Teaching a Schema Translator to Produce O/R Views

Peter Mork[1], Philip A. Bernstein[2], and Sergey Melnik[2]

[1]The MITRE Corporation and [2]Microsoft Research
pmork@mitre.org, {philbe, melnik}@microsoft.com

Abstract. This paper describes a rule-based algorithm to derive a relational schema from an extended entity-relationship model. Our work is based on an approach by Atzeni and Torlone in which the source EER model is imported into a universal metamodel, a series of transformations are performed to eliminate constructs not appearing in the relational metamodel, and the result is exported. Our algorithm includes novel features that are needed for practical object to relational mapping systems: First, it generates forward- and reverse-views that transform instances of the source model into instances of the target and back again. These views automate the object-to-relational (O/R) mapping. Second, it supports a flexible mapping of inheritance hierarchies to flat relations that subsumes and extends prior approaches. Third, it propagates incremental updates of the source model into incremental updates of the target. We prove the algorithm's correctness and demonstrate its practicality in an implementation.

1. Introduction

Object-to-relational (O/R) mapping systems are now mainstream technology. The Java Data Objects (JDO) specification is supported by many vendors of Enterprise Java Beans [20]. The Hibernate system is in widespread use [17]. Ruby on Rails includes the Active Record package [29]. And Microsoft recently released an ER-to-relational mapper in the next version of ADO.NET [1].

Developers often start building new applications by designing a conceptual model *E* of the application and translating it into a relational schema *R* to persist the data. In JDO and Hibernate, *E* is expressed as a set of Java classes. In ADO.NET, *E* is expressed in the Entity Data Model (EDM), a variant of the extended entity-relationship (EER) model [6]. Thus, the object-oriented (OO) constructs in *E* can include inheritance, associations, complex types, and nested collections, all of which have to be mapped to relational structures.

The problem of translating schemas between metamodels, or schema definition languages, has received attention in [2][3][9][16][23][25][26]**.** However, published approaches lack solutions to several issues that are required for practical applications: bidirectional semantic mappings, flexible translation of inheritance hierarchies, and incremental schema modification. These problems are non-trivial. They require architectural and algorithmic advances, which are the main subject of this paper. A preliminary, short description of the work reported here appears in [5].

Our basic strategy follows the rule-based approach of Atzeni and Torlone in [2]. Using this approach, we define a universal metamodel that has all of the main modeling constructs in the metamodels of interest, in our case EER and relational. New constructs can be added to the universal metamodel to support a new metamodel or extend an existing one. We then define a collection of transformation rules. For example, one simple rule transforms an entity type into a complex type (e.g., a relation). The goal is to execute a series of transformation rules whose composition eliminates from the source model all modeling constructs absent in the target metamodel. The result of this *translation step* is exported into the desired syntax.

Our first contribution is the generation of instance-level transformations between the source schema and generated target schema. While there are solutions to this problem (e.g., [3][25][26]), they require passing the instances through an intermediate generic representation. This is impractical for large databases and does not generate the view definitions that are required to drive EER-to-relational mapping systems. We take a different approach. We augment each transformation rule applied in the translation step to generate not only target schema elements but also forward- and reverse-views that describe how each eliminated construct of the source model is represented in the target. We have proved that these views are correct, i.e., do not lose information, and give one example proof in this paper.

The series of transformation rules executed in the translation step produces a series of elementary views. These views are composed via view unfolding to generate the final forward- and reverse-views between the source and target schemas. The correctness of the composition is ensured by the correctness of the elementary views. The composed views are expressed in terms of the universal metamodel. They are fed into a component that translates them into the native mapping language.

Our second contribution is a rich set of transformations for inheritance mapping. It allows the data architect 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 inheritance mapping strategies we know of, including horizontal and vertical partitioning [22], their combinations, and many new strategies. The transformations are driven by a data structure called a mapping matrix. We present 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 EER constructs.

Our third contribution is a technique for propagating incremental updates of the source model into incremental updates of the target. To do this, we ensure that an unchanged target object has the same id each time it is generated, so we can 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 arises when the schema translation process is interactive. A data architect analyzes different translation choices, switching back and forth between the source and target schemas, which may be large and thus require careful on-screen layout. Since it is unacceptable to regenerate the target schema and discard the layout information after changes in the schema translation, incremental update propagation is required.

