Model Independent Schema and Data Translation

Paolo Atzeni(1), Paolo Cappellari(1), Philip A. Bernstein(2)

(1) Università Roma Tre
Roma, Italy
[atzeni, cappellari]@dia.uniroma3.it

(2) Microsoft Research
Redmond, WA, USA
philbe@microsoft.com

Abstract

We illustrate the development of ExtendedModelGen, the model management operator that translates schemas and instances from one model to another, for example, from object to SQL or from SQL to XSD. The operator can be used to generate database wrappers (e.g. object or XML to relational), default user interfaces (e.g. relational to forms), or default database schemas from other representations. The approach handles the translation of instances: given a source instance I, referring to a schema S expressed in a source model, and a target model TM, it generates a schema S' expressed in TM that is "equivalent" to S and an instance I' of S' equivalent to I. A wide family of models is handled, by using a metamodel in which models can be succinctly and precisely described. The approach makes use of a dictionary to represent models, schemas, instances as well as the rules that implement translations. Rules are written in a Datalog dialect with oid invention and can therefore be easily customized.

1 Introduction

Many application settings involve the management of heterogeneous data and the need to translate actual data and their description from one framework to another. In the database world, we often have different systems that we need to use to handle our data, following different models, and we might need to exchange data from one to another. Even variations of models are often enough to create difficulties: for example, while most database systems are now object relational, it is the case that the actual features offered by different systems almost never coincide and so any migration requires a conversion. Recent developments of technology in various directions have introduced more heterogeneity and more need for translations. To mention a major area, the growth of XML as a tool for data representation and exchange has generated a set of issues, including *(i)* the need to have object-oriented wrappers for XML data, *(ii)* the translation from nested XML documents into flat relational databases and vice versa, *(iii)* the conversion from one company standard to another (for example, a standard could require that attributes are used for simple values and sub-elements for nesting whereas another that attributes are not used and all data is in sub-elements).

In all these frameworks, there is the need to translate both schemas and instances from one model to another. The models of interest can be significantly different, including those traditionally used in databases, as well as many others, such as those for XML documents, Web site structure descriptions, data warehouses, and restrictions or variants of each of them: for example, in *(iii)* above, we would have two models, one with attributes and another without.

A slightly more restricted problem arises in the design process, where attention can often be limited to schemas, but on the other hand an even larger variety of models may arise as specialists use many different formalisms: it would be sufficient to mention the E-R model, with its countless variations, UML (often used under some restrictions), other conceptual models. The interest here is indeed huge: for example, almost every database designer translates schemas from (his/her preferred version of) the ER model or UML to the relational model.

The common requirement to the above settings is the need to handle different *models* and to be able to translate schemas from one model to another: given two models M_1 and M_2 and a schema S_1 of M_1 we want to produce the translation of S_1 into M_2, that is, a schema S_2 of M_2 that properly represent S_1. If data is of interest, one wants to be able to translate

Submitted for publication, February 2005

them as well: given a database instance I_1 of S_1 we want to produce also an instance I_2 of S_2 that properly represent I_1.

These translation problems are often tackled by means of ad-hoc solutions, for example by writing a program for each specific application, but this is clearly very laborious, time consuming, and hard to maintain. Bernstein et al. [5, 6] have recently argued for generic solutions for them, as well for other problems that require the management of descriptions of application artifacts. Indeed, they proposed a high level approach, called *model management*, based on a set of operators to be applied to schemas.[1] A specific operator in the family is *ModelGen*, which translates schemas from a source to a target model, exactly as we required above with respect to the schema level; ModelGen is also supposed to provide as a result an explicit mapping between the constructs in the source schema and those in the target one. We use the term *ExtendedModelGen* to refer to the extension of it that also translates database instances. Clearly, like other model management operators, ExtendedModelGen is useful only if it is model-independent, so that it works for all data models of interest.

An early approach to ModelGen was proposed by Atzeni and Torlone [2, 3] who developed a tool, called MDM, to manage heterogeneous schemas based on a notion of *metamodel*. A metamodel is a set of constructs (the *metaconstructs*) that can be used to define models. The translation of a schema from one model to another is then defined in terms of translations over the metaconstructs, in such a way that the same translation is used for a given metaconstruct in all the models where it appears. In this approach, a translation is performed by eliminating constructs not allowed in the target model, and possibly introducing new constructs. Translations are built from elementary transformations, each of which is essentially an elimination step. Other authors have proposed similar approaches, including Claypool and Rundensteiner et al. [9, 10] Song et al. [17], Bézivin et al [7].

The ideas at the basis of the MDM tool and of similar approaches are interesting but offer only a partial solution to our problem. The main limitation is obviously the fact that they refer only to the schema level, but there is a number of reasons for which a direct extension of them would not work. Indeed, the representation of the models and transformations is hidden within the tool's source code, and so any extension would be very complex, from various points of view: first, only the designers of the tool can extend the models and define the transformations; as a consequence, instance level transformations would have to be recoded in a similar way; also correctness of the rules has to be accepted by users as a dogma, without even any possibility of inspection; finally, any customization would require changes in the tool's source code. Morevoer, MDM does not consider at all the issue of generating mappings between the source and target constructs, an issue that is essentially to support any form of effective use of ModelGen in a practical setting.

Only few attempts exists at the management of heterogeneuos translations at the instance level, including Cluet et al. [11], Milo and Zohar [15], Popa et al. [16], none of which is indeed general enough to tackle the problem we would like to, with the definition of models within a very large family and a strong support to the translation of both schemas and instances.

