6 General Schema Manipulation Operators

In the last chapter, we saw techniques for taking two schemas and finding matches and mappings between them. Mappings and matches are directly useful for translating data from one schema to another, which is the focus of a good deal of this book. However, we may wish to do more with schemas than simply map data from one to the other. For instance, it may be useful to take two schemas and create a single merged schema that can represent all of their data. We may wish to translate a schema from one data model to another (e.g., relational to XML). Given three schemas A, B, and C and mappings $A \rightarrow B$ and $B \rightarrow C$, we may wish to compose the mappings to create a mapping $A \rightarrow C$.

In general, one can define an algebra of generic operators that could be applied to schemas and mappings across a broad range of applications. The operators would take as input schemas and/or mappings, and return schemas or mappings. Such an algebra would be analogous to the algebra we have for operating on data itself and which powers all modern query processors. This algebra has been termed the set of *model management operators*: model is a more general term referring to a schema, a set of classes in an object-oriented language, an ontology, etc.

We illustrate the use of a generic set of model management operators in the context of data integration with the following example.

Example 6.1

Consider the process of setting up a data integration application and maintaining it over time (see Figure 6.1). Suppose that our goal is to build a data integration application that spans two sources, S_1 and S_2.

We start by creating a mediated schema that contains the information in S_1 and S_2 and mappings from each of the sources to the mediated schema. We could proceed in two steps. First, we use the model management match operator (which does schema matching and mapping as discussed in Chapter 5) to create a schema mapping, M_{12}, between S_1 and S_2 (see Figure 6.1(1)). Second, we create a mediated schema, G, by applying a schema merge operator to schemas S_1 and S_2 and the mapping M_{12} (Figure 6.1(2)). The merge operator will create a schema that includes all the information in S_1 and S_2, but that combines any information present in both schemas so it is not duplicated. The merge operator will also create mappings from S_1 and S_2 to G, M_{1G}, and M_{2G}.

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00006-5

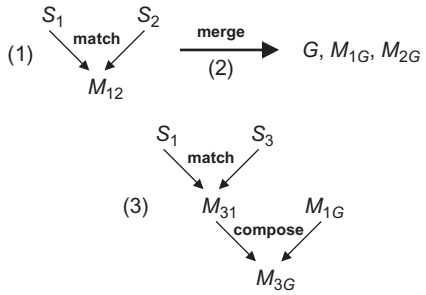

FIGURE 6.1 Common tasks in data integration can be modeled as sequences of model management operators. To set up an integration application involving S_1 and S_2 we first create a mapping between them (1), and then merge the two schemas with the **merge** operator to create the mediated schema, G (2). To add a new source S_3, we can first **match** it to one of the existing ones (S_1) and **compose** the mappings to obtain a mapping from S_3 to G (3).

Now suppose we need to add another source, S_3, to our system, and suppose that S_3 is very similar to S_1. We could add S_3 in two steps: (1) since S_3 and S_1 are relatively similar, it would be easy to use match to create a mapping, M_{31}, between S_1 and S_3, and (2) use a compose operator to create a mapping M_{3G} by composing mappings M_{31} and M_{1G} (Figure 6.1(3)).

Data integration, whether virtual or using a warehouse, is a key application for the model management operators. However, there are other tasks in which the generic operators are also useful:

- **Message-passing systems**: A message-passing system expects messages to be passed in particular formats (e.g., a particular XML schema), but the data itself typically reside in back-end systems such as relational databases. Hence, the data need to be transformed from the model and schema in which it is stored into the target message format. On the recipient side, the reverse transformation needs to be performed. Web services and message queueing middleware are commonly used instances of message passing systems.
- **Wrapper generation tools** (Chapter 9): A wrapper needs to transform the data from one data model to another and possibly apply a schema mapping. For example, a common scenario is to translate between an object model in a programming language into a relational schema.

A significant amount of labor in enterprises is spent on manually performing these operations in the tasks above and others. Developing a generic set of model management operators can significantly simplify these tasks.

