A Vision for Management of Complex Models

Philip A. Bernstein
Microsoft Research
Redmond, WA 98052-6399
philbe@microsoft.com

Alon Y. Levy
University of Washington
Seattle, WA, 98195
alon@cs.washington.edu

Rachel A. Pottinger
University of Washington
Seattle, WA, 98195
rap@cs.washington.edu

Technical Report
MSR-TR-2000-53

Abstract

Many problems encountered when building applications of database systems involve the manipulation of models. By “model,” we mean a complex structure that represents a design artifact, such as a relational schema, object-oriented interface, UML model, XML DTD, web-site schema, semantic network, complex document, or software configuration. Many uses of models involve managing changes in models and transformations of data from one model into another. These uses require an explicit representation of “mappings” between models. We propose to make database systems easier to use for these applications by making “model” and “model mapping” first-class objects with special operations that simplify their use. We call this capability model management.

In addition to making the case for model management, our main contribution is a sketch of a proposed data model. The data model consists of formal, object-oriented structures for representing models and model mappings, and of high-level algebraic operations on those structures, such as matching, differencing, merging, function application, selection, inversion and instantiation. We focus on structure and semantics, not implementation.

1 Introduction

Many of the problems encountered when building applications of database systems (DBMSs) involve the manipulation of models. By "model," we mean a complex discrete structure that repesents a design artifact. For example, a model could be an XML DTD, web-site schema, interface definition, relational schema, database transformation script, workflow definition, semantic network, software configuration or complex document. Many uses of models involve managing the change in models and the transformation of data from one model into another. These uses require an explicit representation of *mappings* between models. We believe there is an opportunity to make DBMSs easier to use for these applications by making "model" and "mapping" first-class objects with high-level operations that simplify their use. We call this capability *model management*.

This paper makes two main contributions. First, it argues that general-purpose model management functions are needed to reduce the amount of programming required to manipulate models. We expect the development of these functions will follow the usual evolution from components, to middleware, to integrated system. That is, they could be implemented initially as a library of reusable components, on top of a DBMS. If successful in this form, they could later be tied together in a "model management system," implemented first as an extensible framework separate from the DBMS, and eventually as an integrated part of the DBMS.

The second contribution is a proposed data model that captures model management functions. The data model consists of formal structures for representing models and mappings between models, and of algebraic operations on those structures. We present the overall shape of the data model in just enough detail to justify our thesis that general-purpose model management is a worthwhile and achievable goal. We expect the details of the data model will take years to work out.

Today's model management applications store models in either a file system or some form of DBMS, such as a relational DBMS, object-oriened (OO) DBMS, or repository system [Ber98]. A DBMS-based solution is an advance over file systems because, among other things, it allows applications to replace some object-at-a-time navigation by set-oriented queries. Beyond that, repository systems help applications avoid navigational code associated with specialized relationship semantics and versioning. However, despite these functionality advances in DBMSs, model management applications still include a lot of complex code for navigating graph-like structures. The main technical goal of model management is to greatly reduce the amount of this navigational code.

Producing, understanding, tuning, and maintaining navigational code is a serious drag on programmer productivity, making model management applications expensive to build. Sometimes, it is too hard to write the code at all, and users simply do not get the applications they need. In commercially hot areas, where representations and uses of models change rapidly, some model management applications cannot be developed fast enough to meet the market need, or they're obsolete before they're finished being built.

To solve these problems of development cost and timeliness, we propose raising the level of abstraction beyond what's offered by current DBMSs, by introducing high-level operations on models and model mappings. Examples of these operations are matching, merging, function application, selection, and composition. These operations are not especially novel. There is research literature on each of them, in the DB field and elsewhere, much of which is relevant to the design and implementation of model management functions. We believe they can and should be generalized and integrated, to support a generic model management interface.

To illustrate the pervasiveness and scope of model management, we offer some examples of models and model mappings that arise in various applications:

- mapping an XML schema (or DTD) to a relational schema, to drive transformation of XML elements into rows of relational tables;
- mapping an XML schema of one application to that of another, to guide the exchange of XML instances between the applications;
- mapping a web page wrapper to a DB schema, to guide the translation of queries on the schema to the underlying web sites;
- mapping a web site's content to its page layout, to drive the generation of web pages;
- mapping a business process definition to a workflow description, to generate scripts that execute the workflow application;
- mapping data sources into data warehouse tables to generate programs that transform production data and load it into a data warehouse;
- mapping a query posed against a high-level semantic model into an equivalent query posed against a logical DB schema;

- mapping the DB schema of one software release into the DB schema of the next release, to guide the migration of DBs;
- mapping the object model of one application to the object model of another application, to generate a wrapper that exposes the first application's object model on the second;
- mapping source makefiles into target makefiles, to drive the transformation of make scripts and thereby help port complex applications from one programming environment to another; and
- mapping the components of a complex application to the components of a system where it will be deployed, to drive the generation of installation, upgrade, and de-installation programs.

By building generic functions to create models and mappings and manipulate them as single objects, we can provide a better environment for the above tasks. Said differently, instead of implementing common model manipulation operations again and again for different schema types, we can implement them once as generic operations in a more abstract model management data model.

Although our approach reduces the need for procedural code that manipulates models, it does not eliminate it. In particular, it will still be necessary to write certain custom programs that interpret particular mappings. Such programs are sensitive to the semantics of mapping types and are therefore not easily replaced by generic operators. For example, these programs may involve translating an XML attribute to a table column, a data source column to a data target column, a wrapper parameter to a method parameter, etc. Fortunately, these programs are typically applied to individual objects or small structures. So, while writing these programs is not trivial, it is a localized and incremental activity, and far easier than writing custom code to manipulate entire models. The latter usually requires traversing a large model structure. This more complex and performance-sensitive algorithmic part is addressed by the generic functions we propose.

At least initially, we expect that a model management system would primarily manipulate models whose home is on other platforms (see Figure 1). Such a target platform could be another DBMS, a web site, an XML environment, a programming environment, etc. — any system that manipulates models such as those listed in the above example bullets. The glue between the systems is provided by simple adapters that (1) import or export a model in the model management system from or to a schema in the target platform, or (2) interpret a mapping in the model management system to transform instances of one target model to those of another. One of the many challenges in this area is to find appropriate architectures for coupling these systems.

We do not exclude the possibility that model management is built into the target platforms themselves. However, we think that most of the benefits can be obtained with this loosely coupled architecture, which is clearly less disruptive to target systems.

The leverage of building model management functionality is that it is highly generic and therefore widely applicable. Still, to be competitive with more customized approaches, it must be specializable so it can exploit the semantics of a particular data type (e.g., relational schemas). Making it both generic and easily specializable is another challenge.

The types of model management applications discussed in this paper are usually considered examples of "metadata management," where most of the effort in building the application is in manipulating $descriptions$ of a thing of interest, rather than the thing itself. We intentionally avoid using the term metadata in this paper because it is so overloaded. One person's metadata is another person's data. Are keywords data or metadata? Model management takes a different cut at the problem. It focuses attention on a particular kind of metadata, structure and semantics of descriptive information. We see much leverage to be gained looking at this kind of metadata in isolation.

In describing the data model, we focus on structure and semantics. Although we have little to say at this point about its implementation, we emphasize that we see model management being implemented on top of today's most advanced DB interfaces – OO or object-relational – exploiting such features as recursion and deduction. It is not a replacement for these technologies.

We begin with a scenario in Section 2, which both motivates the need for a coherent system and describes some of the models and operators that such a system should provide. In Sections 3, 4, and 5 respectively, we discuss the formal structure of models, mappings, and operations. Finally, in Section 6 we discuss how previous work relates to a model management system and how it can tackle some of the many open questions.