This paper proposes a framework for the development of an effective implementation of ExtendedModelGen (the first approach in this sense, to the best of the authors' knowledge), based on the following ideas, each of which is novel in model management:

- a *dictionary* is available and visible, and it actually includes three parts (i) the "metamodel" level that contains the description of models, (ii) the "model" level that contains the description of schemas; the structure of the model level can be automatically generated from the metamodel; (iii) the "instance" part, that contains data for the various schemas, with a structure that is a variation of that for schemas at the model level
- the elementary translations are also visible and independent of the engine that executes them; they are implemented by means of rules in a Datalog variant with Skolem functions for the invention of identifiers; this solution allows for the rapid development and effective maintenance of rules for the translation at the schema level and for the (almost automatic) generation of rules that implement translations at the instance level from those at the schema level.

For a brief discussion of a preliminary version of the tool (considering only the schema level), in terms of a demo description, see Atzeni et al. [1].

The paper is organized as follows. Section 2 illustrates the problems we tackle by showing an application scenario. In Section 3 we illustrate the main features of the MDM approach we refer to as a starting point. In Section 4 we highlight the main features and contributions of the paper. In Section 5 we illustrate the structure of the dictionary we use to handle schemas and instances. In Section 6 we describe the process used to translate schema and instances and then in Sections 7 and 8 we illustrate the rules used for

[1] We use here a terminology that differs from that used by Bernstein [5]: we use a more traditional database terminology, where a *schema* is the description of the structure of the database and a *data model* (briefly, a *model*) is a set of constructs you are allowed to use to define your schemas; with this terminology, examples of models are the relational one or (any variation of) the ER one. Bernstein [5] uses the term *model* for what we call schema and *metamodel* for what we call model. As a higher level is also needed in this framework, then we will have a *metamodel* as well, which he would call *metametamodel*.

	Employees		
	EmpNo	Name	Dept
E#1	134	Smith	D#1
E#2	201	Jones	D#2
E#3	255	Black	D#1
E#4	302	Brown	NULL

	Departments	
	Name	Address
D#1	A	5, Pine St
D#2	B	10, Walnut St

Figure 1: A simple object-relational database

the translation of schemas and instances, respectively. In Section 9 we discuss the experiments we have conducted over a prototype implementation. Finally, in Section 10 we draw our conclusions.

2 The problem

As we said in the introduction, many different models exist, and we often need to exchange and share design artifacts (schemas, in database terms, but sometimes also program definitions, UML diagrams, user interface specifications, document schemas or descriptions) and even actual data (database instances as well as programs inputs and outputs or XML documents).

Let us show a simple application scenario for our approach. Real problems are obviously more complex, but we have to consider a very small case, in order to pinpoint the details while limi3ting the space used. Also, the example illustrates two major issues, common to many transformations: the need for replacing system-managed identifiers with values, and the need for generating new values. Consider an object-relational database used to represent information about employees and departments by means of tables with identifiers

Employees(EmpNo,Name,Dept:*Departments)
Departments(Name,Address)

Each of the tables is assumed to have a system managed identifier, and the notation Dept:*Departments specifies that Employees includes in each tuple a reference to a tuple of Departments in terms of such an identifier. A possible instance for the schema is shown in Fig. 1, where we also show for each tuple the (system managed) identifier. We also assume that null references are allowed: the null reference is used as Dept for employee E#4 as he/she does not belong to any department.

Now, if we want to translate this database into the relational model, we can follow the well known intuitive technique that replaces explicit references with actual key values. However, some of the details of this transformation depend upon the specific features we have in the source and target model. Just to give an idea of the problem, let us recall that in the object world, and even in object-relational databases, keys

Employees		
EmpNo	Name	Dept
134	Smith	A
201	Jones	B
255	Black	A
302	Brown	NULL

Departments	
Name	Address
A	5, Pine St
B	10, Walnut St

Figure 2: A translation into the relational model

Employees			
EmpID	EmpNo	Name	Dept
1	134	Smith	1
2	201	Jones	2
3	255	Black	1
4	302	Brown	NULL

Departments		
DeptID	Name	Address
1	A	5, Pine St
2	B	10, Walnut St

Figure 3: A translation with new key attributes

(or visible identifiers) are sometimes ignored; in the example: can we be sure that employee numbers identify employees and names identify departments? It depends on whether the model allows for keys and on whether keys have actually been specified. If both keys have been specified on the object tables, then a reasonable result would be that in Fig.2, with a schema with tables that closely correspond to the object tables in the source database and

- EmpNo is the primary key of Employees
- Name is the primary key of Departments
- there is a referential integrity constraint from the attribute Dept in Employees to (the primary key of) Departments

If instead the keys have not been specified in the object-relational database (and we assume, as usual, that they are required in the relational model), then the most natural way to implement the translation involves the use of an additional attribute for each table (see Fig. 3), to be used as identifier (however, a visible one, as opposed to the system managed ones of the object-relational model). In this case, we would still have the referential integrity constraint from the attribute Dept in Employees to Departments, but the values used would be those of the new attribute used as a unique identifier.

3 Background

As we said in the introduction, while there is no complete general approach available to the solution of the problem we consider, there have been a few partial efforts. In particular, we used as a starting point the

MDM proposal [2], and therefore we briefly recall here its principles. A *metamodel* in such a work is a set of constructs that can be used to define models, which are instances of the metamodel. The approach is based on Hull and King's observation [13] that the constructs used in most known models can be expressed by a limited set of generic (i.e. model-independent) *metaconstructs*: lexical, abstract, aggregation, generalization, function. In MDM, a metamodel is defined by these generic metaconstructs. Each model is defined by its constructs and the metaconstructs they refer to. The models in the examples in Section 2 could be defined as follows:

- the relational model (under a reasonably simplified view) involves (i) aggregations of lexicals (the tables), with some specification for the respective components (the columns), for example to specify whether they are part of the key or whether nulls are allowed; (ii) foreign keys
- a simplified version of the object-relational model has (i) abstracts (tables with system-managed identifiers; both tables in the example); (ii) aggregations of lexicals and references to abstracts (value-based tables, not used in the example for the sake of space); (iii) lexical attributes of abstracts (for example Name and Address, each of which can be specified as part of the key; (iv) reference attributes for abstracts, which are essentially functions from abstracts to abstracts (in the example, the Dept attribute in table EMPLOYEES).