6.1 Model Management Operators

We begin by describing the basic terminology of model management and presenting its main operators. Model management operators manipulate two kinds of objects:

models and *mappings* between models. Models describe the data representation used in a particular application (e.g., a relational schema of a data source). Models are specific instances of a particular representation system, which is called a *meta-model*: for instance, the relational data model, the XML data model, and the Java object model. Common language encodings of meta-models include relational DDLs, XML schemas, Java class definitions, entity-relationship models, and RDF. For example, the relational meta-model specifies that a schema include a set of relations, each of which consists of a set of typed attributes. In this chapter we will encode all models as directed graphs whose nodes are elements. The edges in the graph represent binary relationships between the elements, such as is-a; has-a and type-of represent the different constructs of the meta-models. For example, a relation element will have a has-a relationship with the attributes it contains.

Mappings describe the semantic relationship between a pair of models. Mappings are used to translate data instances from one model into another or to reformulate queries posed on one model into queries on another. Mappings can be described in a variety of formalisms, such as the GLAV formalism we described in Section 3.2.4.

The model management operators should be composable so that sequences of operators can be executed as scripts on behalf of the user. Some of the operators of model management are completely automatic (e.g., compose), while others might require some human feedback (e.g., match and merge). The goal of model management is not to completely automate these tasks, but to remove as much of the human labor as possible. We describe the most commonly studied operators below, and toward the end of the chapter we describe the ultimate vision of building a general-purpose *model management system* that is an "engine" for these operators.

Match: takes as input two models S_1 and S_2 and produces a schema mapping, M_{12}, between them. As explained in Chapter 3, a mapping expresses a constraint on which pairs of instances of S_1 and S_2 are consistent with each other. The match operator assumes that the two input models are in the same meta-model. The system may solicit some feedback from the user in the process of creating the model. Chapter 5 describes techniques for the match operator.

Compose: takes as input a mapping M_{12} between models S_1 and S_2 and a mapping M_{23} between models S_2 and S_3 and outputs a direct mapping M_{13} between S_1 and S_3. As discussed in Section 17.5, the algorithms for the compose operator are highly dependent on the particular meta-model.

Merge: takes as input two models S_1 and S_2 and a mapping M_{12} between S_1 and S_2. merge outputs a combined model S_3 that contains all the information modeled by S_1 and S_2 but but does not duplicate information that is present in both models. We discuss merge in Section 6.2.

ModelGen: takes as input a model S_1 in a given meta-model $\mathcal{M}_1$ and creates a model S_2 in a different meta-model, $\mathcal{M}_2$. For example, modelGen may translate a relational model into XML or an Java object model into a relational schema. We describe modelGen in Section 6.3.

Invert: takes as input a mapping M_{12} from S_1 to S_2 and outputs a mapping M_{21} from S_2 to S_1. The mapping M_{21} should be able to take an instance of S_2 and compute an instance of S_1. We discuss invert in Section 6.4.

We now discuss these model management operators in more detail. The bibliographic notes contain references to other works in the literature. It should be noted that our list of operators is not exhaustive. For instance, there have been proposals for a diff operator that returns the difference between two schemas.

6.2 Merge

The merge operator takes as input two models, S_1 and S_2, and a mapping M_{12} between them, and outputs a model that contains the union of the information in S_1 and S_2, but collapses the information that is present in both. The mapping M_{12} determines which information exists in both models and which does not.

Figure 6.2 depicts an example that illustrates some of the decisions that need to be made when we merge two models. Figure 6.2 shows two models of actor entities and a mapping between them. We depict each of the models as a graph of elements. The solid lines indicate the has-a relationship between the actor element and its subelements. The dashed lines depict the attribute correspondences that make up the mapping between the two models. The graph representation is agnostic of a particular meta-model. Hence, for example, the graph may be a depiction of a relational schema of an actor table or an XML schema where the element actor has the specified subelements.

