A Generic Approach for Representing Model-Based Superimposed Information

Shawn Bowers
{shawn@cse.ogi.edu}
Oregon Graduate Institute
Research Proficiency Exam
May 1, 2000

Abstract

Superimposed information is used to supplement existing sources of information, while maintaining connections back to the original data. It can be used for organizing, highlighting, annotating, connecting, and reusing selected elements within information sources. With the growing amount of digital information available today, especially on the web, more and more applications are being developed that use superimposed information. However, generic support to store, create, update, and transform superimposed information does not exist. Superimposed applications must implement their own representation and storage capabilities, which also makes superimposed information difficult to share across applications.

We seek to represent superimposed information for a wide variety of model-based applications (e.g., XML and CASE tools, Topic Map navigators, SQL, and even spreadsheet editors) that use superimposed information. Our solution leverages a metamodel that is used to define multiple superimposed-information models. We employ a flat representation scheme, using the Resource Description Framework, to represent models (defined by our metamodel), schemas, and instances of superimposed information. As a result, we are able to provide model-based applications with a uniform, generic representation of superimposed information. While the generic representation is stored explicitly, it need not be visible to an application, which we support through a variety of application programming interfaces. The APIs provide ease of use and rapid superimposed-application development. The representation also enables mappings between superimposed layers, which allows applications to dynamically share superimposed information.

1. Introduction

Imagine trying to use the Web to plan a vacation to British Columbia. As you come across relevant information, you will use some strategy to keep track of what you've found. For example, you might write down URLs and other information (such as reviews and fares) on a piece of paper or even bookmark sites of interest. And although your vacation will eventually get planned, you will inevitably waste time by forgetting or loosing information, visiting the same site multiple times (thinking you haven't already seen it), and by having to make expense calculations by hand.

But what if you could use tools you already have to help you with your task? For example, instead of manually keeping track of what you've found, you drag and drop selected information from Web sites into a scratchpad tool that allows you to group information and attach annotations such as "Hotels in Vancouver," "Restaurants in Victoria," "Day trips from Nanaimo," etc. 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 rate, but wonder if the hotel has 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 will automatically lift 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. After using the calendar to select the best itinerary, you choose to integrate the plan into your bookmark list so that you can later access the information directly from your Web browser. You also decide to lift all of the address and phone numbers relevant to the trip into your address book (e.g., to load into your Palm Pilot) and integrate the payment information you've collected into your checkbook software to develop a budget for your vacation.

Each tool in this example leverages *superimposed information* [12, 15] to help manage and organize data. Superimposed information is a layer (the *superimposed layer*) of data placed over existing information sources (the *base layer*) to select, combine, highlight, supplement, and provide additional links to selected information elements within the underlying sources. Superimposed information has the following characteristics.

- It can contain additional information about elements in the base layer (e.g., by organizing, annotating, or highlighting elements).
- It can have varying degrees of structure.
- It does not modify the base layer.
- It can contain *marks* that connect the superimposed layer to elements within the existing information sources (see Figure 1).

Additionally, each tool in our example is model-based. That is, each tool exploits a particular model that prescribes a certain organization to superimposed information. For example, scratchpads manage groups of links and annotations, spreadsheets model rows and columns of cells, and bookmark browsers organize information into folders that contain bookmark entries. Some superimposed models are application-specific, while others are generic across multiple tools. For example, the Resource

Description Framework (RDF) [14], Topic Maps [8], and the Extensible Markup Language (XML) [9], when combined with an addressing mechanism such as XLink [13], are all generic superimposed models. A common feature of these models is their use of schema to type information, although the schema is optional. For example, RDF uses RDF Schema [10], Topic Maps can contain Topic Map definitions, and XML provides Document Type Descriptions (DTDs).

Our goal is to develop a generic representation scheme for model-based superimposed information. Representing superimposed information generically for a wide range of applications enables:

- **Generic Support:** We can employ technology to store, create, manipulate, and retrieve superimposed information represented generically, without forcing each model-based application to represent information using the same model. With generic support, superimposed applications need not implement their own superimposed information management.
- **Application-Specific APIs:** Based on an application developer's desires, we can provide specialized programming interfaces to superimposed information, which provide capabilities for schema and instance creation, update, storage, and retrieval based on the application's model. By using an application-specific API, superimposed applications aren't required to use the generic representation directly, are simpler to develop, and still have the benefits of generic representation.
- **Information Sharing:** By having a common representation of superimposed information, applications can easily exchange their data. The idea of sharing information is central in our illustration of superimposed tools. For example, information originally represented in the scratchpad is used by the spreadsheet, which is then used by the checkbook program.

The remainder of this paper describes our approach to representing superimposed information and how the representation is used to enable information sharing and generic support with application-specific APIs. Section 2 describes the superimposed information metamodel, which is essential to defining our representation scheme. The metamodel serves as a formalism from which superimposed models can be defined. Section 3 describes our approach to representing superimposed information. We employ a flat representation scheme, using RDF and RDF Schema, which allows us to easily store models, schemas, and data (termed *instances*) of a superimposed layer. The representation scheme can be used across multiple models and represents the entire superimposed layer in a uniform way. Additionally, we provide a visual method, based on the Unified Modeling Language (UML) [18], to help model engineers describe superimposed models. In Section 4, we use basic production rules over the representation scheme to formally specify and implement superimposed-layer mappings from one representation to another. Section 5 discusses our approach to providing generic support for two superimposed applications. We introduce TRIM Store, which is used to provide creation, update, storage, and retrieval capabilities using

our representation scheme. We also describe our use of application-specific APIs and describe a mechanism to automatically generate interfaces based on an application's superimposed model. Our approach to providing generic support with application-specific capabilities is unobtrusive and can simplify the development of superimposed applications. We discuss related work in Section 6 and summarize our conclusions and future work in Section 7.

2. The Superimposed-Information Metamodel

Figure 1 shows the basic arrangement of the superimposed and base layers. The base layer consists of information sources that are referenced by marks in the superimposed layer. Marks reference information at an appropriate granularity within a source, as permitted by an addressing scheme. Example information-source types include HTML pages, XML elements within XML documents, PDF files, and Microsoft PowerPoint presentations. The superimposed layer consists of three levels: model, schema, and instances. For a given superimposed layer, the model specifies the constructs that can be used to structure data. We use the term *data* to refer to both schema-level and instance-level data.

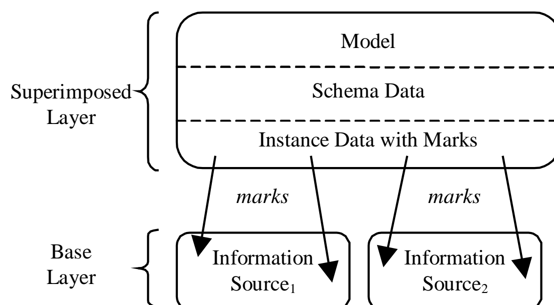


Figure 1. The superimposed and base layers.

To describe multiple superimposed models, we define a level of abstraction above the model, called a metamodel, which is used to define the model of interest to the superimposed application. Figure 2 shows the role of the metamodel for three superimposed layers representing RDF, Topic Maps, and XML.

The metamodel describes the basic abstractions 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 relationship between instance-level data and schema-level data. Each level of the architecture can be viewed as an instantiation of the levels above it. More specifically, model constructs (i.e., schema and instance constructs) are particular instantiations of the abstractions defined by the metamodel, schema-level data are particular instantiations of the model's

schema constructs, and instance-level data are instantiations of the model's instance constructs and can conform to the schema-level data.

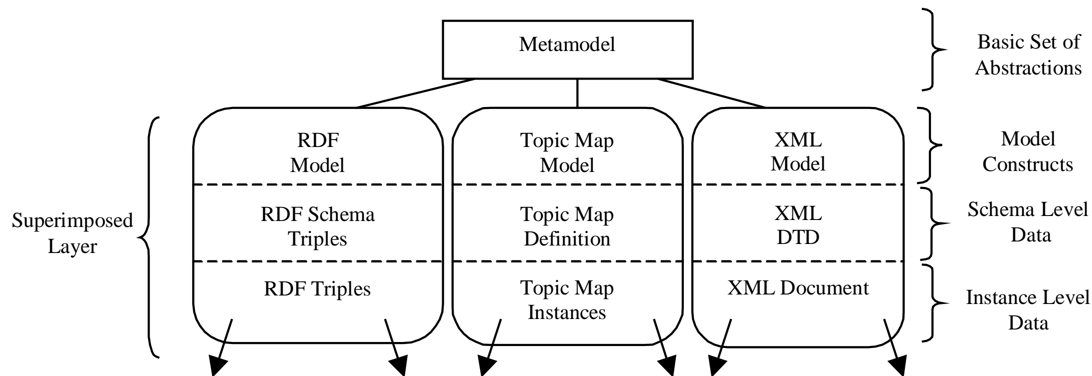


Figure 2. The RDF, Topic Map, and XML models within a superimposed layer.

Figure 3 shows an example of model, schema, and instance data for XML. Notice that we use an "open" DTD, which means that additional elements and attributes not defined in the DTD can be included in XML documents.

