Translating Schemas and Data between Metamodels

Peter Mork
The MITRE Corporation
7515 Colshire Dr
McLean, VA 22102, USA
pmork@mitre.org

Philip A. Bernstein
Microsoft Research
1 Microsoft Way
Redmond, WA 98052, USA
philbe@microsoft.com

Sergey Melnik
Microsoft Research
1 Microsoft Way
Redmond, WA 98052, USA
melnik@microsoft.com

ABSTRACT

ModelGen is an operator that automatically translates a source model expressed in a source metamodel into an equivalent target model expressed in a different metamodel. For example, given an XML schema, ModelGen can automatically generate an equivalent relational schema or Java interface. This paper describes a new algorithm for ModelGen with several novel properties. It automatically determines a series of transformations to generate the target model. It generates forward- and reverse-views that transform instances of the source model into instances of the target and back again. It supports rich mappings of inheritance hierarchies to flat relations. And it supports incremental modification of a source-to-target mapping. We prove its correctness and demonstrate its practicality in an implementation.

1. INTRODUCTION

In this paper, we address the problem of automatically translating a model expressed in one formalism into an equivalent model expressed in another formalism. For example, a database architect may develop an entity-relationship (ER) diagram, from which a relational schema must be designed. Similarly, the architect needs to provide application developers with interface definitions expressed in an object-oriented (OO) programming language, such as Java or C#. Finally, the architect may want to generate a schema from these interface definitions to support data exchange expressed as an XML Schema Definition (XSD).

In many schema translation tasks, producing the schema in the target formalism is only one half of the job. The other half is producing a mapping that describes how the source schema constructs relate to the target constructs and specifies how to translate data between the source and target representation. For example, an XSD schema produced for data exchange needs to be instantiated by serializing an object graph. Similarly, in object-to-relational schema translation, data transformations are required to shred objects into relations and reassemble them.

Feedback we have received from product developers suggests that generating correct data transformations is one of the most challenging issues in schema translation. For example, in object-to-relational schema translation, there exist a variety of strategies for translating each object-oriented construct, such as inheritance, associations, complex types, or nested collections. Combinations of these strategies yield a huge space of scenarios. Producing data transformations for this space of scenarios is difficult and error-prone. A particularly hard issue is supporting flexible inheritance strategies since minor changes in inheritance translation may have a disruptive effect on the generated data transformations.

In the model-management framework [4], schema translation is encapsulated in the operator ModelGen. It automatically translates a source model expressed in a source metamodel into an equivalent target model expressed in a different metamodel. We use the terms model and metamodel instead of schema and data model for consistency with the metadata field and for clarity. A model can be a database schema, interface definition, or object model. The latter two are not normally called 'schemas.' Similarly, the phrase 'data model' is a uniquely database term. 'Metamodel' is more neutral and makes clearer its relationship to models, which are instances of a metamodel.

ModelGen is a *generic* operator, in the sense that it can work with a range of source and target metamodels, such as ER, SQL, OO interfaces, XSD, RDFS, or types supported by the .NET Common Language Runtime (CLR). Implementing ModelGen in a completely generic fashion seems out of reach due to the myriad semantic details of each metamodel and data transformation language. Instead, we developed an extensible, rule-driven core that can be customized to specific model-translation tasks with moderate effort. As a proof of concept, we customized our generic ModelGen implementation to build an object-to-relational schema translation tool. The tool is integrated with a high-quality user interface that runs inside Microsoft Visual Studio 2005 and produces provably correct mappings. We demonstrated the tool in [5]. This paper reports on the techniques we used to build and customize ModelGen.

Our basic strategy follows the approach of Atzeni and Torlone in [1]. Using this approach, we define a universal metamodel, called the super-metamodel, which has all of the main modeling constructs found in popular metamodels. In this respect, it is the union of its component metamodels. To support a new metamodel, new constructs can be added to it.

The super-metamodel is used to define transformations generically. For example, the concepts of Entity in the ER metamodel and Class in an OO metamodel correspond to a single construct in the super-metamodel, called an Abstract. The concepts of Tuple in SQL and Struct in an OO metamodel correspond to the super-metamodel construct Structure. Therefore, a transformation from Abstract to Structure can be used to translate either an entity or class into a tuple or struct. Thus, many fewer transformations are required than if we must specify transformations for each [source metamodel, target metamodel] pair.

In the Atzeni and Torlone approach, translation proceeds in three steps: (1) transform the source model into an instance of the super-metamodel; (2) identify and then execute a series of transformations that eliminates from the source model all modeling

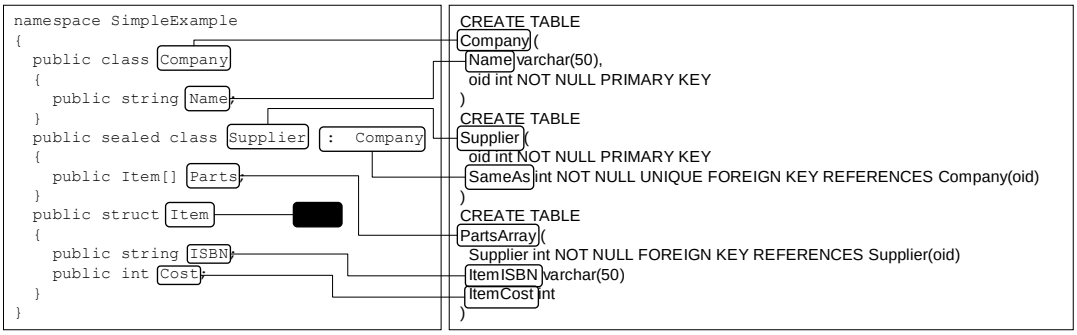


Figure 1: The C# input on the left is transformed into the SQL output on the right.

constructs absent in the target metamodel; (3) transform this intermediate model into the target metamodel.

Our first contribution is the generation of instance-level transformations between the source schema and generated target schema. While there are solutions that can transform instances of the given source model into instances of the generated target model (e.g., [2],[19],[20]), they require passing the instances through an intermediate generic representation. This is impractical for large databases and unnecessary when only view definitions are required. To avoid this, we augment each transformation rule applied in step (2) above to generate not only target schema elements but also elementary forward- and reverse-views in Datalog that describe how each eliminated construct of the source model is represented in the target. We prove that these views are correct, i.e., do not lose information: The composition of the forward-view and reverse-view is the identity.

The final forward- and reverse-views between the source and target schemas are obtained by composing the elementary views (via view unfolding). The correctness of the composition is ensured by the correctness of the elementary views. The composed views are expressed in terms of the super-metamodel. They are fed into a custom component that translates them into the native mapping language, e.g., SQL or XQuery. In our customized object-to-relational (OR) tool, the resulting native views are in SQL.

Our second contribution is a rich set of transformations for inheritance mapping. It allows the engineer to decide on the number of relations used for representing a sub-class hierarchy and to assign each direct or inherited property of a class independently to any relation. These transformations allow a per-class choice of inheritance mapping strategy. They subsume all OR inheritance strategies we know of, including horizontal and vertical partitioning [14], their combinations, and many new strategies. The transformations are driven by a data structure called a mapping matrix. We developed algorithms for populating mapping matrices from per-class annotations of the inheritance hierarchy and generating provably correct elementary views. The complexity of inheritance mapping is encapsulated in a single transformation rule. Since the final views are obtained by composition, inheritance mappings do not interfere with mapping strategies for other OO constructs.

Our third contribution is an extensible plan generator. Each transformation rule encapsulates a small piece of a complete transformation. As new modeling features are introduced or new strategies identified, new rules may be necessary. As a result, the series of rules executed in the translation step needs to be modified. We have developed an A* search algorithm that, given a set of constructs to eliminate and a target metamodel, generates a transformation plan that (a) eliminates all constructs absent from the target metamodel, (b) includes no irrelevant transformations, and (c) minimizes a cost function, such as plan length. The search algorithm is invoked only once for each [set of transformations, source metamodel, target metamodel] triple. In addition to enabling extensibility of our schema translator, this plan generator is generic in the sense that it works with a range of source and target metamodels, such as ER, Extended ER (EER), SQL, OO, XSD, RDFS, Java or .NET types. Implementing it using our approach (or that of [1]) requires an extensible set of rules, which in turn requires an extensible plan generation algorithm to select the best series of rules to apply.

Our fourth contribution is a technique for propagating incremental updates of the source model into incremental updates of the target model. To do this, we ensure that an unchanged target object has the same id each time it is generated, thereby allowing us to reuse the previous version instead of creating a new one. This avoids losing a user's customizations of the target and makes incremental updating fast. This practical requirement stems from the fact that schema translation is typically an interactive process in which the architect analyzes different translation choices. She switches back and forth between the source and target schema, both of which may be large and thus require careful on-screen layout. In such scenarios, it is unacceptable to regenerate the target schema and discard the layout information upon changes in schema translation strategy, i.e., interactive schema translation is required.