The translation of a schema from one model to another is defined in terms of translations over the metaconstructs. A major concept in the approach is the *supermodel*, a model that has constructs corresponding to all the metaconstructs known to the system. Thus, each model is a specialization of the supermodel, so a schema in any model is also a schema in the supermodel, apart from the specific names used for constructs. Indeed, the supermodel acts as a "pivot" model, so that it is sufficient to have translations from each model to and from the supermodel, rather than translations for any pairs of models (and so there is the need for a linear and not a square number of translations). Moreover, since every schema in any model belongs to the supermodel, the only needed translations are those within the supermodel with the target models in mind; a translation is performed by eliminating constructs not allowed in the target model, and possibly introducing new constructs.

Indeed, each translation in MDM is built from elementary transformations, which are essentially elimination steps. So, a possibility would be to have two elementary transformations (i) one that eliminates references to abstracts by adding aggregations of abstracts (in other terms, replacing functions with relationships), and then (ii) another that replaces abstracts and aggregations of abstracts with aggregations of lexicals and foreign keys (the traditional steps in the translation from the ER model to the relational one). Essentially, MDM handles a library of elementary transformations and uses them to implement complex transformations.

4 Overview of the approach

The main innovative issues of this paper, as anticipated in the introduction, are the structure (and visibility) of the dictionary and the way basic translations are specified in (a variation of) Datalog, so that schema level rules are easily extended to be used at the instance level.

In the next sections we will develop in some detail the above issues, as follows. We will first (Section 5) illustrate the structure of the dictionary, at the schema level and then at the instance level. The two parts have a close correspondence, which turns out to be very effective in the generation of rules. Also, the dictionary has a meta-level part, which contains a description of the metaconstructs of interest and of the models and can be used as a "core," to generate the structure of the rest of the dictionary. This organization of the dictionary also allows for the automatic generation of some of the rules, as we comment shortly.

Then, in the subsequent sections we describe the translation process (Section 6) and the actual rules. In Section 7 we will illustrate the rules for the translation of schemas, where a number of original contributions arise. First, since rules are written in Datalog, they are visible and independent of the main engine that interprets them. In fact, they can be added or modified dynamically, whenever new metaconstructs are defined or when new translation techniques are needed or discovered. Rules can also be easily customized: for example, one can add "selection conditions" that specify the elements of a schema to which a transformation has to be applied; this is particularly useful if a construct can be handled in several ways. Third, the system itself can verify basic properties of sets of transformations (e.g., some form of correctness) by reasoning on the basis of the set of constructs appearing in the bodies and the heads of Datalog rules. Finally, Skolem functions, used to generate new identifiers when needed and to correlate them in various rules, can be materialized and stored in the dictionary; so they would represent the mappings between source and target schemas in each translation, a feature that is considered very important for model management operators (see Bernstein [5]).

Then, in Section 8 we illustrate the most novel contribution of the paper, built upon the results of the previous sections: we show how rules for the translation of instances can be obtained from the rules for the translation of schemas. These rules also make use of Skolem functions, both at the schema level (that is,

those used for schema level rules), to relate instance elements to the constructs they refer to, and at the instance level, to establish correspondences (mappings) between elements of source and target instances.

We have tested the ideas discussed in this paper, over a prototype in which we implemented the dictionary and the rules, both at the schema and at the instance level. In Section 9 we will briefly discuss the architecture of the prototype and the experiments.

5 The dictionary

Let us illustrate the major aspects of the dictionary. It is implemented as a relational database, as in this way Datalog rules can be easily formulated.

5.1 Description of schemas

Schemas are described in the dictionary by means of a table for each metaconstruct, with columns that describe the various properties of a construct and the other constructs it refers to. Going back to our example, we would have a number of tables to handle schemas in the object-relational model, partly shown in Fig.4. Let us comment on them:

- SM_Abstract describes the object tables (those with system-managed identifiers); for each of them it keeps the name and two identifiers that are used in all tables: OID, which identifies the construct, and sOID (abbreviation for SchemaOID), to indicate the schema the construct belong to; in this way (as indicated by the ellipsis for additional rows), a single table can store descriptions of constructs for all the schemas of interest;
- SM_AttributeOfAbstract has information about the attributes of the two object tables: for each attribute we keep a reference to the abstract it belongs to (the AbsOID attribute) and two booleans that specify whether it belongs to the key and whether it allows null values. In some cases these last two attributes could have special values: if keys were not specified (or not specifiable), then IsKey would be identically *false*; similarly for null values.
- SM_RefAttributeOfAbstract stores information on the reference attributes, in a similar way, with the difference that there is no IsKey (as we assume that reference attributes do not participate in keys), and that the abstract referred to is mentioned (AbsToOID, which in the row of interest has a value 102, that is, object table Departments).
- the dictionary also contains other tables for the object relational model, to store value-based relations, but they are not shown here as they do not contain any tuple for the schema under consideration (as both tables have identifiers and are therefore handled by SM_Abstracts

The above tables describe schemas in terms of supermodel constructs, because translations, in order to be reusable, as we will see, refer to the supermodel. Indeed, the dictionary also keeps track of each model described by means of its own terminology, by means of tables that are essentially isomorphic to the one above; for example, there would be a table OR_ObjectTable(OID, sOID, Name) with tuples that contain the same data as the tuples in SM_Abstracts(OID, sOID, Name) corresponding to the object-relational model. Clearly SM_Abstract would also contain tuples referring to other models, for example the ER model, where entities also correspond to abstracts. The use of the supermodel allows us to write transformation rules that refer to SM_Abstracts rather than to OR_ObjectTable or ER_Entity (and so need not be duplicated). The transfer of data between the model specific representation to the supermodel one can be done by means of "copy" transformations that, as we will discuss later, can be generated automatically.