2 A Motivating Scenario

We begin by describing a scenario in which a model management system would play a central role and consider-

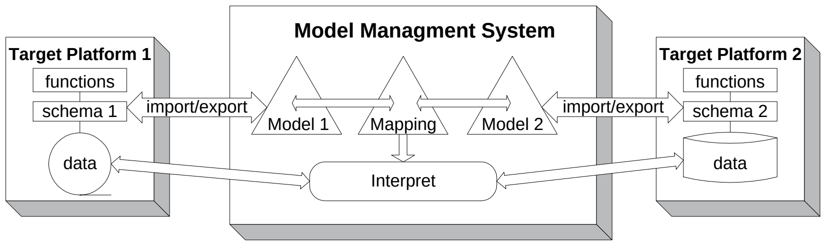


Figure 1: The High Level Architecture of Model Management

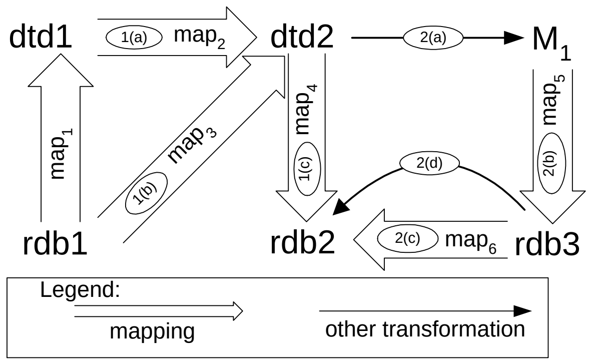
ably reduce the coding burden of the application. We refer here to several model management operations whose semantics will be elaborated later.

Consider an online merchant selling books. The data about its titles, customers, and orders are stored in a relational DB, whose schema is represented as a model, rdb1. The relational data is mapped into an XML DTD, dtd1, to serve the data onto their web site and to share data in a broader marketplace of online merchants. The DTD conforms to the standard recommended by the marketplace participants. Denote the mapping from rdb1 into dtd1 by map_1.

In a model management system, rdb1, dtd1, and map_1 would be represented as first class objects. For now, consider each one to be a set of objects in an OO system. This explicit representation enables engineers to pose queries that help familiarize them with the mapping. For example, one could ask for all XML element types that have data originating from the relation Books. Another immediate benefit of the representation of the mapping is that we can define a generic inversion operator. Suppose the company strikes a partnership with another online book merchant and starts receiving data in XML according to the standard DTD. If map_1 includes expressions that tell how to create elements of dtd1 from rows of rdb1, then map_1^{-1} will include expressions that tell how to create rows in rdb1 from elements of dtd1.

A more complex (and likely) scenario is that the company starts a partnership in a slightly different domain (e.g., CDs) with another online merchant, which exports its data in dtd2. In this case, the DTDs of the two companies will differ in two ways: (i) while they still both talk about customers and orders, one has DTD elements concerning books while the other concerns CDs, and (ii) since nobody fully conforms to standards (and even if they do, there are many versions of the same standard), the two DTDs contain slight differences even on customers and orders.

Suppose our task is to create a relational schema rdb2 for the data from the CD merchant. We will do this by using the model management operators to create the model for rdb2. Assume for the moment that only differences of type (i) exist between the DTDs. In this case, we could proceed as follows (see Figure 2):



1(a). map_2 = Match(dtd1, dtd2)
1(b). $map_3 = map_1 \circ map_2$
1(c). $<map_4$, rdb2$>$ = DeepCopy(map_3^{-1})
2(a). $M_1 =$ dtd2 $-$ range(Match(dtd1,dtd2))
2(b). map_5 = Default(M_1, rdb3)
2(c). map_6 = Match(rdb2, rdb3)
2(d). Merge(rdb2, rdb3, map_6)

Figure 2: An example sequence of model management operations.

1. For the parts of the DTDs that match exactly, use the inverse of map_1 to create rdb2. This can be done in three steps, using the operations on models and mappings:
   (a) Use the generic model matching function Match to create map_2 = Match(dtd1,dtd2), which is a mapping from dtd1 to dtd2 that identifies the maximal subsets of the two DTDs that exactly match.
   (b) Create $map_3 = map_1 \circ map_2$, the composition of map_1 and map_2, from rdb1 to dtd2. Since map_2 includes only exact matches, map_3 maps only to the subset of dtd2 that exactly matches dtd1.

(c) Set $< map_4$, rdb2> = DeepCopy(map_3^{-1}), which creates a new copy of map_3^{-1} and rdb1, called map_4 and rdb2 respectively. It is "deep" in the sense that it copies not only objects in map_3^{-1} but also objects map_3^{-1} connects to in rdb1.

2. For the parts of dtd2 that don't match dtd1, use a default mapping from DTDs to relational schemas. Using our operations, this can be done as follows:

   (a) Use the set-difference operator to create a model M_1 representing the part of dtd2 that doesn't match with dtd1: M_1 = dtd2 − Range(Match(dtd1,dtd2)).

   (b) Instantiate the default DTD-to-RDB transformation by calling map_5 = Default(M_1, rdb3), which creates a mapping map_5 and a relational schema rdb3 with new names. The domain of map_5 is M_1 and its target is rdb3.

   (c) Create map_6 = Match(rdb2, rdb3) to align the overlapping tables and keys of the two schemas, in preparation for merging them.

   (d) Call Merge(rdb2, rdb3, map_6) to merge rdb3 into rdb2 based on map_6.

Other model management operations can be useful in complications of the above situation:

- If there are differences even in the common parts of the DTDs (of type (ii) above), we would like the match function to propose a set of possible matches between the two DTDs and rank them. The engineer can then choose the appropriate one and, possibly, modify it manually. The system should also be able to take input from the engineer that constrains the possible matches between the two DTDs.
- After creating rdb2, we have two separate relational DBs for the two companies. Suppose that later on, one company acquires the other, and therefore wants to merge their DBs. In this case, we would like a Merge(rdb1,rdb2) operation to propose a relational schema that merges the original ones. This is more complex than 2(c,d) above, because the DB schemas are probably not disjoint and may have different internal structure. Furthermore, we would like the system to automatically generate a mapping from the old schema to the merged one, to facilitate the migration of applications to the new merged company.
- There may be several possible mappings from XML DTDs to relational schemas. In this case, instead of using Default(dtd,rdb), we may experiment with several mappings and pose *what-if* queries on their results. In particular, we would want to estimate the cost of processing certain queries on the resulting schemas, using a generic function Estimate(queries, schema).

Companies are facing an increasing need to share data with others in various contexts, especially with the growth of business-to-business online applications. Hence, these operations are becoming more important all the time.

The above scenario might sound like pie-in-the-sky that is hopeless to achieve. For example, it sounds extremely hard to develop a generic algorithm that finds the *best* match of two distinct models or that inverts an *arbitrary* mapping. But optimal and complete algorithms are not essential ingredients for success. Success would be realized by a platform for manipulating models and mappings using high level operators that users can further customize manually. Since there are many published, mostly-heuristic algorithms for all the above operations (cf. Section 6), this goal seems well within reach.

3 Models

In this section we describe a first attempt to design a data model for model management. We begin by discussing models. In the next section we discuss mappings between models.

At an abstract level, we think of models as labeled directed graphs, where the labels can have rather complex structure. We therefore regard an object-oriented data model to be the natural platform on which to define model management functionality. That is, we represent models by sets of objects that are connected via relationships. Using an OO data model yields the usual benefits: behavioral encapsulation, polymorphism, convenient sharing of type information, low-impedance language integration, etc. On the whole, we are agnostic about the choice of object model. However, to explain the concepts of interest to us, we need some special features in the object model. Rather than extending an existing object model, we will define a simple one that has only the features we need. For example, we do not include types (as distinct from classes), inheritance, collection-valued properties, or structure-valued properties.