Our final contribution is an implementation of a generic ModelGen component for fast development of end-to-end model-translation solutions. We report on the customization effort it took us to build an OR schema translation tool using ModelGen.

The rest of this paper is structured as follows. Section 2 presents an illustrative scenario. Section 3 describes our super-metamodel and provides a state-based semantics for it. Section 4 specifies our syntax for transformations and shows that each transformation is correct. Section 5 describes how we support multiple strategies for mapping inheritance hierarchies into relations. Section 6 describes how to transform models incrementally. Section 7 explains our search algorithm to identify transformation plans. Section 8 discusses our implementation. Section 9 discusses related work and Section 10 is the conclusion.

2. EXAMPLE

Before delving into our algorithms, we illustrate the complete process by showing how to translate a CLR namespace into an equivalent SQL schema. This namespace, shown on the left side of Figure 1 includes two class definitions and a simple structured type. For simplicity, we will assume that the order of values in an array is not relevant.

The first task is to identify which CLR constructs are not supported by SQL. For example, SQL disallows classes, inheritance, arrays, and non-lexical attributes. To remove each of these constructs, we need to identify a transformation plan guaranteed to produce sets of structured types whose attributes are lexical.

Using our A* search algorithm, we identify a suitable transformation plan consisting of five transformation rules: (1) Convert each array of values into a join table. (2) Remove inheritance structures, in this case using vertical partitioning; alternative strategies are considered in Section 5. (3) Replace each class with a relation that includes an explicit object identifier, or oid. (4) Replace each containment relationship with a foreign-key relationship. (5) Replace each attribute that references another relation with a foreign-key relationship.

In Figure 1, rule (1) translates the `Item` array into a `PartsArray` table with `From` and `To` attributes that reference `Supplier` and `Item`. Rule (2) translates the inheritance relationship between `Supplier` and `Company` into a `SameAs` reference from `Supplier` to `Company`. Rule (3) translates `Company` and `Supplier` from classes to relations, and adds a primary key to each. Rule (4) isn't used in this example. Rule (5) converts `SameAs` and `From` into foreign keys that reference the primary keys of `Company` and `Supplier`, respectively. Because no primary key exists for `Item`, rule (5) translates `To` into the attributes of `Item`.

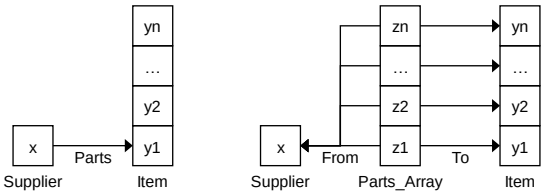


Figure 2: Transformation from an array to a join table.

The first transformation is displayed in Figure 2. Each array is replaced by a join table with two columns. The first column (`From`) references the instance containing the array. The second (`To`) references the values stored in the array.

This schema-level transformation has associated instance-level transformations that are used to generate forward- and reverse-views. Note that the `Parts` attribute has been mapped to the join-relation `PartsArray`.

Forward-view: For each `Supplier x` and related `Item y`, the first line generates a tuple `z` using a Skolem function `f`. The second and third lines populate the tuple using `x` and `y`.

```
PartsArray(z=f(x,y)) :- Parts(x,y)
From(z,x) :- Parts(x,y)
To(z,y) :- Parts(x,y)
```

Reverse-view: In this simple example, the reverse-view simply inverts the forward view:

```
Parts(x,y) :- From(z,x), To(z,y)
```

Composing the forward- and reverse-views maps `Parts(x,y)` to itself. Thus, the mapping does not lose information. The same holds for the other transformations. So running them in sequence also avoids losing information.

These five transformations are executed and the result is exported as a DDL statement, as shown in the right side of Figure 1. Note that all of the elements defined in the CLR namespace appear in the SQL schema except the `Item` structure. Instances of this structure can be generated by projecting `ISBN` and `Cost` from the `PartsArray` relation.

3. BACKGROUND

Before we can define transformation rules, we need to describe the super-metamodel in which they are expressed. The super-metamodel we use in this paper supports many of the constructs present in popular metamodels, enough to illustrate our techniques. It is not intended to be complete, i.e. capture all of the features of rich metamodels such as XSD or SQL with complex constraints and triggers. First, we formally define our metamodel constructs. We then provide a relational schema for them.

We assume the existence of a finite set of lexical values $\boldsymbol{L}$, a countably infinite set of object instances $\boldsymbol{ID}$, and a countably infinite set of attributes $\boldsymbol{F}$. These sets are disjoint.

3.1 Super-Metamodel

A *data instance D* is a triple of the form $<D_L, D_{ID}, \boldsymbol{D_A}>$ where $D_L \subseteq \boldsymbol{L}$, $D_{ID} \subseteq \boldsymbol{ID}$ and $\boldsymbol{D_A}$ is a set of triples of the form $<x \in D_{ID}, a \in \boldsymbol{F}, y \in D_{ID} \cup D_L>$. A data instance can be interpreted as an edge-labeled graph whose nodes are lexical values or object instances and each edge links an object instance to a lexical or object instance. A *data element* is any component of a data instance such as a lexical, object value, labeled edge, or collection of values.

A *model* is a (possibly infinite) set of data instances. E.g., a model can be a CLR type definition or SQL schema. A *metamodel* is a (possibly infinite) set of models (e.g., CLR or SQL themselves).

The *super-metamodel* is the set of all models that can be expressed as a triple of the form $<\boldsymbol{T, A, C}>$ in which $\boldsymbol{T}$ is a set of types, $\boldsymbol{A}$ is a set of attributes and $\boldsymbol{C}$ is a set of constraints. A valid instance of such a model is a function $\boldsymbol{I}$ that relates each element of $\boldsymbol{T} \cup \boldsymbol{A}$ to a set of data elements, subject to the constraints in $\boldsymbol{C}$.

The *lexical* types partition $\boldsymbol{L}$ into discrete domains; these lexical types are shared by all models (and metamodels). A *complex* type T defines a new set of instances: $\boldsymbol{I}(T)$ is a finite subset of $\boldsymbol{ID}$. We also allow *collection* types, which are declared with respect to some base type. For a given collection $C(T)$, $\boldsymbol{I}(C)$ is a finite set of collections of T. For example, $List(int)$ is a finite set of (ordered) lists of integers. The super-metamodel in this paper includes list and set collections.

There is a special `NULL` value that is an instance of every type (i.e., $\forall T \in \boldsymbol{T}$, $\texttt{NULL} \in \boldsymbol{I}(T)$). For collection types, `NULL` represents an empty collection. Otherwise, `NULL` represents an empty instance (of an abstract type) or an empty value (for lexical types). This representation is in keeping with SQL and CLR, both of which support explicit `NULL` values.

Types are connected via *attributes*. An attribute $F \in \boldsymbol{F}$ is a total mapping from a complex type D to type R . It is defined formally by $\boldsymbol{I}(F) \subseteq (P_1 \times \boldsymbol{I}(R) - \texttt{NULL}) \cup (P_2 \times \texttt{NULL})$, such that

a) $P_1 \cup P_2 = \boldsymbol{I}(D)$
b) $P_1 \cap P_2 = \varnothing$
c) $\texttt{NULL} \in P_2$

These conditions ensure that $\boldsymbol{I}(D)$ is partitioned into instances P_1 that map to non-null values and instances P_2 that map to NULL (and only NULL). Condition (a) ensures that the mapping is total, and (c) maps NULL to NULL. Note that the domain of $\boldsymbol{I}(F)$ is $\boldsymbol{I}(D)$, and the range, a subset of $\boldsymbol{I}(R)$.

Cardinality constraints can be placed on an attribute. If the *minimum* cardinality of an attribute is One, then $P_2 = \{\texttt{NULL}\}$. If the *maximum* cardinality is One, then F is a (total) function. When the minimum and maximum cardinalities are both One, F is a total function in the more traditional sense—every non-null element of $\boldsymbol{I}(D)$ is mapped to a single non-null element of $\boldsymbol{I}(R)$.

The super-metamodel supports additional constraints, such as inclusions and keys. The simplest inclusion is a *generalization* that relates one abstract type to a set of abstract types (called the specialization): $G(X, \boldsymbol{S})$ indicates that type X is more general than the types of $\boldsymbol{S}$. This introduces a new restriction on the valid instances of a model: $\forall S \in \boldsymbol{S}, \boldsymbol{I}(S) \subseteq \boldsymbol{I}(X)$.

