanceOf', 'folder_type', 'ElementType')
τ('elementTypeName', 'folder_type', 'folder')
τ('nestedElement', 'folder_type', 'folder_type')

Mapping Rules

S('source', τ('elemInstOf', X, 'genus_type'))
⇒
S('target', τ ('elemInstOf', X, folder_type))

S('source', τ('elemInstOf', X, 'species_type'))
⇒
S('target', τ('elemInstOf', X, 'folder_type'))

S('source', τ('nestedElement', X, Y))
⇒
S('target', τ('nestedElement', X, Y))

Figure 18. Example of schema-to-schema mapping between two XML DTDs.

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.

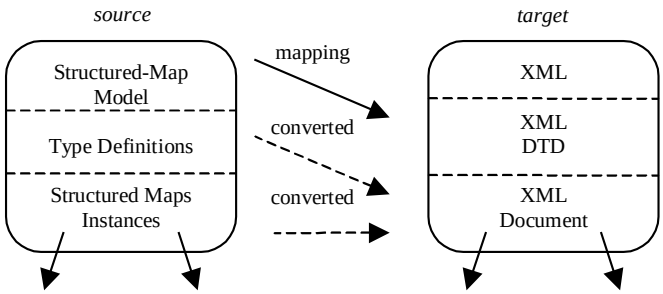


Figure 19. Example of a model to schema mapping.

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". Then, for a particular Topic Instance (e.g., "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.

Source Model

S('source', τ('instanceOf', 'TopicInstance', 'Construct'))
S('source', τ('instanceOf', 'title', 'Connector'))
S('source', τ('instanceOf', 't1', 'TopicInstance'))
S('source', τ('title', 't1', 'painter'))

Target Model

S('target', τ('instanceOf', 'ElementType', 'Construct'))
S('target', τ('instanceOf', 'AttributeType', 'Construct'))
S('target', τ('instanceOf', 'elementTypeName', 'Connector'))
S('target', τ('instanceOf', 'attTypeOf', 'Connector'))
S('target', τ('instanceOf', 'attTypeName', 'Connector'))
S('target', τ('instanceOf', 'Element', 'Construct))
S('target', τ('instanceOf', 'Attribute', 'Construct'))
S('target', τ('instanceOf', 'elemInstOf', 'Conformance'))
S('target', τ('instanceOf', 'attInstOf', 'Conformance'))
S('target', τ('instanceOf', 'tagName', 'Connector'))
S('target', τ('instanceOf', 'attName', 'Connector'))
S('target', τ('instanceOf', 'attValue', 'Connector'))
S('target', τ('instanceOf', 'attributeOf', 'Connector'))

Mapping Rule:

⇒
S('target', τ('instanceOf', 'topic_inst', 'ElementType')),
S('target', τ('elemTypeName', 'topic_inst', 'topic')),
S('target', τ('instanceOf,' 'topicInst_att', 'AttributeType')),
S('target', τ('attTypeName', 'topicInst_att', 'name')),
S('target', τ('attTypeOf', 'topic_inst', 'topicInst_att'))

S('source', τ('instanceOf', X, 'TopicInstance')),
S('source', τ('title', X, Y))
⇒
S('target', τ('instanceOf', X, 'Element')),
S('target', τ('tagName', X, 'topic')),
S('target', τ('elemInstOf', X, 'topic_inst')),
Z = guid(),
S('target', τ('instanceOf', Z, 'Attribute')),
S('target', τ('attName', Z, 'name')),
S('target', τ('attInstOf', Z, 'topicInst_Att')),
S('target', τ('attributeOf', X, Z)),
S('target', τ('attValue', Z, Y))

Figure 20. A model-to-schema mapping rule between the Structured Maps and XML.

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. Work in Progress

In this section, we describe additional uses of the metamodel and the triple representation scheme as well as some of the challenges we are working on.