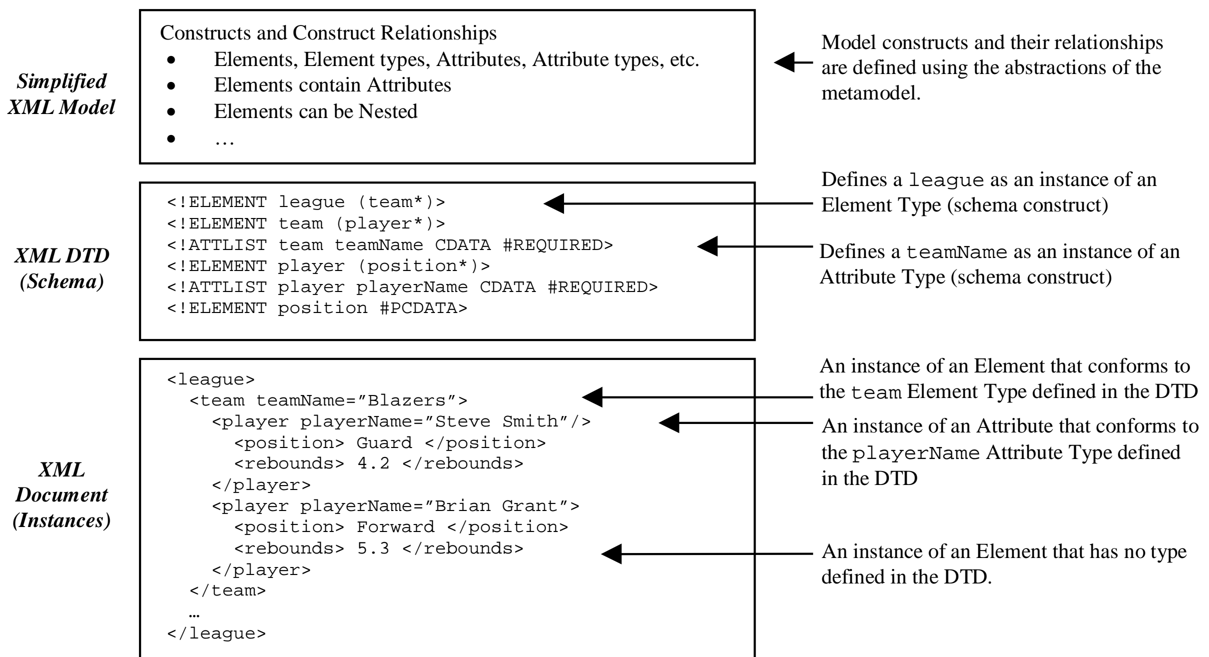


Figure 3. An example of each of the three levels (model, schema, and instance) for XML.

The definition of our superimposed-information metamodel is similar to other metamodel approaches in the object [17, 18] and database communities [1-4, 5, 16] for describing structural models. Examples of structural models include the entity-relationship model, the relational model, the hierarchical model, and the various semantic data models including the UML.

However, superimposed information models have a number of unique features that are additionally supported by our metamodel. One such feature is the inclusion of marks, which are references from the superimposed layer to elements within the base layer and may appear at various places within a superimposed model [12].

Another distinguishing characteristic of superimposed models is their relaxation of schema-first definitions that require schema to be created prior to instances. For example, in a relational database, a table (which represents schema information) must be created before any of its rows. Many superimposed models do not enforce schema-first definitions, including the Topic Map model, in which it is possible to first create a topic and then later attach its type.

Not only do superimposed models relax schema-first definitions, they also allow for data that is not explicitly typed. For example in a Topic Map, a topic can exist without being associated to any type. Finally, multiple levels of schema-instance relationships can occur within some superimposed models. 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.

Figure 4 shows the abstractions of the superimposed-information metamodel. The two basic elements are the *construct*, which represents a basic structural definition within a model, and *structural connector*, which represents a relationship between constructs.

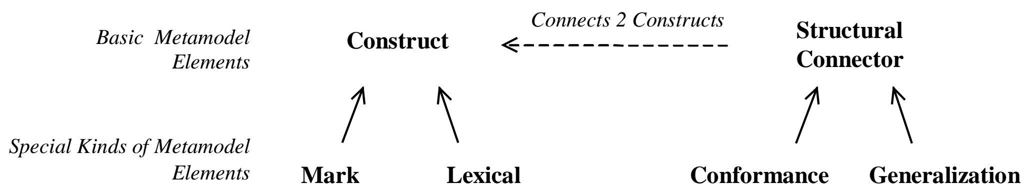


Figure 4. The elements defined by the superimposed-information metamodel.

There are two special constructs: mark and lexical. A *mark* describes a model construct whose instances represent connection-points to the base layer. A *lexical* describes a model construct whose instances contain primitive-value types (e.g., string). There are also two special structural connectors: *conformance*, which specifies a schema-instance relationship, and *generalization*, which specifies inheritance 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).				
Metamodel Elements	*Constructs*	*Lexicals*	*Connectors*	*Conformance Connectors*
XML Model	Element Type Attribute Type Element Attribute Primitive Content	Primitive Content Type	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 and 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 Prim. Content Type*

Element constructs in XML form a hierarchy and are 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 may also wish 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, which allows models to have multiple levels of schema and instance definition. For example, in the Topic Map model we could 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.

3. Representing Superimposed Models, Schemas, and Instances

Superimposed models defined by the metamodel are stored using a representation scheme based on RDF. Although model engineers can specify superimposed models directly using the RDF

representation, we believe it is more convenient to define models visually. Therefore, we also define a visual representation of models using a subset of the UML.

3.1 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 [14].

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 consists of 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 define XML as an interchange format to exchange RDF and RDF Schema triples.

3.2 The Metamodel Defined using RDF

Figure 5 shows the definition of the superimposed-information 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, generalization, and conformance as properties, each with a construct as domain and range. Generalization and conformance are both sub-properties of connector.

Domain and range multiplicity constraints are defined as RDF *ConstraintProperties*. A Constraint Property is a higher-order property that can be used to add constraints (beyond domain and range) to a Property. To create model constructs and connectors as well as schema and instance data, we define a property called *instanceOf*. Notice that the domain and range of instanceOf is not defined, which means that it can be used over any resource and contain any value. We also create two lexical constructs: string and number. If desired, a model engineer can specify a new primitive type (e.g., PCDATA) using a

similar definition, providing new primitive types are primitive (i.e., data of the type must have some form of string representation like a float, integer, or date).

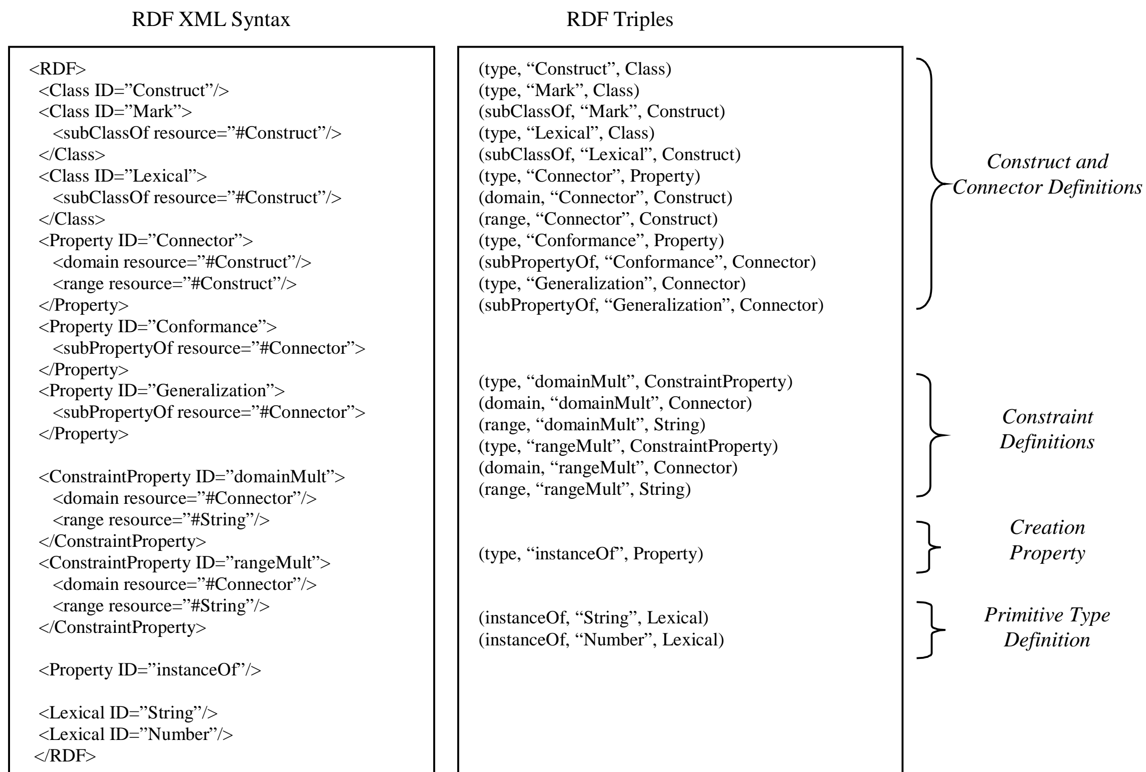


Figure 5. The superimposed-information metamodel represented in RDF XML and as RDF Triples.

3.3 The RDF and Visual Model Representation

Figure 6 depicts both the RDF and visual representations of the XML model of Section 2 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 primitive type as a range (e.g., string or number) and implicitly have a domain multiplicity of zero-or-one and a range multiplicity of one. UML stereotypes are used to distinguish marks and lexicals from constructs, and conformance connectors from regular connectors. Additionally, UML generalization relationships require a name (which is not a general requirement of UML).

The schema-level constructs of Figure 6 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. The conformance connectors only represent a structural connection between the constructs and do not fully define the meaning of the

conformance (e.g., if an Element conforms to an Element Type, then each nested Element should conform to the appropriate nested Element Type).