Finally, we discuss the implementation of our O/R translation algorithm. We developed an extensible, rule-driven core that can be customized to specific model-translation tasks with moderate effort. To support efficient rule execution, we wrap the native meta-model APIs so that the rules directly manipulate the objects representing the model elements, avoiding the conversion penalty often incurred by using rule-based systems.

The rest of this paper is structured as follows. Section 2 describes our universal metamodel. Section 3 specifies our syntax for transformations and gives an example correctness proof for one of them. Section 4 describes how we support multiple strategies for mapping inheritance hierarchies into relations. Section 5 explains how we do incremental updating. Section 6 discusses our implementation. Section 7 discusses related work and Section 8 is the conclusion.

2. METAMODEL

Before we can define any transformation rules, we need to describe the universal metamodel in which they are expressed. The universal metamodel we use in this paper, called $\mathcal{U}$, is similar to the universal metamodel in [21]. $\mathcal{U}$ supports most of the standard constructs found 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, but it can easily be extended to incorporate additional features.

Table 1 lists the basic constructs of $\mathcal{U}$ and examples of their use in popular metamodels. We base our discussion of the semantics of $\mathcal{U}$ on its relational schema shown in Fig. 1. A detailed description and formal semantics for $\mathcal{U}$ appear in [24].

In $\mathcal{U}$ there are three simple types: Atomic types are called *lexicals*, which we assume to be uniform across all metamodels. The remaining simple types are collections, either *lists* or *sets* of some base type. For example, in SQL, apart from lexicals, the only simple type is a set whose base type is a tuple.

Complex types are either *structured* types (e.g., relations) or *abstract* types (e.g., entities). Complex types are related to other types via *attributes* and *containment*. For an attribute *A* The domain of *A* is the complex type on which *A* is defined, and the range of *A* is the type associated with *A*. An attribute can have minimum and maximum cardinality constraints. For example, in SQL every attribute's domain must be a structured type and its maximum cardinality must be one. A containment is

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	class	complex type
List Type			array	list
Set Type	table			
Attribute	column	attribute, relationship	field	attribute
Containment		aggregation		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 key of the superset type.

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 tells whether children are disjoint and cover the parent.
Specialization(SpecID, GenID, TypeID)
GenID references the parent generalization.
TypeID references the associated specialized type.

Fig. 1. Relational schema for universal metamodel $\mathcal{U}$

similar to an attribute; it establishes a (named) structural relationship between the parent type and the child type such that each such instance of the child type is nested within an instance of the parent type.

The constraints supported by $\mathcal{U}$ include key constraints, inclusion dependencies and generalizations. Each *key constraint* consists of a set of attributes that uniquely identify instances of some complex type. Multiple candidate keys can be defined for a complex type, but at most one primary key can be defined. An *inclusion dependency* establishes a relationship between a key and another complex type. For each attribute in an inclusion dependency there is a corresponding attribute in the related key. For any instance of a model containing an inclusion dependency, the projection of the inclusion attributes must be a subset of the projection of the key attributes. Finally, a *generalization* establishes a relationship between a complex type (the supertype) and a set of more specialized subtypes. Each subtype inherits any attributes or containment relationships associated with the supertype.

3. TRANSFORMATIONS

Using the Atzeni-Torlone approach, schema translation has four steps: (1) manually or automatically generate a valid transformation plan consisting of a sequence of transformations (2) import the source model (3) [translation step] execute the transformations in the plan, and (4) export the result. In this section and the next, we explain step (3), the transformations, which is the core of the algorithm and where most of our innovations lie. Due to lack of space, we omit a description of step (1), our A*-based algorithm for automatic generation of a transformation plan; it appears in [24]. We briefly discuss steps (2) and (4) in Section 6 on Implementation.

3.1. Defining a Transformation

Each step of a transformation plan is a transformation that removes certain constructs from the model and generates other constructs plus view definitions. 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 rule that produces an elementary forward view that expresses the target model as a view over the source, and *R* is a rule that produces an elementary reverse view that expresses the source as a view over the target.

Rules in *D* contain predicates, each of which is a construct in $\mathcal{U}$. Each rule is of the form "<body> ⇒ <head>", where <body> and <head> are conjunctions of predicates. 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:

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

AbstractType and **StructuredType** are predicates from Fig. 1, and *id* and *name* are variables. The Skolem function newAS(*id*) generates a new type ID for the structured type definition based on the abstract type's *id*. Skolem function names are prefixed by "new" to aid readability.