A *key* defined on T identifies a set of attributes whose values uniquely identify an instance of T. $K(T, \boldsymbol{A})$ indicates that values of the attributes in $\boldsymbol{A}$ uniquely identify instances of type T. The interpretation of a key is straightforward when all elements of $\boldsymbol{A}$ have a maximum cardinality of One. More formally, we say t_1 and t_2 *agree on* A if $(\exists v)(\langle t_1, v\rangle \in \boldsymbol{I}(A) \wedge \langle t_2, v\rangle \in \boldsymbol{I}(A))$. The key constraint $K(T, \boldsymbol{A})$ says that if t_1 and t_2 agree on every $A \in \boldsymbol{A}$, then $t_1 = t_2$.

Since some metamodels distinguish between *abstract* and *structured* types, we preserve this distinction in the super-metamodel. We assume that the set of all attributes of each structured type constitutes a key (although this violates the SQL metamodel).

Finally, an *inclusion dependency* imposes a restriction on a set of attributes: $Inc(T,K,B)$ indicates that type T references the attributes of key K, using a partial function B to relate attributes of T to key attributes of K. Its semantics is that the values of the attributes of T are a subset of the values of the attributes of K.

Note that a relation is not modeled as a cross-product of types. Instead, a relation is a structured type, and therefore a complex type, whose instances are tuples. Attributes are used to relate the tuple object to its corresponding values. This allows us to transform abstract types into structured types (and vice versa). In this framework, NULL is a tuple whose attribute values are all NULL.

3.2 Relational Schema

The previous section presented a super-metamodel that has features common to many popular metamodels, summarized in Table 2. A relational schema for this super-metamodel is in Figure 3. In this schema, each model element is uniquely identified via an object identifier, and by convention, the first attribute of each predicate is an object identifier. The types in the super-metamodel are organized into a hierarchy. A collection type is either a list or set, and a simple type is either a collection or a lexical type. A complex type is either an abstract or structured type.

Table 1: Relationships among common metamodels

Construct	SQL	EER	Java	XSD
Lexical Type	int, varchar	scalar	int, string	integer, string
Structured Type	tuple			element
Abstract Type		entity type	class	complex type
List Type			array	list
Set Type	table			
Attribute	column	attribute, relationship	field	attribute
Containment				nesting

Simple types include lexicals and collections:
LexicalType(TypeID, TypeName)
ListType(TypeID, TypeName, BaseType)
SetType(TypeID, TypeName, BaseType)

Complex types can be structured or abstract:
StructuredType(TypeID, TypeName)
AbstractType(TypeID, TypeName)

Complex types have *attributes* and can be nested:
Attribute(AttrID, AttrName, Domain, Range, MinCard, MaxCard)
Containment(ConID, AttrName, Parent, Child, MinCard, MaxCard)
- Domain/Parent must be a complex type.
- Range/Child can be any type.
- Min/MaxCard are Zero, One or N and apply to the range/child.

A *key* indicates a set of attributes that identify a complex object:
KeyConstraint(KeyID, TypeID, IsPrimary)
- TypeID references the type for which this is a key.
- Primary indicates if this is the primary key for the type.

KeyAttribute(KeyAttrID, KeyID, AttrID)
- KeyID references the key for which this is an attribute.
- AttrID references an attribute of the associated type.

An *inclusion dependency* establishes a subset relationship:
InclusionDependency(IncID, TypeID, KeyID)
- TypeID references the type for which this dependency holds.
- KeyID references the associated key.

InclusionAttribute(IncAttrID, IncID, AttrID, KeyAttrID)
- IncID references the inclusion for which this is an attribute.
- AttrID references an attribute of the associated type.
- KeyAttrID: references a key attribute of the associated key.

Generalization is used to extend a type or construct a union:
Generalization(GenID, TypeID, IsDisjoint, IsTotal)
A type can serve as the parent for multiple generalizations. Disjoint and Total describe the generalization.
Specialization(SpecID, GenID, TypeID)
- GenID references the parent generalization.
- TypeID references the associated specialized type.

Figure 3: Relational schema for super-metamodel

4. TRANSFORMATIONS

Using the Atzeni-Torlone approach, an implementation of ModelGen has four basic steps: (1) import the source model, (2) manually or automatically generate a valid transformation plan

which consists of a sequence of transformations (3) execute the transformations in the plan, and (4) export the result. Step 1 is trivial; it is just a 1:1 translation of the input model into super-metamodel constructs. All of the remaining steps revolve around the transformations, which is also where most of our innovations lie. So we discuss them now, in this section and the next. We explain step (2), automatic generation of a transformation plan, in Section 7. We discuss step (4) in Section 8 on Implementation.

4.1 Defining a Transformation

Each step of a transformation plan is a transformation that removes certain constructs from the model and generates other constructs. The transformation is expressed as a set of Datalog rules. Thus, constructs are expressed as predicates, each of which is a super-metamodel construct in Figure 3. For example, the following is a simplified version of the rule that replaces an abstract type, such as a class definition, by a structured type, such as relation definition:

StructuredType(newAS(id), name) :- AbstractType(id, name)

The Skolem function newAS(id) generates a new TypeID for the structured type definition based on the abstract type's TypeID. All Skolem functions names are prefixed by new to aid readability.

A *transformation* is a triple of the form <*D, F, R*> where *D* is a set of rules that expresses a model transformation, *F* is a forward view that expresses the target model as a view over the source, and *R* is a reverse view that expresses the source as a view over the target.

Datalog is verbose when many rules have the same body. To avoid this, we introduce a condensed format for expressing rules in *D*. Each rule is of the form "<body> ⇒ <head>". Unlike normal Datalog, we allow the head to contain multiple terms. A rule in *D* with body *b* and *n* terms in the head is equivalent to a Datalog program with *n* rules, each of which has one term in the head implied by *b*. For example, A(x,y) ⇒ B(x), C(f(y)) is equivalent to the Datalog program B(x) :- A(x,y) and C(f(y)) :- A(x,y).

Some of the rules in each model transformation *D* also populate a binary predicate Map, whose transitive closure identifies all of the elements derived from a given source element. For example, adding Map to the rule that replaces an abstract type by a structured type, we get:

AbstractType(id, name)
⇒ StructuredType(newAS(id), name), Map(id, newAS(id))

Map(id, newAS(id)) says that the element identified by id is mapped to a new element identified by newAS(id).

After executing all of the transformations, we can extract from the transitive closure of Map those tuples that relate source elements to target elements. Tools that display the source and target models can use this mapping to offer various user-oriented features, such as the ability to navigate back and forth between corresponding elements or to copy annotations such as layout hints or comments.

Datalog rules add tuples to the head predicates but never delete them. Since we need to delete tuples that correspond to constructs being replaced in a model, we use a unary predicate Delete that identifies elements to delete. After all the rules of a transformation are executed, a non-Datalog post-processing step deletes the elements identified in the Delete predicate. For example, in the rule that replaces an abstract type by a structured type, the predicate Delete removes the abstract type that is being replaced, as follows:

AbstractType(id, name)
⇒ StructuredType(newAS(id), name), Map(id, newAS(id)), Delete(id)

The rules in a model transformation *D* are schema-level mappings. Forward- and reverse-views are instance-level mappings. The predicates and variables in a view are variables in the rules of *D*. For example, a simplified version of the forward-view for replacing an abstract type by a structured type is "id(x) ⇒ newAS[id](x)". This rule says that if x is an object of the abstract type identified by id, then it is also a tuple of the structured type identified by newAS[id]. To generate such views in Datalog, we can define predicates that create their components, such as the following:

ViewHead(newRule(newAS(id)), newPredicate(id, "x"))
ViewBody(newRule(newAS(id)), newPredicate(newAS(id), "x"))

We can then conjoin these to the head of the rule that replaces an abstract type by a structured type.

Since views are expressed using predicates of a non-standard metamodel, namely our super-metamodel, we need to define their semantics. To do this, we represent the model before and after a transformation as a *model graph*. Its nodes correspond to simple and complex types. Its edges correspond to attributes. For example, in Figure 4, R is a structured type with attributes a and b. The value of b is a structured type S with attributes c and d; attributes a, c, and d have lexical type.

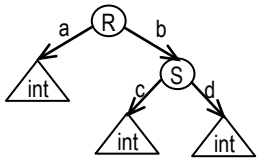


Figure 4 A Model Graph

An instance of a model graph is an *instance graph*, which is comprised of a set of values for each node and a set of value pairs for each edge. A *view* is a formula that defines how to populate the nodes and edges of one instance graph from those of another.

A transformation is correct if the forward-view converts every instance $\mathcal{I}_s$ of the source model into a valid instance $\mathcal{I}_t$ of the target model, and the reverse-view converts $\mathcal{I}_t$ back into $\mathcal{I}_s$ without loss of information. That is, the composition of the forward- and reverse-views is the identity. We do not require the converse—there may be instances of the target model that cannot be converted into instances of the source model. Note that this definition of correctness is more stringent than the one in [20], which only requires that the forward view generate a valid instance of the target model.

