Applying Model Management to Classical Meta Data Problems

Philip A. Bernstein

Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
philbe@microsoft.com

Abstract

Model management is a new approach to meta data management that offers a higher level programming interface than current techniques. The main abstractions are models (e.g., schemas, interface definitions) and mappings between models. It treats these abstractions as bulk objects and offers such operators as Match, Merge, Diff, Compose, Apply, and ModelGen. This paper extends earlier treatments of these operators and applies them to three classical meta data management problems: schema integration, schema evolution, and round-trip engineering.

1 Introduction

Many information system problems involve the design, integration, and maintenance of complex application artifacts, such as application programs, databases, web sites, workflow scripts, formatted messages, and user interfaces. Engineers who perform this work use tools to manipulate formal descriptions, or *models*, of these artifacts, such as object diagrams, interface definitions, database schemas, web site layouts, control flow diagrams, XML schemas, and form definitions. This manipulation usually involves designing transformations between models, which in turn requires an explicit representation of *mappings*, which describe how two models are related to each other. Some examples are:

- mapping between class definitions and relational schemas to generate object wrappers,
- mapping between XML schemas to drive message translation,
- mapping between data sources and a mediated schema to drive heterogeneous data integration,
- mapping between a database schema and its next release to guide data migration or view evolution,
- mapping between an entity-relationship (ER) model and a SQL schema to navigate between a database design and its implementation,
- mapping source makefiles into target makefiles, to drive the transformation of make scripts from one programming environment to another, and
- mapping interfaces of real-time devices to the interfaces required by a system management environment to enable it to communicate with the device.

Following conventional usage, we classify these as *meta data management* applications, because they mostly involve manipulating descriptions of data, rather than the data itself.

Today's approach to implementing such applications is to translate the given models into an object-oriented representation and manipulate the models and mappings in that representation. The manipulation includes designing mappings between the models, generating a model from another model along with a mapping between them, modifying a model or mapping, interpreting a mapping, and generating code from a mapping. Database query languages offer little help for this kind of manipulation. Therefore, most of it is programmed using object-at-a-time primitives.

We have proposed to avoid this object-at-a-time programming by treating models and mappings as abstractions that can be manipulated by model-at-a-time and mapping-at-a-time operators [6]. We believe that an implementation of these abstractions and operators, called a *model management system*, could offer an order-of-magnitude improvement in programmer productivity for meta data applications.

The approach is meant to be *generic* in the sense that a single implementation is applicable to all of the data models in the above examples. This is possible because the same modeling concepts are used in virtually all modeling environments, such as UML, extended ER (EER), and XML Schema. Thus, an implementation that uses a representation of models that includes most of those concepts would be applicable to all such environments.

There are many published approaches to the list of meta data problems above and others like them. We borrow from these approaches by abstracting their algorithms into a small set of operators and generalizing them across applications and, to some extent, across data models. We

Proceedings of the 2003 CIDR Conference

thereby hope to offer a more powerful database platform for such applications than is available today.

In a model management system, models and mappings are syntactic structures. They are expressed in a type system, but do not have additional semantics based on a constraint language or query language. Despite this limited expressiveness, model management operators are powerful enough to avoid most object-at-a-time programming in meta data applications. And it is precisely this limited expressiveness that makes the semantics and implementation of the operators tractable.

Still, for a complete solution, meta data problems often require some semantic processing, typically the manipulation of formulas in a mathematical system, such as logic or state machines. To cope with this, model management offers an extension mechanism to exploit the power of an inferencing engine for any such mathematical system.

Before diving into details, we offer a short preview to see what model management consists of and how it can yield programmer productivity improvements. First, we summarize the main model management operators:

- Match – takes two models as input and returns a mapping between them
- Compose – takes a mapping between models A and B and a mapping between models B and C, and returns a mapping between A and C
- Diff – takes a model A and mapping between A and some model B, and returns the sub-model of A that does not participate in the mapping
- ModelGen – takes a model A, and returns a new model B based on A (typically in a different data model than A's) and a mapping between A and B
- Merge – takes two models A and B and a mapping between them, and returns the union C of A and B along with mappings between C and A, and C and B.

Second, to see how the operators might be used, consider the following example [7]: Suppose we are given a mapping map_1 from a data source S_1 to a data warehouse S_W, and want to map a second source S_2 to S_W, where S_2 is similar to S_1. See Figure 1. (We use S_1, S_W, and S_2 to name both the schemas and databases.) First we call Match(S_1, S_2) to obtain a mapping map_2 between S_1 and S_2, which shows where S_2 is the same as S_1. Second, we call Compose(map_1, map_2) to obtain a mapping map_3 between S_2 and S_W, which maps to S_W those objects of S_2 that correspond to objects of S_1. To map the other objects of S_2 to S_W, we call Diff(S_2, map_3) to find the sub-model S_3 of S_2 that is not mapped by map_3 to S_W, and map_4 to identify corresponding objects of S_2 and S_3. We can then call other operators to generate a warehouse schema for S_3 and merge it into S_W. The latter details are omitted, but we will see similar operator sequences later in the paper.

The main purpose of this paper is to define the semantics of the operators in enough detail to make the above sketchy example concrete, and to present additional examples to demonstrate that model management is a credible approach to solving problems of this type. Although this paper is not the first overview of model management, it is the most complete proposal to date. Past papers presented a short vision [5,6], an example of applying model management to a data warehouse loading scenario [7], an application of Merge to mediated schemas [22], and an initial mathematical semantics for model management [1]. We also studied the match operator [23], which has developed into a separate research area. This paper offers the following new contributions to the overall program:

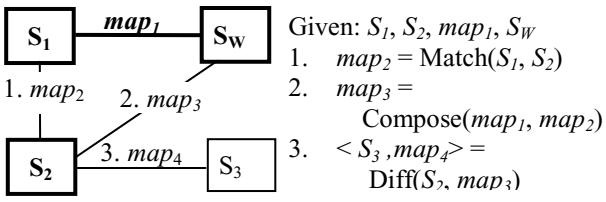


Figure 1 Using model management to help generate a data warehouse loading script

- The first full description of all of the model management operators.
- New details about two of the operators, Diff and Compose, and a new proposed operator, ModelGen.
- Applications of model management to three well known meta data problems: schema integration, schema evolution, and round-trip engineering.

We regard the latter as particularly important, since they offer the first detailed demonstration that model management can help solve a wide range of meta data problems.

The paper is organized as follows: Section 2 describes the two main structures of model management, models and mappings. Section 3 describes the operators on models and mappings. Section 4 presents walkthroughs of solutions to schema integration, schema evolution, and round-trip engineering. Section 5 gives a few thoughts about implementing model management. Section 6 discusses related work. Section 7 is the conclusion.

2 Models and Mappings

2.1 Models

For the purposes of this paper, the exact choice of model representation is not important. However, there are several technical requirements on the representation of models, which the definitions of mappings and model management operators depend on.

First, a model must contain a set of objects, each of which has an identity. A model needs to be a set so that its content is well-defined (i.e., some objects are in the set while others are not). By requiring that objects have identity, we can define a mapping between models in terms of mappings between objects or combinations of objects.

