The VLDB Journal
DOI 10.1007/s00778-008-0105-2

SPECIAL ISSUE PAPER

Model-independent schema translation

Paolo Atzeni · Paolo Cappellari · Riccardo Torlone · Philip A. Bernstein · Giorgio Gianforme

Received: 23 September 2007 / Revised: 21 February 2008 / Accepted: 9 May 2008

Abstract We discuss a proposal for the implementation of the model management operator ModelGen, which translates schemas from one model to another, for example from object-oriented to SQL or from SQL to XML schema descriptions. The operator can be used to generate database wrappers (e.g., object-oriented or XML to relational), default user interfaces (e.g., relational to forms), or default database schemas from other representations. The approach translates schemas from a model to another, within a predefined, but large and extensible, set of models: given a source 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. A wide family of models is handled by using a metamodel in which models can be succinctly and precisely described. The approach expresses the translation as Datalog rules and exposes the source and target of the translation in a generic relational dictionary. This makes the translation transparent, easy to customize and model-independent. The proposal includes automatic generation of translations as composition of basic steps.

Keywords Schema translation · Data models · Model management

P. Atzeni and R. Torlone partially supported by MIUR, Università Roma Tre and an IBM Faculty Award, and G. Gianforme by a Microsoft Research Fellowship.

P. Atzeni (✉) · P. Cappellari · R. Torlone · G. Gianforme
Università Roma Tre, Rome, Italy
e-mail: atzeni@dia.uniroma3.it

P. A. Bernstein
Microsoft Research, Redmond, WA, USA

1 Introduction

To manage heterogeneous data, many applications need to translate data and their descriptions from one model (i.e., data model) to another. Even small variations of models are often enough to create difficulties. For example, while most database systems are now object-relational, the actual features offered by different systems rarely coincide, so data migration requires a conversion. Every new database technology introduces more heterogeneity and thus more need for translations. For example, the growth of XML has led to such issues, including the need to have object-oriented wrappers for XML data, the translation from nested XML documents into flat relational databases and vice versa, and the conversion from one company standard to another, such as using attributes for simple values and sub-elements for nesting versus representing all data in sub-elements. Other popular models lead to similar issues, such as Web site descriptions, data warehouses, and forms. In all these settings, there is the need for translations from one model to another. This problems belongs to the larger context Bernstein [11] termed *model management*, an approach to meta data management that considers schemas as the primary objects of interest and proposes a set of operators to manipulate them.

According to the model management terminology, this paper considers the *ModelGen* operator [11], defined as follows: given two models M_1 and M_2 and a schema S_1 of M_1, *ModelGen* translates S_1 into a schema S_2 of M_2 that properly represents S_1. A possible extension of the problem, considers also the data level: given a database instance I_1 of S_1, the extended operator produces an instance I_2 of S_2 that has the same information content as I_1.

As there are many different models, what we need is an approach that is generic across models and can handle the idiosyncrasies of each model. Ideally, one implementation

Fig. 1 A simple ER schema

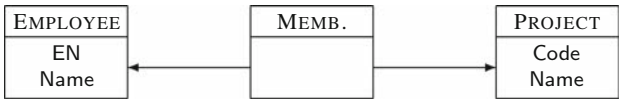


Fig. 2 The translation of the schema in Fig. 1 into a simple object-oriented model

should work for a wide range of models, rather than implementing a custom solution for each pair of models.

We illustrate the problem with some of its major features by means of a couple of short examples. Let us consider two popular data models, Entity-relationship (ER) and object-oriented (OO). Indeed, each of them is not "a model," but "a family of models," as there are many different proposals for each of them: OO with or without keys, binary and n-ary ER models, OO and ER with or without inheritance, and so on. Let us assume that our OO model has classes with simple fields and uses (directed) references as the means of relating classes to one another. In general, if we have an ER schema and we want to translate it into an OO schema, we essentially map entities to classes and replace relationships with references. Our approach is based on the extension and generalization of these simple ideas, as follows:

- Entities are mapped to classes (and vice versa in the reverse translation) because the two types of constructs play the same role (or, in other terms, "they have the same meaning"). Indeed, this situation is common: most known models have constructs that can be classified according to a rather small set of *metaconstructs* [5]. In fact, the same happens for attributes of entities and fields of classes. In both cases, this part of the translation is trivial (in the strict sense: there is no translation, just renaming of the type of construct, from "entity" to "class" and from "attribute" to "field").
- Relationships are replaced by references, and here things are more complex. As usually relationships are more sophisticated (they can be n-ary, many-to-many, or have attributes), the introduction of new classes (beside those corresponding to entities) may be needed. So, if for example we want to translate the ER schema in Fig. 1, which involves a many-to-many relationship, then we will have to introduce a new class for such a relationship (see Fig. 2).

An interesting issue is that the actual translation can depend on the details of the source and target model. For example, if we go from an object model to the relational one,

Fig. 3 A simple OO schema

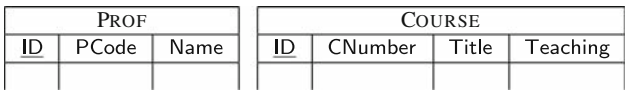


Fig. 4 The relational translation of the OO schema in Fig. 3 with new key attributes

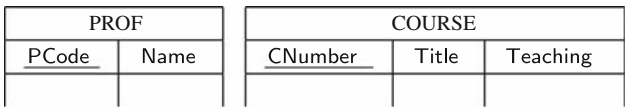


Fig. 5 The relational translation of the OO schema in Fig. 3 without new key attributes

then we have to implement different translations depending on whether the source model requires the specification of identifiers or not. Let us refer to the schema in Fig. 3. If the object model does not allow (or it allows but does not require) the specification of identifiers, then, in the translation to the relational model, we have to add new key attributes in the tables generated for each class. If instead identifiers are required (and, in the example, PCode and CNumber are specified as identifiers), no new attributes are needed. Figures 4 and 5 show the two possibilities in the translated schema. In both cases, referential integrity is needed, in the former case involving the new attributes and in the latter over the existing identifiers.

The example shows that we need to be able to deal with the specific aspects of models, and that translations need to take them into account. We have shown two versions of the object model, one that has visible keys (besides the system-managed identifiers) and one that does not. The translation has to be different as well.

More generally, we are interested in developing a platform that allows the specification of the source and target models of interest (including OO, OR, ER, UML, XSD, and so on), with all relevant details, and to generate the translation of their schemas from one model to another.

This paper describes the schema-related component of *MIDST* (*Model Independent Data and Schema Translation*), a framework for the development of an effective implementation of a generic (i.e., model independent) platform for schema and data translation. The development of data translation has been sketched in a preliminary report [1], but it is beyond the scope of this paper.

In the next section we give an overview of our approach, with a summary of the contributions and a description of the organization of the subsequent sections.

2 Overview

2.1 Constructs and models: a metamodel approach

Our approach is based on the idea of a *metamodel*, defined as a set of constructs that can be used to define models, which are instances of the metamodel. This is based on Hull and King's observation [26] 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 fact, we define a metamodel by means of a set of generic metaconstructs. Each model is defined by its constructs and the metaconstructs they refer to. Simple versions of the models in the examples in Sect. 1 could be defined as follows:

- an ER model involves (i) abstracts (the entities), (ii) aggregations of abstracts (relationships), and (iii) lexicals (attributes of entities and, in most versions of the model, of relationships);
- an OO model has (i) abstracts (classes), (ii) reference attributes for abstracts, which are essentially functions from abstracts to abstracts, and (iii) lexicals (fields or properties of classes);
- the relational model involves (i) aggregations of lexicals (tables), (ii) components of aggregations (columns), which can participate in keys, (iii) foreign keys defined over aggregations and lexicals.

The various constructs are related to one another by means of references (for example, each attribute of an abstract has a reference to the abstract it belongs to) and have properties that specify details of interest (for example, for each attribute we specify whether it is part of the identifier and for each aggregation of abstracts we specify its cardinalities). We will see these aspects in detail in the following sections.

All the information about models and schemas is maintained in a dictionary. We will discuss the dictionary in some detail later in the paper; here we just mention that it has a relational implementation, which is exploited by the specification of translations, written in Datalog.

2.2 The supermodel and translations

A major concept in our 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 and a schema in any model is also a schema in the supermodel, apart from the specific names used for constructs.

It is worth mentioning that while we say that we follow a "metamodel" approach, what we actually implement in our dictionary is the supermodel, as we will see in Sect. 3.

The supermodel gives us two interesting benefits. First, it 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 every pair of models. Thus, a linear and not a quadratic number of translations is needed. Indeed, since every schema in any model is also a schema of the supermodel (modulo construct renaming), the only needed translations are those within the supermodel with the target model in mind: a translation is composed of (a) a "copy" (with construct renaming) from the source model into the supermodel; (b) an actual transformation within the supermodel, whose output includes only constructs allowed in the target model; (c) another copy (again with renaming into the target model). The second advantage is related to the fact that the supermodel emphasizes the common features of models. So, if two source models share a construct, then their translations towards similar target models could share a portion of the translation as well. In our approach, we follow this observation by defining elementary (or *basic*) translations that refer to single constructs (or even specific variants thereof). Then, actual translations are specified as compositions of basic ones, with significant reuse of them.

For example, assume we have as the source an ER model with binary relationships (with attributes) and no generalizations and as the target a simple OO model. To perform the task, we would first translate the source schema by renaming constructs into their corresponding homologous elements (abstracts, binary aggregations, lexicals, generalizations) in the supermodel and then apply the following steps (sketched in Fig. 6):

(1) eliminate attributes of aggregations, by introducing new abstracts and one-to-many aggregations;
(2) eliminate many-to-many aggregations, by introducing new abstracts and one-to-many aggregations;
(3) replace one-to-many aggregations with references between abstracts.

If instead we have a source ER model with generalizations but no attributes on relationships (still binary), then, after the copy in the supermodel, we can apply steps (2) and (3) above, followed by another step that takes care of generalizations:

(4) eliminate generalizations (replacing them with references).

It is important to note that the basic steps are highly reusable. Let us comment on this issue with the help of Fig. 7. The figure shows several models that can be obtained by combining the constructs seen in the previous examples. Indeed, this is just a subset of the models our tool handles (we will come back to this issue at the end of the next section),