6.1. The Application-Specific Data Manipulation Interface (DMI)

The goal of the Data Manipulation Interface (DMI) is to make it simple for applications to store their data using our representation scheme, without managing or creating triples. The DMI thus provides the bridge between the application data and the underlying triple representation. The developer of an application is able to create and manipulate data of interest, naturally, 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 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 implementions. Our strategy is to present the application with a set of classes that represent the data model constructs 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.

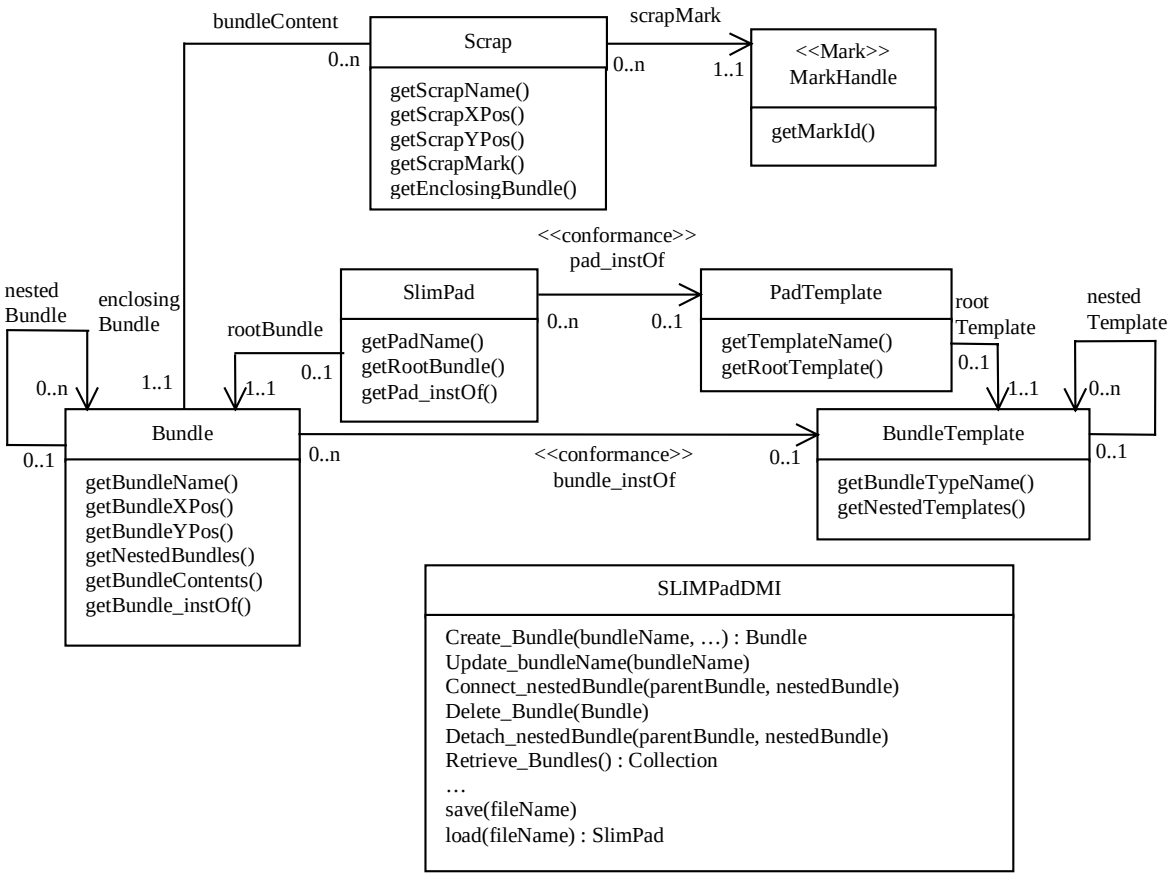


Figure 21. The DMI and corresponding objects for SLIMPad.

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 or other file system to store the underlying triples.