Second, we want the expressiveness of the representation of models to be comparable to that of EER models. That is, objects can have attributes (i.e., properties), and

can be related by is-a (i.e., generalization) relationships, has-a (i.e., aggregation or part-of) relationships, and associations (i.e., relationships with no special semantics). As well, there may be some built-in types of constraints, such as the min and max cardinality of set-valued properties.

Third, since a model is an object structure, it needs to support the usual object-at-a-time operations to create or delete an object, read or write a property, and add or remove a relationship.

Fourth, we expect objects, properties and relationships to have types. Thus, there are (at least) three meta-levels in the picture. Using conventional meta data terminology, we have: instances, which are models; a meta-model that consists of the type definitions for the objects of models; and the meta-meta-model, which is the representation language in which models and meta-models are expressed. We avoid using the term "data model," because it is ambiguous in the meta data world. In some contexts, it means the meta-meta-model, e.g., in a relational database system, the relational data model is the meta-meta-model. In other contexts, it means the meta-model; for example, in a model management system, a relational schema (such as the personnel schema) is a model, which is an instance of the relational meta-model (which says that a relational schema consists of table definitions, columns definitions, etc.), where both the model and meta-model are represented in the meta-meta-model (such as an EER model).

Since a goal of model management is to be as generic as possible, a rich representation is desirable so that when a model is imported from another data model, little or no semantics is lost. However, to ensure model management operators are implementable, some compromises are inevitable between expressiveness and tractability.

To simplify the discussion in this paper, we define a *model* to be a set objects, each of which has properties, has-a relationships, and associations. We assume that a model is identified by its root object and includes exactly the set of objects reachable from the root by paths of has-a relationships. In an implementation, we would expect a richer model comparable to EER models.

2.2 Mappings

Given two models M_1 and M_2, a *morphism* over M_1 and M_2 is a binary relation over the objects of the two models. That is, it is a set of pairs $<o_1, o_2>$ where o_1 and o_2 are in M_1 and M_2 respectively. A *mapping* between models M_1 and M_2 is a model, map_{12}, and two morphisms, one between map_{12} and M_1 and another between map_{12} and M_2. Thus, each object m in mapping map_{12} can relate a set of objects in M_1 to a set of objects in M_2, namely the objects that are related to m via the morphisms. For example, in Figure 2, Map_{ee} is a mapping between models Emp and Employee, where has-a relationships are represented by solid lines and morphisms by dashed lines.

In effect, a mapping reifies the concept of a relationship between models. That is, instead of representing the relationship as a set of pairs (of objects), a mapping represents it as a set of objects (each of which can relate objects in the two models). In our experience, this reification is often needed for satisfactory expressiveness. For example, if the mapping in Figure 2 were represented as a relationship, it would presumably include the pairs <Name, FirstName> and <Name, LastName>, which loses the structure in Map_{ee} that shows FirstName and LastName as components of Name.

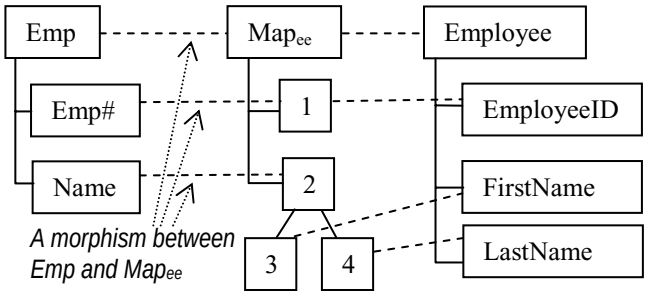


Figure 2 An example of a mapping

In addition to enabling more structural expressiveness, reifying a mapping also allows us to attach custom semantics to it. We can do this by having a property called *Expression* for each object m in a mapping, which is an expression whose variables include the objects that m directly or indirectly references in M_1 and M_2. For example, in Figure 2 we could associate an expression with object 2 that says Name equals the concatenation of FirstName and LastName. We will have more to say about the nature of these expressions at the end of Section 3.

Despite these benefits of reifying mappings as models, we expect there is value in specializing model management operators to operate directly on morphisms, rather than mappings. However, such a specialization is outside the scope of this paper. Thus, the operators discussed here work on models and mappings, but not on morphisms (separately from the mappings that contain them).

3 Model Management Algebra

3.1 Match

The operator Match takes two models as input and returns a mapping between them. The mapping identifies combinations of objects in the input models that are either equal or similar, based on some externally provided definition of equality and similarity. In some cases, the definition is quite simple. For example, the equality of two objects may be based on equality of their identifiers or names. In other cases, it is quite complex and perhaps subjective. For example, the equality of database schema objects for databases that were independently developed by different enterprises may depend on different terminologies used to name objects.

This range of definitions of equality leads to two versions of the match operator: Elementary Match and Complex Match. Elementary Match is based on the simple definition of equality. It is used where that simple definition is likely to yield an accurate mapping, e.g.,

when one model is known to be an incremental modification of another model.

Complex Match is based on complex definitions of equality. Although it need not set the Expression property on mapping objects, it should at least distinguish sets of objects that are equal (=) from those that are only similar (≅). By similar, we mean that they are related but we do not express exactly how. For example, in Figure 3, object 1 says that Emp# and EmployeeID are equal, while object 2 says that Name is similar to a combination of FirstName and LastName. A human mapping designer might update object 2's Expression property to say that Name equals the concatenation of FirstName and LastName.

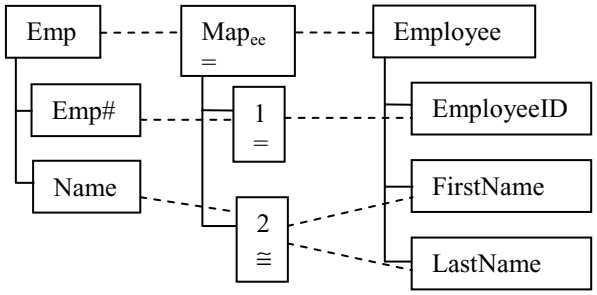


Figure 3 A mapping output from Complex Match

In practice, Complex Match is not an algorithm that returns a mapping but rather is a design environment to help a human designer develop a mapping. It potentially benefits from using technology from a variety of fields: graph isomorphism to identify structural similarity in large models; natural language processing to identify similarity of names or to analyze text documentation of a model; domain-specific thesauri; and machine learning and data mining to use similarity of data instances to infer the equality of model objects. A recent survey of approaches to Complex Match is [23].

3.2 Diff

Intuitively, the difference between two models is the set of objects in one model that do not correspond to any object in the other model. One part of computing a difference is determining which objects do correspond. This is the main function of Match. Rather than repeating this semantics as part of the diff operator, we compute a difference relative to a given mapping, which may have been computed by an invocation of Match. Thus, given a mapping map_1 between models M_1 and M_2, the operator Diff(M_1, map_1) returns the objects of M_1 that are not referenced in map_1's morphism between M_1 and map_1.

There are three problems with this definition of Diff, which require changing it a bit. First, the root of map_1 always references an object (often the root) of M_1, so the result of Diff(M_1, map_1) would not include that object. This is inconvenient, because it makes it hard to align the result of Diff with M_1 in subsequent operations. We will see examples of this in Section 4. Therefore, we alter the definition of Diff to require that the result includes the object of M_1 referenced by map_1's root.

