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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

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
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:

- 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.

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.

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 \( \langle o_1, o_2 \rangle \) 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, \( Map_{oe} \) 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 repre-
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 (\(\cong\)). 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.

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].

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\((Employee, Map_{ee})\).

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 \(M_1\).
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 $m_1$ 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].
To compute the composition, for each object $m_2$ in $map_2$, we identify each object $m_1$ in $map_1$ where $\text{range}(m_1) \cap \text{domain}(m_2) \neq \emptyset$, which means that $\text{range}(m_1)$ 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 $m_{1i}$, ..., $m_{1n}$ in $map_1$ together supply all of domain($m_2$), and each $m_{1i}$ ($1 \leq i \leq n$) supplies at least one object to domain($m_2$). That is, $\bigcup_{i=1}^{n} \text{range}(m_{1i}) \supseteq \text{domain}(m_2)$ and $(\text{range}(m_{1i}) \cap \text{domain}(m_2)) \neq \emptyset$ for $1 \leq i \leq n$. Then $m_2$ should generate an output object $m_3$ in $map_3$ such that $\text{range}(m_3) = \text{range}(m_2)$ and $\text{domain}(m_3) = \bigcup_{i=1}^{n} \text{domain}(m_{1i})$.

For example, in Figure 8, range(4) and range(5) can supply all of domain(11). That is, range(4) $\cup$ range(5) = \{7, 8, 9\} $\supseteq$ domain(11) = \{7, 9\}. Then object 11 should generate an output object $m_3$ in $map_3$ such that range($m_3$) = range($m_2$) = \{13\} and domain($m_3$) = domain($m_2$) $\cup$ domain(5) = \{1, 2\}.

<table>
<thead>
<tr>
<th>$M_1$</th>
<th>$map_1$</th>
<th>$M_2$</th>
<th>$map_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
</tbody>
</table>

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($m_1$) $\cap$ domain($m_2$) $\neq \emptyset$ (\{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 $M_3$ and $M_2$ 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($m_1$) $\cap$ domain($m_3$) $\neq \emptyset$.

3. (Define domains) For each $m_3$ in $map_3$:
   a. if $\bigcup_{m_1 \in \text{Input}(m_3)} \text{range}(m_1) \supseteq \text{domain}(m_3)$, then set $\text{domain}(m_3) = \bigcup_{m_1 \in \text{Input}(m_3)} \text{domain}(m_1)$.
   b. else if $m_1$ is not needed as a support object (because none of its descendants satisfies (3a)), then delete it, else set domain($m_3$) = range($m_3$) = $\emptyset$.

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 $map_3$).

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($m_3$) = $\emptyset$.”

A definition of composition that allows a more flexible choice of inputs to $m_3$ 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.

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).

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].

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.

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.

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.

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 walkthroughs 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].

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.

The problem is to create: a schema $S_3$ that represents all of the information expressed in two given database...
schems, $S_1$ 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.

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

1. $map_{12} = \text{Match}(S_1, S_2)$. 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. $<S_3, map_{13}, map_{23}> = \text{Merge}(S_1, S_2, map_{12})$. 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\textsubscript{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 11 shows the result of merging Emp and Employee with respect to Map\textsubscript{ee}. (The mappings between Emp’ and Emp and between Emp’ and Employee are omitted, to avoid cluttering the figure.) Since Map\textsubscript{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\textsubscript{ee}, they are simply copied to the output. Since Map\textsubscript{ee} says that Name is similar to FirstName and LastName, these objects are partially integrated in $S_12$ under an object labeled $\equiv$, which is a placeholder for an expression that relates Name to FirstName and LastName.

The sub-structure rooted by “$\equiv$” 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 $\equiv$ 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 $\equiv$, 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.

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 $S_j$, a set of view schemas $V_i$ over $S_j$, and a mapping $map_l$ that maps objects of $S_j$ to objects of $V_i$. (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 $V_2$.

![Diagram](image)

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

1. $map_3 = \text{Match}(S_1, S_2)$. 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. $map_4 = map_1 \circ map_3$. 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 $map_3$) 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 $S_2$.

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. $<V_2', map_2> = \text{ DeepCopy}(V_1, map_4)$. This makes a copy $V_2'$ of $V_1$ along with a copy $map_2$ of $map_4$.

4. $<V_2', map_2> = \text{ Diff}(V_2, map_3)$. Identify the orphans.

5. For each $e$ in $\text{Enumerate}(map_3)$, delete $\text{domain}(e)$ from $V_2$. This enumerates the orphans and deletes them. Notice that we are treating $map_3$ 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 $S_1$ is missing an attribute that is used to populate an attribute of a view in $V_1$. 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 $\text{Apply}(f, map_2)$. Thus, $f$ gives us a way of exploiting non-generic model semantics while still working within the framework of the model management algebra.

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-enginereers 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 $G_2$ replacing $G_2$. 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. $map_3 = \text{Match}(G_1, G_2)$. 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. $\text{map}_4 = \text{map}_1 \cdot \text{map}_3$. Mapping $\text{map}_4$, between $S_1$ and $G_2$, includes a copy of each object in $\text{map}_1$ all of whose incident $G_1$ objects are still present in $G_2$. 