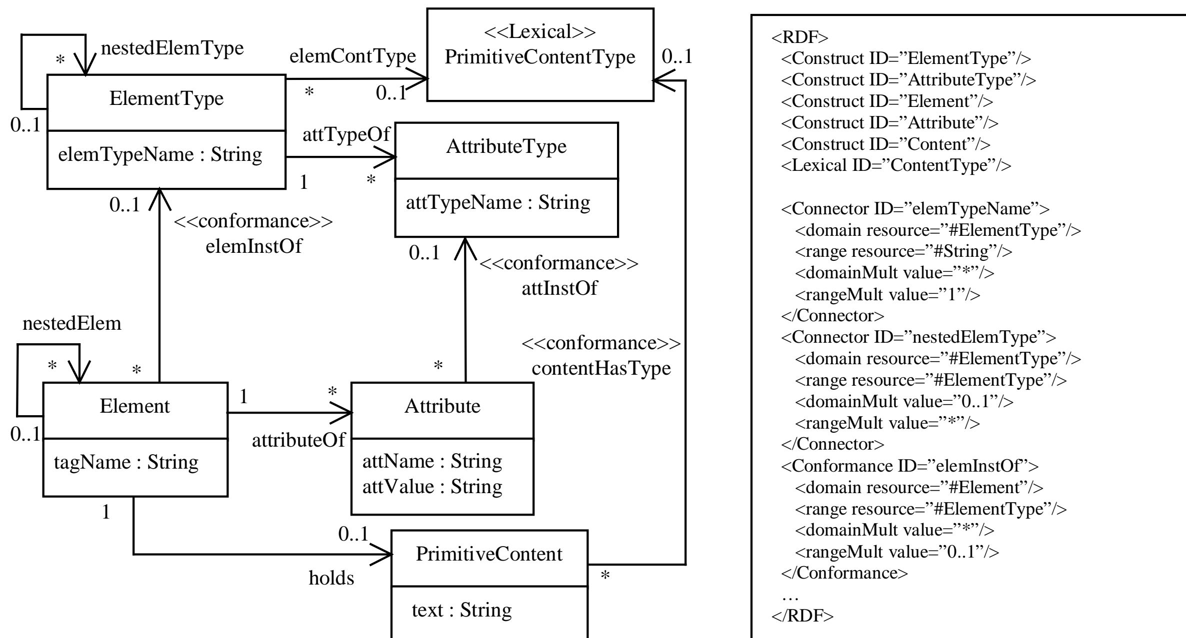
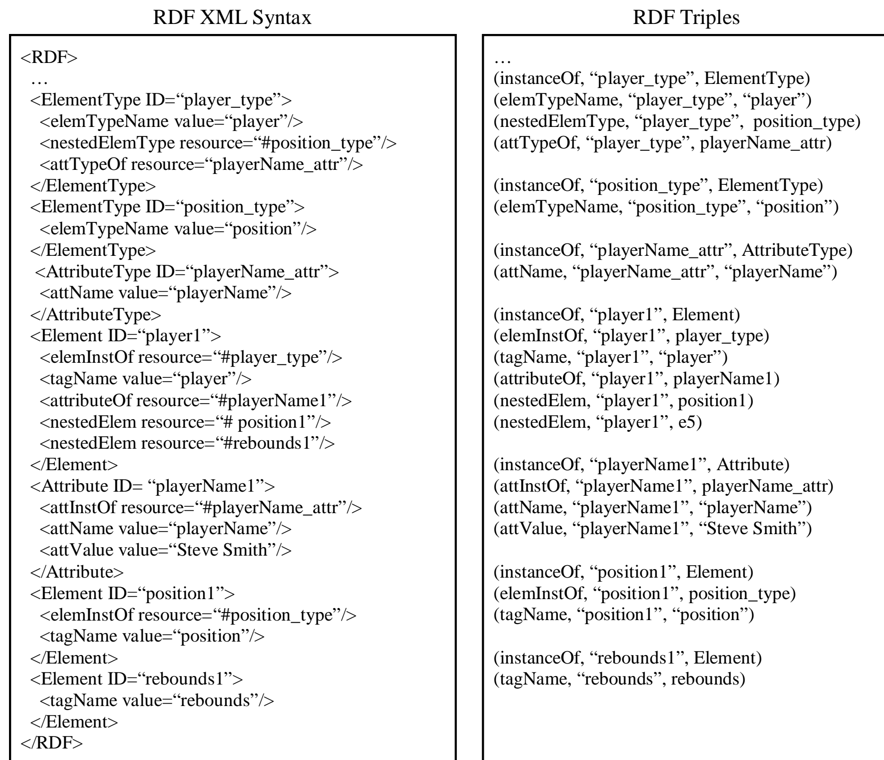


Figure 6. The XML model represented using UML with a sample of the RDF representation.