Second, recall that a model is the set of objects reachable by paths of has-a relationships from the root. Since the result of Diff may equal any subset of the objects of M_1, some of those objects may not be connected to the Diff result's root. If they are not, the result of Diff is not a model. For example, consider Diff(Employee, Map$_{ee}$) on the models and mapping in Figure 4. Since FirstName and LastName are not referenced by Map$_{ee}$'s morphism between Employee and Map$_{ee}$, they are in the result. However, Name is not in the result, so FirstName and LastName are not connected to the root, Employee, of the result and therefore are not in that model. This is undesirable, since such objects cannot be subsequently processed by other operators, all of which expect a model as input. Therefore, to ensure that the result of Diff is a well-formed model, for every object o in the result, we require the result to include all objects O on a path of has-a relationships from the M_1 object referenced by map_1's root to o. Objects in O that are referenced in map_1's morphism to M_1 are called *support objects*, because they are added only to support the structural integrity of the model. For example, in Figure 5, Name is a support object in the result of Diff(Employee, Map$_{ee}$).

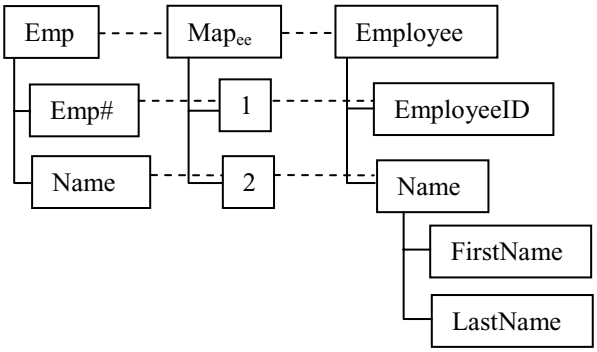


Figure 4 Diff(Employee, map$_{ee}$) includes FirstName and LastName but not Name

Having made this decision, we now have a third problem, namely, in the model that is returned by Diff, how to distinguish support objects from objects that are meant to be in the result of Diff (i.e., that do not participate in map_1)? We could simply mark support objects in the result. But this introduces another structure, namely a marked model. To avoid this complication, we use our two existing structures to represent the result, namely, model and mapping. That is, the result of Diff is a pair <M_1', map_2>, where

- M_1' includes a copy of: the M_1 object r referenced by map_1's root; the set S of objects in M_1 that are not referenced by map_1's morphism between map_1 and M_1; all support objects, i.e., those on a path of has-a relationships from r to an object in S that are not otherwise required in M_1'; every has-a relationship between two objects of M_1 that are also in M_1'; and every association between two objects in S or between an object in S and an object outside of M_1.

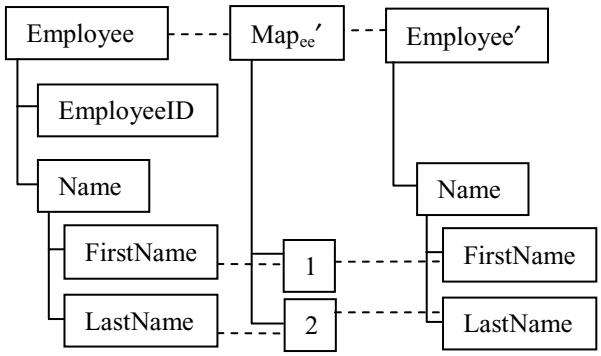


Figure 5 The result of Diff(Employee, Map$_{ee}$) is <Employee′, Map$_{ee}$′>

- *map*$_2$ connects the root of M_1' to r in M_1 and connects each object of S to the corresponding object of M_1'.

For example, given Employee and Map$_{ee}$ in Figure 4, the result of Diff(Employee, Map$_{ee}$) is <Employee′, Map$_{ee}$′> as shown in Figure 5.

3.3 Merge

The merge operation returns a copy of all of the objects of the input models, except that objects of the input models that are equal are collapsed into a single object in the output. Stating this more precisely, given two models M_1 and M_2 and a mapping map_1 between them, Merge(M_1, M_2, map_1) returns a model M_3 such that

- M_3 includes a copy of all of the objects of M_1, M_2, and map_1, except that for each object m of map_1 that declares objects of M_1 and M_2 to be equal, those equal objects are dropped from M_3 and their properties and relationships are added to m. The root of map_1 must declare the roots of M_1 and M_2 to be equal.
- All relationships in M_1, M_2, and map_1 are copied to the corresponding objects in M_3. For example, in Figure 6 Emp′ is the result of Merge(Emp, Employee, Map$_{ee}$) on the models and mappings of Figure 2.
- Merge also returns two mappings, map_{13} between M_1 and M_3 and map_{23} between M_2 and M_3, which relate each object of M_3 to the objects from which it was derived. Thus, the output of Merge is a triple <M_3, map_{13}, map_{23}>. For example, Figure 7 shows the map pings between the merge result in Figure 6 and the two input models of the merge, Emp and Employee.

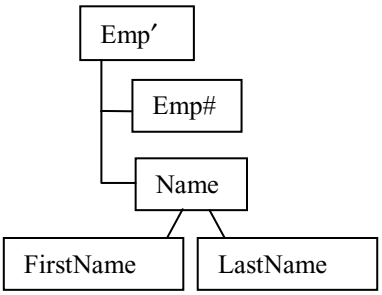


Figure 6 The result of Merge applied to Figure 2

The effect of collapsing objects into a single object can cause the output of Merge to violate basic constraints that models must satisfy. For example, suppose map_1 declares objects m_1 of M_1 and m_2 of M_2 to be equal, and suppose m_1 is of type integer and m_2 is of type image. The type of the merged object m_3 is both integer and image. If a constraint on models is that each object is allowed to have at most one type, then m_3 manifests a constraint violation that must be repaired, either as part of Merge or in a post-processing step. A solution to this specific problem appears in [9]. A more general discussion of constraint violations in merge results appears in [15].

3.4 Compose

The composition operator, represented by •, creates a mapping by combining two other mappings. If map_1 relates models M_1 and M_2, and map_2 relates M_2 and M_3, then the composition $map_3 = map_2 \bullet map_1$ is a mapping that relates M_1 and M_3 (i.e., $map_3(M_1) \equiv map_2(map_1(M_1))$).

To explain the semantics of composition, we will use mathematical function terminology: For each object m_1 in map_1, we refer to the objects that m_1 references in M_1 as its *domain*, and those that m_1 references in M_2 as its *range*. That is, domain(m_1) $\subseteq M_1$ and range(m_1) $\subseteq M_2$. Similarly, for each object m_2 in map_2, domain(m_2) $\subseteq M_2$ and range(m_2) $\subseteq M_3$.

In principle, a composition can be driven by either the left mapping (map_1) or right mapping (map_2). However, in this paper we restrict our attention to right compositions, since that is enough for the examples in Section 4. In a right composition, the structure of map_2 determines the structure of the output mapping.