Each object has a set of properties whose values describe the state of the object. To keep things simple, we assume properties are scalar-valued. Each object also has relationship properties, each of which consists of a set of relationships. Relationships are binary, so each relationship is a pair of object references. Each relationship is directed, from its *origin* object to its *destination* object, but it can be traversed in either direction. As in ODMG [CCB+00], relationships are not first-class objects.

We assume each object is an instance of a class. Each class definition describes the set of properties and relationship properties that are defined for instances of that class. Since relationships are binary, each relationship property definition has an inverse relationship property definition on the class to which it connects and is tagged as either the origin or destination side.

We assume that class definitions are objects. This makes the data model self-describing. It also means that a model can be a mixture of class definitions and ordinary objects.

We use *database* to refer to a set of objects of interest. It may be persistent, or may be simply a set of objects in main memory that are being manipulated by model management operations.

Model contents: Since we want to define operations on models, we need to start by defining the set of objects and relationships contained in a model. We could make this explicit by defining a model to be a particular object that has a relationship to every object in the model. But this is inconvenient, since it implies that every time a submodel is added to a model (e.g., adding a database schema to an application program), a relationship would have to be added from the model object to every object in the submodel. We can avoid this maintenance task entirely by using the ordinary relationships in a model to define the contents of the model.

We could say that a model is simply the set of objects that are reachable from its root object. Thus, when adding a relationship from an object in a model to some submodel, all of the objects in the submodel immediately become part of the model. Unfortunately, this doesn't quite work, because some relationships are between pairs of objects in different models. Our solution is to distinguish between relationships that imply containment within the model from those that do not. This is very similar to the representation of complex objects in some OO DBMSs and repositories. A model, then, is the transitive closure of containment relationships emanating from the model's root. Formally, we assume that each relationship property definition in the schema includes a $containment$ flag, which is set to True for containment relationships, and we define a model as follows:

Definition 1 A $model$ is a set of objects O that is identified by a $root\ object$, r in O, such that $O - \{r\}$ is the set of objects that are reachable by following containment relationships from r. □

In our experience, it is worth restricting the containment flag to be settable only on the origin side of a relationship property, and to require that containment relationships in a model constitute a directed acyclic graph [BBC+99]. This makes the definition of containment correspond to the intuitive notion of set containment. It simplifies the maintenance of a materialized closure, which enables the content of a model to be identified efficiently [DS00]. We expect it also simplifies algorithms for propagating delete, copy and other operations on models (cf. Section 5.1).

Model schemas: We can "pull up" the definition of model to the schema level. A *model schema* consists of a distinguished root class and all classes that are reachable from the root by following a sequence of containment relationship property definitions. Every model is an instance of a model schema whose root is the class of the model's root.

Cross-model relationships: We define a "cross-model" relationship to be one that connects two objects that are contained in different models. By definition of model, such a relationship cannot be a containment relationship. It may seem strange that such a relationship is not fully contained in any model. However, since relationships are not first-class objects, this fact does not cause any special problems.

Examples: We illustrate these concepts with relational schemas and XML DTDs. In the examples, the models and model schemas are shown graphically. The syntax and semantics of these models can be described in many textual languages, the choice of which is irrelevant here.

Example 1 Figure 3 shows a fragment of a model schema for relational schemas. Nodes represent classes and relationships represent relationship types. It models which relations, attributes and integrity constraints exist in the relational schema, as well as the types of the attributes and the foreign keys. Figure 4 shows an instance of the schema in Figure 3, describing a relational DB storing book orders. □

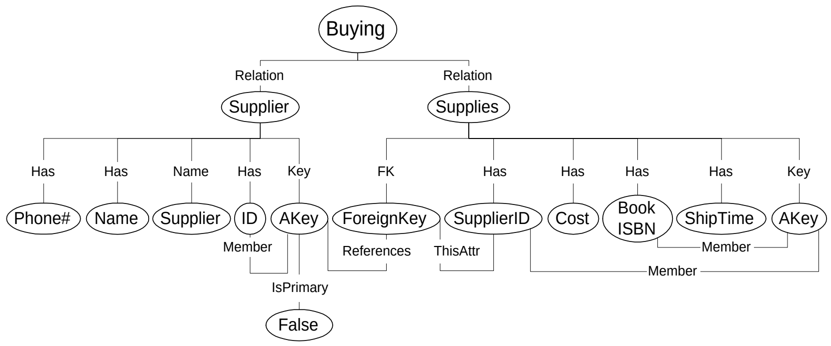


Figure 4: A model describing a relational schema for a relational database storing book orders. This model is an instance of the model schema shown in Figure 3.

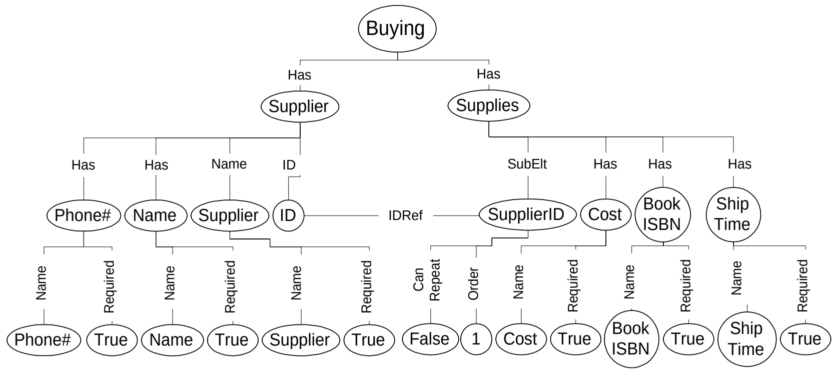


Figure 6: An instance of the model schema of Figure 5, describing XML data for purchasing books.

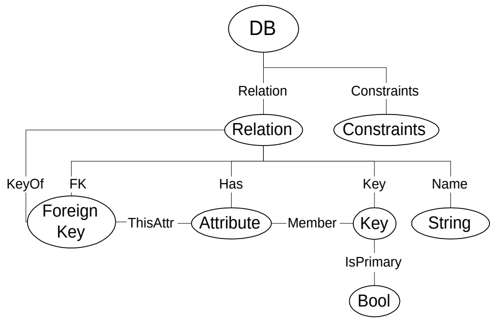


Figure 3: A model schema for relational schemas.

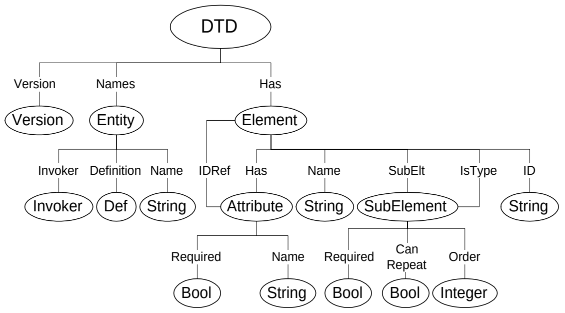


Figure 5: A model schema for XML DTDs.

Example 2 Figure 5 shows a fragment of a model schema for XML DTDs. Figure 6 shows an instance of that schema, describing a DTD for book orders. □

Challenge 1 A fundamental challenge in model management is to develop a mechanism for representing models and for storing these representations. A key issue here is how much of a model's semantics is expressed in its representation. For example, an integrity constraint for a relational schema can be represented as a string in one of the model's properties, or as a logical formula whose interpretation is known to the model management system. Another issue is how much of the semantics of the model we want to describe. For example, in Figure 3 the model schema does not express the constraint that a foreign key can only reference a primary key.