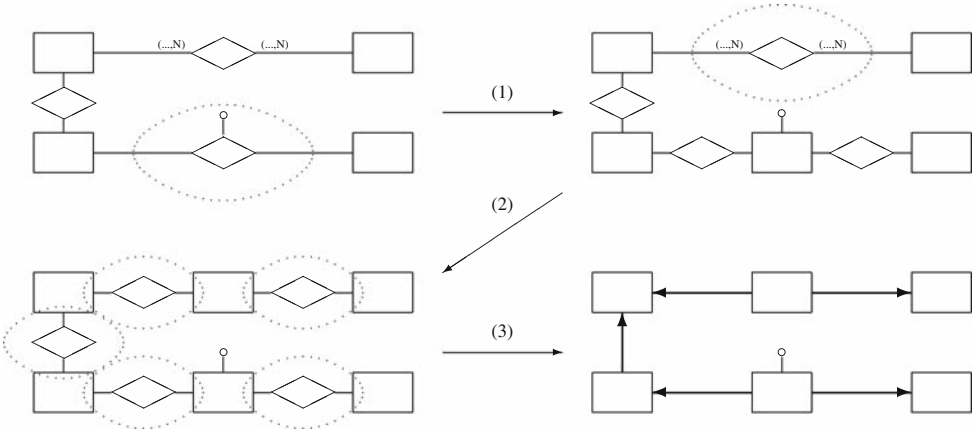


Fig. 6 A translation composed of three steps

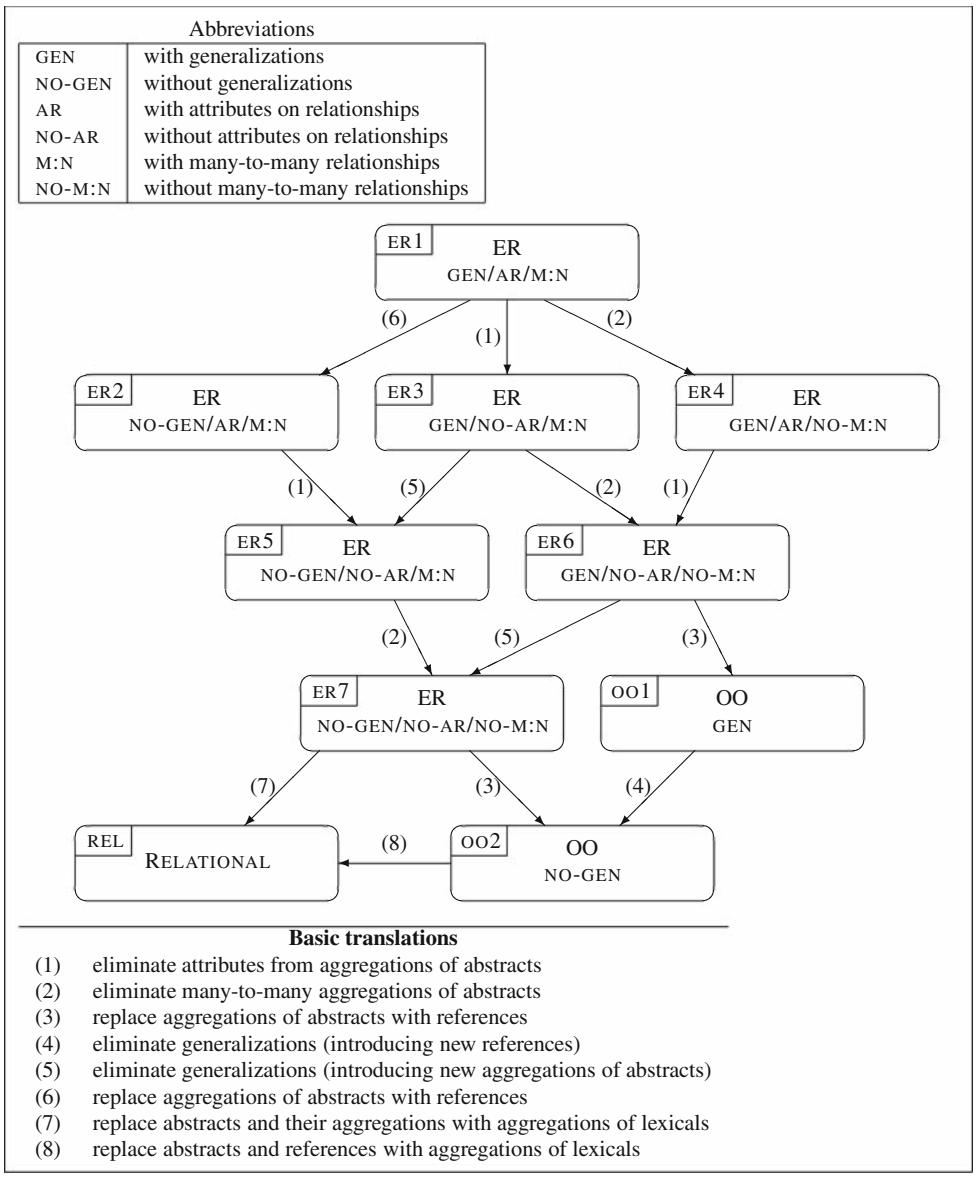


Fig. 7 Some models and translations between them

but it is sufficient to make the point. In the figure, we have also omitted various translations, including the identity one, which would be useful to go from a model to a more complex version of it, for example from ER2 or ER3 to ER1.

The diagram in Fig. 7 shows the translations we have just commented on (with the two source models marked with ER2 and ER3, respectively, and the target model with OO2). Reuse arises in various ways. First, entire translations can be used in various contexts: the translation composed of steps (1), (2) and (3), which we have mentioned for the translation from ER2 to OO2, can be used also to go from the most complex ER model in the picture (the top one, indicated with ER1) to the OO model with generalizations (OO1) in the picture. Second, individual steps can be composed in different ways: for example, if we want to go from ER1 to the relational model (in the bottom left corner), then we could use basic translation (5) to eliminate generalizations, then (1) and (2) to reach a simple ER model (ER7), and finally (7) to get to the relational one.

2.3 Building complex translations

With many possible models and many basic translations, it becomes important to understand how to find a suitable translation given a source and a target model. Intuitively, we could think of using a graph such as that in Fig. 7, with models as nodes and basic translations as edges. Here there are two difficulties. The first one is how to verify what target model is generated by applying a basic step to a source model (for example to verify that transformation (1) indeed generates schemas of model ER3 from schemas of ER1 and that it generates schemas of ER5 from schemas of ER2). The second problem is related to the size of the graph: due to the number of constructs and properties, we have too many models (a combinatorial explosion of them, if the variants of constructs grow) and it would be inefficient to find all associations between basic translations and pairs of models.

We propose a complete solution to the first issue, as follows. We associate a concise *description* with each model, by indicating the constructs it involves with the associated properties (described in terms of propositional formulas), and a *signature* with each basic translation. Then, a notion of *application* of a signature to a model description allows us to obtain the description of the target model. With our basic translations written in a Datalog dialect with OID-invention, as we will see shortly, it turns out that signatures can be automatically generated and the application of signature gives an exact description of the target model.

With respect to the second issue, the complexity of the problem cannot be completely circumvented, but we have devised algorithms that, under reasonable hypotheses, efficiently find a complex translation given a pair of models (source and target). So, for example, given ER2 and OO2, our algorithm properly finds the translation composed of steps (1), (2) and (3), out of a library of many basic translations.

2.4 Specification of basic translations

In the current implementation of our tool, translations are implemented as programs in a Datalog variant with OID-invention, where the latter feature is obtained by means of the use of Skolem functors. Each translation is usually concerned with a very specific task, such as eliminating a certain variant of a construct (possibly introducing another construct), with most of the constructs left unchanged. Therefore, in our programs only a few of the rules concern real translations, whereas most of them just copy constructs from the source schema to the target one. For example, the translation that performs step (3) in Figs. 6 and 7 (the translation from an ER model to an object-oriented one) would involve the rules for the following tasks:

(3-i) copy abstracts;
(3-ii) copy lexical attributes of abstracts;
(3-iii) replace relationships (only one-to-many and one-to-one) with reference attributes;
(3-iv) copy generalizations.

In Fig. 8 we show two of the rules: (3-i), as an example of a copy rule; and (3-iii), the really significant one, which replaces binary one-to-many (or one-to-one) relationships with reference attributes. We will discuss rules in detail in Sect. 4. Here we just make some high-level comments. Rules refer directly to our dictionary, and this is facilitated by the relational implementation: the predicates in the rule correspond to the tables in the dictionary. The body of each rule includes conditions for its applicability. Rule 3-i has no condition, and so it copies all abstracts. Instead, Rule 3-iii, because of the condition in line 13, is applied only to aggregations that have IsFunctional1=TRUE, that is, as we will clarify later, one-to-many and one-to-one relationships. Row 2 of each rule "generates" a new construct instance in the dictionary with a new identifier generated by a Skolem function;[1] Rule 3-i generates a new ABSTRACT for each ABSTRACT in the source schema (and it is a copy, except for the internal identifier), whereas Rule 3-iii generates a new ABSTRACTATTRIBUTE for each BINARYAGGREGATIONOFABSTRACTS, with suitable features.

It is worth noting that the specification of rules in Datalog allows for another way of reuse. In fact, a basic translation is a program made of several Datalog rules, and it is often the case that a rule is used in various programs. This happens for all "copy" rules, such as 3-i above, but also for many other

[1] We will comment on Skolem functors, which may appear both in heads and in bodies of rules, in Sect. 4.

Fig. 8 Two Datalog rules

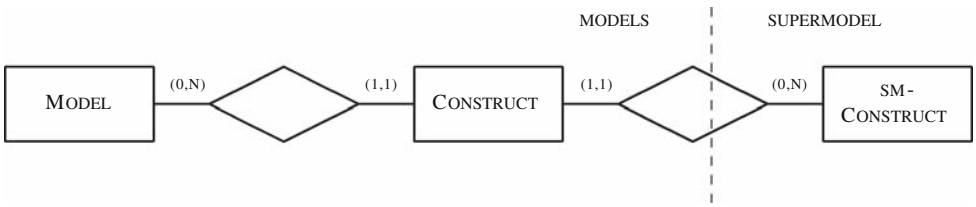
Rule 3-i

```
1.  ABSTRACT(
2.          OID: #abstract_0(absOid),
3.          Abs-Name: name,
4.          Schema: tgt )
5.  ← ABSTRACT(
6.          OID: absOid,
7.          Abs-Name: name,
8.          Schema: src )
```

Rule 3-iii

```
1.  ABSTRACTATTRIBUTE(
2.          OID: #abstractAttribute_1(aggrOid),
3.          Abstract: #abstract_0(absOid1),
4.          Att-Name: aggName,
5.          AbstractTo: #abstract_0(absOid2),
6.          IsOptional: isOpt1,
7.          Schema: tgt )
8.  ← BINARYAGGREGATIONOFABSTRACTS(
9.          OID: aggrOid,
10.         Agg-Name: aggName,
11.         Abstract1: absOid1,
12.         IsOptional1: isOpt1,
13.         IsFunctional1: "TRUE",
14.         Abstract2: absOid2,
15.         Schema: src )
```

Fig. 9 A simplified conceptual view of models and constructs



rules; for example, the two programs that implement steps (7) and (8) in Fig. 7 (which translate into the relational model from a simple ER and a simple OO, respectively) would definitely share a rule that transforms abstracts into aggregations of lexicals (entities or classes into tables) and a rule that transforms attributes of abstracts into lexical components of aggregations (attributes into columns).

2.5 Organization of the paper

The rest of the paper is organized as follows. In Sect. 3, we present the metamodel approach and the dictionary that handles models and schemas. Then, in Sect. 4, we discuss basic translations and their specification in Datalog. In Sect. 5, we discuss how complex translations are generated out of a library of basic translations. Then, in Sect. 6, we briefly describe the features of the tool we have implemented to validate the approach and show its application to a number of examples. In Sect. 7, we discuss related work. Sect. 8 is the conclusion.

3 Models, schemas and the dictionary

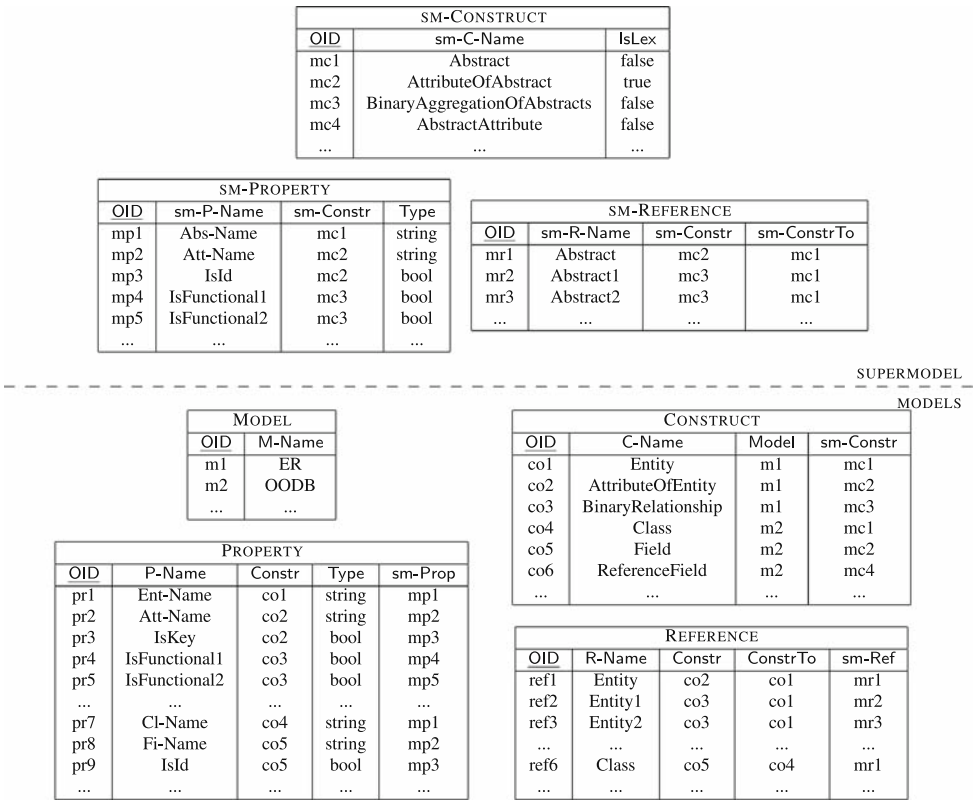
3.1 Description of models

As we observed in the previous section, the starting point of our approach is the idea that a *metamodel* is a set of constructs (called *metaconstructs*) that can be used to define models, which are instances of the metamodel. Therefore, we actually define a model as a set of constructs, each of which corresponds to a metaconstruct. An even more important notion, also mentioned in the previous section, is the *supermodel*: it is a model that has a construct for each metaconstruct, in the most general version. Therefore, each model can be seen as a specialization of the supermodel, except for renaming of constructs.