The semantics of a rule in *D* with body *b* and *n* terms in the head is defined by a Datalog program with *n* rules, each with 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*). We chose our rule syntax because it is less verbose than Datalog when many rules have the same body, which arises often in our transformations. In essence, each rule is a tuple-generating dependency [11] or a second-order dependency without equalities [12], if the Skolem functions are considered existentially quantified.

For some rules, expressing them in logic is impractical, because they are too verbose or hard to understand. Such rules can be implemented in an imperative language. But for succinctness and clarity, we use only the logic notation in this section.

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.

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 rules of a transformation are executed, a non-rule-based post-processing step deletes the elements identified in Delete predicates. For example, in the rule that replaces an abstract type by a structured type, the predicate Delete removes the abstract type 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 the identifier of an instance (i.e., an object) of the abstract type identified by *id*, then it is also the identifier of an instance (i.e., a tuple) of the structured type identified by newAS[*id*]. Notice that we use the same identifier to denote two different types of items, namely objects and tuples, which enables us to express instance-level mappings between them.

To generate such views in rules, 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. However, in this paper we will use the simpler and more readable notation "*id*(*x*) ⇒ newAS[*id*](*x*)".

We represent a 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, on the left side of Fig. 2, R is a structured type with attributes k and a. The value of k is a lexical type and the value of a is a structured type S with attributes b and c. 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 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 I_S of the source schema into a valid instance I_T of the target schema, and the reverse-view converts I_T back into I_S without loss of information. That is, the composition of the forward- and reverse-views is the identity. Unlike [9][13], we do not require the converse; there may be instances of the target model that cannot be converted into instances of the source. Our definition of correctness is more stringent than [26], which requires only that the forward view generates a valid instance of the target.

Sections 3.2–3.3 define two of the main transformations to convert from EER 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 rule predicates that would generate them. Since the forward- and reverse-views for the first transformation are inverses of each other, correctness is immediately apparent. We give a detailed correctness argument for the transformation of Section 3.3.

3.2. 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*))

We are careful in our use of negation, as in ¬**KeyConstraint** above, 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.

3.3. Remove Structured Attribute

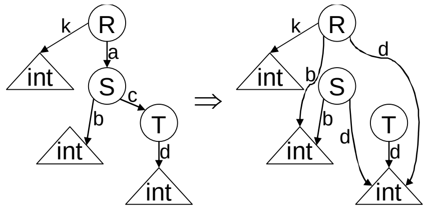

Fig. 2. Removing structured attributes

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. (If the latter is desired even if *S* has a primary key, then a user-defined tag on *a* can be used to ask that the latter rule be applied.) The transformation is applied iteratively to eliminate nested types.

For example, consider three structured types: *R*, *S* and *T* (see Fig. 2). *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*.

The model transformation rules are as follows (we use an underscore in a slot for an existential variable that appears only once in the rule, to avoid useless variables):

StructuredType(*domain*, *name*),
Attribute(*id*, _, *domain*, *range*, _, "One"), ¬ **LexicalType**(*range*, _)
$\Rightarrow$ **MixedTypeHelper**(*domain*, *name*)

Attribute(*id*, *name1*, *domain*, *range1*, *min1*, "One"),
StructuredType(*range1*, *name*), ¬MixedTypeHelper(*range1*, *name*),
Attribute(*attr*, *name2*, *range1*, *range2*, *min2*, "One"), Min(*min1*, *min2*, *min*)
$\Rightarrow$ **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*, _)
$\Rightarrow$ **KeyAttribute**(newSAKeyAttr(*keyAttr*, *attr*), *key*, newSA(*id*, *attr*)),
Map(keyAttr, newSAKeyAttr(*keyAttr*, *attr*))

The first rule identifies all "mixed" structured types—those types that reference another complex (i.e., non-lexical) type. In Fig. 2 *S* is a mixed type, but *T* is a "leaf" type. The second rule replaces an attribute (*id*) that references a leaf type (such as *c*) with the attributes (newSA(*id*, *attr*)) of the leaf type (in this case *d*). The third rule updates any key constraints that referenced the old attribute to reference the new attribute. After the first iteration, *S* becomes a leaf type, and attributes that reference it (such as *a*) are replaced by attributes of *S*. Thus, *a* is replaced with attributes *b* and *d*.