Storing and indexing models also raises challenges. Different storage schemes can be devised for building a model management system over an OO, object-relational or other DBMS. □

4 Mappings Between Models

A key goal of model management is to provide support for managing change in models and for mapping data between different models. Hence, we believe it is crucial that model mappings be manipulated as first-class citizens. Before describing our representation of model mappings, we outline the key elements underlying our approach to modeling mappings.

- We need to manipulate model mappings much like we manipulate models: copy a mapping, delete a mapping, select from a mapping, etc. To avoid defining separate elementary operations on mappings, we require that a mapping be a model.
- A mapping consists of connections between instances of two models, which may be instances of different model schemas (e.g., a mapping between a relational schema and an XML DTD). While we could allow a mapping to connect more than two models, this adds complexity to the data model and is unnecessary for the applications we know of.
- There may be more than one mapping between a given pair of models. For example, we can have two different mappings between a relational schema and an XML DTD, which represent two different ways to encode instances of the relational schema as instances of the DTD.
- A mapping may relate a *set* of objects in one model to a set of objects in another via a language for building complex expressions. For example, a mapping between two relational schemas M_1 and M_2 may specify that a view V_1 over M_1 corresponds to a view V_2 over M_2, where the view definitions are part of the mapping. Moreover, a model may have an associated language for building complex expressions over elements in the model, such as a query language or arithmetic expressions.
- Mappings must be able to nest. This enables the reuse of mappings. That is, it allows a mapping on a model M to be used as a component of a mapping on models that contain M.

Given these points, we define model mappings as follows.

Definition 2 A model mapping map from a model M_1 to a model M_2 is a model where:

1. map has a distinguished root element that has two single-valued relationship properties, domainRoot and rangeRoot, which point to the root objects of M_1 and M_2, respectively. These properties identify the models being related by the mapping.
2. Each object in map (called a *mapping object*) has a property Expr, which is an expression over the objects of M_1 and M_2.
3. Each mapping object in map has two relationship properties, called domain and range, which include all the objects of M_1 and M_2 (respectively) that are referred to in Expr.

□

We do not fix the data of Expr. It could be a string, or it could be an object that is the root of a complex structure that represents the expression.

Although a mapping must be a model, the definition of mapping says nothing about the mapping's containment relationships. This degree of freedom is appropriate because different applications will want to structure mappings in different ways. For example, a mapping between two very similar XML DTDs could mimic the structure of the DTDs being related. By contrast, a mapping between two highly dissimilar DTDs might be flat, where all objects in the mapping are children of the root, since the mapping is not preserving much of the DTDs' structure.

Example 3 Figure 7 shows a mapping between the relational schema and DTD shown in Figures 4 and 6. The skeleton of the mapping is represented in gray in Figure 7(a). Its internal containment structure, shown in Figure 7(b), mimics that of the two models it relates. The root of Mapping connects only to the root of the relational schema and XML DTD, as required by the definition of model mapping. The mapping objects' Expr properties are not shown in the figure and are presumed to be empty. □

An immediate advantage of the explicit representation of models and mappings is that it allows us to pose queries such as "Find all attributes in the XML DTD that map to members of keys in the relational model." In Figure 7, this would be ID of the element Supplier. Since we are formulating our model as a graph, we can answer essentially any queries that follow regular expressions. This is similar to OQL and many XML query languages when the SELECT clause is a subset of a single class.

There are several fundamental issues to consider concerning the representation of mappings.

Interpretation of mappings: There is a spectrum of levels at which to specify mappings. At one extreme, the

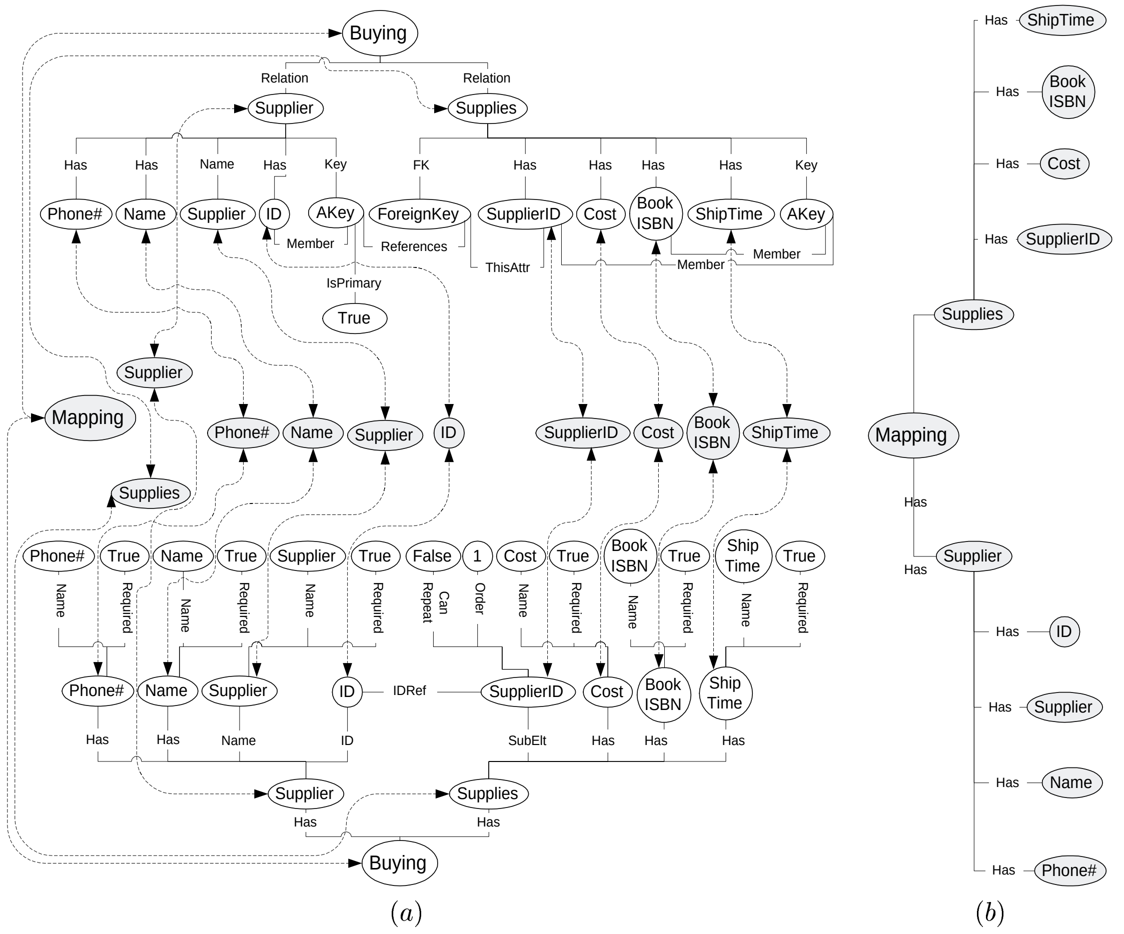


Figure 7: A mapping between the XML DTD of Figure 6 and a relational schema of Figure 4. The skeleton and internal containment structure of the mapping are shown in Figures 7(a) and 7(b) respectively.

mapping could specify the full semantic relationships between the two models. At the other extreme, a mapping is purely structural, specifying only its domain and range (i.e., the objects in the two models that are related to each other), and no mapping semantics (i.e., Expr is null). In the latter case, more semantic information can be attached to the mappings in an application-specific way, but the semantics are not interpreted by the model management system. Instead, they are interpreted by custom reasoning modules that are invoked when necessary.

For example, Figure 7 shows a structural mapping between two schemas. It could be augmented with an equality predicate for the Expr property of each mapping object that connects a relational attribute to an XML element that has no sub-elements. A full semantic relationship between the schema and DTD would need to state precisely how to translate rows of a relation into XML elements, including those with substructure. The mapping shown in Figure 7 is consistent with several such translations (see [FK99, SGT$^+$99, DFS99] for examples).