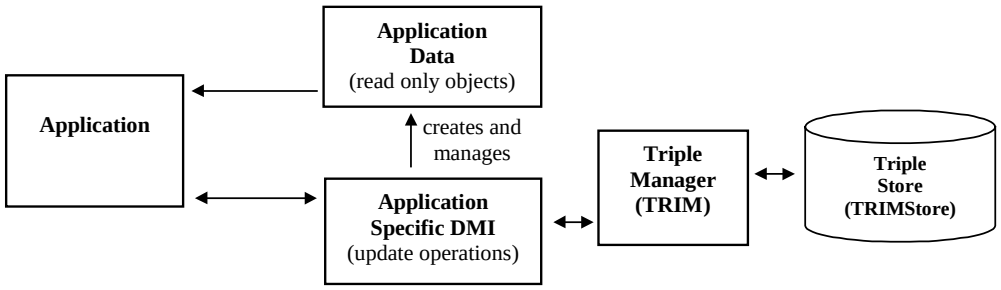


Figure 22. The DMI within the information management architecture.

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 can 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, that Bundle is either the root Bundle of a SLIMPad or it isn't (as specified by the domain multiplicity 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.

Another challenge is to provide programmers the ability to 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). Finally, we are investigating ways to optimize DMI operations.

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 also exploit fixed data models when we transform information. For a particular mapping between triple sets, if we know that 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 makes it easier to write mapping rules (since the resulting predicates are more descriptive) and provides performance benefits. Performance of the mappings is improved because fewer joins (or bindings, in Prolog) are required – we no longer have to join all the triples that represent attributes of a Bundle together with the triple that represents a Bundle, for example. Another benefit of using the specific predicates is that we partition the facts (i.e., the triples) into several predicates (or tables if we use SQL) rather than having a single predicate or table for all triples.

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 it as a fixed model to generate a DMI or to pre-generate predicates. Two challenges in this approach are deciding the correct names to use for the new predicates that represent the constructs and connectors as well as determining which items are 'lifted'. For correct naming, one approach is to use construct identi-

fier names, assuming they were chosen with this purpose in mind. For example, we might have the triple '(instanceOf, "genus", ElementType),' where "genus" is the ElementType instance identifier, used to create a construct with the name 'genus' (and a corresponding mapping rule predicate could be generated with the same name). Alternatively, we could use some other specification that determines the method used to calculate the name (i.e., a key specification). In the case of XML, we might have a rule that tells us to use the elementTypeName property as the construct name for ElementType instances.

For determining the schema items to be lifted to model elements, we can calculate the appropriate items through conformance connectors – we are working on the algorithm for this.

6.3. Abstract Addressing

As was mentioned earlier, our work on the metamodel is part of a larger research program investigating superimposed information. SLIMPad is our first superimposed application and it has been built using our generic architecture for superimposed applications. Here we briefly summarize the architecture (for additional details see [16]) and describe how we can extend the mark management component using the generic representation and the metamodel. Figure 23 shows our current approach to creating and managing marks to various information sources, including XML, HTML, Adobe PDF, Microsoft Word, PowerPoint, and Excel documents.

Each base information source is associated with a particular base application, and each application has a wrapper with a thin interface that allows marks to be created and resolved at various granularities (e.g., whole documents, pages within documents, sections within pages, and so on). When the user creates a mark, the application wrapper generates a mark appropriate for the application type (e.g., there may be many different XML applications, each of which produce marks for the same XML Mark type). The Mark Manager, however, hands the superimposed application a generic mark object, not a typed mark, which encapsulates the base application specific details. This architectural decision allows new mark types to be added and used without having to notify or update the superimposed applications. Each mark type contains the appropriate information required to address information in the base information source.

