Interactive Schema Translation with Instance-Level Mappings

Philip A. Bernstein
Microsoft Research, USA
philbe@microsoft.com

Sergey Melnik
Microsoft Research, USA
melnik@microsoft.com

Peter Mork*
Microsoft Research, USA
pmork@cs.washington.edu

Abstract

We demonstrate a prototype that translates schemas from a source metamodel (e.g., OO, relational, XML) to a target metamodel. The prototype is integrated with Microsoft Visual Studio 2005 to generate relational schemas from an object-oriented design. It has four novel features. First, it produces instance mappings to round-trip the data between the source schema and the generated target schema. It compiles the instance mappings into SQL views to reassemble the objects stored in relational tables. Second, it offers interactive editing, i.e., incremental modifications of the source schema yield incremental modifications of the target schema. Third, it incorporates a novel mechanism for mapping inheritance hierarchies to relations, which supports all known strategies and their combinations. Fourth, it is integrated with a commercial product featuring a high-quality user interface. The schema translation process is driven by high-level rules that eliminate constructs that are absent from the target metamodel.

1 Introduction

We present a prototype that translates a source model, such as an object-oriented schema, into a target model, such as a relational schema, and produces an executable instance-level mapping from the source model to the target model. The mechanism that we developed works with many metamodels such as ER, SQL, OO, and XSD. Example applications include translating an interface definition into a SQL schema to support an object-to-relational mapper, translating a UML or ER model into a C# interface definition to support a modeling tool, or translating a SQL schema into an XML schema for data exchange.

We integrated our prototype with the Microsoft Visual Studio 2005 to support object-to-relational mapping scenarios that are common in business applications. Typically, in such applications the business logic runs on top of an object model, whereas the actual data is persisted in SQL databases. To shield the applications from impedance mismatch and storage management issues, an abstraction layer is used to translate the data access operations on the object model into SQL queries and updates. The mappings produced by our prototype can be used to drive the abstraction layer, and to rewrite queries on objects as queries on relations. The mappings are expressed in a standard logic-based data transformation language and support a wide range of alternatives for persisting objects as relations. An implemented SQL generator compiles the mappings into SQL views, and can also be used at deployment time to compile queries rewritten using the mappings into SQL.

Developing an object-to-relational mapping is an interactive process, during which the source schema undergoes revisions, and various mapping options are explored, such as choosing a horizontal or vertical strategy for mapping inheritance. Typically, the user (a database designer) wants to see right away how her design choices affect the target schema and the generated views. To improve the user experience in such scenarios, our prototype translates schemas in a stateful fashion: the target schema is updated incrementally instead of being re-created upon each modification. In particular, the graphical layout of the target schema is preserved during editing of the source schema.

The choice of the inheritance mapping strategy is one of the main tuning knobs in representing objects as relations. We developed a novel approach for capturing inheritance mapping strategies that subsumes and generalizes the standard techniques. Essentially, it allows the engineer to decide on the number of relations used for representing a subclass hierarchy, and to assign each direct or inherited property of a class independently to any relation. The prototype allows choosing an inheritance mapping strategy on a per-class basis, and supports any combination of the known horizontal/vertical/union strategies.

The system translates schemas by first transforming the source model S into a representation S_0 in a universal metamodel. Then a sequence of rule-based transformations eliminates from S_0 all modeling constructs (e.g., many-to-many relationships, aggregation, or generalization) that are absent from the target metamodel, producing in n steps a

*Current affiliation: University of Washington, USA

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

model S_n. Finally, S_n is cast into the target metamodel, thereby producing the output model S'. This style of model translation was suggested in [2]. A rule-based implementation was demonstrated in [1].

The major novel features of our prototype are generation of instance-level mappings, interactive editing, a general mechanism for dealing with inheritance, and integration with a commercial product featuring a high-quality user interface. Instance-level mappings are computed by composing [5] the elementary data transformations produced upon eliminating each successive modeling construct.

At Microsoft Research, we are building a model-management system that helps engineers to develop meta data applications more effectively using a set of high-level operations on models and mappings [4]. The task of schema translation is abstracted as the model-management operator ModelGen [3]. In the remainder of this proposal we describe the scope of the demonstration and highlight some technical details.

2 What is Demonstrated

We demonstrate the key features of the prototype using a sample business application designed using Microsoft Visual Studio 2005. Figure 2 shows the screenshot of a schema containing several business entities, which correspond directly to C# classes deployed in the application. The schema is presented in the Entity Designer view of the Microsoft Business Framework, which runs on top of Visual Studio 2005. The schema defines types for customers, (credit card) accounts, which are specialized into personal and business accounts, and addresses linked to the accounts. Customers are identified using a compound key (`SSN`, `DOB`). Each customer owns one or more accounts, which are billed to US addresses. `Owns` is an m:n-association, while `billTo` is an aggregation, i.e., US addresses are existentially dependent on the accounts.