RDF XML Syntax

```
<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"/>
    <tagName value="player"/>
    <attributeOf resource="#playerName1"/>
    <nestedElem resource="# position1"/>
    <nestedElem resource="#rebounds1"/>
  </Element>
  <Attribute ID= "playerName1">
    <attInstOf resource="#playerName_attr"/>
    <attName value="playerName"/>
    <attValue value="Steve Smith"/>
  </Attribute>
  <Element ID="position1">
    <elemInstOf resource="#position_type"/>
    <tagName value="position"/>
  </Element>
  <Element ID="rebounds1">
    <tagName value="rebounds"/>
  </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)
(attName, "playerName_attr", "playerName")

(instanceOf, "player1", Element)
(elemInstOf, "player1", player_type)
(tagName, "player1", "player")
(attributeOf, "player1", playerName1)
(nestedElem, "player1", position1)
(nestedElem, "player1", e5)

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

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

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

Figure 7. Schema and instance data of the XML model.

Figure 7 shows example schema- and instance-level data that represent the XML document excerpt of Figure 4. (See Appendix A for the full representation.) Notice that RDF provides a uniform representation of model, schema, and instance by allowing the entire superimposed layer to be completely described with RDF triples.

Figures 8 and 9 describe the Structured-Map and Structured-Bundle 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 [11], 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 schema-first definitions. 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.

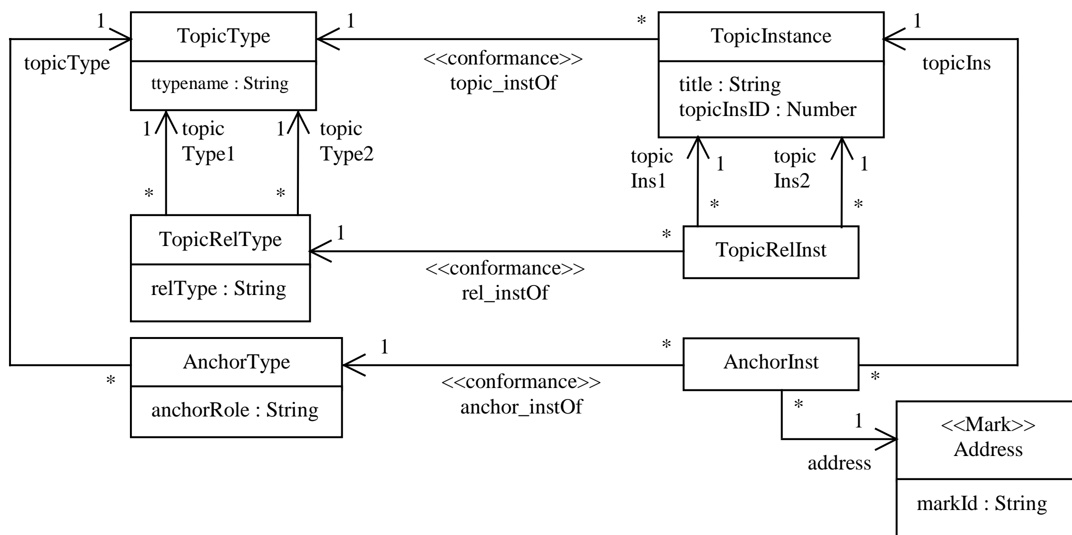


Figure 8. CARTE's Structured Map model.

The Structured-Bundle model is used by SLIMPad (*Superimposed-Layer Information Manager scratchpad*), which is a scratchpad application being built for the Traces project [19]. The goal is to develop specialized applications similar to SLIMPad that enable medical experts to organize important

facts and issues excerpted from a base layer of digital medical documents. Currently, 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.

In SLIMPad, users can create and use templates as schema-level data. By instantiating a template, the user is provided with a set of default bundles organized hierarchically within the scratch pad. However, as shown by the multiplicity constraints, bundles can be created without an associated template. Within the medical domain, we see a number of templates being used for specialized tasks (e.g., templates for making drug-interaction decisions for individual patients differ from templates for managing the state of an Intensive Care Unit). While SLIMPad and CARTE handle multiple schemas, we envision a number of schema-specific versions of SLIMPad designed around a single template (schema), in which the schema-level data is designed by an application developer.

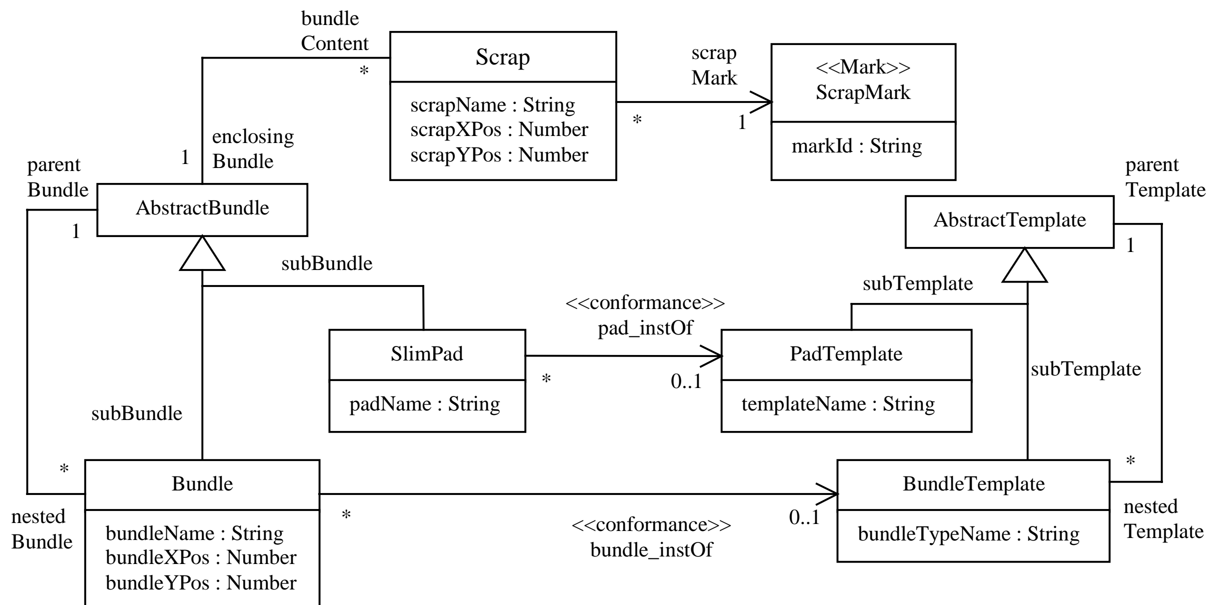


Figure 9. SLIMPad's Structured Bundles Model.

Note that in Figure 9 we use the UML subclass symbol to denote generalization. However, we require that the relationship be named, since we are creating a new connector instance in the model. Additionally, arrows are not included in the association between a Scrap and an AbstractBundle since the

association represents two independent connectors. We use AbstractTemplate and AbstractBundle as constructs to stop the recursion created by the nested Template and nested Bundle connectors, respectively.

4. Superimposed-Layer Mappings

In this section, we describe an approach to information sharing between superimposed applications that use different models and schemas. In general, there will be various ways to map between superimposed layers, depending on the user's desires and the applications of interest. Therefore, instead of trying to automatically generate mappings between superimposed layers, we provide a technique for mappings to be specified manually, and perform the conversions as needed.

4.1 Mapping Rules

Superimposed-layer mappings are specified using production rules. The rules are defined over triples of the RDF representation scheme for superimposed information. Since a triple is 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. It is important to note that we do not require mappings between superimposed layers 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 superimposed layer *L*. For example, *S*('xml', τ('instanceOf', 'Element', 'Construct')) would be true if there were an "Element" construct defined in a superimposed layer named "xml." 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)) $\Rightarrow$ *S*('target', τ('owner', X, Y)) would add a triple τ('owner', X, Y) to the superimposed layer named "target" for every triple that matched τ('creator', X, Y) in the superimposed layer named "source." Therefore, if τ('creator', 'index.html', 'Ora Lassilla') is a triple in the superimposed layer named

"source," then the triple τ('owner', 'index.html', 'Ora lassilla') will be added to the superimposed layer named "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	A set (i.e., database) of triple predicates τ for superimposed layer L.
S	A predicate of the form $S(L, \tau)$ that is true if $\tau \in L$.
M	A mapping that consists of a set of mapping rules.
m	A mapping rule with the form: $T \Rightarrow T'$, where T, T' are sets of S predicates. The rule can be read as follows: if the left hand side matches (i.e., each $S \in T$ is true) then for each $S(L, \tau) \in T'$ add τ to L.

Table 3 describes the functions that are used to perform mappings. The conversion function applies a set of mapping rules to a source and target superimposed layer. Conversion generates a new superimposed layer that contains the generated triples of the mappings. In our current implementation, Prolog performs the conversion function.

Table 3. Functions used to provide mappings.

Function	Definition
Conversion : $M \times L_s \times L_t \rightarrow L_r$	Conversion takes a mapping M and applies the mapping rules of M to a source layer L_s and a target layer L_t, and returns a new layer L_r. Conversion is a basic rule-based algorithm that computes the fixed-point of applying mapping rules to the source and target superimposed layers.
ExtractModel : $L \rightarrow L'$	$L' = L_1 \cup L_2$ where: L_1 = {t | t∈L and t=τ('instanceOf',X,Y} where X='Construct', 'Mark', 'Lexical', 'Connector', 'Conformance', or 'Generalization'} L_2 = {u | u,t∈L and t=τ('instanceOf',X,Y) where Y='Connector', 'Conformance', or 'Generalization' and u=τ(P,X,Z)}.
ExtractSchema : $L \rightarrow L'$	$L' = L_1 \cup L_2$ where: L_1 = {v | t,u,v∈L and t=τ('instanceOf',P,'Conformance'} and u=τ(P,Y,Z) and v=τ('instanceOf',Z,X)} L_2 = {u | u ∈L and t1,t2∈ L_1 and t1=τ('instanceOf',S1,X) and t2=τ('instanceOf',S2,Y) and u=τ(P,S1,S2)}.
guid $\rightarrow x$	A 0-ary Skolem function that returns a unique identifier x.

The extract model function is used to extract model information from a superimposed layer. It can also be applied to the result of a conversion. For example, in an inter-model mapping, we want the resulting superimposed layer to contain the model of the target layer. By using the extract model function, we can add the appropriate triples of the target layer to the superimposed layer returned by the conversion function.

Similarly, the extract schema function gathers schema information from the target layer. 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 this approach works for cases where

there is only one level of schema and instance.) To add target model and schema information to the result of an inter-schema mapping we would use the extract model and schema functions as follows: conversion(M, L_s, L_t) ∪ extractModel(L_t) ∪ extractSchema(L_t), where L_t is the target layer, L_s the source layer, and the result of the clause is a new superimposed layer.

4.2 Inter-Model, Inter-Schema, and Model-to-Schema Mappings

Figure 11 illustrates three types of mappings. Each example shows information from a source layer being mapped to a target layer to convert data from the source layer into data that conforms to the target layer. Although we focus on conversion, it is also possible to perform integration between superimposed layers. Integration goes a step further by combining the source and target data. The mapping rules can be used to provide integration at both the schema and instance levels.

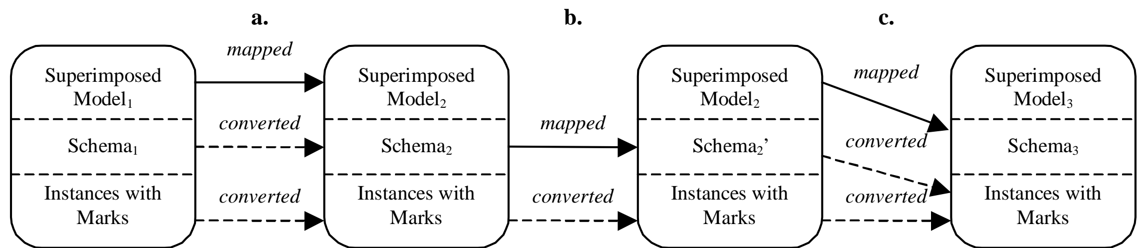


Figure 10. Three superimposed layer mappings: (a) inter-model, (b) inter-schema, and (c) model-to-schema.

Figure 10(a) is an inter-model mapping in which the schema- and instance-level data of the source superimposed-layer are converted to valid schema- and instance-level data of the target superimposed-layer. Figure 11 shows an inter-model mapping between the Structured-Bundle model and the Structured-Map model. The goal of the mapping is to allow SLIMPad data to be used with CARTE.

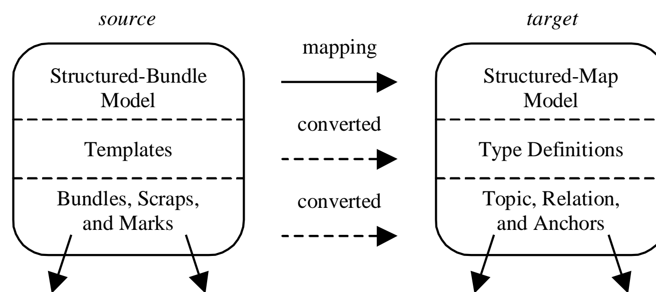


Figure 11. An inter-model mapping between the Structured Bundles and Structured Maps.

Figure 12 illustrates four mapping rules between the Structured-Bundle model (shown as the source) and the Structured-Map model (shown as the target) using the UML model representation. The first mapping rule, Figure 12(a), specifies a mapping between the Bundle Template schema-construct and the Topic Type schema-construct. That is, all Bundle Templates in the source superimposed-layer will be

converted to Topic Types in the resulting superimposed layer. Figure 12(b) is a mapping between Bundles and Topic Instances, which are both at the instance-level. Figure 12(c) is a mapping between two conformance connectors: bundle_instOf and topic_instOf. The mapping states that each bundle_instOf relationship in the source-layer should be converted to a topic_instOf relationship in the target-layer. Finally, the mapping in Figure 12(d) shows the nestedTemplate connector being mapped to a Topic Relationship Type. As the data is converted, a new Topic Relationship Type will be created with the relType attribute set to the string "nested_template," the domain of the nestedTemplate connector assigned to the range of topicType1, and the range of the NestedTemplate connector assigned to the range of topicType2.

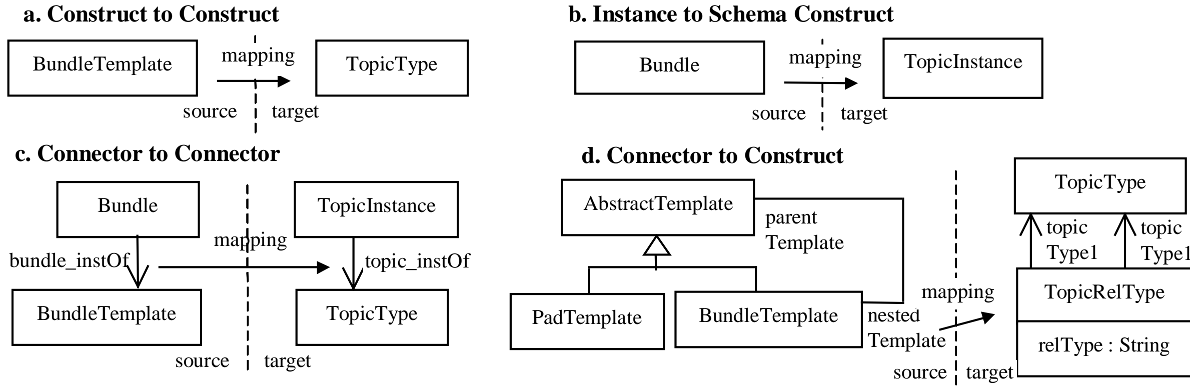


Figure 12. Inter-model mappings from Structured Bundles to Structured Maps represented visually.

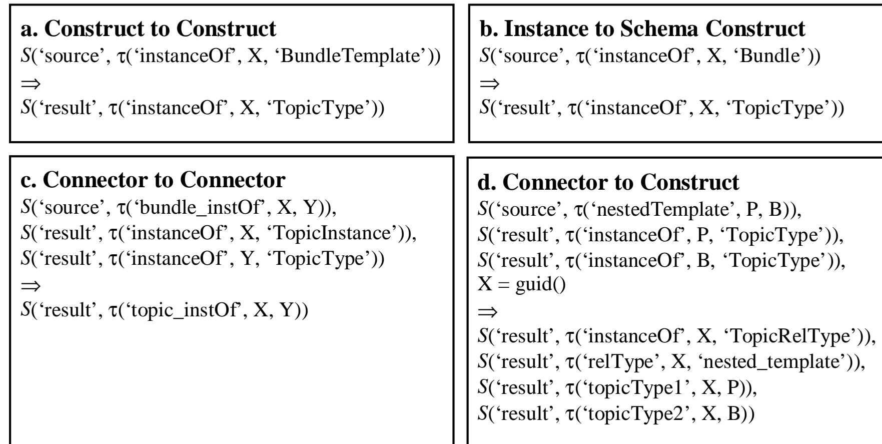


Figure 13. Inter-model mappings from Structured Bundles to Structured Maps.

Figure 13 shows the mapping rules that specify the mappings of Figure 12. We use the constant 'source' to represent the Structured Bundles superimposed layer, the constant 'target' to represent the Structured Maps superimposed layer, and the constant 'result' to represent the new superimposed layer

that is created from the mappings. (See Appendix B for an example of the full mapping between the models.)