A conceptual view of the essentials of this idea is shown in Fig. 9: the supermodel portion is predefined, but can be extended, whereas models are defined by specifying their respective constructs, each of which refers to a construct of the supermodel (SM-construct) and so to a metaconstruct. It is important to observe that our approach is independent of the specific supermodel that is adopted, as new metaconstructs and so SM-constructs can be added. This allows us to show simplified examples for the set of constructs, without losing the generality of the approach.

In order to make things concrete and to comment on some details, we show in Fig. 10 the relational implementation of a portion of the dictionary, as we defined it in our tool. The actual implementation has more tables and more columns for each of them. We concentrate on the principles and omit marginal details.The SM- CONSTRUCT table shows (a subset of) the generic constructs (which correspond to the metaconstructs) we have "Abstract," "AttributeOfAbstract," "BinaryAggregationOfAbstracts," and so on; these are the categories according to which the constructs of interest can be classified. Then, each construct in the CONSTRUCT table refers to an SM-construct (by means of the sm-Constr column whose values contain foreign keys of SM- CONSTRUCT) and to a model (by means of Model). For example, the first row in CONSTRUCT has value "mc1" for sm-Constr in order

Fig. 10 The relational implementation of a portion of the dictionary



SM-CONSTRUCT

OID	sm-C-Name	IsLex
mc1	Abstract	false
mc2	AttributeOfAbstract	true
mc3	BinaryAggregationOfAbstracts	false
mc4	AbstractAttribute	false
...	...	...

SM-PROPERTY

OID	sm-P-Name	sm-Constr	Type
mp1	Abs-Name	mc1	string
mp2	Att-Name	mc2	string
mp3	IsId	mc2	bool
mp4	IsFunctional1	mc3	bool
mp5	IsFunctional2	mc3	bool
...	...	...	...

SM-REFERENCE

OID	sm-R-Name	sm-Constr	sm-ConstrTo
mr1	Abstract	mc2	mc1
mr2	Abstract1	mc3	mc1
mr3	Abstract2	mc3	mc1
...	...	...	...

SUPERMODEL

MODELS

MODEL

OID	M-Name
m1	ER
m2	OODB
...	...

CONSTRUCT

OID	C-Name	Model	sm-Constr
co1	Entity	m1	mc1
co2	AttributeOfEntity	m1	mc2
co3	BinaryRelationship	m1	mc3
co4	Class	m2	mc1
co5	Field	m2	mc2
co6	ReferenceField	m2	mc4
...	...	...	...

PROPERTY

OID	P-Name	Constr	Type	sm-Prop
pr1	Ent-Name	co1	string	mp1
pr2	Att-Name	co2	string	mp2
pr3	IsKey	co2	bool	mp3
pr4	IsFunctional1	co3	bool	mp4
pr5	IsFunctional2	co3	bool	mp5
...	...	...	...	...
pr7	Cl-Name	co4	string	mp1
pr8	Fi-Name	co5	string	mp2
pr9	IsId	co5	bool	mp3
...	...	...	...	...

REFERENCE

OID	R-Name	Constr	ConstrTo	sm-Ref
ref1	Entity	co2	co1	mr1
ref2	Entity1	co3	co1	mr2
ref3	Entity2	co3	co1	mr3
...	...	...	...	...
ref6	Class	co5	co4	mr1
...	...	...	...	...

to specify that "Entity" is a construct (of the "ER" model, as indicated by "m1" in the Model column) that refers to the "Abstract" SM-construct. It is worth noting (fourth row of the same table) that "Class" is a construct belonging to another model ("OODB") but referring to the same SM-construct.

Tables SM- PROPERTY and SM- REFERENCE describe, at the supermodel level, the main features of constructs, properties and relationships among them. We discuss each of them in turn. Each SM-construct has some associated properties, described by SM- PROPERTY, which will then require values for each instance of each construct corresponding to it. For example, the first row of SM- PROPERTY tells us that each "Abstract" (mc1) has an "Abs-Name," whereas the third says that for each "AttributeOfAbstract" (mc2) we can specify whether it is part of the identifier of the "Abstract" or not (property "IsId"). Correspondingly, at the model level, we have that each "Entity" has a name (first row in table PROPERTY) and that for each "AttributeOfEntity" we can tell whether it is part of the key (third row in PROPERTY). In the latter case the property has a different name ("IsKey" rather than "IsId"). It is worth observing that "Class" and "Field" have the same features as "Entity" and "AttributeOfEntity," respectively, because they correspond to the same pair of SM-constructs, namely "Abstract" and "AttributeOfAbstract." Other interesting properties are specified in the fourth and fifth rows of SM- PROPERTY. They allow for the specification of cardinalities of binary aggregations by saying whether the participation of an abstract is "functional" or not: a many-to-many relationship will have two *false* values, a one-to-one two *true* ones, and a one-to-many a *true* and a *false*.

Table SM- REFERENCE describes how SM-constructs are related to one another by specifying the references between them. For example, the first row of SM- REFERENCE says that each "AttributeOfAbstract" (construct mc2) refers to an "Abstract" (mc1); this reference is named "Abstract" because its value for each attribute will be the abstract it belongs to. Again, we have the issue repeated at the model level as well: the first row in table REFERENCE specifies that "AttributeOfEntity" (construct "co2," corresponding to the "AttributeOfAbstract" SM-construct) has a reference to "co1" ("Entity"). The same holds for "Class" and "Field," again, as they have the same respective SM-constructs. The second and third rows of SM- REFERENCE describe the fact that each binary aggregation of abstracts involves two abstracts and the homologous happens for binary relationships in the second and third rows of REFERENCE.

The close correspondence between the two parts of our dictionary is a consequence of the way it is managed. The supermodel part (top of Fig. 10) is its core; it is predefined (but can be extended) and used as the basis for the definition of specific models. Essentially, the dictionary is initialized with the available SM-constructs, with their properties and references. Initially, the model-specific part of the

Fig. 11 The dictionary for a simple ER model

SCHEMA

OID	S-Name
s1	SchemaER1
s2	SchemaER2

ENTITY

OID	Ent-Name	Schema
e1	School	s1
e2	Professor	s1
e3	Course	s1
e4	Employee	s2
e5	Project	s2

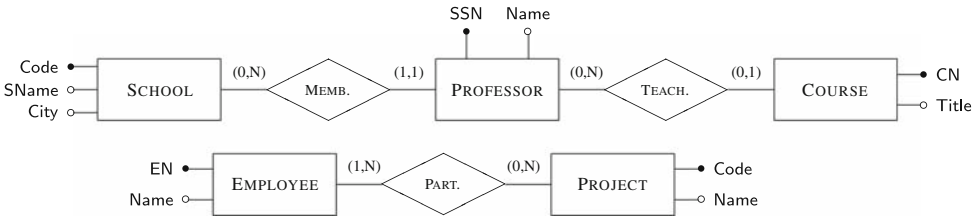
ATTRIBUTEOFENTITY

OID	Entity	Att-Name	Type	isKey	Schema
a1	e1	Code	int	true	s1
a2	e1	SName	string	false	s1
a3	e1	City	string	false	s1
a4	e2	SSN	int	true	s1
a5	e2	Name	string	false	s1
a6	e3	CN	int	true	s1
a7	e3	Title	string	false	s1
a8	e4	EN	int	true	s2
...	...	...	...	...	...

BINARYRELATIONSHIP

OID	Rel-Name	Entity1	IsOpt1	IsFunctional1	Entity2	...	Schema
b1	Membership	e2	false	true	e1	...	s1
b2	Teaching	e3	true	true	e2	...	s1
b3	Participation	e4	false	false	e5	...	s2

Fig. 12 The ER schemas described in the dictionary in Fig. 11



dictionary is empty and then individual models can be defined by specifying the constructs they include by referring to the SM-constructs they refer to. In this way, the model part (bottom of Fig. 10) is populated with rows that correspond to those in the supermodel part, except for the specific names, such as "Entity" or "AttributeOfEntity," which are model specific names for the SM-constructs "Abstract" and "AttributeOfAbstract," respectively. This structure causes some redundancy between the two portions of the dictionary, but this is not a great problem, as the model part is generated automatically: the definition of a model can be seen as a list of supermodel constructs, each with a specific name.

An additional feature we have is the possibility of specifying conditions on the properties for a construct, in order to put restrictions on a model. For example, to define an object model that does not allow the specification of identifying fields, we could add a condition that says that the property "IsId" associated with "Field" is identically "false." These restrictions can be expressed as propositional formulas over the properties of constructs. We will make this observation more precise in Sect. 5.2.

3.2 Description of schemas

The model portion of our dictionary is a metadictionary in the sense that it can be the basis for the description of model-specific dictionaries, with one table for each construct, with their respective properties and references. For example, the schema of a dictionary for the binary ER model mentioned above includes tables ENTITY, ATTRIBUTEOFENTITY, and BINARYRELATIONSHIP, as shown in Fig. 11. The content of the dictionary describes two schemas, shown in Fig. 12. The dictionary for a model has a structure that can be automatically generated when the model is defined. At the initialization of the tool, this portion of the dictionary is empty, and the tables for each are created when the model is defined.

Figure 13 shows a similar dictionary, for an object data model, with very simple constructs, namely class, field (as discussed above), and reference field. The corresponding schema is shown in Fig. 14.

In the same way as the supermodel gives a unified view of all the constructs of interest, it is useful to have an integrated view of schemas in the various models, describing them in terms of their SM-constructs rather than constructs as we have done so far. As we anticipated in Sect. 2, this gives great benefits to the translation process. The dictionary for the supermodel has tables whose names are those of the SM-constructs and whose columns correspond to their properties and references. We show an excerpt of the dictionary in Fig. 15. Its content is obtained by putting together the information in the model-specific dictionaries. For example, Fig. 15 shows the portion of the supermodel that suffices for the descriptions of the schemas in the versions of the ER and OO model shown in Figs. 11 and 13. In particular, the table

Fig. 13 The dictionary for a simple OO model

SCHEMA

OID	S-Name
s3	SchemaOO1

CLASS

OID	Cl-Name	Schema
c1	School	s3
c2	Professor	s3
c3	Course	s3

FIELD

OID	Class	Fi-Name	Type	isId	Schema
f1	c1	Code	int	true	s3
f2	c1	SName	string	false	s3
f3	c1	City	string	false	s3
f4	c2	SSN	int	true	s3
...	...	...	...	...	...

REFERENCEFIELD

OID	Class	Ref-Name	ClassTo	isOpt	Schema
r1	c2	Memb.	c1	false	s3
r2	c3	Teach.	c2	true	s3

Fig. 14 The OO schema described in the dictionary in Fig. 13

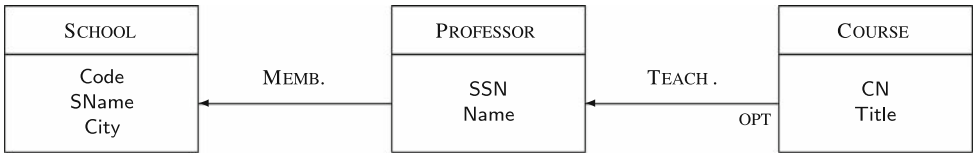


Fig. 15 A model-generic dictionary, based on the supermodel

SCHEMA

OID	SchemaName	Model
s1	SchemaER1	m1
s2	SchemaER2	m1
s3	SchemaOO1	m2

ABSTRACT

OID	Abs-Name	Schema
e1	School	s1
e2	Professor	s1
e3	Course	s1
e4	Employee	s2
e5	Project	s2
c1	School	s3
...	...	...