5.2 Description of instances

Instances are described in a portion of the dictionary that has a structure similar to that for schemas, with some differences. The basic idea is that instances of constructs are represented by means of internal identifiers and that lexical constructs (that is, those that have a visible value) also have the associated value. A portion of the representation of the instance in Fig.1 is shown in Fig.5 (with some abbreviations in the table names) Let us comment the main points:

- each table has a i-sOID (for instance of SchemaOID) attribute, instead of the sOID attribute we had at the schema level; indeed, our dictionary can handle various schemas for a model and various instances for each schema;
- each instance has a reference to the construct it instantiates; for example, the first table in the Fig.5 has an AbsOID column, whose values are identifiers for the abstracts in the schema; in fact, the first four rows have a value 101 for it, and 101 is, in Fig.4, the identifier for object-table Employees, whereas the other two have a value 102, the identifier for Departments;
- "properties" of schema elements (such as IsKey and IsNullable) do not have a counterpart at the instance level;
- all identifiers (both the OID of the construct and the references to other constructs) that appear at the schema level are transformed into identifiers for instances; e.g., table SM_RefAttributeOfAbstract in Fig.4 has (among others) columns (i) OID (the identifier for

SM_Abstracts

OID	sOID	Name
101	1	Employees
102	1	Departments
...	...	...

SM_AttributeOfAbstract

OID	sOID	Name	IsKey	IsNullable	AbsOID	Type
201	1	EmpNo	T	F	101	Integer
202	1	Name	F	F	101	String
203	1	Name	T	F	102	String
204	1	Address	F	F	102	String
...	...	...	...	...	...	...

SM_RefAttributeOfAbstract

OID	sOID	Name	IsNullable	AbsOID	AbsToOID
301	1	Dept	T	101	102
...	...	...	...	...	...

Figure 4: An object-relational schema represented in the dictionary

SM_InstOfAbstract

OID	i-sOID	AbsOID
1001	1	101
1002	1	101
1003	1	101
1004	1	101
1005	1	102
1006	1	102
...	...	...

SM_InstOfAttributeOfAbstract

OID	i-sOID	AttOID	i-AbsOID	Value
2001	1	201	1001	134
2002	1	202	1001	Smith
2003	1	201	1002	201
2004	1	202	1002	Jones
...	...	...	...	...
2011	1	203	1005	A
2012	1	204	1005	5, Pine St
2013	1	203	1006	B
...	...	...	...	...

SM_InstOfRefAttributeOfAbstract

OID	i-sOID	RefAttOID	i-AbsOID	i-AbsToOID
3001	1	301	1001	1005
3002	1	301	1002	1006
...	...	...	...	...

Figure 5: Representation of an object relational instance

the row), (ii) AbsOID (the identifier of the abstract to which the attributes belongs), and (iii) AbsToOID (the identifier of the abstract to which the attributes "points"); in Fig.5 each of them is taken one level down: (i) each row is still identified by an OID column, but this is the identifier for the instance; (ii) each value of i-AbsOID indicates the instance of abstract the attribute is associated with (for example, in the first tuple in the figure, 1001 is the identifier for employee Smith); (iii) i-AbsToOID indicates the instance of abstract the attribute refers to (in the first tuple, 1005 is the identifier of department A);

- if the construct is lexical (that is, has an associated value [13]), then the table has a Value column; in Fig.5 this is the case for table SM_InstanceOfAttributeOfAbstract, the only lexical construct; it should be noted that we use only one Value column for all the attributes, regardless of their type; this is indeed reasonable if we assume we have the availability of serializabilty functions that transform values of any type into values of a common one (for example strings).

The representation for instances we have just seen is clearly an "internal" one, into which or from which actual database instances or documents have to be transformed. The idea we are developing is that of import/export features that allow for the upload/download of instances and schemas for a given model.

6 The translation process

The translation process from a source schema and instance (in a source model) to a target schema and instance (in a target model) is based on the supermodel, so extending the MDM approach (which would do this but only at the schema level):

1. the source schema and instance are translated (essentially, copied) into the supermodel; this is straightforward, as each model is subsumed by the supermodel
2. the actual translation to the target schema and instance is performed within the supermodel
3. the target schema and instance are copied into the target model; this is also a reasonably simple phase, for the same reasons as for 1 above.

Given the complete description of models in the dictionary and their correspondence to the supermodel, "copy transformations" that perform the "copy" tasks (phases 1 and 3) can be automatically generated. We will show an example in the next section.

Therefore, the only transformations that have to be explicitly specified in our approach are those within the supermodel. We will discuss basic transformations in detail in the next section; we now comment

briefly on the composition of elementary translations. The idea is that we have basic steps that, being very focused, can be easily reused. For example, the translation for the first scenario we illustrated in Section 2, requiring a translation from an object-relational model (with keys) to a relational one (with nulls values), could be implemented by means of two steps:[2]

B_1 from the object-relational (OR) model to the binary entity-relationship (ER) model (transforming reference attributes into relationships and copying abstracts and lexical attributes)

B_2 from the binary ER model to the relational one

If instead we wanted to use variations of these models, as we said in Section 2, for example an OR model without keys, we could proceed with three steps, the first and third being B_1 (written to work properly also if there are no keys in the OR model, generating a schema in a variation of the ER model with no keys) and B_2 above, and the second the following:

B_3 from the binary ER model without keys to the binary ER model with keys (adding "new" key attributes to entities and generating new values for them).

It is worth noting that the reuse of basic transformations appears both when referring to variations of models, as in the case above, and for translations that are apparently unrelated. For example, if we had an n-ary ER model (that is, one that allows n-ary relationships, not just binary ones), and had again the relational model as a target, then we could have two steps, the second one being implemented by B_2 and the first one by

B_4 from the n-ary ER model to the binary one, replacing n-ary relationships with entities

Basic translations can be described in terms of their "signatures" (essentially, which constructs they refer to and which they introduce), in order to reason on properties of basic transformations and of complex translations; the declarative specification of rules in Datalog allows for an automatic generation of signatures for basic transformations.