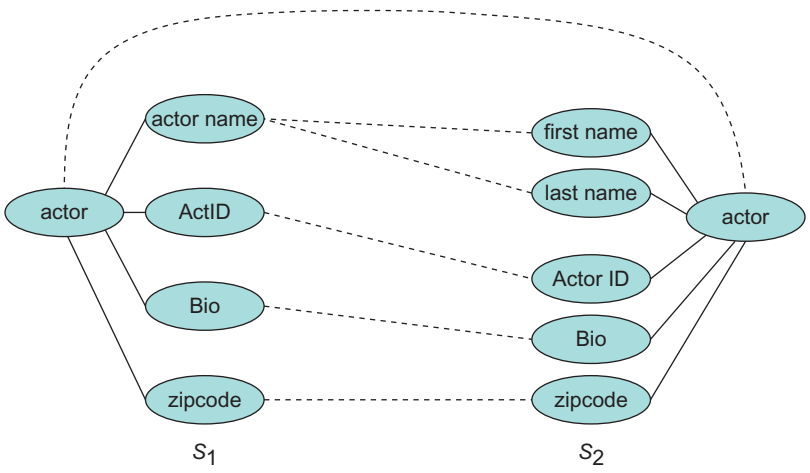

FIGURE 6.2 Two models of actors that need to be merged. The models are depicted as graphs of elements. The solid lines indicate the **has-a** relationship between the **actor** element and its subelements. The dashed lines depict the attribute correspondences that make up the mapping between the two models.

The merge operator illustrates the challenges involved in defining a *generic* model management operator. At first glance, the merge operator needs to address three kinds of conflicts. We consider each in turn.

The first conflict that the merge operator needs to consider is a *representational conflict*. In our example, the name of the actor is represented as a single field in one model and as a pair of fields in the other. This conflict is not inherent to merge and really depends on the mapping between the two models. For example, the mapping will specify whether first name and last name are subelements of name, or whether the concatenation of first name and last name are equivalent to name. In fact, different mappings between the input models can result in different merged models. This is the reason that merge is defined to take a mapping as input. In addition to the mapping, the input to merge may include rules stating that one model should be preferred over the other when we encounter representational conflicts. For example, if we merge the schema of a data source into the mediated schema, we prefer that the mediated schema changes as little as possible.

The second kind of conflict that merge needs to deal with is a meta-model conflict. For example, suppose that S_1 in our example is an XML schema and S_2 and the merged model are relational. Since relational schemas do not have the notion of a subattribute, there is no obvious way of mapping S_1 into the merged model.

The mismatch at the meta-model level is also not inherent to the merge operator. In fact, the problem of translating models from one meta-model to another occurs quite frequently. For that reason, model management includes a separate operator, modelGen, to do the specific task of converting between meta-models.

Finally, the third kind of conflict is one that is fundamental to merge and hence the focus of the generic merge algorithms proposed in the literature. The fundamental conflicts are considered with respect to the common representation we use to encode all of the models (and which must be able to capture all of the aspects of their meta-models)—sometimes this is termed the *meta-meta-model*. The meta-meta-model is the representation system used by the model management operators. In our running example, a fundamental conflict involves the modeling of zipcode in the merged model. Suppose that in S_1 zipcode is an integer, while in S_2 it is a string. Our meta-meta-model only allows attributes to have a single type, and therefore we have a conflict.

The main thrust of algorithms for merge have been on identifying fundamental conflicts and devising rules for resolving them. We give a couple of examples below.

Cardinality constraints: A cardinality constraint on a relationship R in the meta-meta-model restricts the number of elements, y, for which $R(x, y)$ holds for a given x. For example, in the relational model, the cardinality constraint requires that every element be the origin of at most one type-of relationship (i.e., have a single type). Note that cardinality constraints can provide either lower or upper bounds on the cardinality.

One solution to a violation of such a constraint is to create a new type that inherits from more than one. For example, we would create a new type in our meta-meta

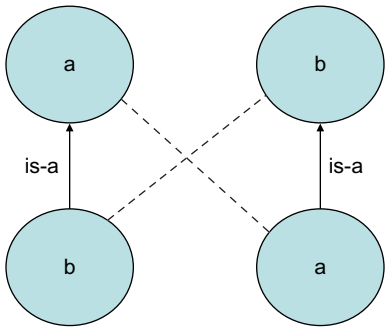

FIGURE 6.3 A cycle resulting from a merge.

model that inherits from both string and integer, and we would declare the attribute zipcode to be of the new type.