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

$id[x, z], attr_i[z, y] \Rightarrow \text{newSA}(id, attr_i)[x, y]$

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

$\text{newSA}(id, attr_1)[x, t_1], attr_1(s, t_1), \ldots, \text{newSA}(id, attr_k)[x, t_k], attr_k(s, t_k) \Rightarrow attr[x, s]$

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

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

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 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:

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

Since a is a function, $a(r, s_i)=a(r, s_j)$ for all i,j. 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:

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

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

3.4. Additional Transformations

In addition to the transformations in Sections 3.2-3.3, we have a transformation to replace a multi-valued attribute by a join relation and another to eliminate containments. They are quite simple, like converting an abstract type to a structured type, and are described in detail in [24]. We also implemented transformations to address more target metamodels. 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 3.2.

Replace an attribute with a maximum cardinality of N by a new attribute with a 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 unique key values, one for each value of the attribute; a set-valued attribute provides a single key value, namely, the set itself.

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 is T. This transformation creates an explicit association between values and their original positions in the list.

Stratify sets. This transformation takes a set of sets and converts it into a set of indexed structures; each nested set is assigned a unique identifier, which is associated with the values in that set. This transformation is needed to support the nested relational model.

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

3.5. Composing Transformations

The execution of 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., $V_F \circ v_f$ and $V_R \circ v_r$, using ordinary view unfolding, thereby generating V_F and V_R.

Table 2. A mapping matrix from classes to relations

	Person	Customer	Full-Time	Part-Time
P	id, name		id, name	id, name
C		id, name, account		
E			id, salary, hire, exempt	id, salary, hire
PT				id, hours
rel	{P}	{C}	{P, E}	{P, E, PT}
attr*	id, name	id, name, account	id, name, salary, hire, exempt	id, name, salary, hire, hours

4. INHERITANCE MAPPINGS

So far, we have assumed that all instances of a given source model construct are transformed using the same transformation rule. We now consider a more general strategy for mapping inheritance hierarchies of abstract types into structured types that allows the user to customize the transformation. Since this is the familiar object-to-relational mapping problem, we use the terms class and relation instead of abstract type and structured type.

Several strategies for mapping classes to relations exist. For example, consider the inheritance hierarchy in Fig. 3. Typical strategies for mapping these classes to flat relations include the following [17]: relation per concrete class (a.k.a. horizontal partitioning), in which each relation contains one column for every attribute, inherited or otherwise; relation per subclass (a.k.a. vertical partitioning), in which each relation contains a column only for the class' directly defined attributes; and relation per hierarchy, in which one relation stores all classes with a discriminator column to indicate which rows are in which concrete classes.

These simple strategies reflect only a few of the storage possibilities. For example, in Fig. 3, the designer has indicated that the system should partition Person (and its subclasses) using a horizontal strategy (⇔). However, Employee (and its subclasses) should be partitioned vertically (⇕), except for Full-Time whose attributes should be stored with those of the base class (∅).

Based on these declarations, we automatically generate the inheritance mapping shown in Table 2. Each column of this table corresponds to a class. Each of the first 4 rows corresponds to a database relation. (The rows *rel* and *attr** are discussed below.) Reading down a column, we can easily verify that every concrete class' (non-key) attributes are stored in some relation. Reading across a row, we can determine the relational structure. For example, because of horizontal parti-

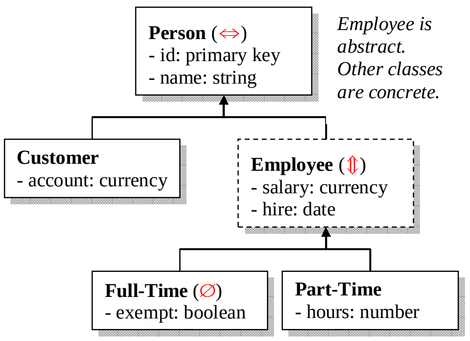

Fig. 3. An inheritance hierarchy

tioning, relation C contains all attributes (direct and inherited) of Customer. Similarly, vertical partitioning is used for Employee, so E is the only relation to contain salary and hire information.

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 $r \in R$.