Figure 10(b) is an inter-schema mapping in which 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 14 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 is to reuse existing tools for browsing bookmark lists on taxonomies.

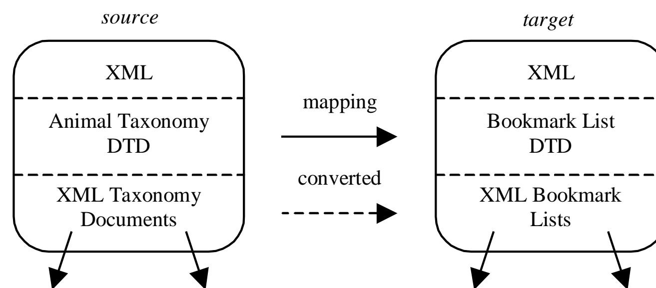


Figure 14. An example of a schema to schema mapping between two DTDs.

Figure 15 demonstrates three rules that are used to perform part of the mapping of Figure 14. 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.

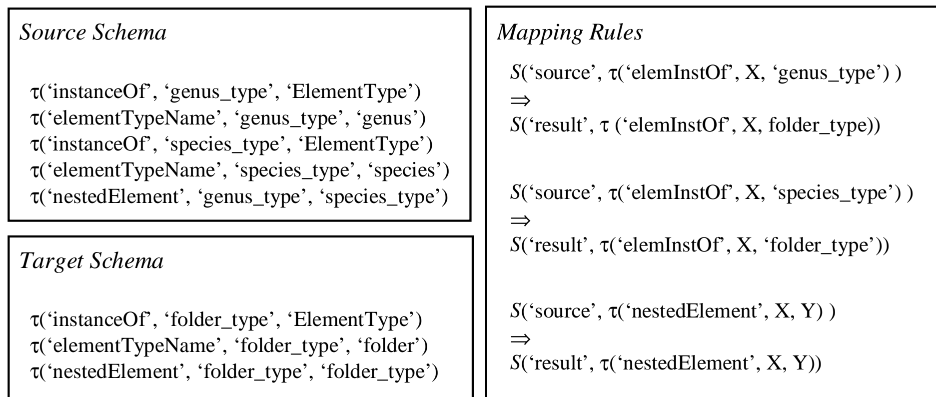


Figure 15. Example of schema-to-schema mapping between two XML model schemas.

The last mapping of Figure 10, shown as Figure 10(c), is called a model-to-schema mapping. Here, the model of the source layer is mapped to schema-level data in the target layer, which allows the schema- and instance-level data of the source to be converted to valid instance-level data in the target. Figure 16 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 doing this mapping is to use XML as the interchange format for Topic Maps.

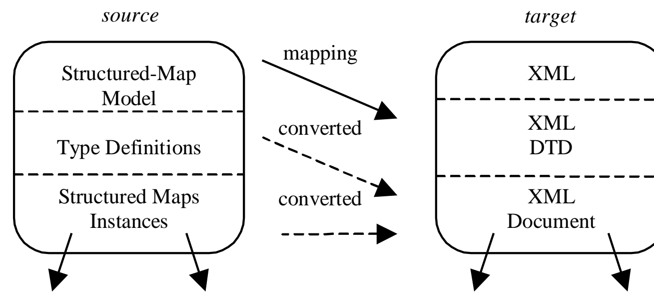


Figure 16. Example of a model to schema mapping.

To specify the mapping, we could 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 would result in the XML tag <topic name="painter"/>. Figure 17 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.

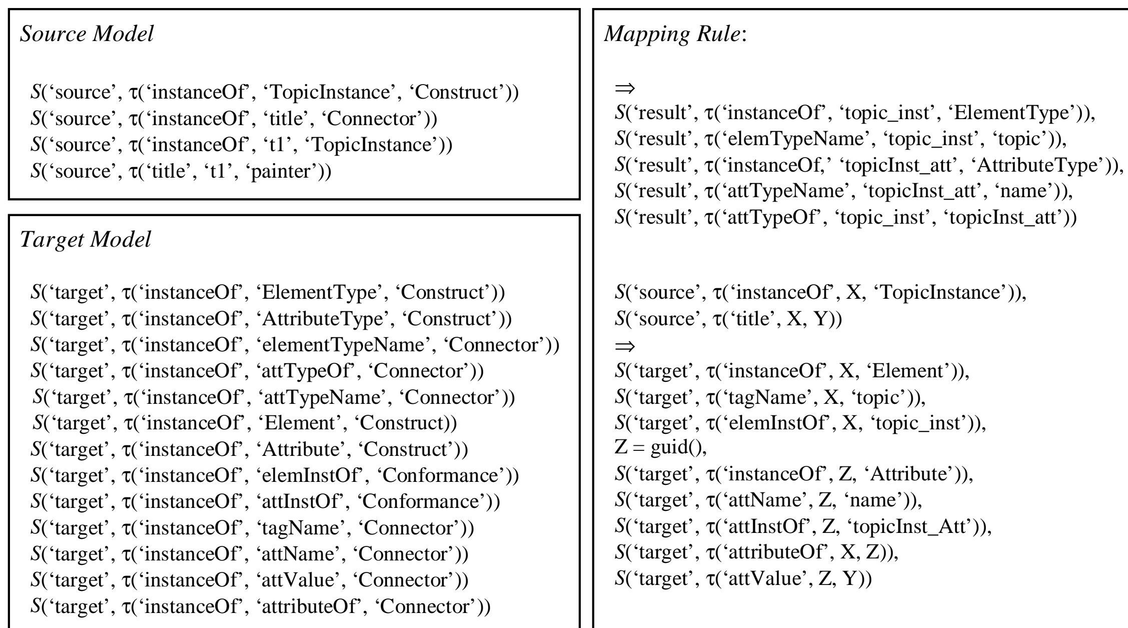


Figure 17. A model-to-schema mapping rule between the Structured Maps and XML.

5. Application Support for the use of Superimposed Information

This section describes our approach to providing applications with support for the management of superimposed information. Our goal is to provide generic support with application-specific programming

interfaces. Generic support leverages our representation scheme to create, store, update, and retrieve RDF triples across models. Application-specific APIs provide applications with specialized capabilities to manage superimposed information based on the application's model (or schema). By providing specialized interfaces, applications are able to create, store, manipulate, and retrieve superimposed information without directly using our representation scheme, which is automatically managed for the application.

Figure 18(a) depicts the general strategy we use to provide support for the use of superimposed information in SLIMPad and CARTE. On the left of Figure 18(a) sits the superimposed application, which can directly access Application Data. The Application Data represents superimposed information presented to the application in its desired format. For example, SLIMPad (shown in Figure 18(b)), which is written in Visual Basic, accesses superimposed information through ActiveX Objects. On the other hand, CARTE (shown in Figure 18(c)) uses relational tables to access superimposed information. The information represented by Application Data is redundant since it is also represented as RDF triples. However, by directly using the Application Data, applications can access superimposed information in a natural way.

The Superimposed Layer Information Manager (SLIM) is used to create and manipulate Application Data. SLIM provides the management capabilities required by the superimposed application and maintains consistency between RDF-triples and Application Data. For CARTE, SQL is used as the SLIM component to manipulate relational tables that represent Application Data. SLIMPad uses an ActiveX component as SLIM (called the SLIMPad API) to create and update Application Data. Together, SLIM and the Application Data represent the application-specific API.

Lying between SLIM and the RDF representation is the Storage Manager, which provides generic support for the management of RDF triples for SLIM. For example, CARTE uses Microsoft Access as a Storage Manager, which represents the RDF triples as a relational table. Similarly, SLIMPad uses the TRIM Store (Triple Manager), which is a Java component, to provide general management of RDF triples that are represented as Java objects. TRIM Store provides creation, update, storage, and simple retrieval capabilities over the RDF triples.

An advantage to the general architecture in Figure 18(a) is that it is unobtrusive. For example, CARTE was originally developed to store its superimposed information directly as relational tables. We were able to use the same relational tables, but convert them into views, using SQL, over the RDF-triple

table. (Note that for CARTE, SQL along with the relational tables act as an application-specific API). CARTE is still able to access the original tables as if nothing changed, and so its application logic is unaffected by adding superimposed information support, while gaining the benefits of information sharing.

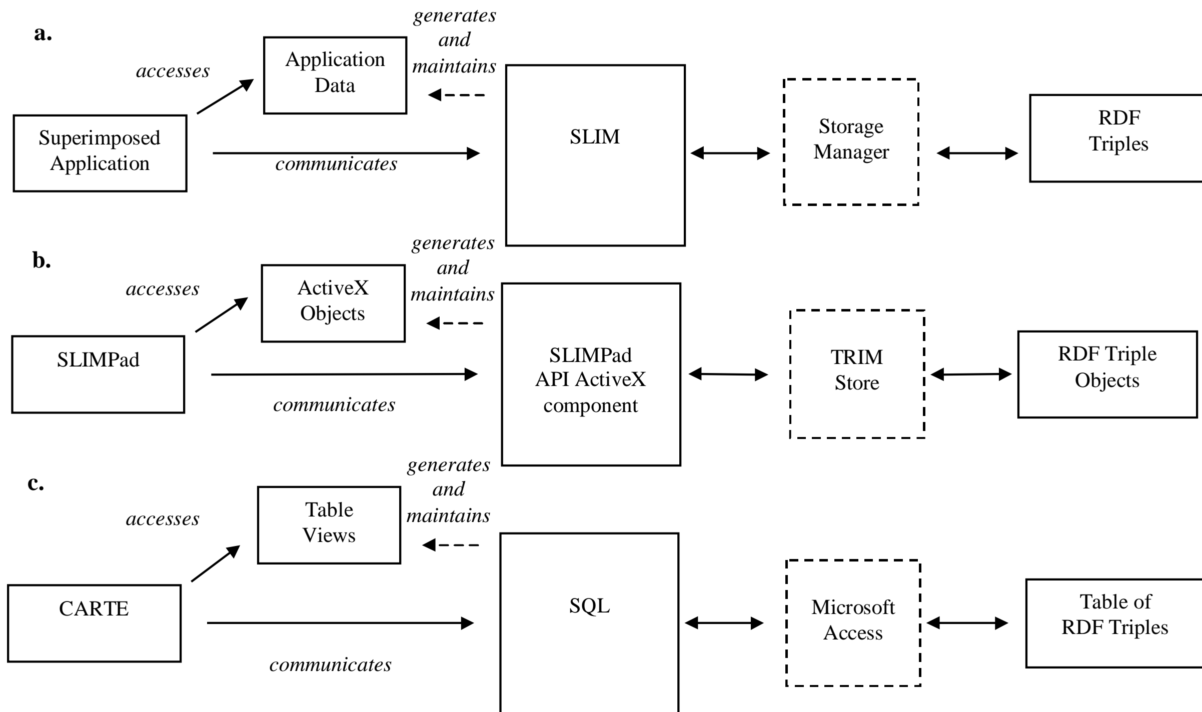


Figure 18. Superimposed application support represented: (a) generically, (b) with SLIMPad, and (c) with CARTE.

The SLIMPad application, when compared to CARTE, requires a greater degree of support for the use of superimposed information. To create an application-specific API for SLIMPad, we developed a prototype compiler [20] that automatically generates both the Application Data component and the SLIM component based on an application's superimposed model (represented in the RDF XML format). The compiler implicitly uses a model-to-schema mapping between the target model (e.g., the Structured-Bundle model) and the target language, which is currently Java. We map constructs in the given model to Java classes, which are immutable to the application, and model connectors to private attributes of the class representing the domain of the connector. Additionally, a method is provided for read-access to the connector's range. For example, using the Structured-Bundle model, we generate a Bundle class with an attribute called "bundleName" and a method called "getBundleName" that returns a string. The SLIM component generated by the compiler allows for schema and instance creation and update based on the construct and connectors of the model. For example, the SLIMPad API provides a method called

"create_bundle" to create a Bundle object with a string as a parameter for the bundleName. When creation and update is performed through the generated SLIM API, the TRIM Store is notified of the change, which maintains the underlying superimposed-information stored as RDF.

Providing applications with specialized APIs is a flexible approach for application developers managing superimposed information. Currently, we use only a fixed model-to-schema mapping between a superimposed model and a target language. However, with our approach it is possible to employ other types of mappings. For example, an application developer could create a schema for the Structured-Bundle model with SLIMPad, store the schema-level data, and then use it to generate a new specialized API based on an inter-model mapping to the target language. The API would enable specialized instance creation and update capabilities for the designer's schema. Additionally, APIs could be generated for various target languages, based on the needs of the application designer. For example, instead of generating the instance creation and update API for Java, the compiler could produce SQL to create the appropriate relational tables representing the designer's schema data.

Providing an application-specific API along with generic support for SLIMPad, we were able to focus on the basic functionality and user interface of SLIMPad. In general, we believe that providing generic support and application-specific APIs not only supports the use of superimposed information, but also simplifies the development of superimposed applications while allowing applications to share information.

6. Comparison of Related Work

In this section, we discuss how our work compares to existing approaches for providing support for multiple models. A number of metamodels have been developed (see Atzeni and Torlone [1-4], Barsalou and Gangopadhyay [5], McBrien and Poulovassilis [16], and the Meta Object Facility [17]) with the 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 both database data models [1-4, 5, 16] and object-oriented models [17-18] 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 superimposed 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 major 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. McBrien and Poulovassilis use the Hypergraph Data Model (HDM) to store schemas defined in diverse models. Their primary goal is to perform inter-model transformations, which are specified using first-order logic expressions, to map between semantically equivalent schemas. They also 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). However, 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 using the HDM. Atzeni and Torlone employ procedural inter-model mapping specifications. Similar to our approach, their specifications can be used to implement the conversions. However, they require complete mappings between models, whereas we allow partial mappings to allow for a wider range of cases, and they do not provide support for other types of mappings (e.g., inter-schema or model-to-schema mappings).

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 UML 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 [6-7] 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.

Both the MOF and the HDM provide mechanisms that support schema and instance creation for models described using their metamodels. However, both approaches provide a single, fixed mechanism

to provide schema and instance creation. For example, HDM provides standard method definitions that create schema-level data based on the model. However, neither approach provides application-specific APIs to allow specialized capabilities or generic support for managing information (both approaches assume a database is used for management).

7. Conclusions and Future Work

The following list summarizes the contributions of our work.

1. **The Superimposed-Information Metamodel:** We define a unique metamodel for describing multiple superimposed models. The metamodel allows for the explicit specification of the relationship between schema and instance and allows marks to be placed anywhere within a model.
2. **A Generic Representation Scheme for Superimposed Information:** Leveraging the superimposed information metamodel, we define a single representation scheme, based on RDF, for superimposed information. Our representation scheme uniformly represents model, schema, and instance data and can be used generically by diverse superimposed applications.
3. **Visual Superimposed-Model Definition:** We provide model engineers the ability to define superimposed models visually by using a subset of the UML. We believe that being able to define and view models visually is a benefit, and in addition, model engineers can leverage a number of existing tools that support UML.
4. **Information Sharing:** We enable information sharing in two ways. First, using a common representation format for superimposed information makes it possible for applications to dynamically share data. Second, applications can share superimposed information that differs in schema or model through mappings.