Sections 4.2-4.5 define the transformations that are needed to convert from CLR to SQL. For each transformation, we give its model transformation and its forward-/reverse-views. We write the views as instance transformations and omit the verbose Datalog predicates that would generate them. Since the forward- and reverse-views for the first three transformations are inverses of each other, correctness is immediately apparent. We give a detailed correctness argument for the transformation of Section 4.5.

4.2 Convert Abstract to Structured Type

This transformation replaces each abstract type with a structured type. To preserve object identity, a new oid attribute is added to the structured type, unless the abstract type already included a primary key. The model transformation rules are as follows:

AbstractType(id, name)
⇒ StructuredType(newAS(id), name), Map(id, newAS(id)), Delete(id)

AbstractType(id, name), ¬KeyConstraint(_, id, "True")
⇒ Attribute(newOID(id), "oid", newAS(id), "Int", "1", "1"),
KeyConstraint(NewASKey(id), NewAS(id), "True"),
KeyAttribute(NewASKeyAttr(id), NewASKey(id), NewOID(id))

To avoid generating useless unique variables, we use an underscore in a slot for an existential variable that appears only once in the rule. Our use of negated predicates, such as ¬KeyConstraint above, is done carefully to ensure that stratification is possible.

The forward view is: id(x) ⇒ newAS[id](x), newOID[id](x, newID(x)). The last predicate says that newOID[id] is an attribute whose value for the tuple x is newID(x).

The reverse view is: newAS[id](x) ⇒ id(x). Notice that we do not need to map back the new oid attribute of the structured type, since it is not needed for information preservation of the source abstract type. It is immediately apparent that the forward- and reverse-views are inverses of each other and hence are correct.

4.3 Replace Multi-value Attr. by Join Table

This transformation replaces each attribute with a maximum cardinality of *N* by a join table containing two attributes: From and To. An example application of this rule appeared in Figure 2. The model transformation rule is as follows:

Attribute(id, name, d, r, min, "N")
⇒ StructuredType(newJT(id), name),
Attribute(newJTFrom(id), "From", newJT(id), d, "1", "1"),
Attribute(newJTTo(id), "To", newJT(id), r, min, "1"),
Map(id, newJT(id)), Delete(id)

The forward view is:

id(x,y) ⇒ newJT[id] (newTuple(x,y)), newJTFrom[id] (newTuple(x,y),x),
newJTTo[id] (newTuple(x,y),y)

The reverse view is: newJTFrom[id](z,x), newJTTo[id] (z,y) ⇒ id(x,y)

It is immediately apparent that the forward- and reverse-views are inverses of each other and hence are correct.

4.4 Remove Containment

This transformation replaces a containment relationship between a parent type and child by an attribute in the child that refers to its parent. This attribute is added to any existing keys defined on the child. Moreover, if the parent can only contain a single child, the new attribute constitutes a key. The model transformation rules are as follows:

Containment(id, name, parent, child, min, max)
⇒ Attribute(newCA(id), "Parent", parent, child, "One", "One"),
Map(id, newCA(id)), Delete(id)

Containment(id, name, parent, child, min, max),
KeyConstraint(key, child, _, schema)
⇒ KeyAttribute(newCAKeyAttr(id), key, newCA(id))

Containment(id, name, parent, child, min, "One")
⇒ KeyConstraint(newCAKey(id), child, "False"),
KeyAttribute(newCAK2(id), newCAKey(id), newCA(id))

The forward view is: id(x,y) ⇒ newCA[id](y,x)

The reverse view is: newCA[id](x,y) ⇒ id(y,x)

Like the previous two rules, the forward- and reverse-views are inverses of each other and hence are correct.

4.5 Remove Structured Attribute

This transformation replaces an attribute a that references a structured type S all of whose attributes are lexicals. It replaces a by lexical attributes that uniquely identify a tuple of S. If S has a primary key, then a is replaced by the key attributes of S and there is an inclusion dependency from the new attributes to that key. Otherwise, a is replaced by all of S's attributes. The transformation is applied iteratively to eliminate nested structured types.

For example, consider three structured types: *R*, *S* and *T* (see Figure 5). *R* references *S* using attribute *a* and has primary key *k* (an Int). *S* has no primary key, but it has two attributes *b* (an Int) and *c* (which references *T*). *T* has a primary key attribute *d* (an Int). Applying the transformation to *S.c* replaces that attribute by *S.d* and adds an inclusion dependency from *S.d* to *T.d*. Now all attributes of *S* are lexicals. So we can apply the transformation again to replace *R.a* by *R.b* and *R.d*.

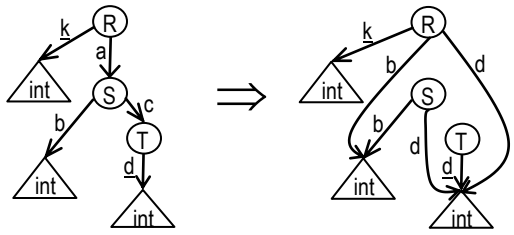


Figure 5 Removing two structured attributes

The model transformation rules are as follows:

StructuredType(domain, name), Attribute(id, _, domain, range, _, "One"),
¬ LexicalType(range, _)
⇒ MixedStructuredType(domain, name)

Attribute(id, name, domain, range1, min1, "One"),
StructuredType(range1, name), ¬ MixedStructuredType(range1, name),
Attribute(attr, _, range1, range2, min2, "One"), Min(min1, min2, min)
⇒ Attribute(newSA(id, attr), newName(name1, name2), domain,
range2, min, "One") , Map(id, newSA(id, attr)), Delete(id)

Attribute(id, _, _, range, _, "One"), KeyAttribute(keyAttr, key, id),
StructuredType(range, _)
⇒ KeyAttribute(newSAKeyAttr(keyAttr, attr), key, newSA(id, attr)),
Map(keyAttr, newSAKeyAttr(keyAttr, attr))

For each id and $attr_i$ that satisfy the second model transformation rule, there is a forward view:

id[x, z], $attr_i$[z, y] ⇒ newSA(id, $attr_i$)[x, y]

In the following reverse view, either $attr_1$ … $attr_k$ are the attributes in the key of structured type range1, or range1 has no key and k attributes in total:

newSA(id, $attr_1$)[x, t_1], $attr_1$(s, t_1), ..., newSA(id, $attr_k$)[x, t_k], $attr_k$(s, t_k)
⇒ attr[x, s]

To explain the above view definitions and argue their correctness, we simplify the notation by replacing the terms id, attri, and newSA(id, attri) in the view definitions by the symbols a, b_i, and ab_i, yielding the following (see Figure 6):

$a(r,s),\ b_i(s,t) \Rightarrow ab_i(r,t)$ // forward views

$ab_1(r, t_1),\ b_1(s, t_1),\ \ldots\ ab_k(r, t_k),\ b_k(s, t_k) \Rightarrow a(r,s)$ // reverse view

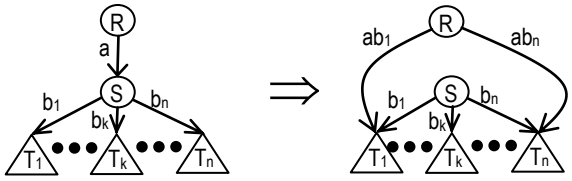


Figure 6 Model graphs before and after applying the transformation that removes a structured attribute

As shown in Figure 6, the structure S has n attributes, k of which are key attributes (if there is a key). The attribute R.a that refers to the structure S is replaced by new attributes that correspond one-to-one with the attributes of the S. To show that the forward- and reverse-views are correct, we need to show that their composition is the identity. We form the composition by substituting the forward view for each ab_i in the reverse view, yielding:

$a(r,s_1),\ b_1(s_1, t_1),\ b_1(s, t_1),\ \ldots,\ a(r,s_k),\ b_k(s_k, t_k),\ b_k(s, t_k) \Rightarrow a(r,s)$

a is a function, so $s_1 = s_2 = \ldots = s_k$. Replacing the s_i's by s_1 we get:

$a(r,s_1),\ b_1(s_1, t_1),\ b_1(s, t_1),\ \ldots,\ a(r,s_1),\ b_k(s_1, t_k),\ b_k(s, t_k) \Rightarrow a(r,s)$

Since $b_1, \ldots b_k$ is either a key or comprises all the attributes of s, we have $s = s_1$. Replacing the s_1's by s we get:

$a(r,s),\ b_1(s, t_1),\ \ldots,\ b_k(s, t_k) \Rightarrow a(r,s)$

Since there must exist values for $t_1, \ldots, t_k$ in s, the above rule reduces to a(r,s) :- a(r,s), which is the identity.

The above argument glosses over one annoying corner case: If nulls are allowed, the transformation does not distinguish between R.a being a null pointer or being a non-null pointer that points to a structure containing all nulls. One can fix this by having the transformation add a Boolean attribute to R to distinguish these cases.

4.6 Additional Transformations

The transformations presented in this paper are sufficient for transforming from CLR to SQL. We have additional transformations to address more target metamodels. Due to space limitations, we provide a brief summary of some of them.