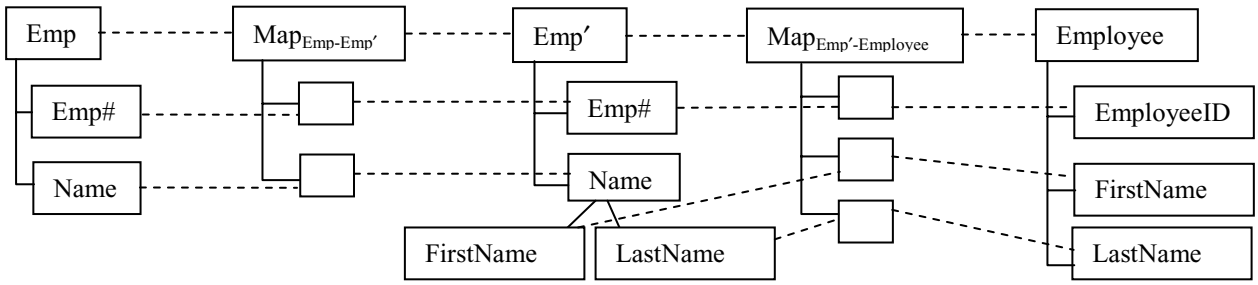


Figure 7 The merge result, Emp′, of Figure 2 with its mappings to the input models Emp and Employee

To compute the composition, for each object m_2 in map_2, we identify each object m_1 in map_1 where range(m_1) ∩ domain(m_2) ≠ ∅, which means that range(m_1) can supply at least one object to domain(m_2). For example, in Figure 8, the ranges of 4, 5, and 6 in map_1 can each supply one object to domain(11) in map_2. Suppose objects m_{11}, …, m_{1n} in map_1 together supply all of domain(m_2), and each m_{1i} (1≤i≤n) supplies at least one object to domain(m_2). That is, $\bigcup_{1\le i\le n} range(m_{1i}) \supseteq domain(m_2)$ and (range(m_{1i}) ∩ domain(m_2)) ≠ ∅ for 1≤i≤n. Then m_2 should generate an output object m_3 in map_3 such that range(m_3) = range(m_2) and domain(m_3) = $\bigcup_{1\le i\le n} domain(m_{1i})$.

For example, in Figure 8, range(4) and range(5) can supply all of domain(11). That is, range(4) ∪ range(5) = {7, 8, 9} ⊇ domain(11) = {7, 9}. Then object 11 should generate an output object m_3 in map_3 (not shown in the figure), such that range(m_3) = range(m_2) = {13} and domain(m_3) = domain(4) ∪ domain(5) = {1,2}.

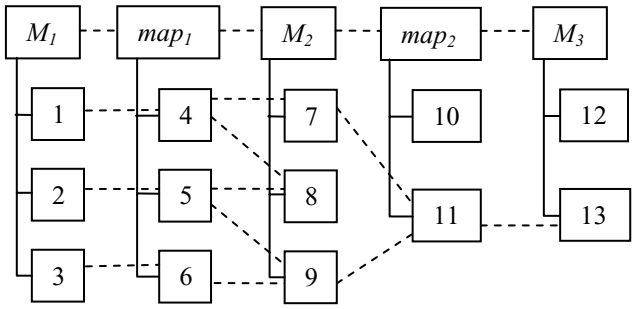


Figure 8 Mappings map$_1$ and map$_2$ can be composed

There is a problem, though: for a given m_2 in map_2, there may be more than one set of objects m_{11}, …, m_{1n} in map_1 that can supply all of domain(m_2). For example, in Figure 8, {4, 5} and {4, 6} can each supply all of domain(11). When defining composition, which set do we choose? In this paper, rather than choosing among them, we use all of them. That is, we compose each m_2 in map_2 with the union of all objects m_1 in map_1 where range(m_1) ∩ domain(m_2) ≠ ∅ ({4,5,6} in the example). This semantics supports all of the application scenarios in Section 4.

Given this decision, we define the right composition map_3 of map_1 and map_2 constructively as follows:

1. (Copy) Create a copy map_3 of map_2. Note that map_3 has the same morphisms to M_2 and M_3 as map_2 and, therefore, the same domains and ranges.
2. (Precompute Input) For each object m_3 in map_3, let Input(m_3) be the set of all objects m_1 in map_1 such that range(m_1) ∩ domain(m_2) ≠ ∅.
3. (Define domains) For each m_3 in map_3,
   a. if $\bigcup_{m_{1i}\in Input(m_3)} range(m_{1i}) \supseteq domain(m_3)$, then set domain($m_3$) = $\bigcup_{m_{1i}\in Input(m_3)} domain(m_{1i})$.
   b. else if m_3 is not needed as a support object (because none of its descendants satisfies (3a)), then delete it, else set domain(m_3) = range(m_3) = ∅.

Step 3 defines the domain of each object m_3 in map_3. Input(m_3) is the set of all objects in map_1 whose range intersects the domain of m_3. If the union of the ranges of Input(m_3) contains the domain of m_3, then the union of the domains of Input(m_3) becomes the domain of m_3. Otherwise, m_3 is not in the composition, so it is either deleted (if it is not a support object, required to maintain the well-formed-ness of map_3), or its domain and range are cleared (since it does not compose with objects in map_1).

Sometimes it is useful to keep every object of map_2 in map_3 even though its Input set does not cover its domain. This is called a *right outer composition*, because all objects of the right operand, map_2, are retained. Its semantics is the same as right composition, except that step 3b is replaced by "else set domain(m_3) = ∅."

A definition of composition that allows a more flexible choice of inputs to m_2 is in [7]. It is more complex than the one above and is not required for the examples in Section 4, so we omit it here.

3.5 Apply

The operator Apply takes a model and an arbitrary function f as inputs and applies f to every object of the model. In many cases, f modifies the model, for example, by modifying certain properties and relationships of each object. The purpose of Apply is to reduce the need for application programs to do object-at-a-time navigation over a model. There can be variations of the operator for different traversal strategies, such as pre-order or post-order over has-a relationships with the proviso that it does not visit any object twice (in the event of cycles).

3.6 Copy

The operator Copy takes a model as input and returns a copy of that model. The returned model includes all of the relationships of the input model, including those that connect its objects to objects outside the model.

One variation of Copy is of special interest to us, namely DeepCopy. It takes a model and mapping as input, where the mapping is incident to the model. It returns a copy of both the model and mapping as output. In essence, DeepCopy treats the input model and mapping as a single model, creating a copy of both of them together. To see the need for DeepCopy, consider how complicated it would be to get its effect without it, by copying the model and mapping independently. Several other variations of Copy are discussed in [6].

3.7 ModelGen

Applications of model management usually involve the generation of a model in one meta-model from a model in another meta-model. Examples are the generation of a SQL schema from an ER diagram, interface definitions from a UML model, or HTML links from a web site map. A model generator is usually meta-model specific. For example, the behavior of an ER-to-SQL generator very much depends on the source and target being ER and SQL models respectively. Therefore, one would not

expect model generation to be a generic, i.e., meta-model-independent, operator.

Still, there is some common structure across all model generators worth abstracting. One is that the generation step should produce not only the output model but also a mapping from the input model to the output model. This allows later operators to propagate changes from one model to the other. For example, if an application developer modifies a SQL schema, it helps to know how the modified objects relate to the ER model, so the ER model can be made consistent with the revised SQL schema. This scenario is developed in some detail in Section 4.3.

A second common structure is that most model generators simply traverse the input model in a predetermined order, much like Apply, and generate output model objects based on the type of input object it is visiting. For example, a SQL generator might generate a table definition for each entity type, a column definition for each attribute type, a foreign key for each 1:n relationship type, and so on. In effect, the generator is a case-statement, where the case-statement variable is the type of the object being visited. If the case-statement is encapsulated as a function, it can be executed using the operator Apply.