A mapping matrix M describes how to map the attributes of classes 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 relation (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.

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: $c(x), a(x, y) \Rightarrow r(x), r.a(x, y)$. 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) $r \in rel(c) \rightarrow PK(c) \subseteq M[\mathrm{r,c}]$

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 there exists $r \in (rel(c\ ) \cap Flagged)$, so we can use $r.flag$ to identify instances of c:

$\mathrm{r}(x), \mathrm{r.flag}(x, \mathrm{TypeID(c)}) \Rightarrow \mathrm{c}(x)$

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:

$$\bigwedge_{r \in rel(c)} \mathrm{r(x)} \bigwedge_{r \notin rel(c)} \neg \mathrm{r(x)} \Rightarrow \mathrm{c(x)}$$

In relational algebra, this is the join of all $r \in rel(c)$ composed with the anti-semijoin of $r \notin rel(c)$, which can be further simplified exploiting the inclusion dependencies between the relations in $rel(c)$. In both cases, the reverse view is an inverse of the forward view. The reverse-view for a given attribute is read directly from the mapping matrix. It is simply the union of its appearances in M:

$\{ \text{r.a}(x, y) \Rightarrow \text{c.a}(x, y) \mid c \in C, r \in R, a \in M[c,r] \}$

The mapping matrix *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 *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: ⇕, ⇔, or ∅. Strategy ⇕ does vertical partitioning, the default strategy: each inherited property is stored in the relation associated with the ancestor class that defines it. Strategy ⇔ yields horizontal partitioning: the direct instances of the class are stored in one relation, which contains all of its inherited properties. Strategy ∅ means that no relation is created: the data is stored in the relation 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 matrices 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 in Fig. 4 (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.

The steps of the algorithm are as follows:

1. Each class labeled horizontal or vertical requires its own relation
2. A relation that contains concrete class c must include c's key so that c can be reassembled from all relations that store its attributes.
3. This is the definition of horizontal partitioning
4. These are the attributes of c that need to be assigned to some relation
5. These attributes have already been assigned to a relation $r_{p,}$ so use that relation.
6. The remaining attributes of c are assigned to c's target relation
7. Now populate the matrix for c's children

```
procedure PopulateMappingMatrix(c: class, r: target relation)
if (strategy($c$) $\in \{\Updownarrow, \Leftrightarrow\}$) then r = ⟨new relation⟩ end if                  // 1
if (c is concrete) then
  $M[r, c] = M[r, c] \cup$ ⟨key attributes of $c$⟩                                                  // 2
  if (strategy($c$) = $\Leftrightarrow$)
  then $M[r, c] = M[r, c] \cup$ ⟨declared and inherited attributes of $c$⟩                           // 3
  else
    toPlace = attrs = ⟨declared and inherited non-key attributes of $c$⟩                             // 4
    for each relation $r_p$ created for ancestor class of $c$, traversing bottom-up
      for each cell $M[r_p, p]$ do
        $M[r_p, c] = M[r_p, c] \cup (M[r_p, p] \cap$ toPlace)                                        // 5
        toPlace = toPlace – $M[r_p, p]$
      end for
    end for
    $M[r, c] = M[r, c] \cup$ toPlace                                                                 // 6
  end if
end if
for each child $c'$ of $c$ do PopulateMappingMatrix($c'$, $r$) end for                               // 7
return
```

Fig. 4. PopulateMappingMatrix generates a mapping matrix from an annotated schema

5. INCREMENTAL UPDATING

Translating a model between metamodels can be an interactive process, where the user 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 loses any customization the user performed 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. This incremental updating also improves performance. For example, our implementation uses a main memory object-oriented database system, in which a full regeneration of the target schema from a large source model can take a minute or so.

Let m_0 be a source model and $m_1, \ldots, m_n$ 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 $f_1, \ldots, f_n$ 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), but 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 is 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.

6. IMPLEMENTATION

Our implementation runs inside an integrated development environment. It has a graphical model editor and an in-memory object-oriented database system (OODB) that simplifies data sharing between tools and supports undo/redo and transactions.

The EER model and schemas are stored as objects in the OODB. We wrote relational wrappers that expose an updateable view of the objects. The wrappers are generic code that use reflection and on-the-fly generated intermediate language code. We then wrote rules that translate between those wrappers and the relational representation of $\mathcal{U}$. As others have noted [2][23][25], this translation is very straightforward; since there is a 1:1 mapping between constructs of the source and target metamodels and universal metamodel (i.e., $\mathcal{U}$), the translation rules are trivial.