ATTRIBUTEOFABSTRACT

OID	Abstract	Att-Name	Type	isId	Schema
a1	e1	Code	int	true	s1
a2	e1	SName	string	false	s1
a3	e1	City	string	false	s1
a4	e2	SSN	int	true	s1
a5	e2	Name	string	false	s1
...	...	...	...	...	...
a8	e4	EN	int	true	s2
...	...	...	...	...	...
f1	c1	Code	int	true	s3
...	...	...	...	...	...

ABSTRACTATTRIBUTE

OID	Abstract	Att-Name	AbstractTo	isOpt	Schema
r1	c2	Memb.	c1	false	s3
r2	c3	Teach.	c2	true	s3

BINARYAGGREGATIONOFABSTRACTS

OID	Agg-Name	Abstract1	IsOptional1	IsFunctional1	Abstract2	...	Schema
b1	Membership	e2	false	true	e1	...	s1
b2	Teaching	e3	true	true	e2	...	s1
b3	Participation	e4	false	false	e5	...	s2

ABSTRACT is the union (modulo suitable renaming) of tables ENTITY and CLASS in Figs. 11 and 13, respectively.[2] Similarly, ATTRIBUTEOFABSTRACT is the union of ATTRIBUTEOFENTITY and FIELD.

We summarize our approach to the description of schemas and models by means of Fig. 16. We have a dictionary composed of four parts, with two coordinates: schemas (lower portion) versus models (upper portion) and model-specific (left portion) versus supermodel (right portion).

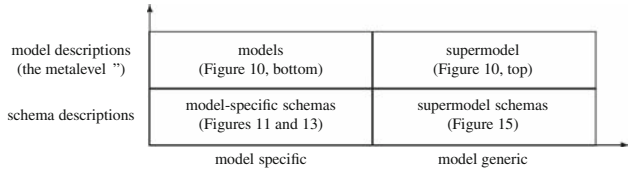


Fig. 16 The four parts of the dictionary

3.3 Generality of the approach

The above discussion confirms that it is indeed possible to describe many models and variations thereof by means of just a few more constructs. In the current version of our tool we have nine constructs, the four shown in Fig. 15 (with ATTRIBUTEOFABSTRACT replaced by the more general

[2] In the figures, for the sake of understandability, we have used, for each construct and schema, the same identifier in the supermodel dictionary and in the model specific one. So, for example, "s1" is used both for a schema in the ER model and for the corresponding one in the supermodel.

Fig. 17 Constructs and models

	Entity-Relationship	BinaryEntity-Relationship	Object(UML ClassDiagram)	Object-Relational	Relational	XSD
Abstract	Entity	Entity	Class	TypedTable		Root-Element
Lexical	Attribute	Attribute	Field	Column	Column	Simple-Element
BinaryAggregation-OfAbstracts		Binary-Relationship				
AbstractAttribute			ReferenceField	Reference		
Aggregation-OfAbstracts	Relationship					
Generalization	Generalization	Generalization	Generalization	Generalization		
Aggregation				Table	Table	
ForeignKey				ForeignKey	ForeignKey	ForeignKey
Structure-OfAttributes			Structured-Field	Structured-Column		Complex-Element

LEXICAL, used for all value-based simple constructs) plus additional ones for representing n-ary aggregations of abstracts, generalizations, aggregations of lexicals, foreign keys, and structured (and possibly nested and/or multivalued) attributes. The relational implementation has a few additional tables, as some constructs require two tables, because of normalization. For example, n-ary aggregations require two tables, one for the aggregations and one for the components of each of them. We summarize constructs and (families[3] of) models in Fig. 17, where we show a matrix, whose rows correspond to the constructs and columns to the families we have experimented with. In the cells, we use the specific name used for the construct in the family (for example, Abstract is called Entity in the ER model). The various models within a family differ from one another (i) on the basis of the presence or absence of specific constructs and (ii) on the basis of details of (constraints on) them. To give an example for (i) let us recall that versions of the ER model could have generalizations, or not have them, and the OR model could have structured columns or just simple ones. For (ii) we can just mention again the various restrictions on relationships in the binary ER model (general vs. one-to-many), which can be specified by means of constraints on the properties. It is also worth mentioning that a given construct can be used in different ways (again, on the basis of conditions on the properties) in different families: for example, we have used a multivalued structured attribute in the XSD family and a monovalued one in the OR family (we have a property isSet which is constrained to *false*).

An interesting issue to consider here is "How universal is this approach?" or, in other words, "How can we guarantee that we can deal with every possible model?" A major point is that the metamodel is extensible, which is both a weakness and a strength. It is a weakness because it confirms that it is impossible to say you have a complete one. However, it is a strength because it allows the addition of features when needed. This applies both to the details of the models of interest and to the families of models. With respect to the first issue, let us give an example: if one wants to handle XSD in full detail, then the metamodel and the supermodel need to be complex at least as XSD is. In fact, the level of detail can vary greatly and it can be chosen on the basis of the context of interest.

With respect to the second issue it is worth mentioning that the approach can be used to handle metamodels in other contexts, with the same techniques. Indeed, we have had preliminary experiences with semantic Web models [3], with the management of annotations [32], and with adaptive systems [21]: for each of them, we defined a new set of constructs (and so a different metamodel and supermodel) and new basic translations, but we used the same framework and the same engine.

In summary, the point is that the approach is independent of the specific supermodel. The supermodel we have mainly experimented with so far is a supermodel for database models and covers a reasonable family of them. If models were more detailed (as is the case for a fully fledged XSD model) then the supermodel would be more complex. Also, other supermodels can be used in other contexts.

4 Basic translations

Section 2 gave a general idea of the specification of translations in our proposal as a composition of basic steps. In this section we give some more details of basic translations and their implementation in Datalog. In the next section we will discuss how to obtain an automatic generation of complex translations out of a library of basic steps.

We have already shown a couple of Datalog rules in Sect. 2, and commented on some aspects. Let us now go into more detail, again referring to the rules in Fig. 8.

[3] The notion of *family* of models is intuitive now and will be made more precise in Sect. 5.

We first comment on our syntax. We use a non-positional notation for rules, so we indicate the names of the fields and omit those that are not needed (rather than using anonymous variables). Our rules generate constructs for a target schema (*tgt*) from those in a source schema (*src*). We may assume that variables *tgt* and *src* are bound to constants when the rule is executed. Each predicate has an OID argument, as we saw in Fig. 8. For each schema we have a different set of identifiers for the constructs. So, when a construct is produced by a rule, it has to have a "new" identifier. It is generated by means of a Skolem functor, denoted by the # sign in the rules.

We have the following restrictions on our rules. First, we have the standard "safety" requirements [38]: the literal in the head must have all fields, and each of them with a constant or a variable that appears in the body (in a positive literal) or a Skolem term. Similarly, all Skolem terms in the head or in the body have arguments that are constants or variables that appear in the body. Moreover, our Datalog programs are assumed to be coherent with respect to referential constraints: if there is a rule that produces a construct C that refers to a construct C', then there is another rule that generates a suitable C' that guarantees the satisfaction of the constraint. In the example in Fig. 8, rule (3-iii) is acceptable because there is rule (3-i) that copies abstracts and guarantees that references to abstracts by (3-iii) are not dangling.

The body of rule (3-iii) unifies only with binary aggregations that have TRUE as a value for IsFunctional1: this is the condition that holds for all one-to-one and one-to-many relationships.[4] For each of them it generates a reference attribute from the abstract that participates with cardinality one (IsFunctional1=TRUE) to the other abstract.[5] This basic translation is designed for models that do not have many-to-many relationships. If we apply this rule to a model with many-to-many relationships, without applying a step that removes them before (step (2) in the examples above and in Fig. 7), then we would lose the information carried by those relationships. We will formalize this point later in Sect. 5 in such a way that we could say that step (3) ignores many-to-many relationships.

Let us comment more on the two rules in Fig. 8. Rule (3-i) generates a new abstract (belonging to the target schema) for each abstract in the source schema. The Skolem functor #abstract_0 is responsible for the generation of a new identifier. Skolem functors produce injective functions, with the additional constraint that different functions have disjoint ranges, so that a value is generated only by the same function with the same argument values. For the sake of readability (and also for some implementation issues omitted here), we include the name of the target construct (abstract in this case) in the name of the functor, and use a suffix to distinguish the various functors associated with a construct. The _0 suffix always denotes the "copy" functor.

Rule (3-iii) replaces each binary non-many-to-many relationship (BINARYAGGREGATIONOFABSTRACTS, in supermodel terminology) with a reference (ABSTRACTATTRIBUTE). The rule has a variety of Skolem functors. The head has three Skolem terms, which make use of two functors: #referenceAttribute_1, for the OID field, and #abstract_0 twice, for Abstract and AbstractTo respectively. The two play different roles. The first Skolem term generates a new value, as we saw for the previous rule. Indeed, this is the case for all functors appearing in the OID field of the head of a rule. The other two terms correlate the element being created with elements created by rule (3-i), namely new abstracts generated in the target schema as copies of abstracts in the source one. The new ABSTRACTATTRIBUTE being generated belongs to the ABSTRACT in the target schema generated for the ABSTRACT denoted by variable *absOid1* in the source schema and refers to the target ABSTRACT generated for the source ABSTRACT denoted by *absOid2*.

Our approach to rules allows for a lot of reusability, at various levels. First of all, we have already seen that individual basic translations can be used in different contexts; in the space of models in Fig. 7, each translation can be used in many simple steps. For example, translation (3) can be used to transform relationships into references, for going from different variants of the ER model to homologous variants of the OO one. In the figure, it can be used to go from ER6 to OO1 or from ER7 to OO2.

Second, as each translation step is composed of a number of Datalog rules, some of which are just "copy" rules, they can be used in many basic translations. This is easily the case for plain copy rules, but can be applied also to "conditional" ones, that is, copy rules that are applied only to a subset of the constructs. For example, the rule that eliminates many-to-many relationships copies all the relationships that are not many-to-many; this can be done with a copy rule extended with an additional condition in the body.

In some cases, basic translations can be written with respect to a "core" set of Datalog rules, with copy rules added automatically, given the set of constructs in the supermodel. In this way, the approach would become partially independent of the current supermodel, especially with respect to its extensions. For example, in our case, assume that initially the supermodel does not include generalizations; our basic translation (3), which replaces binary aggregations with references, could be defined by means of the specification of rule (3-iii), with rule (3-i), which copies abstracts, added

[4] As we saw in Sect. 3, binary aggregations have properties IsFunctional1 and IsFunctional2 for specifying cardinalities. We assume that it cannot be the case that IsFunctional2=TRUE and IsFunctional1=FALSE; therefore if IsFunctional1 is FALSE then the relationship is many-to-many and if it is TRUE it is not many-to-many.

[5] For one-to-one relationships, both abstracts participate with cardinality one. We choose one of them, the first one, even if the converse would be appropriate as well.

automatically because of referential integrity in the supermodel, and rule (3-ii), which copies attributes of abstracts, added because attributes of abstract are "compatible" with abstracts. Then, if the supermodel were extended with generalizations, the basic translation would be extended with rule (3-iv), which copies generalizations.

Most of our rules, such as the two we saw, are recursive according to the standard definition. However, recursion is only "apparent." A literal occurs in both the head and the body, but the construct generated by an application of the rule belongs to the target schema, so the rule cannot be applied to it again, as the body refers to the source schema. A really recursive application happens only for rules that have atoms that refer to the target schema also in their body. In the following, we will use the term *strongly recursive* for these rules.

In our experiments, we have developed a set of basic translations to handle the models that can be defined with our current metamodel. They are listed in Appendix 1. In the next Section we will discuss arguments to confirm the adequacy of this set of rules. Then in Sect. 6.2 we discuss a few complex translations built out of these basic ones.