7 Translation rules for schemas

The basic transformations we discussed in the previous section are each specified by a set of Datalog rules, which consider the various constructs at hand. Rules are used both at the schema and at the instance level; let us concentrate first on the schema level and then on the instance level. Each rule generates elements of a schema for a given metaconstruct (the one that appears in the head of the rule), and we might have more rules for the same metaconstruct.

7.1 Rules for copying constructs

Both the copy phases mentioned in the previous section and the actual translation in the supermodel are specified by means of sets of rules, with one major difference: as we said, copy rules depend only on the structures of the constructs and can therefore be automatically generated. Let us briefly comment on this issue, introducing at the same time the syntax of our rules and some important features.

The portion of the object-relational model whose schemas can be described by means of the tables in Fig.4 could be copied (at the schema level, as we are concentrating on it for the time being) to and from the supermodel by means of three rules, one for each of the tables. The following is the rule for copying the first table from the specific model to the supermodel:

SM_Abstracts (OID:#abstract_0(*oid*),
  sOID:*target*, Name:*name*) ⟵
 OR_ObjectTable(OID:*oid*,
  sOID:*source*, Name:*name*)

For the sake of readability, we use a non-positional notation for Datalog, with attribute names explicitly mentioned. An important observation is that the OID for SM_Abstracts is generated by means of a Skolem function, which produces "new" values whenever applied to "new" arguments (so it is injective), and produce the same value if applied twice to the same arguments. As we will see, there are usually several different Skolem functions for a construct, and this justifies the suffix ('0' in this case) for their name. Another useful comment on the rules is the following: a rule transforms constructs from a source schema to a target one; the involved schemas appear in the tables and are specified as actual parameters when a rule is invoked and end up being instantiated for the variables *source* and *target*, respectively.

Another rule, for the copy of lexical attributes, is the following:

SM_AttributeOfAbstracts(
  OID:#attribute_0(*oid*), sOID:*target*,
  Name:*name*, IsKey:*isK*, IsNullable:*isN*,
  AbsOID:#abstract_0(*otOID*)) ⟵
 OR_LexicalAttribute (OID:*oid*,
  sOID:*source*, Name:*name*, IsKey:*isK*,
  IsNullable:*isN*, OTableOID:*otOID*)

Let us observe that we have two Skolem functors here, the first one with a similar goal as the one we saw above (generating the new oid for the copied construct), whereas the second one is needed to maintain coherence of the references: if the attribute in the OR scheme refers to object-table *otOid*, then the copied attribute in the supermodel schema has to refer to the abstract copied from *otOid*, which is generated by the

[2]For the sake of readability, we refer to model-specific constructs (e.g., entities and relationships), whereas the process to be effective and allow for wider reuse, refers indeed to the supermodel constructs (e.g., abstracts and aggregations of abstracts, respectively).

Skolem function applied to it: #abstract_0($otOid$).

Summarizing, let us say that a copy phase is handled by means of a set of copy rules, which can be automatically generated on the basis of the information on models and constructs handled by the dictionary

7.2 Rules for actual translations

The translation phases are also defined in terms of Datalog rules, with the main difference that they have to be explicitly specified, and not just generated, as for the copy phases. A basic transformation is composed of a set of Datalog rules, at least one for each of the constructs in the target model. Let us consider the basic step for translating from the binary ER model to the relational one. With the versions of the models we currently handle, it requires Datalog rules for the following tasks (which correspond to the known tasks in the logical design of relational databases from ER schemas [4]):

1. generate tables for entities (we should say, in supermodel terms, "generate aggregations of lexicals for abstracts", but we feel that the model specific terminology is more understandable)
2. generate columns of tables for attributes of entities; each column would belong to the table generated, by the previous rule, for the entity each attribute belongs to;
3. generate tables for many-to-many relationships;
4. generate columns of the tables that represent many-to-many relationships for the attributes that form the key of the involved entities;
5. generate foreign key constraints associated with the previous rule;
6. generate columns for each one-to-many (and one-to-one) relationship, in the table corresponding to the entity that participate with a "one" cardinality, for the key attributes of the other entity;
7. generate foreign key constraints associated with the previous rule;
8. generate columns for the attributes of the relationships. (two rules, indeed, one for many-to-many relationships and one for the others)

Let us show one of the rules, the last one, for attributes of many-to-many relationships:

SM_AttributeOfAggregationOfLexicals(
OID:#attribute_4($attOid$), sOID:$target$,
Name:$name$, IsKey:FALSE, IsNullable:isN,
AggOID:#aggregation_2($aggOid$)) ⟵
SM_AttributeOfBinaryAggr...OfAbs...(
OID:$attOid$, sOID:$source$, Name:$name$,
IsNullable:isN, AggOID:$aggOid$),
SM_BinaryAggregationOfAbstracts(
OID:$aggOid$, sOID:$source$,
isFunct1:FALSE, isFunct2:FALSE)

The rule has two literals in the body, the first with the details for the attribute and the second for specifying that the rule is applied only to many-to-many relationships (this is implemented by referring to the two boolean properties isFunct1 and isFunct2, which specify the cardinality of the relationship). With respect to the head, there are two points to note. First, there is a constant value, FALSE, for IsKey: this is because the attribute generated here never belongs to a key, as it originates from an attribute of a relationship. The second point is that the Skolem functors are not those for copies we saw in the previous section (which have conventionally '0' as the suffix), but more specific; in general, there are various functors associated with a given construct, as it may be generated from constructs of various types; for example, attributes of lexicals can be generated for attributes of abstract in the basic case ("attributes of entities become columns of tables," and this is functor '2', as '1' is used for functors that copy within the supermodel, an issue we discuss below), for implementing ER-relationships in the relational model (functor '3'), and for attributes of relationships (this case, functor '4'). the various functors for a given costructs, with the same name and different suffix, may have different sets of arguments and, more important, have pairwise disjoint ranges, that is generate different values.