Convert structured types to abstract types. This transformation is the inverse of the one presented in Section 4.2.

Replace an attribute with maximum cardinality N by a new attribute with maximum cardinality of One. If the range of the old attribute was T, the range of the new attribute is a set of T. The difference between the old and new attributes is evident when the attribute participates in a key constraint. A multi-valued attribute provides multiple key values (i.e., each value is unique); a set-valued attribute provides a single key value.

Replace a list of T with a set of indexed structures. The new structured type has two attributes, Index and Value. The range of the former is Integer, and the latter, T.

Stratify sets. This transformation takes a set of sets and converts it into a set of indexed structures (as in the preceding rule). This transformation is needed to support the nested relational model.

Add an attribute to an otherwise empty complex type. This new attribute has minimum and maximum cardinality of Zero.

Remove multiple-containment. Whenever type T is contained in multiple parent types, this transformation creates a new specialization of T. Each old containment relationship is transformed into a new containment relationship that references exactly one of the new specializations of T. For example, if type A is contained in both B and C, this transformation creates types B-A and C-A, which are contained in B and C, respectively.

4.7 Composing Transformations

A transformation plan is a sequence of *n* transformations. The first transformation takes the initial model m_0 as input and the last transformation produces the final model m_n as output. Our goal is to generate a forward view V_F that defines m_n as a function of m_0 and a reverse view V_R that defines m_0 as a function of m_n. Given the forward- and reverse-views, this can be done incrementally. The initial transformation from m_0 to m_1 defines the initial views V_F and V_R. Suppose we have forward- and reverse-views V_F and V_R for the first *i-1* transformations. For the i^{th} transformation, its forward view v_f and reverse view v_r are composed with V_F and V_R, i.e., $V_F \bullet v_f$ and $V_R \bullet v_r$, using ordinary view unfolding, thereby generating V_F and V_R.

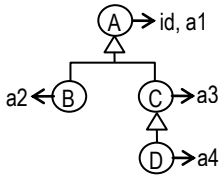
5. INHERITANCE MAPPINGS

The transformations in the last section assumed that every violation of the target metamodel is resolved using the same transformations. We now consider a more general strategy for mapping inheritance hierarchies of abstract types into structured types that allows the user to customize the mapping. Since this is the familiar object-to-relational mapping problem, we use the terms class and relation instead of abstract type and structured type.

For a given hierarchy, let *C* be the set of all classes in the (source) hierarchy and let *R* be the set of target relations. The predicate `c(x)` indicates that `x` is a direct instance of $c \in C$. Similarly, `r(x)` indicates that `x` is a tuple of $r \in R$.

5.1 Mapping Matrices

A mapping matrix *M* describes how the attributes of classes are mapped to attributes of relations. The mapping matrix contains one column for each concrete $c \in C$ and one row for each $r \in R$. Each cell $M[r, c]$ of the mapping matrix indicates which attributes of *c* appear in *r*. For example, to map a class's direct and inherited attributes to one table (a.k.a., horizontal partitioning), all of the attributes of *c* appear in a single cell of *M*. To flatten a hierarchy, *R* contains a single relation, so *M* has just one row.



For example, consider a simple inheritance structure with four classes: *A* is an abstract class (i.e., it does not appear in the mapping matrix) with two attributes: *a1* and primary key *id*. *B* is a subclass of *A* with additional attribute *a2*. Finally, *C* is a subclass of *A* with additional attribute *a3* and *D* is a

subclass of *C* with attribute *a4*. We map these classes into three relations, *R*, *S* and *T* as shown in the top half of Table 2.

Table 2: A Sample Mapping Matrix

	B	*C*	*D*
R	*id,a1*	*id,a1*	*id,a3*
S	*id,a2*		*id,a1*
T		*id,a3*	*id,a4*
rel	*{R,S}*	*{R,T}*	*{R,S,T}*
*attr**	*id,a1,a2*	*id,a1,a3*	*id,a1,a3,a4*

5.2 Generating View Definitions

To explain the construction of view definitions from *M*, we need some additional notation: *PK*(*c*) returns the primary key of *c*, *attr**(*c*) returns the direct and indirect attributes of *c*, *rel(c)* returns the relations used to store instances of *c* (the non-empty cells of column *c*), and *r.a* refers to attribute *a* of relation *r*. *Flagged* is the set of all relations that contain a *flag* attribute, the values of which are type identifiers. The type identifier of *c* is *TypeID*(*c*).

The forward-view for this transformation can be directly inferred from *M*. For each attribute *a* in a cell *M*[*r*, *c*] the forward-view is: c(x), a(x,y) ⇒ r(x), r.a(x,y). The reverse-view is more complex and is based on the following constraints on *M*.

a) $\bigcup_{r \in R} M[r,c] = attr^*(c)$

b) $r \in rel(c) \rightarrow PK(c) \subseteq M[r,c]$

c) $rel(c_1) = rel(c_2) \rightarrow c_1 = c_2 \vee rel(c_1) \subseteq Flagged$

Constraint (a) says that every attribute of *c* must appear in some relation. Constraint (b) guarantees that an instance of *c* stored in multiple relations can be reconstructed using its primary key, which we assume can be used to uniquely identify instances. Constraint (c) says that if two distinct classes have the same *rel*(*c*) value, then each of them is distinguished by a type id in *Flagged*.

To test these constraints in our example, consider the last two rows of Table 2. Constraint (a) holds since every attribute in the bottom row appears in the corresponding column of *M*. Constraint (b) holds because *id* appears in every non-empty cell. Constraint (c) holds because no two classes have the same signature.

Constraint (c) guarantees that the mapping is invertible, so there exists a correct reverse-view for the mapping. There are two cases: For a given $c \in C$, either there is another class c' with $rel(c') = rel(c)$, or not. If so, then $\exists r \in (rel(c) \cap Flagged)$, so we can use *r.flag* to identify instances of *c*: r(x), flag(x, TypeID(c)) ⇒ c(x).

Otherwise, *rel*(c) is unique, so the instances of c are those that are in all *rel*(c) relations and in no other relation, that is: $\bigwedge_{r \in rel(c)} r(x) \bigwedge_{r \notin rel(c)} \neg r(x) \Rightarrow c(x)$.

In relational algebra, this is the join of all $r \in rel(c)$ composed with the anti-semijoin of $r \notin rel(c)$. In both cases, the reverse view is an inverse of the forward view provided that the class has an attribute. The reverse-view for a given attribute is read directly from the mapping matrix. It is simply the union of its appearances in M.

$(\forall c \in C, r \in R: a \in M[c,r])$ r.a(x, y) ⇒ c.a(x, y)

In our example, the forward- and reverse-mappings for *B* are:

B(x) ⇒ R(x), S(x)

R(x), S(x), ¬T(x) ⇒ B(x)

5.3 Generating *M* from Class Annotations

The mapping table *M* is very general, but can be hard to populate to satisfy the required constraints (a) – (c) above. So instead of asking users to populate the mapping table *M*, we offer them easy-to-understand class annotations from which *M* is populated automatically. Each class can be annotated by one of three strategies: *Own*, *All*, or *None*. *Own* does vertical partitioning, the default strategy: each inherited property is stored in the relation associated with the ancestor class that defines it. *All* yields horizontal partitioning: the direct instances of the class are stored in one relation, which contains all of its inherited properties. *None* means that no relation is created: the data is stored in the table for the parent class. The strategy selection propagates down the inheritance hierarchy, unless overridden by another strategy in descendant classes. These annotations exploit the flexibility of the inheritance mapping tables only partially, but are easy to communicate to schema designers.

Let strategy(*c*) be the strategy choice for class *c*. For a given annotated schema, the mapping matrix is generated by the procedure PopulateMappingMatrix below (for brevity, we focus on strategy annotations for classes only, omitting attributes). The root classes must be annotated as ⇳ or ⇔. For every root class *c*, PopulateMappingMatrix(*c*, undefined) should be called. After that, for each two equal columns of the matrix (if such exist), the first relation from the top of the matrix that has a non-empty cell in those columns is added to Flagged.

```
procedure PopulateMappingMatrix(c: class, r: target relation)
    if (strategy(c) ∈ {⇳, ⇔}) then r = ⟨new relation⟩ end if
    if (c is concrete) then
      M[r, c] = M[r, c] ∪ ⟨key attributes of c⟩
      if (strategy(c) = ⇔) then
        M[r, c] = M[r, c] ∪ ⟨declared and inherited attributes of c⟩
      else
        placed = attrs = ⟨declared and inherited non-key attributes of c⟩
        for each cell M[r′, c′] populated by ascendant of c do
          M[r′, c] = M[r′, c] ∪ (M[r′, c′] ∩ attrs)
          placed = placed – M[r′, c′]
        end for
        M[r, c] = M[r, c] ∪ (attrs – placed)
    end if
    for each child c′ of c do
      PopulateMappingMatrix(c′, r)
    end for
```

6. INCREMENTAL UPDATING

Translating a model between metamodels can be an interactive process, where the user (e.g., a database designer) incrementally revises the source model and/or various mapping options, such as the strategy for mapping inheritance. Typically, a user wants to immediately view how design choices affect the generated result. The system could simply regenerate the target model from the revised input. However, this regeneration can be slow, especially if the models are being stored in a database. For example, our implementation uses a main memory object-oriented database system, in which a full regeneration of the target schema can take a minute or more. Also, it loses any customization the user per-

formed on the target, such as changing the layout of a diagrammatic view of the model or adding comments. We can improve the user's experience in such scenarios by translating models in a stateful fashion: the target model is updated incrementally instead of being re-created from scratch by each modification.