Acyclicity constraints: A meta-meta-model may require that certain relationships be acyclic. For example, the is-a relationship is required to be acyclic in our meta-meta-model. However, consider the simple example in Figure 6.3. In the schema on the left b is said to be a subset of a, while in the schema on the right the opposite is-a relationship is asserted. If we merge the two models and inherit both of these relationships, we will get an is-a cycle between a and b.

A typical solution to resolving cycle constraints is to collapse all the elements in the cycle into one element in the merged model. In our example, we would merge a and b into a single element. Of course, we may have a domain-specific rule that specifies a different solution.

6.3 ModelGen

The modelGen operator takes a model in one meta-model and translates it into another meta-model. Translations between meta-models are ubiquitous in model management scenarios because we typically handle data from a variety of sources. Other model management operators, such as match and merge, often assume that modelGen has been applied to their inputs or that it will be applied to their result if needed. The most common scenarios of modelGen are translating between XML and relational models or between object models of a programming languages (such as Java or C#) and a relational schema.

Figure 6.4 depicts an example that demonstrates some of the challenges that modelGen needs to address. The figure shows two models of companies and their suppliers. On the left, the figure shows the source model, a set of Java class definitions, that is the input to modelGen. Our goal is to generate the relational schema shown on the right of the figure.

The challenges for modelGen arise because some of the constructs of the source meta-model are not supported in the target meta-model. In our example, we have the following difficulties:

```
public class Company {
   public string name;
}

public class Supplier
     extends Company {
   public item[] parts;
}


public class Item {
   public string ISBN;
   public int Cost;
}
```

```
CREATE TABLE Company(
 Name varchar(50),
 oid int NOT NULL PRIMARY KEY)

CREATE TABLE Supplier(
 oid int NOT NULL PRIMARY KEY,
 isSameAs int NOT NULL UNIQUE
 FOREIGN KEY REFERENCES Company
    (oid))

CREATE TABLE PartsArray(
 Supplier int NOT NULL
 FOREIGN KEY REFERENCES Company
    (oid),
 itemISBN varchar(50),
 itemCost int)
```

FIGURE 6.4 The operator **modelGen** will translate the Java class hierarchy on the left to the relational schema on the right.

1. To begin, since there is no notion of classes in SQL, the target schema needs to encode classes as relational tables.
2. Furthermore, SQL does not support inheritance. In our example, the class `Supplier` is a subclass of `Company`. The target schema needs to encode inheritance in relations (using one of several known encodings).
3. Finally, SQL does not support array types. However, in the input model, the class `Supplier` has an attribute `parts` that is an array of items.

To translate the Java model into a relational schema, modelGen needs to perform the following transformations.

T1. Create relational structures from classes. For example, instead of a class `Company`, the resulting schema will have a table Company with an attribute denoting an object ID and the name as a field.

T2. The inheritance relation between `Company` and `Supplier` is represented using vertical partitioning. Specifically, the resulting schema will include a table Supplier that contains the attributes that are specific to `Supplier`. Each supplier will also be represented in the Company table, and there is a foreign key relationship from the Supplier table to the Company table. Therefore, getting all the information about a supplier requires accessing attributes from both the Company and Supplier tables.

T3. The `parts` array is represented as a join relationship. The table PartsArray includes a tuple for every (supplier, part) pair. The supplier is represented as an attribute (Supplier) that is a foreign key into the Company table, and the part is represented by its two attributes, ISBN and cost.