5 Properties of translations and their generation

5.1 Correctness, a difficult problem

In data translation (and integration) frameworks, correctness is usually modeled in terms of information-capacity dominance and equivalence. See Hull [24,25] for the fundamental notions and results and Miller et al. [28,29] for their role in schema integration and translation. In this context, it turns out that various problems are undecidable if they refer to models that are sufficiently general [25, p. 53], [29, p. 11–13]. Also, a lot of work has been devoted to the correctness of specific translations, with ongoing efforts for recently introduced models. See for example the recent contributions by Barbosa et al. [7,8] on XML-to-relational mappings and by Bohannon et al. [15] on transformations within the XML world. Undecidability results have emerged even in discussions on translations from one specific model to another specific one [7,15].

Therefore, given the generality of our approach, it seems hopeless to aim at showing correctness in general. However, this is only a partial limitation, as we are developing a platform to support translations, and some responsibilities can be left to its users (specifically, rule designers, who are expert users, as we will see in Sect. 6.1), with system support. We briefly elaborate on this issue.

We follow the initial method of Atzeni and Torlone [6], which uses an "axiomatic" approach. It assumes the basic translations to be correct, a reasonable assumption as they refer to well-known elementary steps developed by the rule designer. So given a suitable description of models and rules in terms of the involved constructs, complex translations can be proven correct by induction. A similar argument is mentioned by Batini and Lenzerini [10] in support of using transformations as a preliminary step in data integration.

In MIDST, we have the additional benefit that transformations are expressed at a high-level, as Datalog rules. So, rather than taking on faith the features of each basic transformation (as in [6]), we can automatically detect which constructs are used in the body and generated in the head of a Datalog rule and then derive a concise representation of the rule, called *signature*. We have shown elsewhere [4] that a Datalog-based approach allows a formalization of the notions of interest. Let us show the main points, to make this discussion self contained.

5.2 Concise description of models

As we saw in Sect. 3, we define our models in terms of the constructs they involve, taken from the supermodel. Each construct has a fixed set of references and properties, which are all boolean and can be constrained by means of propositional formulas. In a concise but precise way, a model can therefore be defined by listing the constructs it involves, each with an associated proposition, which can be seen as a restriction (a "check" in SQL terminology) over the occurrences of the construct. For example, in binary aggregations of abstracts, we have, as we saw in Sect. 3, properties IsFunctional1 and IsFunctional2, which are used to specify the maximum cardinality of the two components of the aggregation. Specifically, many-to-many aggregations (relationships) have FALSE for both, and one-to-many have TRUE for one and FALSE for the other. Therefore, if we want to specify in a model that binary aggregations have no restrictions on cardinalities (so one-to-one, one-to-many, and many-to-many are all allowed), then we associate the *true* proposition with the BINARYAGGREGATIONOFABSTRACTS construct. If instead we want to forbid many-to-many, then we would associate the proposition IsFunctional1 $\vee$ IsFunctional2, which says that, for each occurrence of the construct, at least one of IsFunctional1 and IsFunctional2 has to be TRUE.

In the following, we write the *description* DESC(M) of a model M in the form $\{C_1(f_1), \ldots, C_n(f_n)\}$, where $C_1, \ldots, C_n$ are the constructs in the model and $f_1, \ldots, f_n$ are the propositions associated with them, respectively.[6]

With suitable shorthands for the names of constructs and properties (and distinguishing between the various types of lexicals), the descriptions of some of the models seen in the previous examples and shown in Fig. 7 are the following:

[6] Also, when no confusion arises, we will often blur the distinction between M and DESC(M), and write M for the sake of brevity.

ER1 {ABS (*true*), LEXATTOFABS (*true*), BINAGG (*true*), LEXATTOFBINAGG (*true*), GEN(*true*)}: a binary ER model with all constructs and no restrictions on them;

ER4 {ABS (*true*), LEXATTOFABS(*true*), BINAGG(isF1 ∨ isF2), LEXATTOFBINAGG(*true*), GEN(*true*)}; here the proposition isF1 ∨ isF2 for BINAGG specifies that relationships are not many-to-many, as we saw above;

ER6 {ABS(*true*), LEXATTOFABS(*true*), BINAGG(isF1 ∨ isF2), GEN(*true*)}; here, with respect to model ER4, we do not have LEXATTOFBINAGG (attributes of aggregations); so this is a binary ER model with no many-to-many relationships and no attributes on relationships;

REL {AGGREG(*true*), LEXCOMPOFAGG(*true*)}; this model has tables and columns, with no restrictions.

We can define a partial order on models and a lattice on the space of models, by means of the following relation, which is reflexive, antisymmetric and transitive:

- $M_1 \sqsubseteq M_2$ (read M_2 *subsumes* M_1) if for every $C(f_1) \in M_1$ there is $C(f_2) \in M_2$ such that $f_1 \wedge f_2$ is equivalent to f_1

In words, $M_1 \sqsubseteq M_2$ means that M_2 has at least the constructs of M_1 ("for every $C(f_1) \in M_1$ there is $C(f_2) \in M_2$") and, for those in M_1, it allows at least the same variants ("$f_1 \wedge f_2$ is equivalent to f_1"). For the example models, we have that ER6 $\sqsubseteq$ ER4 $\sqsubseteq$ ER1.

A lattice can be defined on the space of models with respect to the following operators (modulo equivalence of propositions):

$$
\begin{aligned}
M_1 \sqcup M_2 = {} & \{C(f) \mid C(f) \in M_1 \text{ and no } C(f') \in M_2\} \cup \\
& \{C(f) \mid C(f) \in M_2 \text{ and no } C(f') \in M_1\} \cup \\
& \{C(f_1 \vee f_2) \mid C(f_1) \in M_1 \text{ and } C(f_2) \in M_2\} \\
M_1 \sqcap M_2 = {} & \{C(f_1 \wedge f_2) \mid C(f_1) \in M_1, C(f_2) \in M_2 \\
& \text{and } f_1 \wedge f_2 \text{ is satisfiable }\}
\end{aligned}
$$

The notion of description can be used for schemas as well. The *description* of a schema includes the constructs appearing in it, each one with a proposition that is the disjunction of the propositions associated in the schema with such a construct. For example, if a schema contains two binary aggregations of abstract, both with IsFunct1 equal to TRUE and one with IsFunct2 equal to TRUE and the other to FALSE, then the associated proposition would be the disjunction of isF1 ∧ isF2 and isF1 ∧ ¬isF2 and so isF1; thus, the description of the schema would contain BINAGG(isF1). It turns out that the description of a schema is also the description of a model and that the description of a model is the least upper bound (⊔) of the descriptions of its schemas.

5.3 Signatures and applications of Datalog programs

This formalization of model and schema descriptions can be extremely useful in our framework together with a notion of *signature* for a Datalog program with the associated concept of *application* of a signature to a model.

As a preliminary step, let us define the *signature of an atom* in a Datalog rule. Given an atom C(ARGS) (where C is a construct name and ARGS is the list of its arguments), consider the fields in ARGS that correspond to properties (ignoring the others); let them be $p_1 : v_1, \ldots, p_k : v_k$; each v_i is either a variable or a boolean constant *true* or *false*. Then, the signature of C(ARGS) has the form $C(f)$, where f is a proposition that is the conjunction of literals corresponding to the properties in $p_1, \ldots, p_k$ that are associated with a constant; each of them is positive if the constant is *true* and negated if it is *false*. If there are no constants, then the proposition is *true*. For example (with some intuitive abbreviations), the signature of the atom in the body of Rule 3-i is ABS(*true*), because abstracts have no properties, whereas the one in the body of Rule 3-iii is BINAGG(isF1), because the *true* constant is associated with IsFunctional1.

Consider a Datalog rule R with an atom C(ARGS) as the head and a list of atoms $\langle C_{j_1}(\text{ARGS}_1), \ldots, C_{j_h}(\text{ARGS}_h)\rangle$ as the body; comparison terms do not affect the signature, and so we can ignore them.

The signature SIG(R) of rule R is composed of three parts, (B, H, MAP):

- B (the *body of* SIG(R)) describes the applicability of the rule, by referring to the constructs in the body of R; B is a list $\langle C_{j_1}(f_1), \ldots, C_{j_h}(f_h)\rangle$, where $C_{j_i}(f_i)$ is the signature of the atom $C_{j_i}(\text{ARGS}_i)$.
- H (the *head of* SIG(R)) indicates the conditions that definitely hold on the construct obtained as the result of the application of R, because of constants in its head; H is defined as the signature $C(f)$ of the atom C(ARGS) in the head.
- MAP (the *mapping of* SIG(R)) is a partial function that describes where values of properties in the head originate from. It is denoted as a list of pairs where, for each property in the head that has a variable, there is the indication of the property in the body where it comes from (because of the safety requirements, each variable in the head appears also in the body, and only once). Rule 3-i has an empty MAP, because there are no properties in the head, whereas for Rule 3-iii MAP includes the pair "IsOptional: BINAGG(IsOptional1)", which specifies that values of property IsOptional in each ABSTRACTATTRIBUTE generated by this rule are copied from values of IsOptional1 of BINARYAGGREGATIONOF-

ABSTRACTS. As a consequence, if the source model has always a constant for the property IsOptional1 of BINARYAGGREGATIONOFABSTRACTS, then the same constant always appears in the target model for property IsOptional of ABSTRACTATTRIBUTE.

The *application* $\text{APPL}_R()$ of the signature $\text{SIG}(R)$ of a rule R is a function from model descriptions to model descriptions that illustrates the behavior of the rule. The function $\text{APPL}_R()$ is defined on the basis of $\text{SIG}(R)$ of R. For the current development, the important things to know are that it is well defined and describes the transformation induced by a rule. For example, rule 3-iii applied to a model that contains a BINARYAGGREGATIONOFABSTRACTS construct with property IsFunctional1 that can have the TRUE value (so that the rule is applicable), generates a construct description for ABSTRACTATTRIBUTE. The proposition associated with this construct depends on the proposition associated with BINARYAGGREGATIONOFABSTRACTS in the source model: if there is some restriction on IsOptional1 then the same restriction would appear for IsOptional in the target construct. If rule 3-iii is applied to a model that has no BINARYAGGREGATIONOFABSTRACTS or has them but with FALSE value for IsOptional1, then $\text{APPL}_R()$ produces the empty model, as the body of the rule would not unify with any construct.

Then, we can define the application $\text{APPL}_\mathbf{P}()$ of a program $\mathbf{P}$, consisting of a set of Datalog rules $R_1, \ldots, R_n$, to a model M as the least upper bound of the applications of the R_i's to M: $\text{APPL}_\mathbf{P}(M) = \bigsqcup_{i=1}^{n} \text{APPL}_{R_i}(M)$.

After observing that descriptions of models and signatures of rules can be automatically generated, we can mention the major result from our companion paper [4], which we use as a starting point for the rest of the paper. The claim can be stated as follows: "signatures completely describe the behavior of programs on schemas,[7] in the sense that the application of signatures provides a 'derivation' of schemas that is sound and complete with respect to the signatures of schemas generated by programs."

Consider Fig. 18. Let S be a schema and $\mathbf{P}$ a Datalog program implementing a basic translation. Then, the diagram in Fig. 18 commutes: if from the top-left corner of the diagram we move right and then down (apply program $\mathbf{P}$ to S and then compute the description of the schema $\mathbf{P}(S)$) or down and then right (compute the description of S and then apply the signature of $\mathbf{P}$ to it), we obtain the same result, as $\text{DESC}(\mathbf{P}(S))$ always equals $\text{APPL}_\mathbf{P}(\text{DESC}(S))$.

If we want to be more general and refer to models, the result says that, given a source model M_1 and a program $\mathbf{P}$,

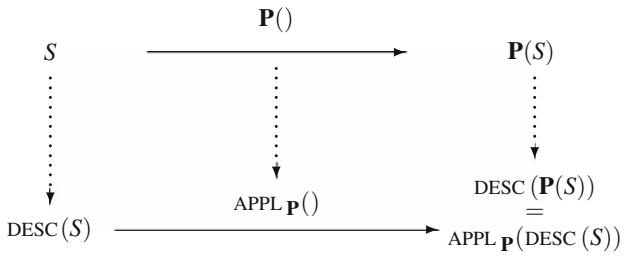


Fig. 18 Models, descriptions, programs, and signatures: a diagram that commutes

we are able to find the signature of the model M_2 to which the schemas obtained by applying $\mathbf{P}$ to schemas of M_1 belong.[8]

In summary, this machinery allows us to know, given a source schema and a basic translation (expressed, as in our case, as a Datalog program), the (minimum) model to which the schema obtained as the result of the translation would belong.