Let m_0 be a source model and m_1, …, m_n be a series of target model snapshots obtained by an application of successive transformations (i.e., a transformation plan). Each transformation is a function that may add or delete schema elements. Let f_i be a function that returns new elements in m_{i+1} given the old ones in m_i. Since f_i uses Skolem functions to generate new elements, whenever it receives the same elements as input, it produces the same outputs. Clearly, invoking a series of such functions f_1, …, f_n preserves this property. That is, re-running the entire series of transformations on m_0 yields precisely the same m_n as the previous run, as the functions in effect cache all generated schema elements.

Now suppose the user modifies m_0 producing m_0'. When m_0' is translated into a target model, the same sequence of transformations is executed as before. In this way, no new objects in the target model are created for the unchanged objects in the source model. Previously created objects are re-used; only their properties are updated. For example, renaming an attribute in the source model causes renaming of some target model elements (e.g., attribute or type names); no new target objects are created.

The mechanism above covers incremental updates to m_0. Deletion is addressed as follows. Let m_n be the schema generated from m_0. Before applying the transformations to m_0', a shallow copy m_{copy} of m_n can be created which identifies all of the objects in m_n. All transformations are re-run on m_0' to produce m_n'. If an element is deleted from m_0 when creating m_0', then some elements previously present in m_{copy} might not appear in m_n'. These target elements can be identified by comparing m_{copy} to m_n'. They are marked as "deleted", but are not physically disposed of. If they appear in m_n at some later run, the elements are resurrected by removing the "deleted" marker. Thus, the properties of generated objects are preserved upon deletion and resurrection. In our implementation, for small changes to the source model, this incremental regeneration of the target takes a fraction of a second.

7. AUTOMATIC PLAN GENERATION

In this section, we focus on automatic generation of a sequence of transformations that comprises the transformation plan which removes all constructs not supported by the target metamodel.

7.1 Metamodel Patterns

The planning algorithm takes as input source and target *metamodel signatures*, *SS* and *TS*, which describe the constructs that are valid for the source and target metamodels. *SS* may include less than all valid source metamodel constructs, if an analysis of the source model indicates some constructs are not present.

Each metamodel signature is a set of *patterns*, each of which is a conjunction of predicates. As in transformation rules, the predicates are super-metamodel constructs (see **Error! Reference source not found.**). A signature contains one pattern for every modeling construct that is valid for that metamodel. For example, the CLR metamodel has five simple patterns, which correspond to classes, structs, fields, arrays and lexical types. A more complex pattern is needed to indicate that attributes in the relational model must reference lexical values:

RA() :- Attribute(_,_,D,R,_,1), StructuredType(D,_), LexicalType(R,_)

The planning algorithm needs to know the patterns consumed and generated by a transformation, called its *input* and *output signatures*. Applying the transformation removes every instance of its input pattern and generates instances of its output patterns.

The input and output patterns for a transformation rule are occasionally parameterized. For example, consider the transformation that converts an attribute with multiple values (such as an array) into a join table. It replaces input pattern Multi-ValueAttr with new parameterized output patterns T1Attr and T2Attr.

MultiValueAttr() :- Attribute(_,_,T1,T2,_,N)

T1Attr[ComplexType:T1] :- Attribute(_,_,X,T1,_,1), StructuredType(X)

T2Attr[Type:T2] :- Attribute(_,_,X,T2,_,1), StructuredType(X)

Parameterized patterns are needed when the specific patterns that are generated by a transformation depend on which patterns have already been removed. For example, if we remove all abstract types before applying the above transformation, we know that the attributes of the join table will not reference abstract types.

By specifying patterns as conjunctive queries, we can construct a pattern hierarchy based on query containment. This hierarchy contains patterns appearing in *SS*, *TS*, or a transformation signature (not all possible patterns). When a transformation is applied, it removes every pattern in its input signature, including any sub-patterns in the hierarchy. Similarly, it generates every pattern in the output signature, including any super-patterns. This observation is at the heart of the heuristic used by the planning algorithm to identify a minimal transformation plan.

7.2 Planning

We use an A* search algorithm to identify a series of transformations that will produce a final model that conforms to the target metamodel. The A* algorithm tries to minimize the number of intermediate states needed to reach a goal state by using a function to estimate the cost of reaching the goal from each state. To use A*, we need to define the state space, the goal state, the actions that transition between states, and the cost function.

We define a *state* to be two sets of patterns: `Invalid` patterns that need to be removed from the model and `Valid` patterns that are allowed by the target metamodel. A* begins at an initial state and chooses actions that generate new states, until it reaches the *goal state*. Our goal is to reduce `Invalid` to the empty set. Thus, we need to define how a transformation modifies `Invalid`.

When a transformation is applied, it removes the patterns in its input signature and introduces the patterns in its output signature. It adds every pattern in the output signature to `Invalid` unless it appears in `Valid`. It might also add sub-patterns to `Invalid`. For example, consider a transformation T that replaces ListType(_,_,T) by SetType(X,_), Attribute(_,_,X,int,1,1), Attribute(_, _,X,T,1,1), i.e., transforming each list into a set of index/value pairs. When T is applied, we need to add Set() :- SetType(X,_) to `Invalid`, and, if there were attributes that referenced the original list, SetAttribute():-SetType(X,_), Attribute(_,_,_,X,_,_) as well.

For each sub-pattern *S* of pattern *P* in the output signature, *S* is added to `Invalid` if every super-pattern of *S* appears in `Present = Invalid∪Valid`, which indicates that *S* is supported by the

patterns present in the current model. (The number of sub-patterns is bounded by the size of the pattern hierarchy.) For example, suppose we 1) remove all multi-value attributes and 2) replace lists with sets. Rule 1 eliminates MultiValueAttr() :- Attribute(_,_,_,_,_,N) and its sub-patterns. Thus, MultiValueSetAttr() :- Attribute(_,_,_,X,_, N), SetType(X) is not in `Invalid`. Since rule 2 adds Set() to `Invalid`, we need to consider sub-patterns such as MultiValueSetAttr and SingleValueSetAttr() :- Attribute(_,_,_,X,_,1), SetType(X). The former is not supported by patterns in the model, because MultiValueAttr() was removed from `Invalid`. The latter is supported, so it is added to `Invalid`.

Each parameterized output signature *S* must first be instantiated. Each parameter of *S* refers to a predicate *Q* in the super-metamodel (e.g., ComplexType) and a variable binding. To instantiate *S*, we first generate $Q' = \{q \subseteq Q \wedge q \in SMM\}$ where *SMM* is the set of all predicates in the super-metamodel (i.e., *Q'* contains extensional predicates only). We then generate a new output pattern for each element of *Q'* that appears in `Present`.

For example, consider T1Attr[ComplexType:T1] from above. If we assume that `Present` contains ComplexType and StructuredType, but not AbstractType, then we instantiate T1Attr as:

Attribute(_,_,X,T1,_,1), StructuredType(X), ComplexType(T1)

Attribute(_,_,X,T1,_,1), StructuredType(X), StructuredType(T1)

In this case, we do not need to deal with attributes whose domain is AbstractType since they are not supported in the current state.

Having identified all of the patterns added by the transformation, we can now determine which patterns are removed by the transformation. Every pattern *P* in the input signature is removed from `Invalid` as are its sub-patterns. Super-patterns of *P* may also need to be removed. For example, if we first remove all multi-value attributes and then remove all single-value attributes that reference a set (e.g., by unnesting), we have also removed all attributes that reference a set. We assume that all non-leaf patterns (excluding predicates in the super-metamodel) are defined to be the union of their children. We therefore remove from `Invalid` every pattern *P'* for which no leaf descendant of *P'* is in `Present`.