There are many advantages to interpreting the semantics of the mappings. For example, a mapping that is fully interpreted (i.e., the Expr property contains formulas in some mathematical system – a logic, algebra, or grammar) can be transformed into a program that translates data from an instance of one model to an instance of the other. Inversion and composition of mappings, as well as matching of models, also rely heavily on the semantics of the mapping.

On the other hand, exploiting the semantics of the mappings adds significant complexity to our data model and to the definition of operations on models and mappings. If we drop the semantics and consider only the structural aspects of mappings, then a model management system still offers many useful functions, such as certain kinds of matching and differencing. Furthermore, in some cases the structural mappings closely mirror the semantics of mapping (e.g., in UML models). Finally, any useful operations that consider only the structure can be implemented generically across model and mapping types.

Our representation provides a placeholder for specifying semantic details, namely the property Expr. However, we do not force the representation of the mapping's semantics. This and the above points raise the following challenge:

Challenge 2 Find appropriate representations of model mappings that trade off expressiveness of semantics with the goal of keeping operations generic across a wide set of model types. □

Directionality of mappings: A mapping is directional if it specifies how to transform data from its *domain* to its *range*. However, the general notion of a mapping only specifies a set of relationships between two models and is not necessarily directional. For example, in a mapping between two relational schemas S_1 and S_2, mapping objects can statements in Expr properties of the form $U_i = V_i$, where U_i and V_i are SQL query expressions over S_1 and S_2 respectively. Such a mapping is not directional, because it does not spell out how to transform rows of S_1 into rows of S_2.

A model management system must be able to manipulate non-directional mappings (e.g., for differencing). However, some of the operations on mappings, such as inversion and some kinds of composition, require mappings to be directional. Directionality is also a convenience, in that it enables us to say that a mapping is a function, total, onto, etc.

We took a middle ground by identifying a domain and a range for each model, without requiring the model to actually be directional. Instead, we envision an orientation operator that takes as input a mapping and produces a directional mapping from its domain to its range whenever possible (and returns the best approximation otherwise).

The issue of directionality raises one of the challenge problems in model management, which can be instantiated to different model and mapping types:

Challenge 3 Develop an algorithm that given a mapping m between two models M_1 and M_2 produces the best directional mapping m' from M_1 to M_2. □

Partial mappings: Often, a mapping does not connect to all objects in the domain model. For example, when creating a mapping between two relational schemas, we cannot always produce a complete directional mapping from the domain to the range. Moreover, we may want to manipulate mappings that are inherently partial. For example, given two relational schemas, an engineer may have constraints on which relations correspond to each other in the two models. These constraints represent a partial (possibly directional) mapping between the schemas. Many model management tasks involve attempting to complete a partial mapping. Therefore, in our representation, we have decided not to require that a mapping be complete. We revisit this issue in Section 5.2 when we discuss the matching operator.

5 Operations on Models

Recall from Section 1 that our main goal in creating a data model for model management is to reduce the amount of programming required to manipulate persistent models. We do this by defining a set of high-level algebraic operations on the two main structures of interest, models and mappings. We have two requirements for these operations. First, each operation should return a model, so that operations can be composed. Second, each operation should be generic, so it works for any model schema and for a wide range of model management applications.

We can think of many useful operations, some of which are quite standard and others that raise many research challenges. We define many of them here, in varying levels of detail. In Section 5.1 we consider operations that generalize basic operations on data to models and mappings. In Section 5.2 we consider operations that are more unique to model management. We focus only on the semantics of the operations. We do not discuss how best to offer the functionality in a programming language. Nor do we discuss implementation, except where it motivates the need for the operation at all. The overall challenge is:

Challenge 4 Design an algebra of useful, composable operations on models. Consider efficient implementations of each operation as well as of combinations of operators. □

5.1 Elementary Operations

Operations are needed to create, query and perform basic manipulations on models.

Create: Since a model is fully identified by its root, creating a model simply requires creating a root. A more powerful version of this operator, CreateModel, creates an entire model structure based on a template, with instances of many of the subclasses of the model's schema. For example, CreateModel could create an empty model of a program consisting of a signature, declaration part, and procedure part. The model's schema needs to be annotated with default values and other (possibly parameterized) directives to create a model instance.

Update: Updates are normally done at a fine grain, by using the basic operations of the underlying object model to manipulate individual objects, properties, and relationships.

Delete: Delete is a bulk operation. It involves deleting the root, and then propagating the delete to objects contained in the model. In effect, it treats containment relationships like SQL referential constraints with the cascade option (to propagate a delete to its contained objects) and restrict option (to deny the propagation to a contained object if another containment relationship is still connected to it, i.e., if the contained object is shared by two models). In SQL, this semantics has some well known problems, which are manifested here as well [CPM96, LM98, LML97].

Select: Given a model M, return a model consisting of the set of submodels of M that conform to a given qualification. The qualification language's power should be similar to that of a query language for XML, or at least OQL. Select also functions as a kind of pattern matching operation, since qualifications amount to patterns of submodels to look for. Folders, tables, and other bulk structures should be defined as models, so that Selection (and other operations) can apply to them.

Project: Project a model on a subset of the model's schema. Since some classes and relationship property types are eliminated, this may cause the model to become disconnected. To handle this, Project could return only objects that are still reachable from the model's root. Or it could return a new root, whose children are the roots of the disconnected submodels.

SetDifference: Return the objects in one model that are not contained in the other model. This was needed in step 2(a) in Section 2. The problem of disconnected results arises here too.

ApplyFunction: Apply a function to all of the objects in a model. For example, one could apply the function $Append$ "_2" $to\ the\ name\ property$ to all of the objects in rdb2 (cf. Section 2). A more complex example could be to update the objects in a model to force it to satisfy a set of constraints.

Copy: Copy is another bulk operation on models. It is tempting to say that to copy a model, all objects contained in the model are copied. But this is not always what is wanted. Some contained objects are "embedded," and should be copied, while others are "linked," and should only be referenced by the copy. For example, if a model represents the source code of an application, then some shared libraries may be regarded as part of the model (i.e., the application) with respect to some operations, such as Diff, but not with respect to Copy. Conversely, some non-contained objects should be copied. For example, the application might reference but not contain a DB schema that should be copied along with the application. We therefore need some additional control over which relationships propagate Copy.

Like containment, we could control this in relationship property definitions. That is, each relationship property definition could have a copy flag that specifies whether to propagate Copy across relationships of that type. This would work well for the examples of the previous paragraph, but not so well for the scenario in Section 2, where we performed a deep copy on a mapping, map_3. In that case, if we set the copy flag on the mapping class's relationship property definition $domain$ (the one that caused rdb1 to be copied with map_3), this would affect all mappings. So, we can either have map_3 be an instance of a more specialized type of mapping that propagates Copy across $domain$, or we need to allow control over copy propagation in the Copy operation itself.

Model enumeration: Although our goal is to capture as much model management functionality as possible in set-at-a-time operations, there will undoubtedly be times when the objects of a model need to be navigated one-by-one. Since a model is a set of objects, one could navigate it using the data manipulation language of the underlying DBMS. However, we can simplify the application programmer's job by defining an Enumerate operation that performs the traversal, thereby bridging the programming gap between models and objects within models. Enumerate can optionally ensure that each object in the model is visited only once, a modest simplification over a manual traversal. Another benefit of Enumerate over manual traversal is that its implementation can be optimized by exploiting its predictable access pattern, for example, to do intelligent prefetch [BPS99].