Since the case-statement is driven by object types, one can go a step further in automating model generation by tagging each meta-model object (which is a type definition) by the desired generation behavior for model objects of that type, as proposed in [10]. Using it, model generation could be encapsulated as a model management operator, which we call *ModelGen*.

3.8 Enumerate

Although our goal is to capture as much model manipulation as possible in model-at-a-time operators, there will be times when iterative object-at-a-time code is needed. To simplify application programming in this case, we offer an operator called Enumerate, which takes a model as input and returns a "cursor" as output. The operator Next, when applied to a cursor, returns an object in the model that was the input to Enumerate, or null when it hits the end of the cursor. Like Apply, Enumerate may offer variations for different traversal orderings.

3.9 Other Data Manipulation Operators

Since models are object structures, they can be manipulated by the usual object-at-a-time operators: read an attribute; traverse a relationship, create an object, update an attribute, add or remove a relationship, etc. In addition, there are two other bulk database operators of interest:

- Select – Return the subset of a model that satisfies a qualification formula. The returned subset includes additional support objects, as in Diff. Like Diff, it also returns a mapping between the returned model and the input model, to identify the non-support objects.
- Delete – This deletes all of the objects in a given model, except for those that are reachable by paths of has-a relationships from other models.

3.10 Semantics

The model management operators defined in Section 3 are purely syntactic. That is, they treat models and mappings as graph structures, not as schemas that are templates for instances. The syntactic orientation is what enables model and mapping manipulation operators to be relatively generic. Still, in most applications, to be useful, models and mappings must ultimately be regarded as templates for instances. That is, they must have semantics. Thus, there is a semantic gap between model management and applications that needs to be filled.

The gap can be partially filled by making the meta-meta-model described in Sections 2.1 more expressive and extending the behavior of the operators to exploit that extra expressiveness. So, rather than knowing only about has-a and association relationships, the meta-meta-model should be extended to include is-a, data types, keys, etc.

Another way to introduce semantics is to use the Expression property in each mapping object *m*. Recall that such an expression's variables are the objects referenced by *m* in the two models being related. To exploit these expressions, the model management operators that generate mappings should be extended to produce expressions for any mapping objects they generate. For example, when Compose combines several objects from the two input mappings into an output mapping object *m*, it would also generate an expression for *m* based on the expressions on the input mapping objects. Similarly, for Diff and Merge.

The expression language is meta-model-specific, e.g., for the relational data model, it could be conjunctive queries. Therefore, the extensions to model management operators that deal with expressions must be meta-model-specific too and should be performed by a meta-model-specific expression manipulation engine. For example, the expression language extension for Compose would call this engine to generate an expression for each output mapping object it creates [16]. Some example walk-throughs of these extensions for SQL queries are given in [7]. However, a general-purpose interface between model management operators and expression manipulation engines has not yet been worked out.

Another approach to adding semantics to mappings is to develop a design tool for the purpose, such as Clio [17,27].

4 Application Scenarios

In this section, we discuss three common meta data management problems that involve the manipulation of models and mappings: schema integration, schema evolution, and round-trip engineering. We describe each problem in terms of models and mappings and show how to use model management operators to solve it.

4.1 Schema Integration

The problem is to create: a schema S_3 that represents all of the information expressed in two given database

schemas, S_1 and S_2; and mappings between S_1 and S_3 and between S_2 and S_3 (see Figure 9). The schema integration literature offers many algorithms for doing this [1,8,23]. They all consist of three main activities: identifying overlapping information in S_1 and S_2; using the identified overlaps to guide a merge of S_1 and S_2; and resolving conflict situations (i.e., where the same information was represented differently in S_1 and S_2) during or after the merge. The main differentiator between these algorithms is in the conflict resolution approaches.

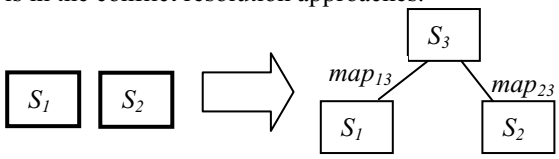


Figure 9 The schema integration problem

If each schema is regarded as a model, then we can express the first two activities using model management operators as follows:

1. map_{12} = Match(S_1, S_2). This step identifies the equal and similar objects in S_1 and S_2. Since Match is creating a mapping between two independently developed schemas, this is best done with a Complex Match operator (rather than Elementary Match).
2. <S_3, map_{13}, map_{23}> = Merge(S_1, S_2, map_{12}). Given the mapping created in the previous step, Merge produces the integrated schema S_3 and the desired mappings.

For example, in Figure 10, Map_{ee} could be the result of Match(Emp, Employee). Notice that this is similar to Figure 3, except that Emp has an additional object Address and Employee has an additional object Phone, neither of which are mapped to objects in the other model.

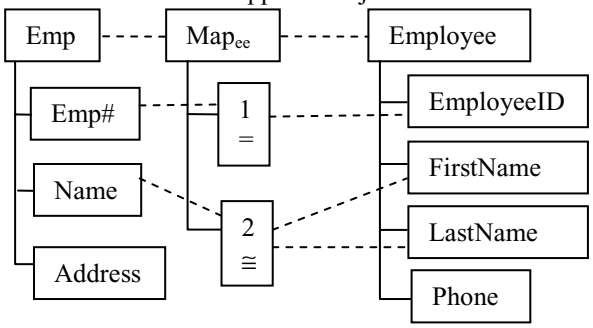


Figure 10 The result of matching Emp and Employee

Figure 11 shows the result of merging Emp and Employee with respect to Map_{ee}. (The mappings between Emp′ and Emp and between Emp′ and Employee are omitted, to avoid cluttering the figure.) Since Map_{ee} says that the Emp# and EmployeeID objects are equal, they are collapsed into a single object Emp#. The two objects have different names; Merge chose the name of the left object, Emp#, one of the many details to nail down in a complete specification of Merge's semantics. Since Address and Phone are not referenced by Map_{ee}, they are simply copied to the output. Since Map_{ee} says that Name is similar to FirstName and LastName, these objects are partially integrated in S_{12} under an object labeled ≅, which is a placeholder for an expression that relates Name to FirstName and LastName.

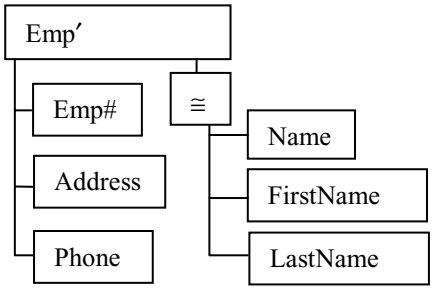


Figure 11 The result of merging Emp and Employee based on Map_{ee} in Figure 10

The sub-structure rooted by "≅" represents a conflict between the two input schemas. A schema integration algorithm needs rules to cope with such conflicts. In this case it could consult a knowledge base that explains that first name concatenated with last name is a name. It could use this knowledge to replace the sub-structure rooted by ≅ either by FirstName and LastName, since together they subsume Name, or by a nested structure Name with sub-objects FirstName and LastName. The latter is probably preferable in a data model that allows nested structures, such as XML Schema. The former is probably necessary when nested structures are not supported, as in SQL. Overall, the resolution strategy depends on the capabilities of the knowledge base and on the expressiveness of the output data model. So this activity is not captured by the generic model management operators. Instead, it should be expressed in an application-specific function.