Continuing with the goal of developing generic model management operators, the approach taken to modelGen is based on identifying a *super-model*. The super-model is a meta-model that includes all the features of the meta-models we expect to encounter. (Clearly, there cannot be a super-model that includes all features, so the super-model is designed with a set of meta-models in mind.) The super-model knows which of its constructs are supported in each of the possible input and output meta-models. For example, the super-model will know that relational schemas include relations and their attributes, while object-oriented schemas are composed of classes with inheritance.[1]

The modelGen operator receives as input a model M in meta-model $\mathcal{M}_1$ and an output meta-model, $\mathcal{M}_2$, into which to translate M. Algorithms for modelGen will proceed in the following three steps:

Step 1: Translate the input model M into the super-model. Denote the result by M'.
Step 2: While the model M' includes a construct that is not supported in the output meta-model $\mathcal{M}_2$, apply transformations that remove the unsupported constructs.
Step 3: If Step 2 was successful in removing all the unsupported constructs, translate M' into $\mathcal{M}_2$.

Hence, the crux of the modelGen algorithm is to express the required transformations, such as T1-T3 above, in the super-model. An additional challenge that the modelGen operator needs to address is to generate a transformation that takes the data in an instance of M in $\mathcal{M}_1$ and produces an instance of it in $\mathcal{M}_2$.

6.4 Invert

In some contexts, such as data exchange, schema mappings are considered to be *directional*, defining a transformation from source schema to a target schema. In these contexts, an important question that arises is whether we can invert the schema mapping. An inverse schema mapping should be able to take an instance of the target and produce the appropriate instance of the source schema. Algorithms for inverting schemas are highly dependent on the precise formalism for specifying mappings. In this section we assume a relational meta-model and mappings that are specified as tuple-generating dependencies (which are an equivalent formalism to GLAV formulas described in Section 3.2.4).

One of the main challenges is to find a definition of invert that is theoretically sound and useful in practice. Let us consider a few definitions.

As a first definition, consider two schemas, S and T, and let M be a mapping from S to T. Recall that M defines a binary relation specifying which pairs of models of S and T are consistent with each other w.r.t. the mapping. Specifically, M defines a relation (I,J), where I is an instance of S and J is an instance of T. When (I,J) is in the relation defined

[1] Note the distinction between the super-model and the meta-meta-model. The super-model is a meta-model that includes the features of many other meta-models. The meta-meta-model is a language for describing meta-models.

by M, we say that $M \models (I,J)$. Hence, a seemingly natural definition is the following. The inverse of M, M^{-1}, is the mapping that defines the relation (J,I), where $M \models (I,J)$.

However, it is easy to see that under this definition, we cannot find inverse mappings that can be expressed as tuple-generating dependencies. Any relation that is defined by a set of tuple-generating dependencies is guaranteed to be *closed down* on the left and *closed up* on the right. Specifically, if I' is a subinstance of I (i.e., I' is a subset of the tuples of I) and J is a subset of the tuples of J', then $M \models (I,J)$ implies that $M \models (I',J')$. But this would mean that the inverse relation would be closed down on the right and closed up on the left, and hence cannot be expressed by tuple-generating dependencies.

A second definition of inverse mapping is based on mapping composition. Specifically, we define M' to be the inverse of M if the composition of M with M' produces the identity mapping. However, this definition turns out to be rather limiting. In particular, it can be shown that a mapping M has an inverse if and only if the following holds: whenever I_1 and I_2 are two distinct instances of S, then their targets under M are also distinct sets. As a consequence, even simple mappings such as the following two mappings do not have inverses:

M_1: P(x,y) → Q(x)
M_2: P(x,y,z) → Q(x,y) ∧ R(y,z)

A more useful definition is based on *quasi-inverses*. Quasi-inverses are also based on the composition being the identity mapping, but in a slightly more relaxed way. Consider a more relaxed equivalence relationship between instances, defined as follows. We define $I_1 \approx I_2$ w.r.t. a mapping M if for every target instance J, $M \models (I_1,J)$ if and only if $M \models (I_2,J)$.

Example 6.2

Consider the mapping M_1 above. Let I_1 be the instance that includes only the tuple $P(1,2)$, and I_2 be the instance that includes only the tuple $P(1,3)$. They are equivalent w.r.t. the mapping M_1.