It is important to mention that Skolem functions are materialized in the dictionary: there is one table for all the Skolem functions associated with each construct, so that it is easy to guarantee the disjointnes of the functions. The tables are used with two goals. First, they are used to guarantee that Skolem functions are indeed functions: assume a functor $F(a)$ appears in a rule; if a value for F on a already exists, then it would be already in the table and then used; otherwise, as the table would store all the identifiers for the object, the value could be obtained by generating a new key value (for example by means of the counter feature offered by various systems). Second, and more important, these tables would essentially represent the mapping between constructs in the source schemas and those in the target schema, an issue that is considered very important in model management [5].

In some cases basic translations are actually interested in only a very specific aspect, and therefore they leave most of the involved schema unchanged. In this case, the transformation includes various Datalog rules that copy constructs, in a way that is even simpler than the one we saw in Section 7.1, as the constructs would all belong to the supermodel, and could be automatically generated as well. For example, the basic transformation that introduces keys in ER schemas (that is, the one used to go from an ER model without keys to one with keys), would include rules for copying all constructs and just a specific rule for adding an attribute to each entity:

SM_AttributeOfAbstract(
OID:#attribute_5($absOid$), sOID:$target$,

Name:*name*+'ID', IsNullable:FALSE,
IsKey:TRUE, AbsOID:#abstract_1(*absOid*)) ⟵
SM_ABSTRACT (OID:*absOid*,
sOID:*source*, Name:*name*)

The new attribute has a name that is obtained by concatenating the name of the entity with the suffix 'ID'; this is essentially what we have done for introducing the attributes EmpID and DeptID in Fig.3.

To conclude the section, let us note that if we apply the all the rules that form the basic transformation steps B_1 and B_2 to the schema we saw in Fig.4 then we obtain the schema in Fig.6, which properly corresponds to the schema of the database we saw in Fig.2.

8 Translation rules for instances

A major feature our proposal includes is the support to the development of rules for translating instances, which leverages upon the close correspondence between the two levels in the dictionary for the representation of schemas and of instances. Indeed, once rules at the schema level are available, then the corresponding ones at the instance level can be generated almost completely, with only the need for some local refinement for specific issues.

Specifically, we have a method that, given a Datalog rule at the schema level, generates a rule to be used for translating instances. Before presenting the algorithm in a rather general way, we illustrate it by means of an example. Let us consider the second rule for the translation from the binary ER model to the relational one, which generates columns of tables for attributes entities. At the schema level, is as follows:

SM_ATTRIBUTEOFAGGREGATIONOFLEXICALS
(OID:#attribute_2(*attOid*), sOID:*target*,
Name:*name*, isKey:*isK*, isNullable:*isN*,
AggOID:#*aggregation_2*(*absOid*)) ⟵
SM_ATTRIBUTEOFABSTRACT (OID:*attOid*,
sOID:*source*, Name:*name*, isKey:*isK*,
isNullable:*isN*, AbsOID:*absOid*)

The corresponding rule at the instance level is the following:

SM_INSTANCEOFATTRIBUTEOFAGGR...OFLEX...
(OID:#i-Attribute_2(*i-AttOid*), i-sOID:*target*,
i-AggOID:#i-Aggregation_2(*i-AbsOid*),
AttOID:#attribute_2(*attOid*),
Value:*value*)) ⟵
SM_ATTRIBUTEOFABSTRACT (OID:*attOid*,
sOID:*sourceSchema*, Name:*name*, isKey:*isK*,
isNullable:*isN*, AbstractOID:*absOid*),
SM_INSTANCEOFATTRIBUTEOFABSTRACT
(OID:*i-AttOid*, i-sOID:*source*,
i-AbsOID:*i-AbsOid*, AttOID:*attOid*, Value:*value*)

Let us note that

- the head is obtained from the head of the schema level rule by applying transformations that take schema literals "down to instances" (see below);
- the body is composed of two parts: a copy of the body at the schema level and its transformation down to instances.

The transformations of schemas "down to instances" derive from the correspondences in the dictionary (as discussed in Section 5.2), as follows

1. references to schemas become references to instances: indeed, both in the head and in the second literal in the body, we have a i-sOID column instead of the sOID;
2. Skolem functors are replaced by "homologous" functors at the instance level, by transforming both the function name and the arguments;
3. instances refer to the constructs they instantiate: both SM_INSTANCEOFATTRIBUTEOFAGGR...OFLEX... and SM_INSTANCEOFATTRIBUTEOFABSTRACT have an AttOID field;
4. "properties" of schema elements do not have a counterpart at the instance level: the rule at the instance level does not refer to the Name, nor to isKey, nor to isNullable;
5. identifiers at the schema level become identifiers at the instance level: OID in the head of the schema level rule refers to a construct (for example, entity EMPLOYEE) whereas in the instance level rule it refers to its instances (individual employees);
6. lexical constructs have a Value attribute.

Let us now illustrate in a general, though informal way, the algorithm. For a schema level rule r with k literals in the body, DOWN generates a rule r', where:

- the head of r' is obtained from the the head of r by applying the DOWN function (see below)
- the body has two parts, each with k literals:
  - the first part contains literals each obtained by applying the DOWN function to a literal in the body of rule r;
  - the second part is a copy of the body of r.

The DOWN function produces an instance level literal from a schema level one. Let us introduce a bit of notation. A literal has the form $P(n_1 : a_1, \ldots, n_h : a_h)$, where P, the *predicate*, is the name of table for a supermodel construct (therefore beginning with the prefix SM_), each n_i (a *name*) is a column of P and each a_i is an *argument*, which can be a constant, a variable,[3] or a Skolem functor. In turn, a Skolem functor has the form $F(p_1, \ldots, p_m)$, where F is a the name of a Skolem function and the each p_j is a constant or a variable.