We wrote our own rules engine, partly because of limitations on the functionality of Datalog engines that were available to us and partly because we wanted native support for rules with compound heads (see Section 3). It supports Skolem functions and user-defined functions. We used the latter to generate forward- and reverse-views.

The size of our implementation is summarized in Fig. 5 (viewed best in the electronic version in color). The rule engine is more than half of our code. It includes the calculus representation, in-memory processing, view unfolding, and parser. The main routines include the rules and plan generator (described in [24]). We coded a few rules in C#, such as the rule to remove structured attributes since the recursion was hard to understand. The logic for mapping inheritance structures into relations includes populating the mapping matrix from class annotations and generating reverse-views with negation when necessary. The import/export routines include 120 lines of rules; the rest is in C#.

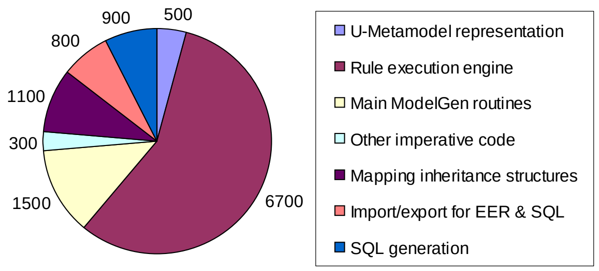

Fig. 5. Code size (in lines of code)

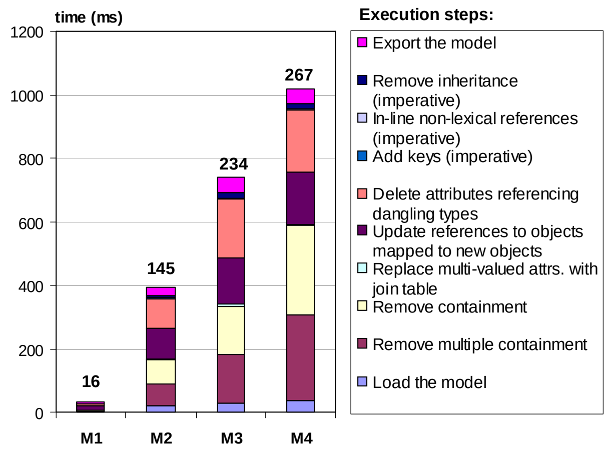

Fig. 6. Execution times in milliseconds

Our implementation is relatively fast. Execution times for four models are shown in Fig. 6. These models use a custom EER model—a rather rich one. For example, it permits a class to contain multiple classes, requiring us to use our transformation that eliminates multiple containment. The number of elements in each model is shown above each bar. The execution time was measured in milliseconds and averaged over 30 runs on a 1.5 GHz machine. The largest model, M4, generates 32 relations—not a huge model, but the result fills many screens.

7. 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 [30]). Later, Rosenthal and Reiner described schema translation as one use of their database design workbench [28]. It is generic but manual (the user selects the transformations), its universal metamodel is less expressive (no inheritance, attributed relationships, or collections), and mappings are not automatically generated.

Atzeni and Torlone [2] showed how to automatically generate the target schema. They introduced the idea of a repertoire of transformations over models expressed in a universal metamodel (abbr. UMM), where each transformation replaces one construct by others. They used a UMM based on one proposed by Hull and King in [19]. 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 and Torlone's work. In [25], Papotti and Torlone generate instance translations via three data-copy steps: (1) copy the source data into XML, in a format that expresses their UMM; (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 [2], transformations are imperative programs. In [3], Atzeni et al. use a similar 3-step technique, except transformations are Datalog rules: (1) copy the source database into their relational data dictionary; (2) reshape the data using SQL queries that express the rules; and (3) copy it to the target.

In contrast to the above two approaches, we generate view definitions that *directly* map the source and target models in both directions and could drive a data transformation runtime such as [1][17]. The views provide access to the source data using the target model, or vice versa, without copying any data at all. If they were executed as data transfer programs, they would move 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.

Transformation strategies from inheritance hierarchies to relations, such as horizontal and vertical partitioning, are well known [15][22]. However, as far as we know, no published strategies allow arbitrary combinations of vertical and horizontal partitioning at each node of an inheritance hierarchy, like the one we proposed here.