5.4 Generation of complex translations

The formal system just introduced allows us to reason about models and translations between them, with the ambitious goal of automatically generating translations from a source schema to a target model. The machinery would allow us to try a brute-force solution, implemented as an exhaustive search where all combinations of basic steps are tried. This would be somehow possible, but computationally infeasible, as the number of basic steps can grow significantly. Indeed, without specific hypotheses, termination is not even guaranteed as rules could introduce and eliminate the same constructs in turn.

However, we can make a couple of observations that derive from our initial experience with manually written translations and show that they can be generalized to work under reasonable hypotheses. First, while we have many possible models, we have few "families" of models, such as ER, OO, and relational. Each of the families has a most general model (the one with all constructs of the family and no restrictions on its properties), which we call the *progenitor* of the family.

The second observation is that, within a family, most translations just eliminate a feature (dropping a construct or reducing its variants), and generate a schema that is subsumed by the input. Let us call *reductions* the translations that have this property; we will make this more precise. For example, in the various translations in Fig. 7, all translations between models in the ER family follow in this category. Clearly, translations between models in different families are not reductions, as

[7] The result we have [4] is more general, as it refers to models, but we prefer to state it here in terms of schemas, as it is simpler and sufficient for the discussion.

[8] Given the lattice of models, a schema belongs to various models. This method allows one to find the minimum model to which all these schemas belong.

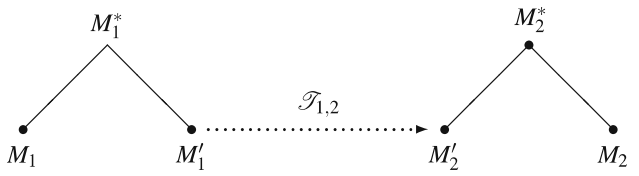


Fig. 19 Translations between families

they typically eliminate one or more constructs and introduce new ones. Let us use the term *transformation* for them.

Now, let us consider translations where the source and target model are in different families. Let us begin with an intuitive argument, which we will refine soon. As the number of families is limited, it is reasonable to assume that for each pair of families (say $\mathscr{F}_1$ and $\mathscr{F}_2$) there is a translation $\mathscr{T}_{1,2}$ from models in family $\mathscr{F}_1$ to a model in $\mathscr{F}_2$. However, given a specific source model M_1 in $\mathscr{F}_1$ and a target model M_2 in $\mathscr{F}_2$, such a translation need not be perfectly suitable, for two reasons: (i) it can ignore constructs in M_1 and (ii) it can generate constructs not in M_2 (albeit in family $\mathscr{F}_2$). Such a translation has been written for going from a model M_1' in $\mathscr{F}_1$ to a model M_2' in $\mathscr{F}_2$. In the worst case, M_1 and M_1' are incomparable (neither is subsumed by the other)—the only thing we can say is that they are both subsumed by the progenitor of the family. The same holds for M_2 and M_2'. Let us proceed with the help of Fig. 19: if we have a schema S_1 of M_1, then S_1 need not be a schema of M_1'; however, S_1 definitely belongs to the progenitor M_1^* of family $\mathscr{F}_1$; thus, if we have a translation (indeed, a reduction) from M_1^* to M_1', we can obtain a schema S_1' of M_1' that is indeed the appropriate translation of S_1 into M_1'. Then, we can apply $\mathscr{T}_{1,2}$ to S_1' and obtain a schema S_2' of M_2'. Again, S_2' need not be a schema of M_2, but it is definitely a schema of the progenitor M_2^* of $\mathscr{F}_2$. So, with a reduction from M_2^* to M_2 we can obtain our desired target schema. In plain words, in this framework translations can be composed of three macrosteps:

1. a reduction within the source family;
2. a transformation from the source family to the target family;
3. a reduction within the target family.

With respect to Fig. 7, we can see that a translation from ER3 to OO2 can be composed of step (2), a reduction within the ER family, step (3), a transformation to the OO family, and (4), a reduction within the OO family.

Let us formalize the various issues.

A *family* $\mathscr{F}$ *of models* is a set of models defined by means of a model M^* (called the *progenitor* of $\mathscr{F}$) and a set of models $M_{*,1}, \ldots, M_{*,k}$ (the *minimal* models of $\mathscr{F}$) and contains all models that are subsumed by M^* and subsume at least one of the $M_{*,i}$'s:

$$\mathscr{F} = \{M \mid M \sqsubseteq M^* \text{ and } M_{*,i} \sqsubseteq M, \quad \text{for some } 1 \leq i \leq k\}$$

In principle, we could think of families as disjoint from one another. However, in practice, it is in some cases convenient to allow some exceptions. Let us start from an observation. The object-relational (OR) model has been proposed in the literature as a generalization of the relational model. Therefore, we could think of just one family, with a rich OR model as the progenitor. While this could work pretty well, we believe that it is more effective to consider the relational model as a separate family, whose progenitor is a minimal model in the OR family. Therefore, we assume hereinafter that families can share models, but only with a minimal model in a family that belongs also to another. Also, as it is often convenient to refer to the *family of a model*, then we assume that in the overlap cases the family is that for which the model is not minimal. Given a model M we will denote its family as FAMILY(M). Similarly, given a schema, we can find the family to which it belongs, since, as we said earlier, there is always a minimum model to which a schema belongs.

With respect to families, in order to be able to perform translations as we have discussed earlier, we need two hypotheses which we state and comment on in turn. We need a definition. Given a schema S and a construct C in S, we say a Datalog program *ignores* C if there is no rule whose body unifies with constructs of S that include a literal that unifies with C. This notion (or, better, its complement) is useful to model the idea that we need translations that take into consideration all elements of the source schema. This notion is aimed at modeling a specific aspect related to information capacity equivalence [24]. In fact, equivalence requires invertibility, which in turn needs injectivity: if constructs are ignored, their presence/absence cannot be distinguished. So a necessary condition for equivalence, and a general condition for translations is that translations do not ignore constructs. It is worth noting that in some cases it is even needed to ignore constructs. Consider for example a very raw relational model with only tables and columns, but no foreign keys. Here any translation from a relational model with foreign keys would essentially ignore them. In this case, our approach requires dummy rules, which generate no constructs, but testify that constructs have not been forgotten.

We are now ready for the first hypothesis on our translations.

Assumption 1 For each pair of families $\mathscr{F}_1$, $\mathscr{F}_2$ there is a model M_1 in $\mathscr{F}_1$ and a translation $\mathscr{T}$ such that, for each schema S_1 of M_1:

1. $\mathscr{T}$ does not ignore any construct of S_1;
2. $\mathscr{T}$ produces a schema that belongs to the progenitor M_2^* of $\mathscr{F}_2$.

This hypothesis requires the existence of f^2 translations, if f is the number of different families. On the one hand this is not a real problem, as f is reasonably small. On the other hand, by using Datalog rules, we reduce the need for actual coding to a minimum. In fact, each of these translations has copy rules for the common constructs and non-copy rules only for the constructs that need to be replaced, namely, those in $\mathscr{F}_1$ and not in $\mathscr{F}_2$, and, whatever the approach, these rules would be needed. In general, what we need is that these f^2 translations can be obtained by combining basic translations (possibly with automatic generation); in practice, it is better to assume that these are indeed basic translations, so they are given, or that they have been manually defined as the "inter-family" translations of interest. Also, in general there might even be pairs of families with more than one translation, but we ignore this issue, as it would not add much to the discussion. When two families share a model (in the case we mentioned above), then the transformation $\mathscr{T}$ between them is, in both directions, the identity, from the minimal model of one to a member of the other or vice versa.

A further observation is useful. Assumption 1 says that $\mathscr{T}$ produces a schema of M_2^*. In most cases, the translation always leads to schemas that belong to a more restricted model, but this is just a simplification. Here we state the assumption in this general way, because it is what suffices for our goals.

Let us now formulate the second hypothesis we need.

Assumption 2 For each family $\mathscr{F}$, for each minimal model $M_{*,i}$ of $\mathscr{F}$, there is a translation from the progenitor M^* of $\mathscr{F}$ to $M_{*,i}$, entirely composed of reductions that do not ignore constructs.

The satisfaction of Assumption 2 can be verified by considering the reductions for the family (that is, the basic translations that are reductions for the progenitor of the family) and performing an exhaustive search on them. This can be done in a fast way, as the number of reductions in a family is small and most of them are commutative. Also, it is important to mention that this test has to be performed when families are defined (or changed) and need not be repeated for each translation nor when an individual model is defined (provided it belongs to an existing family).

An important consequence of Assumption 2 is the possibility of obtaining a translation between any pair of models within a family.

Claim 1 If the set of basic translations satisfies Assumption 2, then, for each family and each pair of models M_1 and M_2 within it, there is a translation from M_1 to M_2 that does not ignore constructs.

Proof Since each model in the family is subsumed by the progenitor M^*, we have that $M_1 \sqsubseteq M^*$. Also, since the family has a set of minimal models, we have that for each model M in the family there is a minimal model M_* that is subsumed by it (either M is minimal, in which case the statement is trivial, or there is another model M' such that $M' \sqsubseteq M$, and we can recursively apply the same argument, at most a finite number of times, as the set of models is finite). Therefore, as M_2 is a model of the family, we have $M_* \sqsubseteq M_2$.

By Assumption 2, there is a translation from M^* to M_* that does not ignore any construct. This translation can be applied to every schema of M_1 (since $M_1 \sqsubseteq M^*$, we have that every schema of M_1 is also a schema of M^*), producing a schema that belongs to M_* and so (as $M_* \sqsubseteq M_2$) also to M_2. □

```
FINDCOMPLETETRANSLATION(S1, M2)
1.    F1 = FAMILY(S1)
2.    F2 = FAMILY(M2)
3.    T = GETTRANSFORMATION(F1, F2)
4.    M1' = GETSOURCE(T)
5.    T1 = GETREDUCTION(F1, M1')
6.    T2 = GETREDUCTION(F2, M2)
7.    return T1 ∘ T ∘ T2
```

Fig. 20 Algorithm FINDCOMPLETETRANSLATION

The previous assumptions and arguments justify the algorithm shown in Fig. 20. The algorithm has an input that is composed of a source schema S_1 and a target model M_2, and refers to a given set of families and a given set of rules. Lines 1 and 2 find the families to which the source schema and the target model belong, respectively. Line 3 finds the transformation $\mathscr{T}$ between the two families, whose existence is guaranteed by Assumption 1. Then, line 4 computes the source model M_1' for transformation $\mathscr{T}$. Next, line 5 finds the sequence of reductions needed to go from the progenitor of $\mathscr{F}_1$ to M_1' (on the basis of Assumption 2) and line 6 does the same within the target family. Finally, the algorithm returns a translation that is the composition of $\mathscr{T}_1$, $\mathscr{T}$, and $\mathscr{T}_2$.

Claim 2 If the set of basic translations satisfies Assumptions 1 and 2, then the algorithm in Fig. 20 is correct, that is, for every schema S_1 and model M_2, it finds a translation of S_1 into M_2 that does not ignore constructs.

The Algorithm is indeed used in our tool for the automatic generation of translations. In the Appendix, we show that the list of basic translations we used satisfies Assumptions 1 and 2 and so the translation can always be generated.

6 Implementation and experimentation

6.1 The MIDST tool

We developed a tool to validate the concepts in previous sections and to test their effectiveness. The main parts of

the MIDST tool are the generic data dictionary, the rule repository and a plug-in-based application that handles the components in a modular way. The main components of a first version, recently demonstrated [2], include a set of modules to support users in defining and managing models, schemas, Skolem functions, translations, import and export of schemas. A second version, just completed, includes two more components: one to extract signatures from rules and models and a second one to generate translation plans.

It is useful to discuss the tool by referring to three categories of users, corresponding to three different levels of expertise.

- The *designer* can define or import/export schemas for available models and perform translations over them.
- The *model engineer*, a more sophisticated user, can define new models by using the available metaconstructs.
- The *metamodel engineer* can add new metaconstructs to the metamodel and define translation rules for them; in this way she can extend the set of models handled by the system. This is clearly an even more expert user.