Given a schema level literal $l_S = P(n_1 : a_1, \ldots, n_h : a_h)$, DOWN produces an instance level literal with:

[3] In general, an argument could also be an expression (for example a string concatenation over constants and variables), but this is not relevant here.

SM_AggregationOfLexicals		
OID	sOID	Name
401	2	Employees
402	2	Departments
...	...	...

SM_AttributeAggregationOfLexicals					
OID	sOID	Name	IsKey	IsNullable	AggOID
501	2	EmpNo	T	F	401
502	2	Name	F	F	401
503	2	DeptName	F	T	401
504	2	Name	T	F	402
505	2	Address	F	F	402
...	...	...	...	...	...

Figure 6: Representation of a relational schema in the dictionary

- a predicate name obtained from P by replacing the SM_ prefix with SM_InstanceOf;[4]
- a set of pairs (name,argument) obtained from the pairs in the input literal l_S where the name is a reference[5] column in table P (that is, the property columns do not contribute); for such a pair $n : a$ in l_S it is the case that a is either a variable or a Skolem functor and Down produces a pair in its output literal of the form $n' : a'$, where n' is obtained from n by adding a "i-" prefix[6] and a' is obtained from a as follows: if a is a variable, then a' is obtained by prefixing i- to its name; if instead it is a Skolem functor, both the function name and its variable parameters are prefixed with i- whereas the constants are left unchanged.
- an additional pair is added if the construct associated with P is lexical (that is, its occurrences at the instance level have values); for the sake of simplicity, let us say here that the pair has just the form Value:v, which is indeed the most common case, where values have to be copied from a source construct to target one; we will comment on more general cases below.

Let us note that the body of the schema level rule may involve selection conditions (referring to variables in literlas) and they are not transformed but just included in the result rule as it includes all the body of the input rule.

As we said above, the plain copy of values from the source to the target instance is not always possible, and in some cases it would not be correct: for example, if we also consider types of lexical attributes (an aspect we have omitted here for the sake of space), then the allowed types in the source and target model need not coincide, and the copy could cause problems. In general, the values for lexical constructs are obtained by means of functions that have to be specified by the "model engineer" that uses this methodology and the associated tool. In most cases, if there is only one lexical attribute in the body (and this is indeed the case), the function is just the identity over the only lexical attribute in the body of the rule, as we have assumed

[4]In figures and examples we abbreviate names when needed.

[5]Let us recall that the columns in the supermodel table are classified as reference or property—see Section 5

[6]The prefix is not added to the OID, as all objects in our dictionary have an identifying column named OID.

SM_I...OfAggr...OfLex...		
OID	i-sOID	AggOID
4001	2	401
4002	2	401
4003	2	401
4004	2	401
4005	2	402
4006	2	402
...	...	...

SM_I...OfAttr...OfAggr...OfLex...				
OID	i-sOID	AttOID	i-AggOID	Value
5001	2	501	4001	134
5002	2	502	4001	Smith
5003	2	503	4001	A
5004	2	501	4002	201
5005	2	502	4002	Jones
5006	2	503	4002	B
...	...	...	...	...
5021	2	504	4005	A
5022	2	505	4005	5, Pine St
5023	2	504	4006	B
...	...	...	...	...

Figure 7: A relational instance obtained from an object relational one

"as a default" in the case above, but this would not be the only possibility. There are indeed cases where the need for a function is absolute, as there is no default origin for the target values. The example we showed as a special cases in Section 2 requires a specific function, as there is no value to be copied: we introduce a new attribute (the identifier) and we have no counterpart in the body. This is apparent in the schema level rule for the introduction of the identifier, which has a body that involves only an abstract, that is, a non-lexical construct.

In Fig.7 we show the result of the translation of the database instance in Fig.5 into the relational model, that is, the result of the application to it of the instance level rules corresponding to the basic transformations B_1 and B_2.

A main point to discuss here is the correctness of our method. Given the generality of the approach, it would be difficult to claim a general correctness; indeed, let us recall that there have been dozens of paper discussing the correctness of translations from one specific model to another specific one, and even the notions of equivalence and relative information ca-

pacity of schemas have been deeply debated and have reached only partial, ad-hoc solutions [12].

Let us mention that the initial approach we refer to, proposed by Atzeni and Torlone [2] assumed an "axiomatic" attitude: assuming the basic translations to be correct (and this would be reasonable, as they refer to known elementary steps developed over the years), and given a suitable description of models and rules in terms the involved constructs, then complex translations can be proven correct by induction. In this work we essentially follow the same idea, by extending it further down to the instance level. Indeed, with respect to the correctness of transformations at the schema level, we have the additional benefit of having them expressed at a high-level, as Datalog rules; in fact, it is possible to automatically detect what constructs are used in the body and what are generated in the head of a Datalog rule and then derive a signature of each basic transformation. Similarly, models are described in terms of constructs. Then, it is possible to describe the model obtained by applying a basic transformation to a given model and then by induction, the model obtained by applying a complex one.

For the correctness at the instance level, we can reason in a similar way. Therefore, the main issue would be the correctness of the basic transformations, as that of complex ones would follow by induction. As the Datalog rules at this level are generated automatically from those at the schema level, a few comments are useful: the algorithm generates rules that consider (and use as a source for generating target instances) all the instances of the elements that are used as sources for target elements; the rules also generate syntactically correct instances of the target elements. The actual correctness of the transformations depends indeed on the specific rules and also on the features of the constructs, and cannot be proven in general. Indeed, the issue here is that the correctness of the basic translations is confirmed by their author (the "model engineer", see next section), who writes them at the schema level and then, after translating them to the instance level (possibly with some refinement) verifies them both at the schema and at the instance level.