Figure 3 shows the relational schema generated by our prototype from the source schema of Figure 2. The relational schema is shown in the Data Source view of the Business Intelligence Development Studio, a part of SQL Server 2005 that runs in Visual Studio 2005.[1] Under the default settings,

- m:n-associations are translated into join tables (so association `owns` becomes a join table JT_Customer_owns)
- aggregation is translated into inverse attributes (hence, `billTo` becomes an inverse attribute Inverse_Account_billTo_Account_AccountNo of Address)
- unique keys are added whenever source entities lack keys (so table Address contains an oid attribute)
- each entity is turned into a relation that stores its non-inherited properties and inherited keys (e.g., table BusinessAccount stores `CompanyName`, but not the inherited `AnnualFee`).

[1] The user interface shown in Figures 2 and 3 is subject to change prior to the official release of the Visual Studio 2005 and SQL Server 2005 products.

We demonstrate how interactive modifications result in incremental updates to the target schema. For example, renaming the property `AccountNo` in the source schema results in renaming of the corresponding relational attributes and constraints in the tables JT_Customer_owns, Account, BusinessAccount, and Address. As another example, changing the association-end multiplicity on `owns` from 0..n ("ZeroOrMany") to 0..1 ("ZeroOrOne") replaces the join table JT_Customer_owns in the target schema by attribute owns_Account_AccountNo in the Customer table.

We show how the default settings can be changed to obtain a wide range of object-to-relational mappings. For example, in the scenario shown in Figures 2 and 3 the source schema is annotated such that no table USAddress is generated: `USAddress` and `Address` objects share the table Address and are distinguished using a flag attribute EType.

Under the default strategy, retrieving objects of type `BusinessAccount` requires a join of the tables Account and BusinessAccount. If most accounts are business accounts, the join can be avoided by storing all direct instances of the `BusinessAccount` class in one self-contained table,

```
BusinessAccount(AccountNo, APR, AnnualFee,
                CompanyName, CompanyAddress)
```

while the direct instances of `Account` and `PersonalAccount` are stored in the table Account. We show how this and other strategies can be selected by annotating the source schema.

We illustrate the effect of choosing different strategies on the generated mappings and SQL views. For example, the `Customer` objects can be reconstructed using the following view on the relational schema of Figure 3:

```
CREATE VIEW CustomerView AS
SELECT T1.SNN, T1.DOB, ..., T5.AccountNo, T5.APR, ...
FROM Customer T1 LEFT OUTER JOIN
(SELECT T3.*, T4.*
 FROM Customer T3, JT_Customer_owns T2, Account T4
 WHERE T4.AccountNo = T2.To_Account_AccountNo AND
       T3.DOB = T2.From_Customer_DOB AND
       T3.SSN = T2.From_Customer_SSN) T5
ON T1.SSN = T5.SSN AND T1.DOB = T5.DOB
```

If we use the above-mentioned strategy for mapping `BusinessAccounts` to a self-contained table, we obtain a different view definition, in which `Accounts` are retrieved as a union of Account and BusinessAccount tables:

```
CREATE VIEW CustomerView AS
SELECT T1.SNN, T1.DOB, ..., T7.AccountNo, T7.APR, ...
FROM Customer T1 LEFT OUTER JOIN
(SELECT T3.*, T6.*
 FROM Customer T3, JT_Customer_owns T2,
      ((SELECT T4.AccountNo FROM Account T4
        WHERE T4.EType IN ('Account', 'PersonalAccount'))
 UNION (SELECT T5.AccountNo
          FROM BusinessAccount T5)) T6
 WHERE T6.AccountNo = T2.To_Account_AccountNo AND
       T3.DOB = T2.From_Customer_DOB AND
       T3.SSN = T2.From_Customer_SSN) T7
ON T1.SSN = T7.SSN AND T1.DOB = T7.DOB
```

3 Technology in Spotlight

In the remaining space, we briefly outline the key ideas behind two of the novel techniques implemented in the prototype: generation of instance-level mappings and a general approach for supporting inheritance.

3.1 Constructing Mappings by Way of Composition

The source schema S_0 represented in a universal metamodel is translated into the target schema S_n using a series of transformations. Each transformation takes as input the current snapshot S_i of the schema and produces as output schema S_{i+1} and the mapping m_{i+1} between S_i and S_{i+1}. The final mapping m between S_0 and S_n is obtained by composing the intermediate mappings as $m = m_1 \circ m_2 \circ \cdots \circ m_n$.