When application-specific conflict resolution functions are used, the apply operator can help by executing a conflict resolution rule on all objects of the output of Merge. The rule tests for an object that is marked by ≅, and if so applies its action to that object and its sub-structure (knowledge-base lookup plus meta-model-specific merge). This avoids the need for the application-specific code to include logic to navigate the model.

To finish the job, the mappings map_{12} and map_{13} that are returned by Merge must be translated into view definitions. To do this, the models and mappings can no longer be regarded only as syntactic structures. Rather, they need semantics. Thus, creating view definitions requires semantic reasoning: the manipulation of expressions that explain the semantics of mappings. In Section 3.10 we explained in broad outline how to do this, though as we said there, the details are beyond the scope of this paper.

4.2 Schema Evolution

The schema evolution problem arises when a change to a database schema breaks views that are defined on it [3, 12]. Stated more precisely, we are given a base schema S_1, a set of view schemas V_1 over S_1, and a mapping map_1 that maps objects of S_1 to objects of V_1. (See Figure 12.)

For example, if S_1 and V_1 are relational schemas, then we would expect each object m of map_1 to contain a relational view definition that tells how to derive a view relation in V_1 from some of the relations in S_1; the morphisms of m would refer to the objects of S_1 and V_1 that are mentioned in m's view definition. Then, given a new version S_2 of S_1, the problem is to define a new version V_2 of V_1 that is consistent with S_2 and a mapping map_2 from S_2 to V_2.

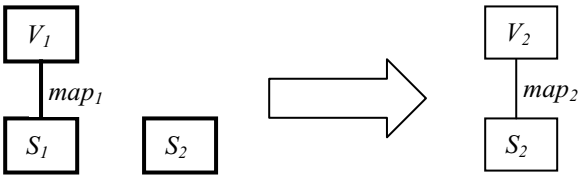


Figure 12 The schema evolution problem

We can solve this problem using model management operators as follows (Figure 13):

1. map_3 = Match(S_1, S_2). This returns a mapping between S_1 and S_2 that identifies what is unchanged in S_2 relative to S_1. If we know that S_2 is an incremental modification of S_1, then this can be done by Elementary Match. If not, then Complex Match is required.
2. $map_4 = map_1 \bullet map_3$. This is a right composition. Intuitively, each mapping object in map_4 describes a part of map_1 that is unaffected by the change from S_1 to S_2. A mapping object m in map_1 survives the composition (i.e., becomes an object of map_4) if every object in S_1 that is connected to m is also connected to some object of S_2 via map_3. If so, then m is transformed into m' in map_4 by replacing each reference from m to an object of S_1 by a reference to the corresponding objects in S_2.

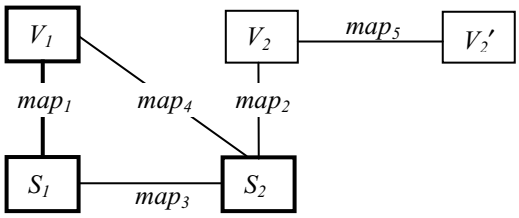


Figure 13 Result of schema evolution solution

Some objects of V_1 may now be "orphans" in the sense that they are not incident to map_4. An orphan arises because it maps via map_1 to an object in S_1 that has no corresponding object in S_2 via map_3. One way to deal with orphans is to eliminate them. Since doing this would corrupt map_1, we first make a copy of V_1 and then delete the orphans from the copy:

3. $<V_2, map_2>$ = DeepCopy(V_1, map_4). This makes a copy V_2 of V_1 along with a copy map_2 of map_4.
4. $<V_2', map_5>$ = Diff(V_2, map_2). Identify the orphans.
5. For each e in Enumerate(map_5), delete *domain(e)* from V_2. This enumerates the orphans and deletes them. Notice that we are treating map_5 as a model.

At this point we have successfully completed the task. An alternative to steps 4 and 5 is to be more selective in deleting view objects, based on knowledge about the syntax and semantics of the mapping expressions. For example, suppose the schemas and views are in the relational data model and S_2 is missing an attribute that is used to populate an attribute of a view in V_2. In the previous approach, if each view is defined by one object in map_1, then the entire view would be an orphan and deleted. Instead, we could drop the attribute from the view without dropping the entire view relation that contains it. To get this effect, we could replace Step 2 above by a right outer composition, so that all objects of map_1 are copied to map_4, even if they connect to S_1 objects that have no counterpart in S_2. Then we can write a function f that encapsulates the semantic knowledge necessary to strip out parts of a view definition and replace steps 4 and 5 by Apply(f, map_2). Thus, f gives us a way of exploiting non-generic model semantics while still working within the framework of the model management algebra.

4.3 Round-Trip Engineering

Consider a design tool that generates a compiled version of a high-level specification, such as an ER modeling tool that generates SQL DDL or a UML modeling tool that generates C++ interfaces. After a developer modifies the generated version of such a specification (e.g., SQL DDL), the modified generated version is no longer consistent with its specification. Repairing the specification is called round-trip engineering, because the tool forward-engineers the specification into a generated version after which the modified generated version is reverse-engineered back to a specification.

Stating this scenario more precisely, we are given a specification S_1, a generated model G_1 that was derived from S_1, a mapping map_1 from S_1 to G_1, and a modified version G_2 of G_1. The problem is to produce a revised specification S_2 that is consistent with G_2 and a mapping map_2 between S_2 and G_2. See Figure 14. Notice that diagrammatically, this is isomorphic to the schema evolution problem; it is exactly like Figure 12, with S_1 and S_2 replacing V_1 and V_2, and G_1 and G_2 replacing S_1 and S_2.

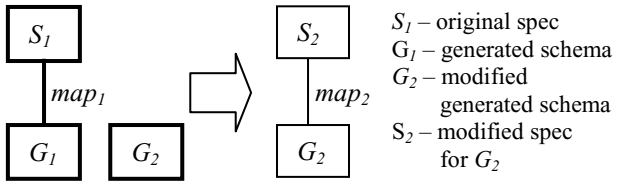


Figure 14 The round-trip engineering problem

As in schema evolution, we start by matching G_1 and G_2, composing the resulting mapping with map_1, and doing a deep copy of the mapping produced by Compose:

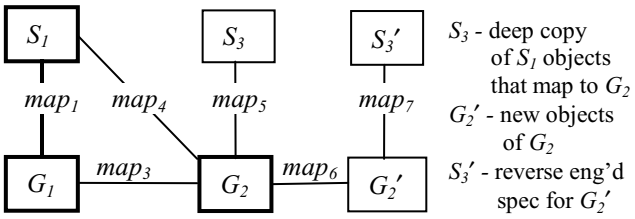
1. map_3 = Match(G_1, G_2). This returns a mapping that identifies what is unchanged in G_2 relative to G_1. Since G_2 is an incremental modification of G_1, Elementary Match should suffice. See Figure 15a.

2. $map_4 = map_1 \bullet map_3$. Mapping map_4, between S_1 and G_2, includes a copy of each object in map_1 all of whose incident G_1 objects are still present in G_2.
3. $<S_3, map_5> = \text{DeepCopy}(S_1, map_4)$. This makes a copy S_3 of S_1 along with a copy map_5 of map_4.