All of the above activities can be done without accessing the tool's source code. The definition of a model or of a schema involves populating tables of the dictionary, whereas the definition of translations involves inserting elements in the rule repository.

Let us present the tool by showing first how models and schemas can be defined and then how translations are performed.

We start with the typical activity of the model engineer, the definition of a model. This is done by "creating" a new model, giving it a name, and then specifying its constructs. This latter activity is the interesting one, and it is done (interactively) in two main steps: (i) choosing a metaconstruct from a pop-up menu and giving it a name within the model, and (ii) adding the desired properties available for the chosen metaconstruct. For instance, suppose the user is creating a simplified version of the ER model, called ERSimple. The model engineer, in order to define the first construct, will probably specify that he wants to add a construct corresponding to the Abstract metaconstruct (see Fig. 21) and call it "Entity." Then, he will define AttributeOfEntity, with reference to Lexical, and so on. During the definition process, the model engineer can add, remove, and alter constructs and construct properties. When the model is complete, the user requests a finalization, during which the system creates the corresponding dictionary structure. Once a model has been defined, the user can also save the description of the model as an XML file, with a specific format. In this way, it is possible to build a repository of models that can be easily imported when the tool is initialized. Figure 22 shows a portion of an XML file that contains the description of a version of the ER model.

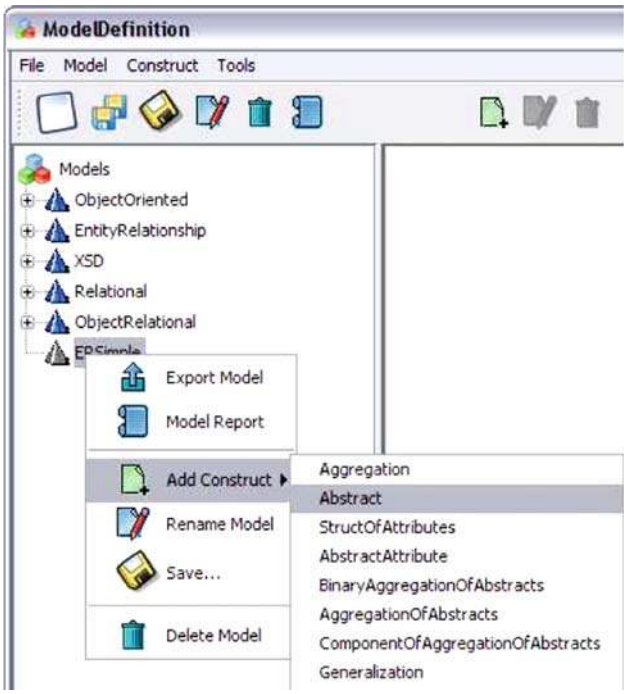


Fig. 21 Creation of a new construct in a model

```
<?xml version="1.0" encoding="UTF-8" ?>
- <model shortName="ERS" name="ERSimple">
  - <constructs>
    - <construct name="Entity" signature="true" bonds="true">
        <metaConstruct name="Abstract" />
        <properties />
        <references />
      </construct>
    - <construct name="AttributeOfEntity" signature="true" bonds="true">
        <metaConstruct name="Lexical" />
      - <properties>
        - <property name="isKey">
            <metaProperty name="isIdentifier" />
          </property>
        + <property name="isOptional">
        + <property name="type">
        </properties>
      - <references>
        - <reference to="Entity" name="abstractOID">
            <metaReference name="abstractOID" />
          </reference>
        </references>
      </construct>
    + <construct name="Relationship" signature="true" bonds="true">
    </constructs>
</model>
```

Fig. 22 The XML file for describing models

Let us now turn our attention to the other main class of users, designers, who define schemas and request their translations. Designers may build schemas through an interactive interface. After choosing the model, they can define the various elements, one at the time, by choosing a construct of the model and then giving a name and the associated properties and references, if needed. For example, the user can define Employee, corresponding to the Entity construct, and add some attributes (AttributeOfEntity) to it, such as SSN

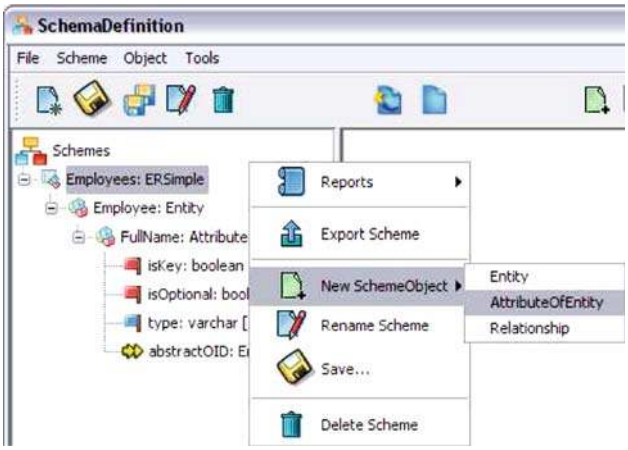


Fig. 23 Creation of a new AttributeOfEntity

Fig. 24 Specification of name, properties and reference for a new AttributeOfEntity

and FullName. The two steps for creating SSN are shown in Figs. 23 (choice of AttributeOfEntity among the constructs) and 24 (specification of the details).

The interactive definition has been useful for testing elementary steps (or for changes to existing schemas), but it would not be effective in practical settings. Therefore, as a major option, we have developed an import (and export) module. It relies on a persistence manager for the supermodel's constructs. Data are handled in an object representation, where each construct is represented by a class. Then, according to the external system of interest, the persistence manager interacts with specific components. We have developed two main import–export components, one for IBM DB2, as a representative of relational and object-relational systems (with also most of the object-oriented features of interest for data models) and the other for XML documents with schemas (according to a reasonable subset of XSD). The development of additional modules would mainly require attention to the specific syntax.

As a support to schema definition and management, the tool offers features for the visualization of schemas, and we will see them in the next subsection, while discussing example translations.

After the schema definition phase, the user has to choose how to translate the schema. In the first version of the tool, the designer had to build complex translations by manually composing basic ones out of a list of the available ones. The system applies them in sequence, generating an intermediate schema for each step, which can be used for documentation and verification. In the second version of the tool, the designer just has to specify the source schema (and so its model) and the target model, and the system finds a translation, on the basis of the algorithm illustrated in Sect. 5. Figure 25 shows a screenshot of the interface used to request an automatic translation. Let us observe that in the upper part there are the description of the input schema (left) and that of the target model (right) and in the lower part the sequence of translations found by the system.

Finally, let us consider the most sophisticated user, the metamodel engineer. She can define new basic transformations by writing Datalog rules or reusing some of the existing ones. The most important task is the definition and management of the supermodel. This is a very delicate task and requires a good knowledge of data models as well as of the supermodel itself. Because of the nature of the supermodel, such tasks are quite rare: after a transitory phase where metaconstructs are introduced into the supermodel, translations and Skolem functions involving the new metaconstructs are created, modifications should tend to zero and reuse should be total.

Translation rules are stored in text files. Any text editor can be used to write Datalog rules first and the basic translation then. A basic translation is a list of file names containing Datalog rules. The application of a basic translation means to apply all its Datalog rules in the same order as they occur in the basic translation. A tool to support the user in the definition of translation rules has been developed. It supports Datalog syntax highlighting and auto-completion of literals, Skolem functions and variables used in the rule. These help functions are possible by leveraging on the metadata associated with the constructs and the Skolem functions stored in the repository. There is also a support to the management

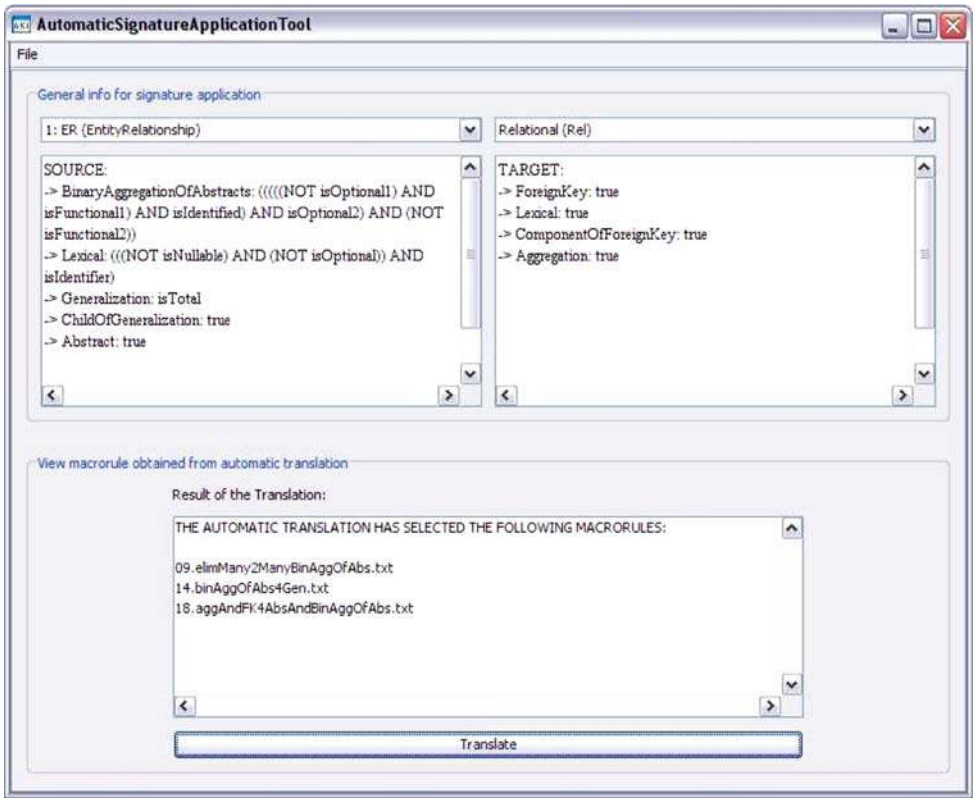


Fig. 25 Request for automatic translation

of basic translations, allowing the user to add or remove a Datalog rule, or change the order in which the Datalog rules occur in the basic translation.

A Datalog translation rule uses Skolem functions. The tool provides a search feature to look up already defined Skolem functions for a specific construct, and allows the creation of new ones by selecting the target construct, giving the function a name and adding a number of parameters. Once a Skolem function is defined, its description is stored as metadata in the dictionary.

We can say few words about the performance of the translation process, although it is not the focus of this work. First of all, it is worth mentioning that we decided to implement our own Datalog engine, because of the need for the OID-invention feature and for the ease of integration with our relational dictionary. The algorithm that generates SQL statements from Datalog rules performs well. It generally takes seconds and it has a linear cost in the size of the input (number of rules). Performance of the translation executions depends on the number of SQL statements (number of rules) to be executed and on the number of join conditions each rule implies. Moreover, the structure of the dictionary and the materialization of Skolem functions do not help performance. However, even if efficiency can be improved, the translation of schemas is performed in a few seconds.

6.2 Experiments

In this section, we discuss the experiments we made with our MIDST tool: we explain the methodology used to test the tool and then illustrate in some detail a few actual examples, which have been also demonstrated recently [2].

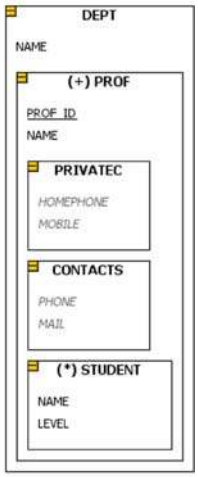
To test all the features of the tool we mainly used synthetic schemas and databases, in order to be cost-effective in the analysis of the various features of models and schemas.

We have tested the tool using two different points of views, one "in-the-small" and the other "in-the-large." For the testing in-the-small, we performed two sets of experiments. The first set was driven by the rules: we tested every single Datalog rule of each basic translation, to verify the correctness of the individual substeps. The second set was driven by model features: we defined many (a few hundred) ad hoc schemas, each one with a specific pattern of constructs, in order to verify that such a pattern is handled as required. In this way, we have verified the correctness of basic translations, which is a requirement of our approach, as we discussed in Sect. 5.