In general, mapping composition is a hard problem [5]. In the case of schema translation we exploit the fact that each mapping m_i is given by a combination of two view definitions. The *forward* view $f(m_i)$ defines S_i as a view of S_{i-1}, whereas the *backward* view $b(m_i)$ defines S_{i-1} as a view of S_i. Hence, m is given by views $f(m)$ and $b(m)$ such that $f(m) = f_1(m) \circ \cdots \circ f_n(m)$ and $b(m) = b_n(m) \circ \cdots \circ b_1(m)$. Views $f(m)$ and $b(m)$ are computed using standard view unfolding algorithms.

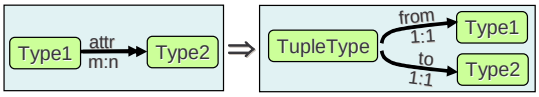
To illustrate the forward and backward views, consider Figure 1. The figure shows schematically a transformation that eliminates m:n-associations. In addition to modifying the schema, the transformation produces an intermediate instance mapping that expresses the new constructs TupleType, from, and to in terms of attr, and vice versa. (Applying this transformation gives rise to the table JT_Customer_owns in Figure 3.)

Mapping composition "links dynamically" the above transformation with the ones that eliminate aggregation, inheritance, and other constructs absent from the relational model. The output mapping is expressed in a neutral predicate calculus representation that can be compiled into a suitable language (e.g., SQL in the demonstration).

3.2 Inheritance Mapping Strategies

Eliminating inheritance is by far the most complex transformation implemented in the prototype. It is driven by a data structure called an *inheritance mapping table* that specifies the target relation for each direct or inherited property of each class. The rows of such tables correspond to classes, the columns correspond to relations.

To illustrate, Tables 1 and 2 define two alternatives for mapping the class hierarchy rooted at `Account` into two relations. For example, the last row of Table 1 tells us how the objects of type `BusinessAccount` are persisted: APR and AnnualFee are stored in relation Account only, CompanyName and CompanyAddr are stored in relation BusinessAccount only, and AccountNo is stored in both relations. To reconstruct the objects, we join the two relations. The second column of Table 1 tells us how to populate the relation Account: we take a union of



Forward view:
TupleType(z) ⇔ ∃x∃y (attr(x, y) ∧ z = Skolem(x, y))
from(z, x) ⇔ ∃y (attr(x, y) ∧ z = Skolem(x, y))
to(z, y) ⇔ ∃x (attr(x, y) ∧ z = Skolem(x, y))

Backward view:
attr(x, y) ⇔ ∃z (from(z, x) ∧ to(z, y))

Figure 1: Eliminating m:n-associations

the objects of types `Account`, `PersonalAccount`, and `BusinessAccount`. The star in Account* indicates that the relation needs to have a flag attribute (EType) to disambiguate the origin of the tuples.

The known inheritance mapping strategies and their combinations are supported by varying the number of the target relations and populating the cells of the table in various ways. The only requirement for populating the cells is that the union of the properties listed in each row covers all direct and inherited properties of the respective class.

We implemented algorithms for generating the mapping expressions for an arbitrary assignment of the table cells. (These mappings are composed with those obtained by other transformations.) Moreover, we developed mechanisms for populating inheritance mapping tables using simple tuning knobs exposed to the engineers. For example, each class can be annotated to use one of the three strategies, called "Own", "All", or "None". "Own" performs 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. The mentioned annotations exploit the flexibility of the inheritance mapping tables only partially, yet are easy to communicate to the engineers.

References

[1] P. Atzeni, P. Cappellari, P. A. Bernstein. ModelGen: Model Independent Schema Translation (Demo). In *Proc. ICDE*, 2005.

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

[3] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. CIDR*, 2003.

[4] S. Melnik, P. A. Bernstein, A. Halevy, E. Rahm. Supporting Executable Mappings in Model Management. In *Proc. ACM SIGMOD*, 2005.

[5] A. Nash, P. A. Bernstein, S. Melnik. Composing Mappings Given by Embedded Dependencies. In *Proc. PODS*, 2005.