3. $<S_3, \text{map}>' = \text{DeepCopy}(S_3, \text{map}_3)$. This makes a copy of $S_3$ along with a copy $\text{map}_3$ of $\text{map}_1$. 

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.

4. $<G_2', \text{map}>' = \text{Diff}(G_2, \text{map}_3)$. This produces a model $G_2'$ that includes objects of $G_2$ that do not participate in the mapping $\text{map}_3$, which are exactly the new objects of $G_2$ plus support objects $O$ that are needed to keep $G_2'$ well-formed. Mapping $\text{map}_3$ maps each object of $G_2'$ not in $O$ to the corresponding object of $G_2$.

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 $\text{map}_3$ 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 $\text{map}_3$.

5. $<S_3', \text{map}>' = \text{ModelGen}(G_2')$. In this case, ModelGen is customized to reverse engineer each object of $G_2'$ into an object of the desired form for integration into $S_3$. For example, if $G_2'$ is a SQL schema and the $S_3'$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 $S_1$ and $S_1'$ into a single model $S_2$, which is half of the desired result. (The other half is $\text{map}_3$, coming soon.) To do this, we need to create a mapping between $S_1$ and $S_1'$ that connects objects of $S_1$ and $S_1'$ 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 $S_1'$ (e.g., an entity type) with the original object for T in $S_1$ (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 $S_1'$ will not map to any object in $S_1$ because it is a new object that was introduced in $G_2'$, and therefore was not present $S_1$. 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. $\text{map}_8 = \text{Match}(G_2, G_2')$. This matches every object in $G_2'$ with its corresponding copy in $G_2$. Unlike $\text{map}_6$, $\text{map}_8$ connects to all objects in $G_2'$, including support objects.

7. $\text{map}_9 = \text{map}_7 \cdot \text{map}_8$. This right composition creates a mapping $\text{map}_9$ between the objects of $G_1$ that are also in $G_2'$ and their corresponding objects of $S_1'$. Since $\text{map}_8$ is incident to all objects of $G_2'$, every object of $\text{map}_7$ generates an object that connects to $G_2$.

8. $\text{map}_{10} = \text{map}_9 \cdot \text{map}_6$. If there are mapping objects of $\text{map}_5$ and $\text{map}_6$ that connect an object of $G_2$ (e.g., T) to both $S_1$ and $S_1'$, then those mapping objects compose and the corresponding objects of $S_1$ and $S_1'$ are related by $\text{map}_{10}$. This should be an “inner” Compose, which only returns objects that connect to both $S_1$ and $S_1'$.

9. $<S_3, \text{map}_{11}, \text{map}_{11}'> = \text{Merge}(S_3, S_3', \text{map}_{10})$. This merges the reverse engineered objects of $S_1'$ (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 $\text{map}_2$ between $G_2$ and $S_2$. This is the union (i.e., merge) of $\text{map}_{11} \cdot \text{map}_3$ and $\text{map}_{11}' \cdot \text{map}_6$. To see why this is what we want, recall that $G_2'$ contains the objects of $G_2$ that do not map to $S_1$ via $\text{map}_3$. Mapping $\text{map}_7$ connected those objects to $S_1'$, as does $\text{map}_8$, 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 $\text{map}_3$ or $\text{map}_6$.

So to start, we need to compute these compositions:

10. $\text{map}_{2'} = \text{map}_{11} \cdot \text{map}_3$
11. $\text{map}_{2''} = \text{map}_{11}' \cdot \text{map}_6$

Next, we need the union of $\text{map}_{2'}$ and $\text{map}_{2''}$. But there is a catch: an object of $G_2$ could be connected to objects in both $\text{map}_3$ and $\text{map}_6$. Continuing our example, table T is such an object because it is mapped to $S_1$ as well as reverse engineered to $S_1'$. 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 $map_1'$ 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. $map_{12} = \text{Match}(map_1', map_2'')$. Objects $m_1'$ in $map_1'$ 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 $map_1'$ 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. $map_2 = \text{Merge}(map_1', map_2'', map_{12})$. The morphisms of $map_1'$ and $map_2''$ should be merged like ordinary relationships. That is, if $map_{12}$ connects $m_1'$ in $map_1'$ and $m_2''$ in $map_2''$, then Merge collapses $m_1'$ and $m_2''$ into a single object $m_2$. Object $m_2$ should have only one copy of the mapping connections that $m_1'$ and $m_2''$ had to $G_2$ and $S_2$.

We now have $map_2$ and $S_3$, so we’re done! Cf. Figure 14.

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:

- 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.

- 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.

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].

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.

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.

References

2. Atzeni, Paolo and Riccardo Torlone: Management of Multiple Models in an Extensible Database Design Tool. EDBT 1996: 79-95
4. Beeri, C. and T. Milo: Schemas for integration and translation of structured and semi-structured data. ICDT, 1999: 296-313,