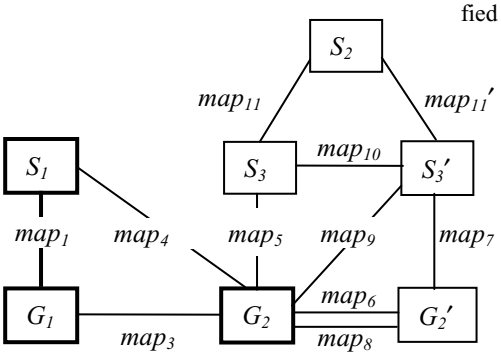
Steps 2 and 3 eliminate from the specification S_3 all objects that do not correspond to generated objects in G_2. One could retain these objects by replacing the composition in step 2 by outer composition. The remaining steps in this section would then proceed without modification.

Next, we need to reverse engineer the new objects that were introduced in G_2 and merge them with S_3. Here is one way to do it (see Figure 15a):

4. $<G_2', map_6> = \text{Diff}(G_2, map_3)$. This produces a model G_2' that includes objects of G_2 that do not participate in the mapping map_3, which are exactly the new objects of G_2, plus support objects O that are needed to keep G_2' well-formed. Mapping map_6 maps each object of G_2' not in O to the corresponding object of G_2.



(a) After Step 5



(b) After Step 8

Figure 15 Result of round-trip engineering solution

For example, suppose G_2 and G_2' are SQL schemas, and G_2' introduced a new column C into table T. In the model management representation G_2 of the schema, C is an object that is a child of object T. Since C is new, it is not connected via map_3 to G_1, so it is in the result of Diff. However, to keep G_2' connected, since C is a child of T, T is also in the result of Diff as a support object, though it is not connected to G_2 via map_6.

5. $<S_3', map_7> = \text{ModelGen}(G_2')$. In this case, ModelGen is customized to reverse engineer each object of G_2' into an object of the desired form for integration into S_2. For example, if G_2' is a SQL schema and the S_i's are ER models, then ModelGen maps each SQL column into an ER attribute, each table into either an entity type or relationship type (depending on the key structure of the table), etc.

We need to merge S_3 and S_3' into a single model S_2, which is half of the desired result. (The other half is map_2, coming soon.) To do this, we need to create a mapping between S_3 and S_3' that connects objects of S_3 and S_3' that represent the same thing. Continuing the example after step 4 above, where G_2' introduces a new column C into table T, the desired mapping should connect the reverse engineered object for T in S_3' (e.g., an entity type) with the original object for T in S_3 (e.g., the entity type that was used to generate T in G_2 in the first place). By contrast, the reverse engineered object for C in S_3' will not map to any object in S_3 because it is a new object that was introduced in G_2', and therefore was not present S_3. We can create the desired mapping by a Match followed by two compositions, after which we can do the merge, as follows (see Figure 15b):

6. $map_8 = \text{Match}(G_2, G_2')$. This matches every object in G_2' with its corresponding copy in G_2. Unlike map_6, map_8 connects to all objects in G_2', including support objects.
7. $map_9 = map_7 \bullet map_8$. This right composition creates a mapping map_9 between the objects of G_2 that are also in G_2' and their corresponding objects of S_3'. Since map_8 is incident to all objects of G_2', every object of map_7 generates a map_9 object that connects to G_2.
8. $map_{10} = map_5 \bullet map_9$. If there are mapping objects of map_5 and map_9 that connect an object of G_2 (e.g., T) to both S_3 and S_3', then those mapping objects compose and the corresponding objects of S_3 and S_3' are related by map_{10}. This should be an "inner" Compose, which only returns objects that connect to both S_3 and S_3'.
9. $<S_2, map_{11}, map_{11}'> = \text{Merge}(S_3, S_3', map_{10})$. This merges the reverse engineered objects of S_3' (which came from the new objects introduced in G_2) with S_3, producing the desired model S_2 (cf. Figure 14).

Finally, we need to produce the desired mapping map_2 between G_2 and S_2. This is the union (i.e., merge) of $map_{11} \bullet map_5$ and $map_{11}' \bullet map_9$. To see why this is what we want, recall that G_2' contains the objects of G_2 that do not map to S_3 via map_5. Mapping map_7 connects those objects to S_3', as does map_9, except on the original objects in G_2 rather than on the copies in G_2'. Hence, every object in G_2 connects to a mapping object in either map_5 or map_9.

So to start, we need to compute these compositions:

10. $map_2' = map_{11} \bullet map_5$
11. $map_2'' = map_{11}' \bullet map_9$

Next, we need the union of map_2' and map_2''. But there is a catch: an object of G_2 could be connected to objects in both map_5 and map_9. Continuing our example, table T is such an object because it is mapped to S_3 as well as reverse engineered to S_3'. Such objects have two mappings

to G_2 via the union of the compositions, which is probably not what is desired. Getting rid of the duplicates is a bit of effort. One way is to merge the mappings. To do this, we need to match map_2' and map_2'' from steps 10 and 11 to find the duplicates (which we can do because mappings are models), and then merge the mappings based on the match result. Here are the steps (not shown in Figure 15):

12. map_{12} = Match(map_2', map_2''). Objects m_2' in map_2' and m_2'' in map_2'' match if they connect to exactly the same objects of G_2 and S_2. To use this matching condition, one needs to regard the morphisms of map_2' and map_2'' as parts of each map's model; e.g., the morphisms could be available as relationships on each map's model. Using this simple match criterion, Elementary Match suffices.
13. map_2 = Merge(map_2', map_2'', map_{12}). The morphisms of map_2' and map_2'' should be merged like ordinary relationships. That is, if map_{12} connects m_2' in map_2' and m_2'' in map_2'', then Merge collapses m_2' and m_2'' into a single object m_2. Object m_2 should have only one copy of the mapping connections that m_2' and m_2'' had to G_2 and S_2.

We now have map_2 and S_2, so we're done! Cf. Figure 14.

5 Implementation

We envision an implementation of models, mappings, and model management operators on a persistent object-oriented system. Given technology trends, an object-relational system is likely to be the best choice, but an XML database system might also be suitable. The system consists of four layers:

Models and mappings – This layer supports the model and mapping abstractions, each implemented as an object-oriented structure, both on disk and heavily cached for fast navigation. The representation of models should be extensible, so that the system can be specialized to more expressive meta-meta-models. And it should be semi-structured, so that models can be imported from more expressive representations without loss of information. This layer supports:

- Models – We need the usual object-at-a-time operations on objects in models, plus GetSubmodels (of a given model) and DeleteSubmodel, where a submodel is a model rooted by an object in another model. Also Copy (deep and shallow) is supported here.
- Mappings - CreateMapping returns a model and two morphisms. GetSource and GetTarget return the morphisms of a given mapping.
- Morphisms – These are accessible and updatable like normal relationships.

Algebraic operators – This layer implements Match, Merge, Diff, Compose, Apply, ModelGen, and Enumerate. It should have an extension mechanism for handling semantics, such as an expression manipulation engine as discussed in Section 3.10.

Model-driven generator of user interface – Much like an advanced drawing tool, one can tag meta-model objects with descriptions of objects and their behavior (e.g., a table definition is a blue rectangle and a column definition is a line within its table's rectangle).

Generic tools over models and mappings – browser, editor, catalog, import/export, scripting.