We now define a mapping M' to be a quasi-inverse of M if the composition of M and M' maps every instance I of S to an instance I' of S, such that $I \approx I'$. The following example illustrates the difference between inverses and quasi-inverses.

Example 6.3

The following mapping is a quasi-inverse of M_1 above:

M_1': Q(x) → ∃ y P(x,y).

Consider the instance I_1 of S that includes the tuple $P(1,2)$. Applying M_1 to I_1 results in the instance J_1 of the target that includes $Q(1)$.

Now, if we apply M_1' to J_1 we'll get the instance I_2 of the source with the tuple $P(1,A)$, where A is an arbitrary constant.

The instances I_1 and I_2 are different from each other, and therefore M_1' could not be considered a strict inverse of M_1. However, $I_1 \approx I_2$, and in fact, the same would hold if we apply M_1 and then M_1' to any instance of S, and therefore M_1' is a quasi-inverse of M_1.

In a sense, a quasi-inverse precisely restores the contents of the source that were relevant to the target. In our example, only the first argument of the relation P is relevant to the target.

Remark 6.1 (Inverses and Data Integration). As noted earlier, the invert operator is motivated in contexts where schema mappings are assumed to be directional. In our discussion of mappings in Chapter 3 we did not assume that mappings are directional. We focused on the relation that the mappings define between the source instances and the mediated schema instances. However, much of the work on query answering in data integration is very related to inverting mappings. In particular, when source descriptions are described in Local-as-View (see Section 3.2.3), the algorithms for answering queries effectively invert the view definitions in order to produce tuples of the mediated schema. One of the key differences is that the algorithms for answering queries described in Chapter 3 invert the views only with respect to the particular query posed to the data integration system. □

6.5 Toward Model Management Systems

One of the original goals underlying the study of the various model management operators was to build a general-purpose model management system that operates on schemas and mappings. This system would support the creation, reuse, evolution, and execution of mappings between models and be able to interact with individual data management systems that store their own schemas and mappings.

Building such a system remains the goal of ongoing research. One of the challenges in developing a model management system is that implementing the operators is often very dependent on the details of the meta-model, and therefore it is hard to build a system that is agnostic to the particular models. One of the areas in which the meta-models differ considerably is the types of integrity constraints they are able to express and the languages they use to express them.

Bibliographic Notes

While isolated work on different model management operators has gone on for a while, the vision for an algebra of operators and a model management system that encapsulates them all was first laid out by Bernstein et al. [72, 76]. In a more recent paper [77], Bernstein and Melnik refine the vision of model management to focus more on semantics of complex mappings and on paying attention to the run-time support of model management.

The run-time support involves applying schema mappings to data instances efficiently and managing updates to data and to schemas. Melnik et al. [428] describes a first model management system, and in [426] they propose semantics of model management operators.

The earliest work related to merge is by Buneman et al. [99], where the semantics of a merged model was specified by the least upper bound in a lattice of models. However, they required the resulting model to adhere to a particular meta-model. Pottinger and Bernstein [481] defined merge in a generic fashion as part of a model management algebra. They identified the different conflicts and proposed a generic merge algorithm that solves the fundamental conflicts for a broader set of cases. The examples in this chapter are taken from [481].

The basic approach to modelGen that we described in Section 6.3 was introduced by Atzeni et al. [40]. Mork et al. [445] extended the work of [40] in several ways. In particular, the modelGen algorithm proposed by Mork et al. produced a translation of data instances that did not have to go through the intermediate super-model and is therefore much more efficient. In addition, their algorithm enabled the designer to explore a much richer set of transformations from class hierarchies to relational schemas. In some sense, one can view object-relational mapping systems (ORMs), such as Hibernate, as special-case instances of modelGen.

Fagin [215] initially studied the invert operator and showed that inverse mappings exist under very limited conditions. Quasi-inverses were described by Fagin et al. [219].

One area closely related to model management, although it tends to use more operational semantics, is that of supporting schema evolution. Schema modification operators [444] capture a record of schema changes and can be used to do a variety of query rewrites [157, 369].