Enumerate takes a model, M, as parameter plus directives regarding the order in which objects should be traversed. It returns a cursor object, which can be used by get-next operations. There are many useful directives that Enumerate could offer, such as:

- Depth-first vs. breadth-first
- Pre-order vs. post-order (returning a root before or after its children)
- Respecting the order of relationships within an ordered relationship property
- Ordering the relationships within a relationship property based on a sort order on certain scalar properties of the target objects
- Ordering the containment relationship property definitions on each class, and using that to drive the enumeration order of relationship properties for each instance of that class.

Enumerate should use the containment relationships to guide the traversal, to avoid straying outside the model. It is conceivable to use non-containment relationships instead. However, this introduces a potentially expensive model membership test as each object is returned by a get-next.

5.2 Matching and Differencing

Since many applications of model management involve tracking changes in models, a key operation to consider is one that accepts as input two models, and returns the mapping that describes the *best* match between them. Unlike the operations described in the previous section, in many contexts the output of a match operation is just an educated guess made by the system. The guess can be based on examining the schema and integrity constraints of the models, or even by inspecting data instances of the two models. Such a guess gives an engineer a starting point for designing a best match.

There are different flavors of the problem of finding matches between models, depending on whether they focus on the commonalities or on the differences.

1. Find the *best mapping* between two models. As we saw in Section 2, such an operator is useful for finding the parts of two DTDs that match.
2. Find the *difference* between two models. This is basically the same as matching, except that the answer needs to highlight the differences. This operation is especially useful for round-trip engineering. As an example of this scenario, a user designs a model (e.g., an ER model) and compiles it into an executable form (e.g., SQL DDL). Later, someone modifies the compiled form (they're not supposed to, but they often do). Now the source model must be updated to reflect the updates to the compiled form (hence the notion of round-trip). Therefore, the compiled (e.g., DDL) form is reverse-engineered into the source (e.g., ER) form, producing an updated source model. The result of reverse-engineering is often not exactly what the designer intended, so the designer runs Diff between the updated source model and the original to see what changed. The result of Diff, called a $delta$, guides the designer through the changes so he or she can clean up the updated source model.
3. The engineer may have some a priori knowledge about the mapping between the two models, and may wish to find a best match that is consistent with the a priori knowledge. For example, the engineer may know that when two relation names are identical, except that one ends with V1 and the other with V2, they refer to the same relation. In some cases, the knowledge may concern which objects in one model *should not* be mapped to the other model. For example, a designer may know that "phone number" in one DTD means home phone while in the other it means business phone.
4. We may already have a partial mapping m between the two models M_1 and M_2 and would like the system to find the best complete mapping that extends m. Such a partial mapping can be viewed as an instance of a priori knowledge. For example, starting with a complete map generated by the system, an engineer might eliminate parts of the map that look wrong and add some new parts to the map. Say she removes from m the bad guess that the class HomePhone in M_1 maps to the class HomeAddress in M_2, and adds to m that HomePhone in M_1 maps to Phone in M_2 with an expression that the property Phone.IsHome = True. Based on this revision and others, she asks the system to try again to complete the mapping.

To summarize, the following is one of the key challenges in model management, which is predicated on the specific type of model.

Challenge 5 Develop algorithms for finding the best matches between two models. Specifically, given two models M_1 and M_2, and a partial mapping m that specifies a priori knowledge about the mapping from M_1 to M_2, find an extension m' of m that is the best complete mapping from M_1 to M_2. The result may be either a single mapping or a set of mappings ranked by some confidence measure. In addition, m' may be accompanied by a justification for how it was generated. □

Matching: Ideally, a Match operation should be generic, so it can apply to any type of model. One approach is to obtain a generic Match operation by encapsulating object mapping criteria in a *similarity relation*, $\cong$, over pairs of objects. A common semantics for $\cong$ is type and value equality. More complex equivalence relations may also be useful, based on dictionaries of synonyms, equivalence relations of pairs of classes, simple name transformations, etc. The $\cong$ relation can be generalized to range over pairs of sets of objects, though we'll assume not in what follows. The concept of best mapping may be based on associating objects of the two models that are $\cong$ and have (nearly) isomorphic relationship structure.

We can define a Match operation that takes as input two models and a $\cong$ relation and produces a mapping that is consistent with $\cong$. This is analogous to developing join algorithms: The join condition is (for the most part) isolated from the semantics of the join, and in many cases efficient join algorithms can be developed without knowing the particular join condition.

Each object in the mapping returned by Match should include an expression that describes how the domain and range objects are related. Often, this will require manipulating expressions in some mathematical system, such as a logic (datalog, description logic), algebra (relational algebra, arithmetic), or grammar (regular expressions, BNF). Ideally, we would like a generic Match algorithm that treats the mathematical system as a black box that it calls to generate a formula (e.g. a view definition) whenever it identifies a combination of domain and range objects that match. Alternatively, we may need a repertoire of Match algorithms, each customized to a particular formal system.

Rather than generate a best mapping consistent with $\cong$, a designer may prefer that the system shows all combinations of objects in the two models that match, from which the designer picks the combinations she likes best. This can be accomplished by an operation Mjoin(M_1, M_2, $\cong$), which has essentially the same semantics as a relational join. We define $an\ equivalence\ class\ of\ M_1\ and\ M_2$ *on* $\cong$ to be a maximal set of objects in M_1 and M_2 such that each pair in the set is related by $\cong$. We then define Mjoin to return a mapping map that consists of

- a root object, which connects to the roots of M_1 and M_2, and
- for each equivalence class E of M_1 and M_2 on $\cong$, an object $o \in o_{map}$ such that o's domain and range are the objects of M_1 and M_2 (respectively) in E and o is a child of the root.

We can extend the Mjoin operation to Left, Right, and Full OuterMjoin, by insisting that every object in M_1, M_2, or both be connected to the resulting mapping. OuterMatch can be defined analogously.

The result of Match could be flat, like that of Mjoin, where all objects in each mapping are children of the root. More likely, a user would want a mapping to mimic the structure of the models being matched. Figure 7 shows such a matching, where the mapping object Supplier connects the element Supplier to the relation Supplier, and the children of the mapping object Supplier connect the children of the element Supplier to the children of the relation Supplier. By mimicking the model structure in the mapping, an application can more easily navigate the mapping systematically. In Figure 7, it was obvious how to use the structure of the models to induce a structure on the mapping. Often, it will not be obvious, due to structural differences between the models being matched.

Challenge 6 What are good structures to use for a matching when the models being matched have different structure? □

Differencing: The differencing operation is essentially a Full OuterMatch, since the latter identifies objects that appear in one model but not the other, i.e., were inserted or deleted. The expression in each object of the map returned by Full OuterMatch is still some form of equivalence that explains how the domain and range objects are related. For example, the expression may say that domain and range objects represent the same conceptual object, but some of their property values or relationships differ. Such an expression has the same meaning as an ordinary Match, where sameness rather than difference is being emphasized. However, it may be desirable to phrase the expression in a way that emphasizes difference rather than sameness. If a mapping object has an empty domain or range, then presumably its associated expression is null.

Suppose we want to compare two models, M_1 and M_2, that are intended to be different versions of the same model. One way to do this is by comparing each of M_1 and M_2 to a base model version M_b from which both were derived. This is known to provide a more accurate view of insertions and deletions. For example, if comparing M_1 to M_b shows that object o is in M_1 but not M_b, then o was inserted during the process of updating M_b to become M_1. If no object p, where $p \cong o$, is in M_2, then o was inserted into M_1 but not into M_2. By contrast, if M_b is not available and we assume that a sequence of updates to M_1 produced M_2, then we might conclude instead that o was deleted during the update sequence.