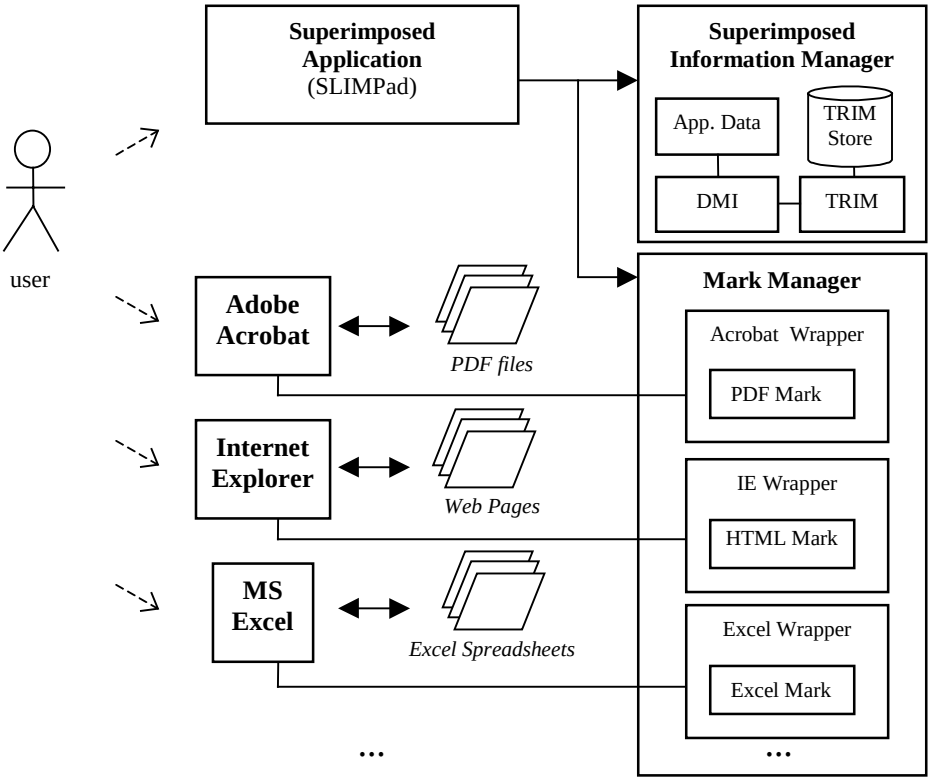


Figure 23. Mark Management as implemented in SLIMPad.

While the current mark architecture is flexible for creating new mark types, we see an opportunity to enhance it by developing a generic approach to addressing base information. 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. The question is should we mark information elements in (one of) the manifestations? Could we devise a scheme for representing canonical marks that could easily accompany the various manifestations?

We believe that we can define an abstract addressing scheme based on our metamodel. And that such an addressing scheme can be used to specify canonical marks. Since the metamodel defines the basic structures used to describe various information, we are looking for basic addressing mechanisms that are appropriate for the structures. 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 con-

struct 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 are currently investigating addressing mechanisms that could be used to support abstract addressing. For example, we are looking at XPath [11] (which itself exploits a derivative of the XML model) to support hierarchical addressing. Another abstract addressing method involves key-indexed collections, in which a key is used to access individual elements. Path expressions are also a form of abstract addressing, which are commonly used in OODB systems. 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 use abstract addressing over these sources.
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 or to a canonical hierarchical addressing scheme, we can create canonical marks (i.e., a single address for multiple document manifestations).

6.4. Extending the Metamodel

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 only 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 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 OODB 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 6.3). We have chosen to introduce sets, bags, and lists as the collection types.

Figure 24 shows three possible approaches for supporting collections in the metamodel.