6 Related Work

Although the model management approach is new, much of the existing literature on meta data management offers either algorithms that can be generalized for use in model management or examples that can be studied as challenges for the model management operators. This literature is too large to cite here, but we can highlight a few areas where there is obvious synergy worth exploring. Some of them were mentioned earlier: schema matching (see the survey in [23]); schema integration [1,8,15,25], which is both an example and a source of algorithms for Match and Merge; and adding semantics to mappings [7,17,21,27]. Others include:

- Data translation [24];
- Differencing [11,19,26]; and
- EER-style representations and their expressive power, which may help select the best representation for models and mappings [2,14,15,18,20].

7 Conclusion

In this paper, we described model management — a new approach to manipulating models (e.g., schemas) and mappings as bulk objects using operators such as Match, Merge, Diff, Compose, Apply, Copy, Enumerate, and ModelGen. We showed how to apply these operators to three classical meta data management problems: schema integration, schema evolution, and round-trip engineering. We believe these example solutions strongly suggest that an implementation of model management would provide major programming productivity gains for a wide variety of meta data management problems. Of course, to make this claim compelling, an implementation is needed. If successful, such an implementation could be the prototype for a new category of database system products.

In addition to implementation, there are many other areas where work is needed to fully realize the potential of this approach. Some of the more pressing ones are:

- Choosing a representation that captures most of the constructs of models and mappings of interest, yet is tractable for model management operators.
- More detailed semantics of model management operators. There is substantial work on Match. Merge, Compose, and ModelGen are less well developed.
- A mathematical semantics of model management. The beginnings of a category-theoretic approach appears in [1], but there is much left to do. A less abstract analysis that can speak to the completeness of the set

of operators would help define the boundary of useful model management computations.

- Mechanisms are needed to fill the gap between models and mappings, which are syntactic structures, and their semantics, which treat models as templates for instances and mappings as transformations of instances. Various theories of conjunctive queries are likely to be helpful.
- Trying to apply model management to especially challenging meta data management problems, to identify limits to the approach and opportunities to extend it.

This is a broad agenda that will take many years and many research groups to develop. Although it will be a lot of work, we believe the potential benefits of the approach make the agenda well worth pursuing.

Acknowledgments

The ideas in this paper have benefited greatly from my ongoing collaborations with Suad Alagic, Alon Halevy, Renée Miller, Rachel Pottinger, and Erhard Rahm. I also thank the many people whose discussions have stimulated me to extend and sharpen these ideas, especially Kajal Claypool, Jayant Madhavan, Sergey Melnik, Peter Mork, John Mylopoulos, Arnie Rosenthal, Elke Rundensteiner, Aamod Sane, and Val Tannen.

8 References

1. Alagic, S. and P.A. Bernstein, "A Model Theory for Generic Schema Management," Proc. DBPL 2001, Springer Verlag LNCS.
2. Atzeni, Paolo and Riccardo Torlone: Management of Multiple Models in an Extensible Database Design Tool. EDBT 1996: 79-95
3. Banerjee, Jay, Won Kim, Hyoung-Joo Kim, Henry F. Korth: Semantics and Implementation of Schema Evolution in Object-Oriented Databases. SIGMOD Conference 1987: 311-322
4. Beeri, C. and T. Milo: Schemas for integration and translation of structured and semi-structured data. ICDT, 1999: 296-313,.
5. Bernstein, P.A.: Generic Model Management – A Database Infrastructure for Schema Manipulation. Springer Verlag LNCS 2172, CoopIS 2001: 1-6.
6. Bernstein, Philip A., Alon Y. Halevy, and Rachel A. Pottinger. A vision of management of complex models. SIGMOD Record 29(4):55-63 (2000).
7. Bernstein, Philip A., Erhard Rahm: Data Warehouse Scenarios for Model Management. ER 2000: 1-15.
8. Biskup, J. and B. Convent. A formal view integration method. SIGMOD 1986: 398-407.
9. Buneman, P., S.B. Davidson, A. Kosky. Theoretical aspects of schema merging. EDBT 1992: 152-167.
10. Cattell, R.G.G., D. K. Barry, M. Berler, J. Eastman, D. Jordan, D. Russell, O. Schadow, T. Stanienda, and F. Velez, editors: *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann Publishers, 2000.
11. Chawathe, Sudarshan S. and Hector Garcia-Molina: Meaningful Change Detection in Structured Data. SIGMOD 1997: 26-37.
12. Claypool, K. T., J. Jin, E. A. Rundensteiner: SERF: Schema Evolution through an Extensible, Re-usable and Flexible Framework. CIKM 1998: 314-321.
13. Claypool, K.T., E.A. Rundensteiner, X. Zhang, H. Su, H.A. Kuno, W-C Lee, G. Mitchell: Gangam - A Solution to Support Multiple Data Models, their Mappings and Maintenance. SIGMOD 2001
14. Hull, Richard and Roger King: Semantic Database Modeling:Survey, Applications, and Research Issues. ACM Computing Surveys 19(3): 201-260 (1987)
15. Larson, James A., Shamkant B. Navathe, and Ramez Elmasri. A theory of attribute equivalence in databases with application to schema integration. Trans. on Soft. Eng. 15(4):449-463 (April 1989).
16. Madhavan, J., P. A. Bernstein, P. Domingos, A.Y. Halevy: Representing and Reasoning About Mappings between Domain Models. 18th National Conference on Artificial Intelligence (AAAI 2002).
17. Miller, R.J., L. M. Haas, M. A. Hernández: Schema Mapping as Query Discovery. VLDB 2000: 77-88.
18. Miller, R. J., Y. E. Ioannidis, Raghu Ramakrishnan: Schema equivalence in heterogeneous systems: bridging theory and practice. Information Systems 19(1): 3-31 (1994)
19. Myers, E.: An O(ND) Difference Algorithm and its Variations. Algorithmica 1(2): 251-266 (1986).
20. Mylopoulos, John, Alexander Borgida, Matthias Jarke, Manolis Koubarakis: Telos: Representing Knowledge About Information Systems. TOIS 8(4): 325-362 (1990).
21. Popa, Lucian, Val Tannen: An Equational Chase for Path-Conjunctive Queries, Constraints, and Views. ICDT 1999: 39-57.
22. Pottinger, Rachel A. and Philip A. Bernstein. Creating a Mediated Schema Based on Initial Correspondences. IEEE Data Engineering Bulletin, Sept. 2002.
23. Rahm, Erhard and Philip A. Bernstein. A survey of approaches to automatic schema matching. VLDB J. 10(4):334-350 (2001).
24. Shu, Nan C., Barron C. Housel, R. W. Taylor, Sakti P. Ghosh, Vincent Y. Lum: EXPRESS: A Data EXtraction, Processing, amd REStructuring System. TODS 2(2): 134-174 (1977).
25. Spaccapietra, Stefano and Christine Parent. View integration: A step forward in solving structural conflicts. TKDE 6(2): 258-274 (April 1994).
26. J. T-L. Wang, D. Shasha, G. J-S. Chang, L. Relihan, K. Zhang, G. Patel: Structural Matching and Discovery in Document Databases. SIGMOD 1997: 560-563
27. Yan, Ling-Ling, Renée J. Miller, Laura M. Haas, Ronald Fagin: Data-Driven Understanding and Refinement of Schema Mappings. SIGMOD 2001.