We can express this three-way calculation as two OuterMatch operations, so the resulting difference is expressed as two mappings. Or we could define a new Diff operation that takes M_1, M_2, and M_b as input and produces a single mapping. In that case, the mapping would need to encode the distinctions of the previous paragraph. For example, a Boolean property on each mapping object could say whether the domain object o was inserted when going from M_b to M_1.

In the literature, the result of a Diff is usually a script that transforms the input model to the output model. The process of constructing the script sometimes involves first producing the mapping, as in [CRGMW96, CGM97].

Challenge 7 Is it better to make Diff a separate operation or a specialization of OuterMatch? Is there a useful way to express a transformation sequence as a model? Can script generation be encapsulated into a useful generic operation? □

5.3 Merge

Merging is the activity of moving the content of a target model into a source model. If the source and target models are disjoint, this amounts to taking the union of the source and target models and assigning it to the source model. In this case, if the roots of the source and target are identical or if they are merely placeholders under which to hang the model, then merging can be achieved by connecting the root of the target model to the children contained by the root of the source model. That is, the effect of Merge(M_1, M_2) is to make a copy of all of the outgoing containment relationships from the root of M_2 and connect them to the root of M_1. This semantics would be enough to satisfy the needs of steps 2(b) and 2(c) in the scenario of Section 2.

When the source and target are not disjoint, Merge faces several issues:

1. Avoiding the creation of duplicate copies of source objects that are already present in the target, and choosing property values for such objects when the source and target state differ.
2. Inserting source objects and relationships that are not present in the target.
3. Possibly deleting target objects and relationships that are not in the source.

Addressing these issues in a generic Merge operation is hard, because many variations are possible, depending on assumptions about the structures being merged. One way to address them is to have Merge take a third parameter which is a delta of the target and source, and use it to drive the merge activity, following the interpretation presented for the definition of potential delta.

Challenge 8 Propose a semantics for Merge that addresses the above issues and is sufficiently generic to subsume most of the known semantic variations. □

5.4 Mapping Composition

Composition of mappings is relatively easy to define for mappings that are single-valued functions, i.e., that map one object of their domain to one object of their range, since ordinary function composition semantics works well. However, we want the result to be a model, so we need containment relationships that connect the objects in the resulting mapping. One approach is to use the containment relationships of one of the two mappings involved in the composition. For example, consider single-valued functions map_1:$M_1 \rightarrow M_2$ and map_2:$M_2 \rightarrow M_3$. The result of composing map_1 and map_2, denoted $map_1 \circ map_2$, is equivalent to the following:

1. Create a shallow copy map_3 of map_1 (i.e., copy the map and its relationships, but not the objects it connects to).
2. For each object $o \in map_3$, if $\exists$ object $q \in map_2$ with $q.domain = o.range$, then set $o.range = q.range$ (i.e., replace $o.range$ by $q.range$). Otherwise set $o.range = \phi$.

Notice that map_3 includes a mapping object for every object in map_1. However, it only includes the range of an object o_2 of map_2 if an object in map_1 composes with o_2. An alternative, equally useful semantics is to guide the above procedure by map_2 instead of map_1. Then the resulting mapping will include every object in map_2 but not map_1. The latter is what is needed for step 1(b) of the scenario in Section 2, because in that case, we do not want the result of the composition to contain any object o for which $o.range = \phi$ (i.e., any object in dtd2 that does not match dtd1).

This definition works even if each object o in map_1 maps a set of objects in M_1 to an object in M_2 (e.g., maps a set of web pages to a relation that they reference). Allowing $o.range$ to be set-valued is more problematic. There are (at least) two ways to interpret $o.range$:

1. Treat map_1 as a single-valued function whose output is a set. The above definition of composition works in this case.
2. Treat map_1 as a multi-valued function whose output is a set of individual objects. So in step (2) of the above definition, we can compose o with a set of objects in map_2 the union of whose domains are covered by $o.range$.

Both semantics appear to be useful, so variations of the composition operation are needed for each of them. Finally, many interesting questions arise when mappings relate complex expressions over the two models, and when we consider composition of non-directional mappings.

6 Related Work

Model management will not be an isolated area of research. Much existing work in the DB literature already deals with aspects of the problem. Model management offers opportunities to extend these works in new directions. This section describes how some of that work fits into the overall vision.

Platforms for model management systems: Many of today's advanced DB architectures and features are relevant to developing an appropriate platform for model management. A model management system should be implemented on a platform that includes OO and object-relational functionality. This will enable it to exploit the usual OO features (inheritance, encapsulation, polymorphism), recursive queries, an extensible set of algebraic operations, an extensible query optimizer, etc. Some model management functions are likely to run faster in client cache than on a server, which is more conducive to today's OO DBs than object-relational ones. Models are usually versioned, making techniques from temporal DBs of interest. Other architectures that combine OO and deductive capabilities in sophisticated ways can also provide significant benefits to model management such as, Telos [MBJK90], ConceptBase [JJ89], Coral [RSSS94], NAIL! [MUG86], and F-Logic [KLW95].

Inferencing in model management: Several key operations in model management involve various forms of inference, such as inverting a mapping, completing a mapping, and determining equivalence of models. For example, a mapping can be thought of as a view of one model in terms of another. Therefore, inverting a mapping resembles the problem of inverting views, which raises the relevance of work on answering queries using views [YL87, TSI96, CKPS95, LMSS95, DG97, FRV96]. Even in the relational case these works need to be extended for the problems faced in model management, not to mention extensions to other contexts. Description Logics provide another formalism that has been shown to be useful for modeling DB schemas and interschema constraints [CL93, Bor95, LRO96]. They provide a formalism for representing and reasoning about intentionally defined sets. Finally, there is a rich literature on tools for testing equivalence and containment of queries [Ull97], starting from the work of Chandra and Merlin [CM77].

Efficient operations on models: A model is the transitive closure of a graph. In many scenarios, it will be important to compute this closure and/or maintain it as a materialized view. There is much literature on computing transitive closures [ADJ90, AJ87, DP97, DR94, DS95, Jag90, IRW93]. Probably, some structure in models can be exploited to improve upon published techniques. For example, many papers assume there is only one kind of edge and that all kinds of updates are equally likely. Transitive closure is also closely related to recursive query processing [BMSU86, BR86, DR94]. It will be important to learn how best to map model management functions on top of recursion, which is now a part of SQL3.

Differencing models: In many cases, differencing models is a lot like differencing graph structures. As shown in [CRGMW96, CGM97], its computational complexity is sensitive to assumptions about the kind of structure that the graph can represent and the available mapping operations. This suggests it will be hard to develop generic algorithms for differencing that are parameterized by the kinds of structures of interest. Luckily, there is a substantial research literature on differencing that can be leveraged to understand the variations that need to be covered by a generic solution, or even to understand if a generic solution is possible [Mye86, SZ90, WSC$^+$97, ZWS95].

Modeling change in models: One of most longstanding topics of DB research is data translation [BKKK87, LCC94, SHT$^+$77]. Recent techniques, such as those of [CJR98, MZ98], are excellent test cases for a generic model management system.

Schema integration: There are many approaches to schema integration which are candidate algorithms for Match [BCV99, JMN$^+$99, MHH00, MWK00, PSU98, MMP95, DDL00]. Concepts such as information capacity of models [MIR93] will also be key to comparing among models.

7 Final Remarks

In this paper, we presented an outline of a data model for model management. The data model has two main abstractions

- *model*, which captures the structure of engineered information artifacts, such as database schemas, interface definitions, XML DTDs, web site designs, semantic networks, complex documents, and software configurations, and
- *mapping*, which captures relationships between models such as transformations and matchings.

The data model includes high-level set-oriented operations that manipuate models and mappings as first class objects, such as copy, select, delete, apply-function, enumerate, compose, match, and merge. These operations should greatly reduce the amount of code required for applications that manipulate models and mappings. Moreover, they should enable people to write model manipulation applications that today seem too daunting.