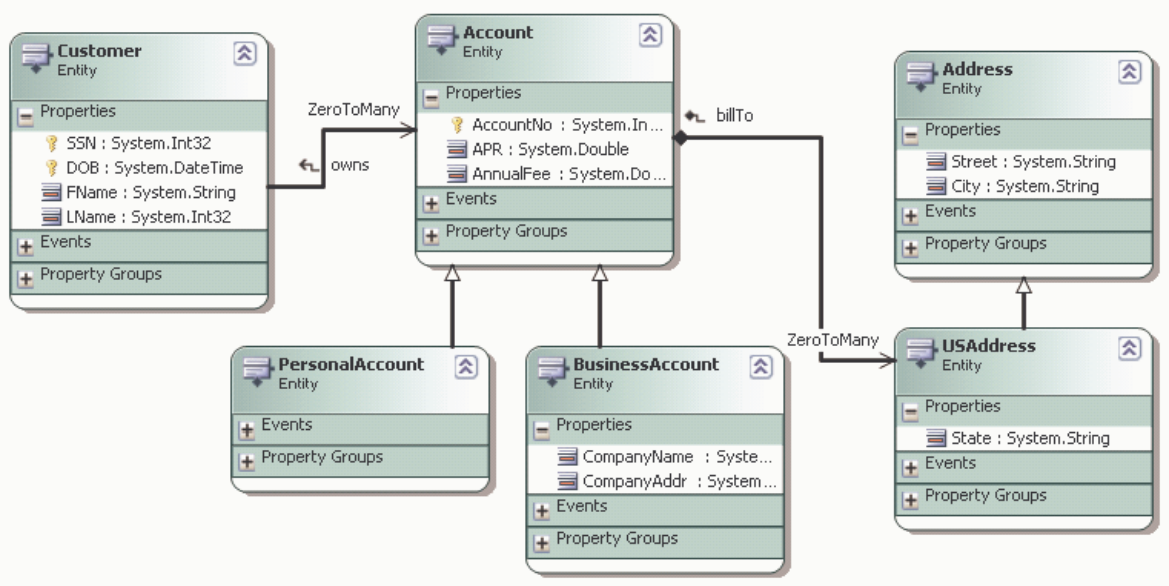


Figure 2: Screenshot showing a source model (object-oriented schema)

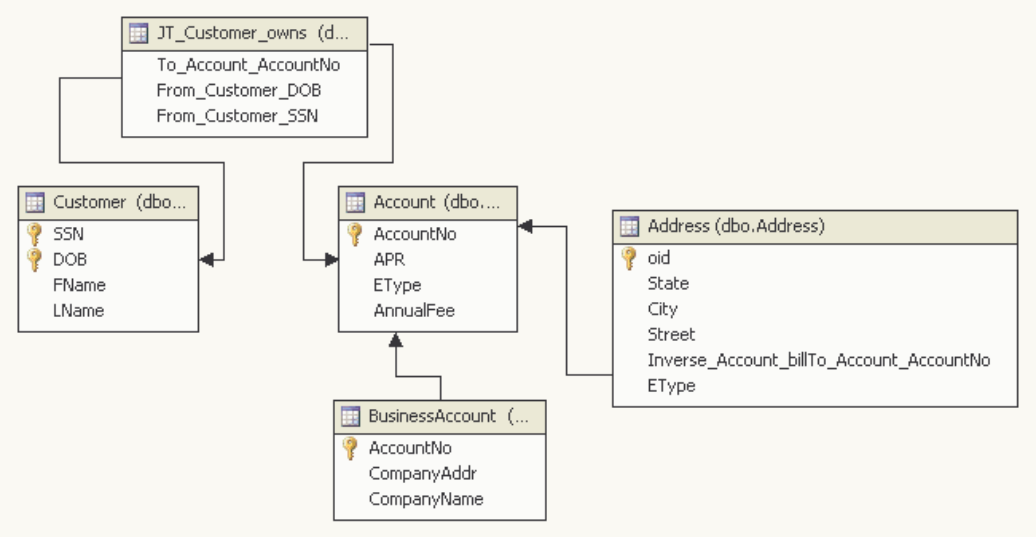


Figure 3: Screenshot showing the generated target model (relational schema)

Entity	Account*	BusinessAccount
Account	{AccountNo,APR,AnnualFee}	
PersonalAccount	{AccountNo,APR,AnnualFee}	
BusinessAccount	{AccountNo,APR,AnnualFee}	{AccountNo,CompanyName,CompanyAddr}

Table 1: Portion of inheritance mapping table used to produce the relational schema of Figure 3

Entity	Account*	BusinessAccount
Account	{AccountNo,APR,AnnualFee}	
PersonalAccount	{AccountNo,APR,AnnualFee}	
BusinessAccount		{AccountNo,APR,AnnualFee,CompanyName,CompanyAddr}

Table 2: Alternative inheritance mapping strategy