5. **Superimposed-Layer Mappings:** We define a formal method, based on production rules, to uniformly specify superimposed-layer mappings. We allow partial mappings as well as mappings between multiple levels of superimposed information.
6. **Generic Support with Application-Specific APIs:** Based on our representation scheme for superimposed information, we were able to develop TRIM Store, which provides generic support for the creation, update, storage, and retrieval of superimposed information across models. Additionally, we do not require applications to directly use the representation scheme by leveraging specialized APIs based on the application's superimposed model or schema. We were successful in integrating superimposed information management into both CARTE and SLIMPad, while providing the benefits of information sharing.

Our plans for future work are centered on further developing application support for the use of superimposed information. We wish to look into specialization and automatic generation approaches to optimize management facilities for a fixed model or fixed model-schema combination. Additionally, we would like to provide applications with the ability to tailor the management facilities they require through a meta DML (data-manipulation language), which would be used by the API compiler.

To leverage the visual representation of models, we are interested in defining parameterized mapping-rules that can be used to specify inter-model mappings visually. We also plan to look at ways to enforce constraints defined within a model, such as cardinality and conformance as well as structural constraints between schema and instances. One possible approach is to allow model engineers and application designers to use a first-order logic language to both specify and check complex constraints between schema- and instance-level data.

Finally, we are working on developing a general architecture to support mark management for superimposed applications, developing marks to new information sources, further developing SLIMPad, and creating new superimposed applications.

8. References

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

[2] Paolo Atzeni and Riccardo Torlone. MDM: a multiple-data-model tool for the management of heterogeneous database schemes. *ACM SIGMOD Conference*, Tucson, Arizona, May 13-15, 1997.

[3] Paolo Atzeni and Riccardo Torlone. Management of multiple models in an extensible database design tool. *5th International Conference on Extending Database Technology EDBT '95*, Lecture Notes in Computer Science Volume 1057, Avignon, France, March 25-29, 1996.

[4] P. Atzeni and R. Torlone, Schema translation between heterogeneous data models in a lattice framework. *6th IFIP TC-2 Working Conference on Database Semantics (DS-6)*, Atlanta, Georgia, May 30-June 2, 1995.

[5] Thierry Barsalou and Dipayan Gangopadhyay. M(DM): an open framework for interoperation of multimodel multidatabase systems. *Eighth International Conference on Data Engineering ICDE'92*, Tempe, Arizona, February 3-7, 1992.

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

[7] Philip A. Bernstein, Brian Harry, Paul Sanders, David Shutt, and Jason Zander. The Microsoft Repository. *Proceedings of the 23rd International Conference on Very Large Databases VLDB '97*, Athens, Greece, August 25-27, 1997.

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

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

[10] Dan Brickley and R.V. Guha, editors. Resource Description Framework Schema (RDFS), W3C Proposed Recommendation 03 March 1999, URL:http://www.w3.org/TR/PR-rdf-schema/.

[11] 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), pages 20-35, 1997.

[12] Lois Delcambre and David Maier. Models for superimposed information. *Advances in Conceptual Modeling ER '99*, Lecture Notes in Computer Science Volume 1727, pages 264-280, Paris, France, November 15-18, 1999.

[13] Steve DeRose, Eve Maler, David Orchard, and Ben Trafford, editors. XML Linking Language (XLINK), W3C Working Draft 21-February-2000, URL: http://www.w3.org/TR/2000/WD-xlink-20000221.
[14] Ora Lassila and Ralph R. Swick, editors. Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February 1999, URL:http://www.w3.org/TR/REC-rdf-syntax.
[15] David Maier and Lois Declambre. Superimposed information for the Internet. *ACM SIGMOD Workshop on The Web and Databases WebDB'99*, pages 1-9, Philadelphia, Pennsylvania, June 3-4, 1999.
[16] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. *11th International Conference on Advanced Information Systems Engineering CAiSE'99*, Lecture Notes in Computer Science Volume 1626, pages 333-348, Heidelberg, Germany, June 14-18, 1999.
[17] Object Management Group. *Meta Object Facility (MOF) Specification*. OMB Document ad/99-09-04. URL:http://www.omg.org/cgi-bin/doc?ad/99-09-04.
[18] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
[19] Tracking footprints through an information space: leveraging the document selections of expert problem solvers. NSF Grant IIS-9817492. Digital Libraries Phase 2. Paul Gorman, Lois Delcambre, and David Maier.
[20] Mat Weaver. SlimML. Term project report for CSE 511. Oregon Graduate Institute of Science and Technology, March 2000.

Appendix A

The XML model represented as RDF Triples, including sample schema- and instance-data.

(instanceOf, “ElementType”, Construct)
(instanceOf, “Element”, Construct)
(instanceOf, “AttributeType”, Construct)
(instanceOf, “Attribute”, Construct)
(instanceOf, “PrimitiveContent”, Construct)
(instanceOf, “PrimitiveContentType”, Lexical)

(instanceOf, “nestedElemType”, Connector)
(domain, “nestedElemType”, ElementType)
(range, “nestedElemType”, ElementType)
(domainMult, “nestedElemType”, “0..1”)
(rangeMult, “nestedElemType”, “*”)

(instanceOf, “elemTypeName”, Connector)
(domain, “elemTypeName”, ElementType)
(range, “elemTypeName”, String)
(domainMult, “elemTypeName”, “*”)
(rangeMult, “elemTypeName”, “1”)

(instanceOf, “attTypeOf”, Connector)
(domain, “attTypeOf”, ElementType)
(range, “attTypeOf”, AttributeType)
(domainMult, “attTypeOf”, “1”)
(rangeMult, “attTypeOf”, “*”)

(instanceOf, “attTypeName”, Connector)
(domain, “attTypeName”, AttributeType)
(range, “attTypeName”, String)
(domainMult, “attTypeName”, “*”)
(rangeMult, “attTypeName”, “1”)

(instanceOf, “elemInstOf”, Conformance)
(domain, “elemInstOf”, Element)
(range, “elemInstOf”, ElementType)
(domainMult, “elemInstOf”, “*”)
(rangeMult, “elemInstOf”, “1”)

(instanceOf, “nestedElem”, Connector)
(domain, “nestedElem”, Element)
(range, “nestedElem”, Element)
(domainMult, “nestedElem”, “0..1”)
(rangeMult, “nestedElem”, “*”)

(instanceOf, “tagName”, Connector)
(domain, “tagName”, Element)
(range, “tagName”, String)
(domainMult, “tagName”, “*”)
(rangeMult, “tagName”, “1”)

(instanceOf, “attributeOf”, Connector)
(domain, “attributeOf”, Element)
(range, “attributeOf”, Attribute)
(domainMult, “attributeOf”, “1”)
(rangeMult, “attributeOf”, “*”)

(instanceOf, “attName”, Connector)
(domain, “attName”, Attribute)
(range, “attName”, String)
(domainMult, “attName”, “*”)
(rangeMult, “attName”, “1”)

(instanceOf, “attValue”, Connector)
(domain, “attValue”, Attribute)
(range, “attValue”, String)
(domainMult, “attValue”, “*”)
(rangeMult, “attValue”, “1”)

(instanceOf, “attInstOf”, Conformance)
(domain, “attInstOf”, Attribute)
(range, “attInstOf”, AttributeType)
(domainMult, “attInstOf”, “0..1”)
(rangeMult, “attInstOf”, “*”)

(instanceOf, “holds”, Connector)
(domain, “holds”, Element)
(range, “holds”, PrimitiveContent)
(domainMult, “holds”, “1”)
(rangeMult, “holds”, “0..1”)

(instanceOf, “elemContType”, Connector)
(domain, “elemContType”, ElementType)
(range, “elemContType”, PrimitiveContentType)
(domainMult, “elemContType”, “*”)
(rangeMult, “elemContType”, “0..1”)

(instanceOf, “contentHasType”, Conformance)
(domain, “contentHasType”, PrimitiveContent)
(range, “contentHasType”, PrimitiveContentType)
(domainMult, “contentHasType”, “*”)
(rangeMult, “contentHasType”, “0..1”)

-- Schema ---
(instanceOf, “league_type”, ElementType)
(elemTypeName, “league_type”, “league”)
(nestedElemType, “league_type”, team_type)

(instanceOf, “team_type”, ElementType)
(elemTypeName, “team_type”, “team”)
(nestedElemType, “team_type”, player_type)
(attTypeOf, “team_type”, “teamName_attr”)

(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”)
(elemContType, “position_type”, PCDATA)

(instanceOf, “teamName_attr”, AttributeType)
(attName, “teamName_attr”, “teamName”)

(instanceOf, “playerName_attr”, AttributeType)
(attName, “playerName_attr”, “playerName”)

(instanceOf, “PCDATA”, PrimitiveContentType)

-- Instances --
(instanceOf, “league1”, Element)
(elemInstOf, “league1”, league_type)
(tagName, “league1”, “league”)
(nestedElem, “league1”, team1)

\(instanceOf, “team1”, Element)
(elemInstOf, “team1”, team_type)
(tagName, “team1”, “team”)
(attributeOf, “team1”, teamName1)
(nestedElem, “team1”, player1)
(nestedElem, “team1”, player2)

(instanceOf, “player1”, Element)
(elemInstOf, “player1”, player_type)
(tagName, “player1”, “player”)
(nestedElem, “player1”, position1)
(nestedElem, “player1”, rebounds1)
(attributeOf, “player1”, playerName1)

(instanceOf, “position1”, Element)
(elemInstOf, “position1”, position_type)
(tagName, “position1”, “position”)
(holds, “position1”, content1)

(instanceOf, “rebounds1”, Element)
(tagName, “rebounds1”, “rebounds”)
(holds, “position1”, content2)

(instanceOf, “player2”, Element)
(elemInstOf, “player2”, player_type)
(tagName, “player2”, “player”)
(nestedElem, “player2”, position2)
(nestedElem, “player2”, rebounds2)
(attributeOf, “player2”, playerName2)

(instanceOf, “position2”, Element)
(elemInstOf, “position2”, position_type)
(tagName, “position2”, “position”)
(holds, “position2”, content3)

(instanceOf, “rebounds2”, Element)
(tagName, “rebounds2”, “rebounds”)
(holds, “position2”, content4)

(instanceOf, “teamName1”, Attribute)
(attInstOf, “teamName1”, “teamName_attr”)
(attName, “teamName1”, “teamName”)
(attValue, “teamName1”, “Blazers”)

(instanceOf, “playerName1”, Attribute)
(attInstOf, “playerName1”, “playerName_attr”)
(attName, “playerName1”, “playerName”)
(attValue, “playerName1”, “Steve Smith”)

(instanceOf, “playerName2”, Attribute)
(attInstOf, “playerName2”, “playerName_attr”)
(attName, “playerName2”, “playerName”)
(attValue, “playerName2”, “Brian Grant”)

(instanceOf, “content1”, PrimitiveContent)
(text, “content1”, “Guard”)
(contentHasType, “content1”, PCDATA)

(instanceOf, “content2”, PrimitiveContent)
(text,”content2”, “4.2”)

(instanceOf, “content3”, PrimitiveContent)
(text,”content3”, “Forward”)

(instanceOf, “content4”, PrimitiveContent)
(text,”content4”, “5.3”)

Appendix B

The following Prolog mapping rules specify an inter-model mapping between the Structured-Bundle model and the Structured-Map model. Included is sample data and the results of the conversion represented as RDF triples (note that we can represent these using *S* predicates as well).