A modern database system supporting object-oriented or object-relational functions should be a very suitable platform on which to implement a model management data model. However, producing such an implementation is not a cake walk. As pointed out in this paper, there are many technical challenges in developing a generic, customizable and efficient implementation. Long term, we expect that a solution to these challenges will result in a substantial layer of software that we can properly think of as a new kind of database system.

Applications that manipulate models are complicated and hard to build. It is slow work. By implementing generic model management functionality along the lines presented in this paper, the database field stands a good chance of improving programmer productivity for these applications by an order of magnitude. It is an exciting prospect.

8 Acknowledgements

We gratefully acknowledge many useful suggestions from Thomas Bergstraesser, Zack Ives, Sergey Melnick, John Mylopoulos, Arnie Rosenthal, Aamod Sane, and especially Erhard Rahm.

References

[ADJ90] Rakesh Agrawal, Shaul Dar, and H. V. Jagadish. Direct transitive closure algorithms: Design and performance evaluation. *TODS*, 15(3):427–458, 1990.

[AJ87] Rakesh Agrawal and H. V. Jagadish. Direct algorithms for computing the transitive closure of database relations. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 255–266. Morgan Kaufmann, 1987.

[BBC$^+$99] P. A. Bernstein, T. Bergstraesser, J. Carlson, P. Sanders S. Pal, and D. Shutt. Microsoft repository version 2 and the open information model. *Information Systems*, 24(2):71–98, 1999.

[BCV99] Sonia Bergamaschi, Silvana Castano, and Maurizio Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, 1999.

[Ber98] P. A. Bernstein. Repositories and object-oriented databases. *SIGMOD Record*, pages 34–46, 1998.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 311–322, 1987.

[BMSU86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–16, 1986.

[Bor95] Alex Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[BPS99] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch for implementing objects on relations. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 327–338. Morgan Kaufmann, 1999.

[BR86] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 16–52, 1986.

[CCB$^+$00] R.G.G. Cattell, Rick Catell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, 2000.

[CGM97] Sudarshan S. Chawathe and Hector Garcia-Molina. Meaningful change detection in structured data. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1997.

[CJR98] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. Serf: Schema evalution through an extensible, re-usable and flexible framework. In *The Proceedings of the 7th International Conference on Information and Knowledge Management (CIKM-98)*, pages 314–321, 1998.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, 1995.

[CL93] T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 1993.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CPM96] Roberta Cochrane, Hamid Pirahesh, and Nelson Mendonça Mattos. Integrating triggers and declarative constraints in SQL database sytems. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 567–578. Morgan Kaufmann, 1996.

[CRGMW96] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1996.

[DDL00] Anhai Doan, Pedro Domings, and Alon Y. Levy. Learning source descriptions for data integration. In *Proceedings of the International Workshop on The Web and Databases (WebDB)*, 2000.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semi-structured data with STORED. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 431–442, 1999.

[DG97] Oliver M. Duschka and Michael R. Genesereth. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, CA, 1997.

[DP97] Guozhu Dong and Chaoyi Pang. Maintaining transitive closure in first order after node-set and edge-set deletions. *Information Proc. Letters*, 62(4):193–199, 1997.

[DR94] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 454–465, 1994.

[DS95] Guozhu Dong and Jianwen Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Information and Computation*, 120(1):101–106, 1995.

[DS00] G. Dong and J. Su. Incremental maintenance of recursive views using relational calculus / sql. *SIGMOD Record*, pages 44–51, 2000.

[FK99] D. Florescu and D. Kossmann. Storing and querying xml data using an rdbms. *IEEE Data Engeneering Bulletin*, 22(3):27–34, September 1999.

[FRV96] Daniela Florescu, Louiqa Rashid, and Patrick Valduriez. Answering queries using OQL view expressions. In *Workshop on Materialized Views, in cooperation with ACM SIGMOD*, Montreal, Canada, 1996.

[IRW93] Yannis E. Ioannidis, Raghu Ramakrishnan, and Linda Winger. Transitive closure algorithms based on graph traversal. *TODS*, 18(3):512–576, 1993.

[Jag90] H. V. Jagadish. A compression technique to materialize transitive closure. *TODS*, 15(4):558–598, 1990.

[JJ89] Matthias Jarke and Manfred A Jeusfeld. Rule representation and management in ConceptBase. *SIGMOD Record*, 18(3):46–51, 1989.

[JMN$^+$99] Jan Jannink, Prasenjit Mitra, Erich Neuhold, Srinivasan Pichai, Rudi Studer, and Gio Wiederhold. An algebra for semantic interoperation of semistructured data. In *IEEE Knowledge and Data Engineering Exchange Workshop (KDEX)*, 1999.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[LCC94] Chien-Tsai Liu, Shi-Kuo Chang, and Panos K. Chrysanthis. Database schema evolution using EVER diagrams. In *Proceedings of the Workshop on Advanced Visual Interfaces*, pages 123–132, New York, New York, USA, 1994.

[LM98] Bertram Ludascher and Wolfgang May. Referential actions: From logical semantics to implementation. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 404–418, 1998.

[LML97] Bertram Ludäscher, Wolfgang May, and Georg Lausen. Referential actions as logic rules. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 217–227. ACM Press, 1997.

[LMSS95] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, San Jose, CA, 1995.

[LRO96] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, Bombay, India, 1996.

[MBJK90] John Mylopoulos, Alexander Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM TOIS*, 8(4):325–362, 1990.

[MHH00] Renee J. Miller, Laura Haas, and Mauricio Hernandez. Schema mapping as query discovery. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 2000.

[MIR93] Renne J. Miller, Yannis E. Ioannidis, and Raghu Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 120–133, 1993.

[MMP95] John Mylopoulos and Renate Motschnig-Pitrik. Partitioning information bases with contexts. In *Proceedings of 3rd CoopIS*, pages 44–54, 1995.

[MUG86] Katherine A. Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the NAIL! system. In *Third International Conference on Logic Programming*, 1986.

[MWK00] Prasenjit Mitra, Gio Wiederhold, and Martin L. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 86–100, 2000.

[Mye86] E. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[MZ98] Tova Milo and Sagit Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, New York City, USA, 1998.

[PSU98] L. Palopoli, D. Saccà, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *Proceedingsof IDEAS'98*, pages 244–253. IEEE Press, Cardiff, United Kingdom, 1998.

[RSSS94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL deductive system. *VLDB Journal*, 3(2):161–210, 1994.

[SGT$^+$99] J. Shanmugasundaram, H. Gang, K. Tufte, C. Zhang, D. J. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, 1999.

[SHT$^+$77] N.C. Shu, B.C. Housel, R.W. Taylor, S.P. Ghosh, and V.Y. Lum. Express: A data extraction, processing and restructuring system. *ACM Transactions on Database Systems*, 2(2):134–174, 1977.

[SZ90] Dennis Shasha and Kaizhong Zhang. Fast algorithms for the unit cost editing distance between trees. *J. Algorithms*, 11(4):581–621, 1990.

[TSI96] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[Ull97] Jeffrey D. Ullman. Information integration using logical views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[WSC$^+$97] Jason Tsong-Li Wang, Dennis Shasha, George Jyh-Shian Chang, Liam Relihan, Kaizhong Zhang, and Girish Patel. Structural matching and discovery in document databases. In *Proc. of ACM SIGMOD Conf. on Management of Data*, pages 560–563, 1997.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 245–254, Brighton, England, 1987.

[ZWS95] Kaizhong Zhang, Jason Tsong-Li Wang, and Dennis Shasha. On the editing distance between undirected acyclic graphs and related problems. In *Proceedings of CPM*, pages 395–407, 1995.