Hull's notion of information capacity [18] is commonly used for judging the information preservation of schema transformations. In [18] a source and target schema are equivalent if there exists an invertible mapping between their instances. Our forward- and reverse-views are examples of such mappings.

Using a UMM called GER, Hainaut has explored schema transformations for EER and relational schemas in a sequence of papers spanning two decades. He presented EER and relational transformations in [13]. Although instance transformations were mentioned, the focus was on schema transformations. Instance mappings for two transformations are presented in [14] as algebraic expressions. In this line of work, instance transformations are mainly used for generating wrappers in evolution and migration scenarios. An updated and more complete description of the framework is in [16].

Poulovasilis and McBrien [27] introduce a universal metamodel, based on a hypergraph. They describe schema transformation steps that have associated instance transformations. Boyd and McBrien [9] apply and enrich these transformations for ModelGen. Although they do give a precise semantics for the transformations, it is 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.

Another rule-based approach was proposed by Bowers and Delcambre [7][8]. They focus on the power and convenience of their UMM, Uni-Level Descriptions, 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.

Barsalou and Gagopadhyay [4] give a language (i.e., UMM) 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.

Claypool and Rundensteiner [10] describe operators to transform schema structures expressed in a graph metamodel. They say the operators can be used to transform instance data, but give no details.

8. CONCLUSION

In this paper, we described a rule-driven platform that can translate an EER model into a relational schema. The main innovations are the ability to (i) generate provably-correct forward and reverse view definitions between the source and target models, (ii) map inheritance hierarchies to flat structures in a more flexible way, and (iii) incrementally generate changes to the target model based on incremental changes to the source model. We implemented the algorithm and demonstrated that it is fast enough for interactive editing and generation of models. We embedded it in a tool for designing object to relational mappings. Commercial deployment is now underway.

9. REFERENCES

[1] ADO.NET, http://msdn.microsoft.com/data/ref/adonetnext/

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

[3] Atzeni, P., P. Cappellari and P. Bernstein. ModelGen: Model Independent Schema Translation. *EDBT 2006*, 368-385

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

[5] Bernstein, P., S. Melnik, P. Mork: Interactive Schema Translation with Instance-Level Mappings (demo), VLDB 2005, 1283-1286

[6] Blakeley, J., S. Muralidhar, A. Nori. The ADO.NET Entity Framework: Making the Conceptual Level Real, ER 2006, LNCS 4215, 552-565

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

[8] 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, 45-58

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

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

[11] Fagin, R.: Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM TODS* 2(3): 262-278 (1977)

[12] Fagin, R., P. G. Kolaitis, L. Popa, and W.C. Tan. Composing Schema Mappings: Second-Order Dependencies to the Rescue. *ACM TODS* 30(4): 994-1055 (2005)

[13] Hainaut, J-L. Entity-Generating Schema Transformations for Entity-Relationship Models. *ER* 1991: 643-670

[14] Hainaut, J-L.. Specification preservation in schema transformations—Application to semantics and statistics. *Data Knowl. Eng.* 16(1): 99-134 (1996)

[15] Hainaut, J-L, J-M Hick, V. Englebert, J. Henrard, and D. Roland. Understanding the Implementation of IS-A Relations. *ER* 1996, 42-57

[16] Hainaut, J-L., The Transformational Approach to Database Engineering. In *Generative and Transformational Tech. in Software Eng.* LNCS 4143: 89-138, 2006

[17] Hibernate, http://www.hibernate.org/

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

[19] Hull, R. and R. King. Semantic Database Modeling: Survey, Applications and Research Issues. *ACM Comp. Surveys* 19(3): 201-260 (1987)

[20] Java Data Objects, http://java.sun.com/products/jdo

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

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

[23] Kensche, D., C. Quix, M. A. Chatti, and M. Jarke: GeRoMe. A Generic Role Based Metamodel for Model Management. *OTM Conferences* (2) 2005: 1206-1224

[24] Mork P., Bernstein, P.A., S. Melnik: A Schema Translator that Produces Object-to-Relational Views. Technical Report MSR-TR-2007-36. http://research.microsoft.com.

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

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

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

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

[29] Ruby on Rails, http://api.rubyonrails.org/

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