```
guid(X) :- retract(count_aux(N)),
     X is N+1,
     asserta(count_aux(X)).
:- dynamic(count_aux/1).
count_aux(0).

/* 'source' = Structured-Bundle model
 * 'result' = Structured-Map model
 */
s('source', t('instanceOf', bt1, 'BundleTemplate')).
s('source', t('bundleTypeName', bt1, 'Produce')).
s('source', t('instanceOf', pt1, 'PadTemplate')).
s('source', t('templateName', pt1, 'Grocery List')).
s('source', t('instanceOf', p1, 'SlimPad')).
s('source', t('padName', p1, 'Weekly List')).
s('source', t('instanceOf', b1, 'Bundle')).
s('source', t('bundleName', b1, 'Fresh Stuff')).
s('source', t('instanceOf', s1, 'Scrap')).
s('source', t('scrapName', s1, 'Mushrooms')).
s('source', t('enclosingBundle', s1, b1)).
s('source', t('scrapMark', s1, 'sm1')).
s('source', t('instanceOf', sm1, 'ScrapMark')).
s('source', t('markId', sm1, 'URLMarkModule@000101')).
s('source', t('nestedTemplate', pt1, bt1)).
s('source', t('nestedBundle', p1, b1)).
s('source', t('pad_instOf', p1, pt1)).
s('source', t('bundle_instOf', b1, bt1)).