This process is summarized below. Invalid is the set of constructs (currently used) that must not appear in the target model, and Valid is the set of constructs that can appear in the target. This procedure modifies Invalid based on a Transformation's input and output signatures. In the first line, the `let` operator introduces an alias. In the second line, Instantiate() is a method that instantiates every parameterized pattern.

```
procedure PatternSearch(Invalid, Valid, Transformation)
 let Present = Invalid ∪ Valid;
 for each P ∈ Transformation.OutputSignature.Instantiate() do
    if P∉Valid then Invalid.Add(P); end if
    for each S ∈ Descendants(P) do
       if Parents(S) ⊆ {P} ∪ Present and P∉Valid then
          Invalid.Add(S); end if
    end for
 end for
 for each P ∈ Transformation.InputSignature do
    Invalid.Remove(P);
    for each S ∈ Descendants(P) do Invalid.Remove(S); end for
 end for
 for each P ∈ Invalid ∩ Patterns do
    if LeafDescendants(P)=∅ then Invalid.Remove(P); end if
 end for
return
```

Having established the effect of applying a transformation to a state, we can now define a heuristic that estimates the distance (number of transformations) from a given state to the goal state. For A* to identify a minimal plan, this heuristic must be *admissible*, which means it never over-estimates the distance to the goal.

We estimate this distance by making several simplifying assumptions. First, we assume that every pattern in the input signature eliminates all of its sub-patterns and super-patterns. This provides an estimate of the number of patterns eliminated by the transformation. We then estimate the distance to the goal by the minimum number of transformations such that the sum of their estimates equals or exceeds the number of patterns in Invalid. This heuristic is admissible because it assumes that every transformation removes as many patterns as possible. Thus, the benefit of each transformation is overestimated, so the number of transformations needed is underestimated.

Our planning algorithm is intended to minimize the number of transformations in the plan. As a side-effect, no transformation appears more than once in a plan because its first appearance can be eliminated without affecting the overall correctness of the plan.

8. IMPLEMENTATION

A block diagram of our implementation is in Figure 8. It has the following components: (1) importers that translate an input model into the super-metamodel; (2) exporters that translate a model from the super-metamodel into its native syntax, (3) transformations, (4) metamodel rules, which are used by (5) the transformation plan generator (described in Section 7), and (6) the engine that executes transformation plans.

As others have noted [1][19], importing (or exporting) schemas to (or from) the super-metamodel is very straightforward, requiring little more than parsing (or generating) syntax and mapping between the corresponding names of constructs in the native metamodel and the super-metamodel.

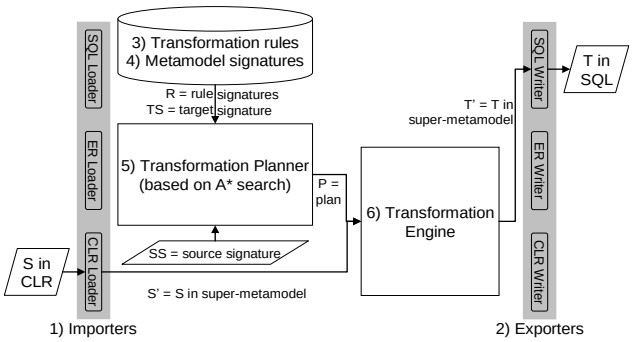


Figure 8: Architecture Overview

Generating data manipulation operations from forward- and reverse-views can be more challenging, depending on the complexity of the transformations and the expressiveness of the target language. If views are all expressible as project-select-join queries, then it is easy to generate SQL from the view definitions. If they have nested structures and union types, then the techniques of [26] can be used. In general, we were satisfied to use existing

code-generation technology and did not attempt to innovate or do much optimization of the generated code. Figure 9 shows an example of our code generator's output for the reverse-view of the Supplier class in the example of Figure 1.

```
CREATE VIEW Supplier AS
 SELECT T1.oid, T1.Name, OJ7.ISBN, OJ7.Cost
 FROM Company T1
 INNER JOIN
  (SELECT T2.oid FROM Supplier T2) OJ4
  ON T1.oid = OJ4.oid
 LEFT OUTER JOIN
  (SELECT T6.oid, T5.ISBN, T5.Cost
  FROM Item T5, Company T6
  WHERE T6.oid = T5.Inverse_Supplier_Parts_Supplier_oid) OJ7
  ON OJ4.oid = OJ7.oid
```

Figure 9 A Generated View

We wanted to use an existing Datalog engine, but were unhappy with the functionality and performance of ones that were available to us, so we built our own. It has native support for Skolem functions, for rules with compound heads (as described in Section 4), and for user-defined functions. We used the latter to generate forward- and reverse-views.

The size of our implementation is summarized in Table 3. The Datalog engine comprises more than half of our implementation. This includes the calculus representation, in-memory processing, view unfolding, and parser. The main ModelGen routines include the rules and plan generator. We ended up coding a few rules in C#, because it was easier to write or to understand. E.g., it was sufficiently hard to understand the recursion in the Datalog rule to remove structured attributes (Section 4.5) that we re-implemented it in 100 lines of C#. The logic for mapping inheritance structures into relations (Section 5) includes populating the mapping table from class annotations and generating reverse-views with negation when type flags are not used. The import/export routines include 120 lines of Datalog; the rest is in C#.

Our implementation runs inside the integrated development environment Microsoft Visual Studio 2005. It uses Visual Studio's in-memory database system and a prototype graphical model editor. The database system enables easier data sharing between tools, but it has significant overhead, which is what pushed us to develop the techniques of Section 0. Screen shots of a source model and generated target model displayed by the implementation appear in [5].

Table 3 Code Size

Component	**Lines of Code**
Super-metamodel representation	500
Datalog engine	6700
Main ModelGen routines	1500
Other imperative code	300
Mapping inheritance structures	1100
Import/export for CLR & SQL	800
SQL generation	900

Table 4 Execution Times in Milliseconds

	M1	M2	M3	M4
Number of elements	16	145	234	267
load the model	3	19	31	36
remove multiple containment	1	70	150	272
remove containment	2	75	154	279
Replace multi-valued attrs. with join table	1	3	5	6
update references to objects mapped to new objects	12	99	147	165
delete attributes referencing dangling types	7	92	184	193
add keys (imperative code)	< .5	1	1	1
in-line non-lexical references (imperative)	< .5	1	3	5
remove inheritance annotation-driven (imperative)	2	6	17	15
export the model	6	28	47	47
TOTAL	50	539	973	1286

Our implementation is relatively fast. Execution times for 4 models are shown in

Table 4. These models use a custom ER model, somewhat richer than CLR. For example, it permits a class to contain multiple classes, requiring us to use our transformation that eliminates multiple containment (see Section 4.6). The first row is the number of elements in each model. The remaining rows are times, measured in milliseconds averaged over 30 runs on a 1.5 GHz machine. The largest model, M4, generates 32 relational tables—not a huge model, but the result fills many screens.

9. RELATED WORK

The problem of translating data between metamodels goes back to the 1970's. Early systems required users to specify a schema-specific mapping between a given source and target schema (e.g., EXPRESS [24]). Later, Rosenthal and Reiner described schema translation as one use of their database design workbench [22]. Like our approach, it is generic, but it is manual (the user selects the transformations), its super-metamodel is less expressive (no inheritance, attributed relationships, or collections), and mappings are not automatically generated.

Atzeni & Torlone [1] showed how to automatically generate the target schema and source-to-target mapping. They introduced the idea of a repertoire of transformations over models expressed in a super-metamodel, where each transformation replaces one construct by others. They used a super-metamodel based on one proposed by Hull and King in [12]. They represented transformation signatures as graphs but transformation semantics was hidden in imperative procedures. They did not generate instance-level transformations, or even schema-level mappings between source and target models, which are main contributions of our work.

Two recent projects have extended Atzeni & Torlone's work. In [19], Papotti & Torlone generate instance translations via three data-copy steps: (1) copy the source data into XML, in a format that expresses their super-metamodel; (2) use XQuery to reshape the XML expressed in the source model into XML expressed in the target model; and (3) copy the reshaped data into the target system. Like [1], transformations are imperative programs. In [2], Atzeni et al. use a similar 3-step technique: (1) copy the source database into their relational data dictionary; (2) reshape the data using SQL queries; and (3) copy it to the target. Like our work, this project also uses Datalog rules to represent transformations.

In contrast to the above two approaches, we generate view definitions that directly map the source and target models in both directions. The views can be used to provide access to the source data using the target model, or vice versa, without copying any data at all. Or they can be executed as data transfer programs to move the data from source-to-target in just one copy step, not three. This is more time efficient and avoids the use of a staging area, which is twice the size of the database itself to accommodate the second step of reshaping the data. Moreover, neither of the above projects offer flexible mapping of inheritance hierarchies or incremental updating of models, which are major features our solution.

The notion of horizontal and vertical partitioning of inheritance hierarchies is well known [14]. However, as far as we know, no published strategies are as flexible as the one we proposed here.

Hull's notion of information capacity [11] is commonly used for judging the information preservation of schema transformations. In [11] a target schema dominates (i.e., has at least as much information capacity as) a source schema if there exists a mapping m from instances of the source to instances of the target, and a mapping m' from instances of the target to instances of the source such that m composed with m' is the identity on instances of the source. Our forward- and reverse-views are examples of such mappings.