As we will briefly discuss in the next section, we have successfully experimented our approach over a significant set of models and translations, and this can be considered an argument to support the effectiveness of the approach. Specifically, we have defined a variety of constructs, which covers the main models and have specified translations for each of them, obtaining the results usually considered as the correct ones.

9 Experimentation

In order to verify the ideas illustrated in this paper, we have implemented a prototype tool, composed of three main parts, the dictionary, the transformation repository and the ExtendedModelGen operator.

The basic management of the dictionary is performed by three generic (i.e., model-independent) modules: DefineModel (used to define new models based on the available constructs), DefineSchema (used to define a schema for a chosen model) and LoadInstance. After a model is defined, by using the description of its constructs and correspondence with meta-constructs, the tool automatically creates the structures needed to handle its schemas and instances; it also automatically generates the "copy rules" discussed in Section 7.

The transformation repository contains artifacts of two kinds (as we saw in Section 7): basic transformations and Datalog rules. Actual translations (i.e., applications of ExtendedModelGen) are then specified by composing basic transformations: as we discussed in Section 6, a translation from the n-ary ER model to the relational one would use two basic transformations, one that replaces n-ary relationships with binary ones and a second from a binary ER model into the relational one. Composition is supported by the tool, which can verify whether the process actually generates schemas in the desired target model and detect redundancies in a sequence of basic transformations.

We have experimented our prototype extensively, and indeed all the examples in this paper have been implemented on it (even if in some cases we presented in the paper a simplified version for the sake of readability).

We have defined a metamodel with a dozen of different metaconstructs, each with a number of properties (for example, attributes with nulls and without nulls are just variants of the same construct). Specifically, we have metaconstructs for aggregations of lexicals and foreign keys over them, abstracts and aggregations of them, attributes for both abstracts and aggregations, aggregations of lexicals and abstracts, generalization over abstracts; components of aggregations and attributes can be nested, structured, or simple, and the simple ones can be lexical or reference (to abstracts). These metaconstructs, with their variants, allows for the definition of a huge number of different models (all the major models and many variations of each). For our experiments, we have defined a set of significant models, each various versions (including the possibility of having or not having nested attributes and generalization hierarchies): extended-ER, UML class diagrams, object-relational, object-oriented and relational.

We have defined the basic translations needed to handle the set of test models. There are more than twenty of them, the most significant being those for eliminating n-ary aggregations of abstracts, eliminating many-to-many aggregations, eliminating attributes from aggregations of abstracts, introducing an attribute (for example in order to have a key for an Abstract or AggregationOfLexicals), replacing aggrega-

tions of lexicals with abstracts and vice versa (and introducing or removing foreign keys as needed), unnesting attributes, eliminating generalizations. Each basic transformation required from five to ten Datalog rules.

These basic transformations allow for the definition of translations between each pair of the test models, with a complete success: each of them produces the expected target schemas, according to the known standard translation used in database literature and practice.

At the instance level, we experimented with the models that handle data, hence object-oriented, object-relational and relational (in the nested and flat version). Here we could verify the correctness of the DOWN function, the simplicity of the management of the rules at the instance level and, in the end, the correctness of data translation, which produced, as for the schema level, the expected results.

On the basis of the experience with nested models, we are currently working on XML descriptions (in terms of XSD and simplifications of it), and we are confident that, apart from the need for specifying many details, translations can be successfully implemented as well.

10 Conclusions

In this paper we have shown how the ExtendedModelGen operator can be implemented, in order to support model-generic translations within a large family of models. The experiments we conducted have confirmed that translations can be effectively performed with our approach. We are currently pursuing a number of directions to extend the work done so far, in various directions, as follows.

Even if we have already considered a significant number of constructs, which include all those mentioned in the paper plus a few more, there are still important constructs we have not considered yet, for example those needed for the fine details of XSD.

A second direction is the formalization of techniques for reasoning on the correctness of complex transformations given properties of the basic ones.

A third direction is the actual development, we have only briefly sketched sofar, of import/export features that allow for the seamless interchange of schemas and instances with other tools.

References

[1] P. Atzeni, P. Cappellari, and P. A. Bernstein. Modelgen: Model independent schema translation. In *ICDE, Tokio*, 2005. To appear.

[2] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. *EDBT*, 79–95, 1996.

[3] P. Atzeni and R. Torlone. Mdm: a multiple-data-model tool for the management of heterogeneous database schemes. *SIGMOD*, pp. 528–531, 1997.

[4] C. Batini, S. Ceri, and S. Navathe. *Database Design with the Entity-Relationship Model*. Benjamin and Cummings Publ. Co., Menlo Park, California, 1992.

[5] P. A. Bernstein. Applying model management to classical meta data problems. *CIDR*, pages 209–220, 2003.

[6] P. A. Bernstein, A. Y. Halevy, and R. Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.

[7] J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The atl transformation-based model management framework. Res. Rep. 03.08, Univ. Nantes, 2003.

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

[9] K. T. Claypool and E. A. Rundensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *ICEIS (1)*, pages 219–224, 2003.

[10] K. T. Claypool, E. A. Rundensteiner, X. Zhang, H. Su, H. A. Kuno, W.-C. Lee, and G. Mitchell. Sangam - a solution to support multiple data models, their mappings and maintenance. In *SIGMOD Conference*, 2001.

[11] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion! In *SIGMOD Conference*, pages 177–188, 1998.

[12] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *ACM-PODS*, pp. 51–61, 1997.

[13] R. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Comp. Surv.*, 19(3):201–260, 1987.

[14] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB'90*, pages 455–468, 1990.

[15] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, pp.122–133, 1998.

[16] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, pages 598–609, 2002.

[17] G. Song, K. Zhang, and R. Wong. Model management though graph transformations. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 75–82, 2004.