/* BundleType -> TopicType */
stmts( [s('result', t('instanceOf', X, 'TopicType'))] ) :-
  s('source', t('instanceOf', X, 'BundleTemplate')).

/* PadTemplate -> TopicType */
stmts( [s('result', t('instanceOf', X, 'TopicType'))] ) :-
  s('source', t('instanceOf', X, 'PadTemplate')).

/* bundleTypeName -> ttypename */
stmts( [s('result', t('ttypename', X, Y))] ) :-
  s('source', t('bundleTypeName', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicType'))] ).

/* templateName -> ttypename */
stmts( [s('result', t('ttypename', X, Y))] ) :-
  s('source', t('templateName', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicType'))] ).

/* Bundle -> TopicInstance */
stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ) :-
  s('source', t('instanceOf', X, 'Bundle')).

/* SlimPad -> TopicInstance */
stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ) :-
  s('source', t('instanceOf', X, 'SlimPad')).

/* bundleName -> title */
stmts( [s('result', t('title', X, Y))] ) :-
  s('source', t('bundleName', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ).

/* padName -> title */
```

```
stmts( [s('result', t('title', X, Y))] ) :-
  s('source', t('padName', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ).

/* -> topicInsID */
stmts( [s('result', t('topicInsID', X, Y))] ) :-
  stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ),
  guid(Y).

/* Scrap -> AnchorType */
stmts( [s('result', t('instanceOf', X, 'AnchorType'))] ) :-
  s('source', t('instanceOf', X, 'Scrap')).

/* scrapName -> anchorRole */
stmts( [s('result', t('anchorRole', X, Y))] ) :-
  s('source', t('scrapName', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'AnchorType'))] ).

/* enclosingBundle -> topicType */
stmts( [s('result', t('topicType', X, Y))] ) :-
  s('source', t('enclosingBundle', X, Z)),
  stmts( [s('result', t('instanceOf', X, 'AnchorType'))] ),
  ( s('source', t('bundle_instOf', Z, Y));
    s('source', t('pad_instOf', Z, Y)) ),
  stmts( [s('result', t('instanceOf', Y, 'TopicType'))] ).

/* ScrapMark -> Address */
stmts( [s('result', t('instanceOf', X, 'Address'))] ) :-
  s('source', t('instanceOf', X, 'ScrapMark')).

/* markId -> markId */
stmts( [s('result', t('markId', X, Y))] ) :-
  stmts( [s('result', t('instanceOf', X, 'Address'))] ),
  s('source', t('markId', X, Y)).

/* scrapMark -> address */
stmts( [s('result', t('address', X, Y)),
  s('result', t('instanceOf', X, 'AnchorInst')),
  s('result', t('anchor_instOf', X, Z)),
  s('result', t('topicIns', X, B))] )
    :-
  s('source', t('scrapMark', Z, Y)),
  stmts( [s('result', t('instanceOf', Z, 'AnchorType'))] ),
  s('source', t('enclosingBundle', Z, B)),
  stmts( [s('result', t('instanceOf', B, 'TopicInstance'))] ),
  guid(ID), atom_concat('ai', ID, X).

/* nestedTemplate -> TopicRelType */
stmts( [s('result', t('instanceOf', X, 'TopicRelType')),
  s('result', t('relType', X, 'nested_template')),
  s('result', t('topicType1', X, P)),
  s('result', t('topicType2', X, B))] )
    :-
  s('source', t('nestedTemplate', P, B)),
  stmts( [s('result', t('instanceOf', P, 'TopicType'))] ),
  stmts( [s('result', t('instanceOf', B, 'TopicType'))] ),
  guid(ID), atom_concat('trt', ID, X).

/* nestedBundle -> TopicRelInst */
stmts( [s('result', t('instanceOf', X, 'TopicRelInst')),
  s('result', t('rel_instOf', X, NT)),
  s('result', t('topicInsID1', X, TI1)),
  s('result', t('topicInsID2', X, TI2))] )
    :-
  s('source', t('nestedBundle', TI1, TI2)),
  stmts( [s('result', t('instanceOf', TI1, 'TopicInstance'))]),
  stmts( [s('result', t('instanceOf', TI2, 'TopicInstance'))]),
  s('source', t('bundle_instOf', TI2, BT)),
  ( s('source', t('pad_instOf', TI1, PT));
    s('source', t('bundle_instOf', TI1, PT)) ),
  stmts( [s('result', t('instanceOf', NT, 'TopicRelType')), _,
```

```
      s('result', t('topicType1', NT, PT)),
      s('result', t('topicType2', NT, BT))] ),
  guid(ID), atom_concat('tri', ID, X).

/* bundle_instOf -> topic_instOf */
stmts( [s('result', t( 'topic_instOf', X, Y ))] ) :-
  s('source', t('bundle_instOf', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ),
  stmts( [s('result', t('instanceOf', Y, 'TopicType'))] ).

/* pad_instOf -> topic_instOf */
stmts( [s('result', t( 'topic_instOf', X, Y ))] ) :-
  s('source', t('pad_instOf', X, Y)),
  stmts( [s('result', t('instanceOf', X, 'TopicInstance'))] ),
  stmts( [s('result', t('instanceOf', Y, 'TopicType'))] ).

/******************
 * Pretty Printing
 ******************/
printMapping :- findall(_,print,_).
print :-
  stmts(S), printEach(S).
printEach( [] ).
printEach( [s(_, S)|Ss] ) :- printStatement(S), nl, printEach(Ss).
printStatement(t(S, P, O)) :-
  write(S), write(', '),
  write(P), write(', '),
  write(O).
```

The results of the conversions that were generated from the mapping specification:

```
?- printMapping.
instanceOf, bt1, TopicType
instanceOf, pt1, TopicType
ttypename, bt1, Produce
ttypename, pt1, Grocery List
instanceOf, b1, TopicInstance
instanceOf, p1, TopicInstance
title, b1, Fresh Stuff
title, p1, Weekly List
topicInsID, b1, 1
topicInsID, p1, 2
instanceOf, s1, AnchorType
anchorRole, s1, Mushrooms
topicType, s1, bt1
instanceOf, sm1, Address
markId, sm1, URLMarkModule@000101
address, ai3, sm1
instanceOf, ai3, AnchorInst
anchor_instOf, ai3, s1
topicIns, ai3, b1
instanceOf, trt4, TopicRelType
relType, trt4, nested_template
topicType1, trt4, pt1
topicType2, trt4, bt1
instanceOf, tri6, TopicRelInst
rel_instOf, tri6, trt5
topicInsID1, tri6, p1
topicInsID2, tri6, b1
topic_instOf, b1, bt1
topic_instOf, p1, pt1

Yes
```