Atzeni & Torlone [1] proposed two algorithms for plan generation. One of them assumes that transformations can be serialized with respect to dependencies, which is not true in general. By contrast, our algorithm can cope with cyclic dependencies among transformations. The second algorithm of Atzeni & Torlone [1] and the one of Papotti & Torlone [20] are both non-deterministic. Therefore, these algorithms either need to explore an exponential number of plans or might miss correct and desirable plans. Our algorithm uses A* [23]. In the worst-case (i.e., a heuristic that consistently returns 0), A* has complexity $O(n!)$. However, in practice our heuristic is a good approximation and a correct solution is found quickly. Note that if no plan exists then the Atzeni & Torlone algorithm is preferable. We optimize for the case in which n is moderately large and a plan exists.

Another rule-based approach was proposed by Bowers & Delcambre [5][7]. They focus on the power and convenience of their super-metamodel, Uni-Level Descriptions (UDL), which they use to define model and instance structures. They suggest using Datalog to query the set of stored models and to test the conformance of models to constraints.

Poulovasilis and McBrien [21] introduce a universal metamodel, based on a hypergraph. They describe schema transformation steps that have associated instance transformations. In [17], they use transformations to translate a given schema from one metamodel to another. They briefly sketch a generic algorithm to translate a relational schema to an ER schema, but not the other way around. They do not discuss view generation. Boyd and McBrien [8] apply and enrich these transformations for ModelGen. Although they do give precise semantics for the transformations, they are quite low-level (e.g., add a node, delete an edge). They do not explain how to abstract them to a practical query language, nor do they describe an implementation.

Jeusfeld and Johnen [13] describe a different generic technique for translating schemas between metamodels. They define an is-a hierarchy of metamodels. Given an schema in a source metamodel, their rule-based algorithm looks for constructs of the target metamodel with a common generalization in their universal metamodel. When there are choices, as there often are, the user is asked to decide. They apply it to reverse engineering a relational schema into an EER model. They do not discuss instance transformations.

In the commercial world, ModelGen is primarily implemented in a non-generic way to translate ER models into relational schemas. Despite the popularity of the approach, and the big literature on ER modeling, we have found surprisingly few papers on algorithms to accomplish the translation. The most complete one we know if is by Markowitz and Shoshani [15], who present a procedure for translating an EER model into a normalized relational schema. Their main focus is on preserving constraints and on intelligent naming of relational attributes. They show that the generated schemas have the same information capacity as the input schema, but they do not give algorithms to generate view definitions. They show how to merge relations that correspond to entity types that are related by inheritance. This is done explicitly, relation by relation, not via a table-driven approach like ours. An implementation is described in [16].

There is a substantial literature on languages for expressing schema transformations. These papers do not provide algorithms for translating an arbitrary schema in one metamodel into an equivalent schema in another metamodel. Rather, they provide operators one can use to write a program that translates a schema and its data into another schema with corresponding data. We close by mentioning a few examples here, and refer the reader to those papers for pointers to other related work.

Barsalou & Gagopadhyay [2] give a language (super-metamodel) to express multiple metamodels. They use it to produce query schemas and views for heterogeneous database integration. Issues of automated schema translation between metamodels and generation of inheritance mappings are not covered.

Miller et al. [18] define schema transformations between schemas expressed in their universal metamodel called Schema Intention Graphs. Using Hull's notion of information capacity they prove their transformations preserve information and can be used to compare the equivalence of two given schemas. However, they do not provide an algorithm to translate a schema of one metamodel into that of another.

Davidson and Kosky [10] define a Horn clause language based on a tuple calculus for expressing database transformations and constraints. They describe an implementation that can execute programs comprised of such transformations and constraints. Like

the previous two papers cited, they do not describe generic algorithms for schema translations between metamodels.

Claypool & Rundensteiner [9] describe operators to transform schema structures expressed in their graph metamodel. The transformation plan is user-defined, not automatically generated. They say that the operators can be used to transform instance data, but give no details. Song et al. [25] propose using graph grammars to translate models based on a user-defined mapping between the models.

10. CONCLUSION

In this paper, we described a rule-driven platform that can translate a model (e.g. database schema) from a source metamodel (e.g., relational, OO, XML) to a target metamodel and can be customized to specific metamodels and mapping languages with moderate effort. The main innovations are the ability to (i) generate provably-correct view definitions between the source and target models, (ii) map inheritance hierarchies to flat structures in a more flexible way, (iii) incrementally generate changes to the target model based on incremental changes to the source model, and (iv) generate transformation plans using a new and improved algorithm based on A*.

We implemented the algorithm and proved that it is fast enough for interactive editing and generation of models. We embedded it in a tool for designing object to relational mappings. Based on this experience, we believe that this schema translation technology is suitable for commercial deployment.

11. REFERENCES

[1] Atzeni, P. and R. Torlone, Management of Multiple Models in an Extensible Database Design Tool. *EDBT 1996*, 79-95.

[2] Atzeni, P., P. Cappellari and P. Bernstein: ModelGen: Model Independent Schema Translation. *EDBT 2006*.

[3] Barsalou, T. and D. Gangopadhyay: M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems. *ICDE 1992*, 218-227

[4] Bernstein, P.A., Applying Model Management to Classical Meta Data Problems. *CIDR* 2003, pp. 209-220.

[5] Bernstein, P., S. Melnik, P. Mork: Interactive Schema Translation with Instance-Level Mappings (demo), VLDB 2005.

[6] Bowers, S., L.M.L. Delcambre. On Modeling Conformance for Flexible Transformation over Data Models, Knowledge Transformation for the Semantic Web (at 15th ECAI), 19-26.

[7] Bowers, S. and L.M.L. Delcambre. The uni-level description: A uniform framework for representing information in multiple data models. ER 2003, LNCS 2813, pp 45-58.

[8] Boyd, M. and P. McBrien: Comparing and Transforming Between Data Models Via an Intermediate Hypergraph Data Model. J. Data Semantics IV: 69-109 (2005)

[9] Claypool, K.T. and E.A. Rundensteiner. Sangam: A Transformation Modeling Framework. DASFAA 2003: 47-54.

[10] Davidson, S.B. and A. Kosky: WOL: A Language for Database Transformations and Constraints. *ICDE 1997*: 55-65

[11] Hull, R. Relative Information Capacity of Simple Relational Database Schemata. *SIAM J. Comput.* 15(3): 856-886 (1986)

[12] Hull, R. and R. King. Semantic Database Modeling: Survey, applications and research issues. *ACM Comp. Surveys* 19(3): 201-260 (1987).

[13] Jeusfeld, M.A. and U.A. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas. *Int. J. Cooperative Inf. Syst.* 4(2-3): 237-258 (1995)

[14] Keller, A.M., R. Jensen, S. Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *SIGMOD 1993*, 523-528

[15] Markowitz, V.M. and A. Shoshani: Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach. *ACM TODS* 17(3): 423-464 (1992)

[16] Markowitz, V.M. and A. Shoshani: An Overview of the Lawrence Berkeley Laboratory Extended Entity-Relationship Database Tools. *ER 1994*: 333-350

[17] McBrien, P., and A. Poulovassilis: A Uniform Approach to Inter-model Transformations. *CAiSE 1999*: 333-348

[18] Miller, R.J., Y.E. Ioannidis, R. Ramakrishnan: Schema equivalence in heterogeneous systems: bridging theory and practice. *Inf. Syst.* 19(1): 3-31 (1994)

[19] Papotti, P. and R. Torlone: An Approach to Heterogeneous Data Translation based on XML Conversion. CAiSE Workshops (1) 2004: 7-19

[20] Papotti, P., R. Torlone. Heterogeneous Data Translation Through XML Conversion. *J. of Web Eng* 4,3: 189-204 (2005)

[21] Poulovassilis, A. and P. McBrien: A General Formal Framework for Schema Transformation. Data Knowl. Eng. 28(1): 47-71 (1998)

[22] Rosenthal, A. and D. Reiner: Tools and Transformations - Rigorous and Otherwise - for Practical Database Design. *ACM TODS 19(2):* 167-211 (1994)

[23] Russell, S. and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 2003.

[24] Shu, N.C., B. Housel, R. Taylor, S. Ghosh, V. Lum: EXPRESS: A Data EXtraction, Processing, and REStructuring System. *ACM TODS 2(2):* 134-174(1977)

[25] Song, G., K. Zhang, and R.Wong. Model management though graph transformations. *IEEE Symp. on Visual Languages and Human Centric Computing*, pp. 75-82, 2004

[26] Velegrakis, Y., *Managing Schema Mappings in Highly Heterogeneous Environments*, Ph.D. thesis, Univ. of Toronto, 2005.