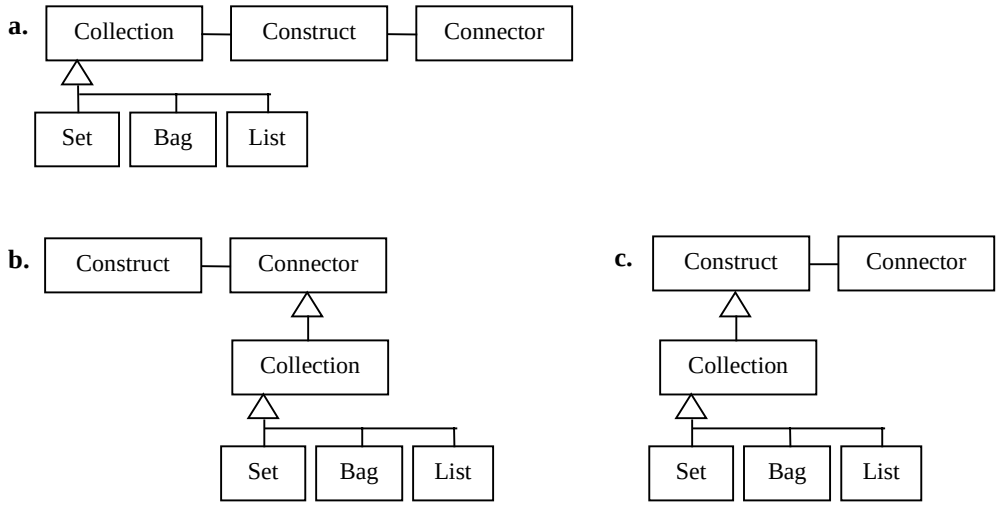


Figure 24. Three approaches to adding collections to the metamodel.

The first approach, shown as Figure 24(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(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 24(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 25 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.

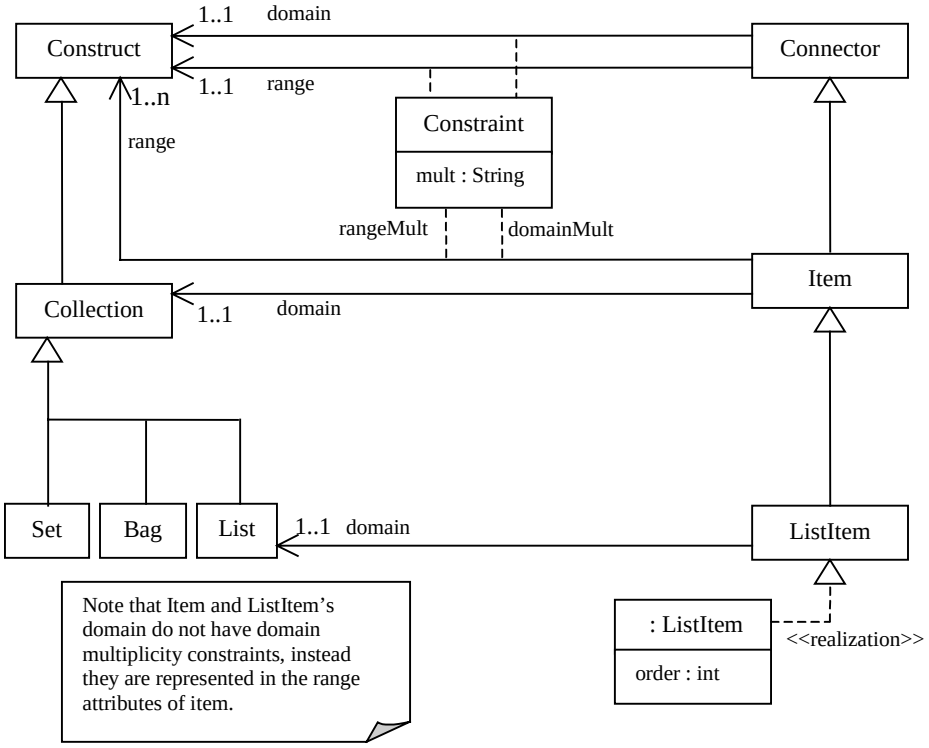


Figure 25. Representing a collection as a type of Construct.

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 im-

ply, 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.

6.5. Visual Mappings

Included in our plans is the development of a high-level, 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, high-level mapping language that is expressive enough to enable complex transformations, while at the same time simple enough to allow for a visual representation (for example, circling or clicking source and target constructs to define mappings at the model-level).

7. Related Work

A number of metamodels have been developed (see Atzeni and Torlone [1-3], Barsalou and Gangopadhyay [4], McBrien and Poulovassilis [20], Cluet et al. [12], and the Meta Object Facility [21]) 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 [1-3, 4, 20, 10, 12, 13] and object-oriented models [21-22] 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 al-

lows 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 [1-3] 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.

8. Conclusion

We have presented a framework for representing and transforming various model-based representation schemes. The framework consists of a metamodel for describing data models, which uniquely allows for the explicit specification of the relationship between schema and instance, and provides a single repre-

sentation 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

We would like to thank David Maier for his helpful discussions, contributions, and comments. We also thank Longxing Deng and Mathew Weaver for their contributions including the development of the SLIMPad architecture.

References

[1] Paolo Atzeni and Riccardo Torlone. A metamodel approach for the management of multiple models and the translation of schemes. In *Information Systems* 18(6):349-362, September 1993.

[2] Paolo Atzeni and Riccardo Torlone. MDM: a multiple-data-model tool for the management of heterogeneous database schemes. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[3] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. In *Proccedings of International Conference on Extending Database Technology (EDBT)*, Lecture Notes in Computer Science Volume 1057, Avignon, France, March 1996.

[4] Thierry Barsalou and Dipayan Gangopadhyay. M(DM): An open framework for interoperation of multimodel multidatabase systems. In *Proceedings of International Conference on Data Engineering (ICDE)*, Tempe, Arizona, February 1992.

[5] Philip A. Bernstein and Thomas Bergstraesser. Meta-data support for data transformations using Microsoft Repository. In *IEEE Data Engineering Bulletin* 22(1):9-14, March 1999.

[6] Philip A. Bernstein, Brian Harry, Paul Sanders, David Shutt, and Jason Zander. The Microsoft Repository. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Athens, Greece, August 1997.

[7] Michel Biezunski, Martin Bryan, and Steve Newcomb, editors. ISO/IEC 13250, Topic Maps Standard, http://www.ornl.gov/sgml/sc34/document/0058.htm.

[8] Tim Bray, Jean Paoli, and C.M. Sperger-McQueen, editors. Extensible Markup Language (XML) 1.0, W3C Recommendation 10-February-1998, http://www.w3.org/TR/REC-xml.

[9] Dan Brickley and R.V. Guha, editors. Resource Description Framework Schema (RDFS), W3C Proposed Recommendation 03 March 1999.

[10] Vassilis Christophides, Sophi Cluet, and Jerome Simeon. On wrapping query languages and efficient XML integration. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Dallas, Texas, May 2000.

[11] James Clark and Steve DeRose, editors. XML Path Language Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath.

[12] Sophie Cluet, Claude Delobel, Jerome Simeon, and Katarzyna Smaga. Your mediators need data conversion! In *Proceedings of ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[13] Susan B. Davidson and Anthony Kosky. WOL: A language for database transformations and constraints. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Birmingham, U.K., April 1997.

[14] Lois Delcambre, David Maier, Radhika Reddy, and Lougie Anderson. Structured maps: modeling explicit semantics over a universe of information. In *International Journal on Digital Libraries* 1(1):20-35, 1997.

[15] Lois Delcambre and David Maier. Models for superimposed information. In *Advances in Conceptual Modeling (ER)*, Lecture Notes in Computer Science Volume 1727, Paris, France, November 1999.

[16] Lois Delcambre, David Maier, Shawn Bowers, Longxing Deng, Mathew Weaver, Paul Gorman, Joan Ash, Mary Lavelle, and Jason A. Lyman. Bundles in captivity: an application of superimposed information. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Hiedelberg, Germany, April 2001. to appear.

[17] PN Gorman, et al. Bundles in the wild: Tools for managing information to solve problems and maintain situation awareness. In *Library Trends* 49(2):266-289, Fall 2000.

[18] Ora Lassila and Ralph R. Swick, editors. Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February 1999, http://www.w3.org/TR/REC-rdf-syntax.

[19] David Maier and Lois Declambre. Superimposed information for the Internet. In *Proceedings of ACM SIGMOD Workshop on The Web and Databases (WebDB)*, pages 1-9, Philadelphia, Pennsylvania, June 1999.

[20] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, Lecture Notes in Computer Science Volume 1626, Heidelberg, Germany, June 1999.

[21] Object Management Group. Meta Object Facility (MOF) Specification. OMB Document ad/99-09-04. http://www.omg.org/cgi-bin/doc?ad/99-09-04.

[22] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.