For the testing in-the-large, we used a set of more complex schemas. We considered some significant models, representatives of the various families (the progenitor and two restricted ones for each family), and defined one schema for each of them, with all the features of such a model. Then, we

Fig. 26 An XSD file and its representation in MIDST

```
- <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <xsd:element name="SCHOOL">
    - <xsd:complexType>
      - <xsd:sequence>
        - <xsd:element name="DEPT" maxOccurs="unbounded">
          - <xsd:complexType>
            - <xsd:sequence>
                <xsd:element name="NAME" nillable="false" type="xsd:string" />
              - <xsd:element name="PROF" nillable="false" minOccurs="1" maxOccurs="unbounded">
                - <xsd:complexType>
                  - <xsd:sequence>
                      <xsd:element name="PROF_ID" nillable="false" type="xsd:string" />
                      <xsd:element name="NAME" nillable="false" type="xsd:string" />
                    - <xsd:element name="CONTACTS" nillable="false">
                      - <xsd:complexType>
                        - <xsd:sequence>
                            <xsd:element name="PHONE" nillable="true" type="xsd:string" />
                            <xsd:element name="MAIL" nillable="true" type="xsd:string" />
                          </xsd:sequence>
                        </xsd:complexType>
                      </xsd:element>
                    + <xsd:element name="PRIVATEC" nillable="false">
                    + <xsd:element name="STUDENT" nillable="false" minOccurs="0" maxOccurs="unbounded">
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    - <xsd:key name="PROF_KEY">
        <xsd:selector xpath="./DEPT/PROF" />
        <xsd:field xpath="PROF_ID" />
      </xsd:key>
    </xsd:element>
  </xsd:schema>
```



translated them into other models of interest. In this case, the translation process for these schemas required the application of a number (from three to eight) of basic translations in the set we mentioned in Sect. 4 (and listed in Appendix 1). Here, we initially built complex translations by manually composing basic ones (as this was the only way in the first version of our tool) and then experimented with the automatic generation, and obtained the same sequences of basic translations. In some cases, when there are various acceptable sequences, the tool generated one of them.

Let us illustrate in detail some complete transformation examples. As we discussed in Sect. 5, our translations are in general composed of (i) reductions within the family of the source model, (ii) a translation to the family of the target model, and (iii) reductions within the target family. In most cases the final portion is not needed, as reductions occur mainly in the source family. Therefore, we comment on a few examples with just two phases and then a final one that has the third phase.

As a first case, let us consider the translation from a binary ER model with all our features, to an object-oriented model with generalizations. In this case, the following reductions would be needed

- eliminate attributes of relationships, possibly introducing new entities and relationships (basic translation 7 in the appendix);
- eliminate many-to-many relationships, introducing additional entities and one-to-many relationships (translation 8)

The schema obtained in this way can be directly translated into the object-oriented model, by means of translation 17. Therefore, the complex translation would be composed of the sequence ⟨7, 8, 17⟩.

If instead the target model were the relational one, then we would need the same reduction steps as above, plus a step to eliminate generalizations before the final translation. There are in fact three different translations to perform this task (12, 13, and 15), according to the various ways of eliminating generalizations [9, pp. 283–287]. In the manual approach, the designer would choose the preferred one; in the automatic one there would be one picked by the tool, with the possibility for the designer to replace it. Then a translation into the relational model would be possible by means of translation 19. A possible sequence in this case would therefore be ⟨7, 8, 12, 19⟩. Our tool also provides a customization feature, where the designer can specify the selective application of rules. In this case, with a two-level generalization, for example Person, Staff, and Professor, we could specify that the first level is replaced with a relationship (translation 15), so the entities Person and Staff are kept, whereas the second level is replaced by merging the child entity into the parent one (translation 13), and so the entity Professor disappears, as its attributes are moved to Staff. In this way, the final relational schema would contain two tables, Person and Staff.

As a more complex example, we show, with also some screenshots, the translation from an XML Schema Description (XSD) to a relational schema. The input file and its graphical visualization via the tool after importing it are shown in Fig. 26. The structure is nested at two levels: each department has a name and one or more professors and, for each professor, we have some contact information and zero or more students. According to our supermodel, this schema is represented by means of an Abstract (Dept), a first level multivalued StructureOfAttributes (Prof), various Lexicals

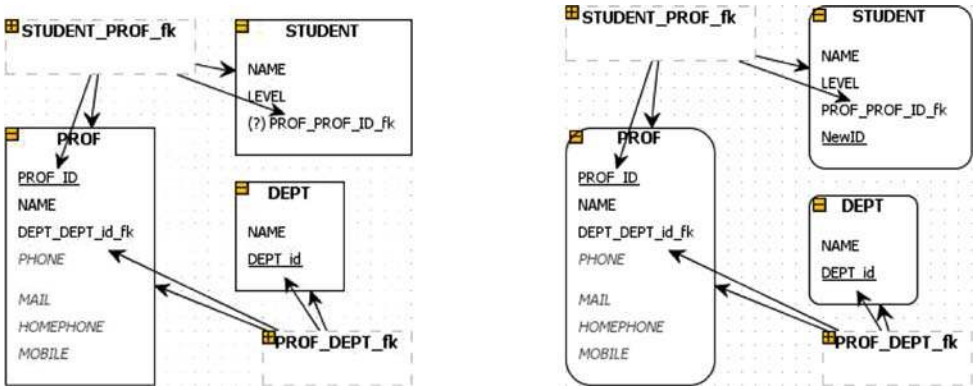


Fig. 27 An intermediate and the final result of the translation from XSD to a relational schema

(including Name of Dept, ProfID, Name of Prof) and some nested StructureOfAttributes (PrivateContacts, Contacts and Students; the latter is multivalued and the other two monovalued). The translation requires the unnesting of sets and structures, introducing new first level elements and foreign keys and flattening the attributes of the structures, respectively (basic translations 1 and 2 in the list). The result of this sequence of steps still belongs to the XSD family and it is depicted in Fig. 27, on the left. Finally we could use the translation from the XSD family to the relational family (translation 20, which replaces elements and their attributes with tables and columns). The final result of the translation is shown in Fig. 27, on the right (in our tool, and so in the figures, boxes with square corners denote abstracts, boxes with rounded corners denote aggregations of lexicals and boxes with dashed lines denote foreign keys).

As a final example, we mention a case where all the three phases are needed. Assume that our set contains basic translation 15, as the only means to eliminate generalizations (so, it does not contain translations 12, 13, 14 and 16). As translation 15 introduces binary aggregations, it cannot be used within the object-oriented family. In this case, if we want to go from an object-oriented model with structured attributes, to an ER model without generalizations, we need first a reduction within the object-oriented family to flatten attributes (translation 2), then we can apply the translation to the ER family (18) and finally the reduction within the ER model to eliminate generalizations (15).

7 Related work

This paper is an extended version of a portion of a conference paper [1] and in some sense also of another, older conference paper [6] (which has never had a journal version). With respect to those papers, Sects. 5 and 6.1 are completely new, and most of the other discussions are richer, in particular the one in Sect. 2 on the space of models, which gives an original perspective on the approach. The conference paper [1] also includes an initial proposal for handling data translation, which is not covered here. The tool presented here in Sect. 6.1 has also been demonstrated [2].

Various proposals exist that consider schema and data translation. However, most of them only consider specific data models. We comment here on related pieces of work that address the problem of model-independent translations.

The term *ModelGen* was coined in [11] which, along with [12], argues for the development of model management systems consisting of generic operators for solving many problems involving metadata and schemas. An example of using ModelGen to help solve a schema evolution problem appears in [11].

An early approach to ModelGen (even before the term was coined) was MDM, proposed by Atzeni and Torlone [6]. The basic idea behind MDM and the similar approaches [14,18,19,37] is useful but offers only a partial solution to our problem. In addition, their representation of the models and transformations is hidden within the tool's imperative source code, not exposed as more declarative, user-comprehensible rules. This leads to several other difficulties. First, only the designers of the tool can extend the models and define the transformations. Thus, instance level transformations would have to be recoded in a similar way. Moreover, correctness of the rules has to be accepted by users as a dogma, since their only expression is in complex imperative code. And any customization would require changes in the tool's source code. All of these problems are overcome by our approach.

There are two concurrent projects to develop ModelGen. The approach of Papotti and Torlone [33] is not rule-based. Rather, their transformations are imperative programs, with the weaknesses described above. Their translation is done by translating the source data into XML, performing an XML-to-XML translation expressed in XQuery to reshape it to be compatible with the target schema, and then translating the XML into the target model. This is similar to our use of a relational database as the "pivot" between the source and target databases.

The approach of Bernstein, Melnik, and Mork [13,31] is rule-based, like ours. However, unlike ours, it is not driven

by a relational dictionary of schemas, models and translation rules. Instead, they focus on flexible mapping of inheritance hierarchies and the incremental regeneration of mappings after the source schema is modified. They also propose view generation and so instance translation.

Bowers and Delcambre [16] present Uni-Level Description (UDL) as a metamodel in which models and translations can be described and managed, with a uniform treatment of models, schemas, and instances. They use it to express specific model-to-model translations of both schemas and instances. Like our approach, their rules are expressed in Datalog. Unlike ours, they are expressed for particular pairs of models.

Other approaches to schema translation based on some form of metamodel, thus sharing features with ours, were proposed by Hainaut [22,23] and Boyd, Poulovassilis and McBrien [17,27,35].

Data exchange is a different but related problem, the development of user-defined custom translations from a given source schema to a given target one, not the automated translation of a source schema to a target model. It is an old database problem, going back at least to the 1970s [36]. Some recent approaches are in Cluet et al. [20], Milo and Zohar [30], and Popa et al. [34].

8 Conclusions

In this paper, we showed MIDST, an implementation of the ModelGen operator that supports model-generic translation of schemas. The experiments we conducted confirmed that translations can be effectively performed with our approach. The main contributions are (i) the visible dictionary, (ii) the specification of rules in Datalog, which makes the specification of translations independent of the engine that executes them, and (iii) the techniques to generate translations out of their specification.

Current work concerns the customization of translation, data level translations and applications of the technique to typical model management scenarios, such as schema evolution and round-trip engineering [11].

Acknowledgments We would like to thank Luigi Bellomarini, Francesca Bugiotti, Fabrizio Celli, Giordano Da Lozzo, Riccardo Pietrucci, Leonardo Puleggi and Luca Santarelli, for their work in the development of the tool and Chiara Russo for contributing to the experimentation and for many helpful discussions.

Appendix 1. Basic translations and their completeness

This appendix lists the set of basic translations used in our experiments, for the supermodel illustrated in Sect. 3.3 (and specifically in Fig. 17).

1. Eliminate multivalued structures of attributes in an abstract, by introducing new abstracts and foreign keys.
2. Eliminate (nested, monovalued) structures of attributes in an abstract, by flattening them.
3. Eliminate (nested, monovalued) structures of attributes in an aggregation, by flattening them.
4. Eliminate foreign keys involving abstracts, by introducing abstract attributes.
5. Eliminate abstract attributes, replacing them with foreign keys involving abstracts.
6. Eliminate lexicals of aggregations of abstracts, by moving them to abstracts (possibly new).
7. Eliminate lexicals of binary aggregations of abstracts, by moving them to abstracts (possibly new).
8. Eliminate many-to-many binary aggregations, by introducing new abstracts and binary aggregations of them.
9. Eliminate n-ary aggregations of abstracts, by introducing new abstracts and binary aggregations of them.
10. Replace binary aggregations of abstracts with aggregations of them.
11. Nest abstracts and abstract attributes within referencing abstracts.
12. Eliminate generalizations, by keeping the leaf abstracts and merging the other abstracts into them.
13. Eliminate generalizations, by keeping the root abstracts and merging the other abstracts into them.
14. Eliminate generalizations, by keeping all abstracts and relating them by means of (n-ary) aggregations (that involve two abstracts).
15. Eliminate generalizations, by keeping all abstracts and relating them by means of binary aggregations.
16. Eliminate generalizations, by keeping all abstracts and relating them by means of abstract attributes.
17. Replace (one-to-many) binary aggregations of abstracts with abstract attributes.
18. Replace abstract attributes with (one-to-many) binary aggregations of abstracts.
19. Replace abstracts and binary (one-to-many) aggregations of them with aggregations (of lexicals) and foreign keys.
20. Replace abstracts and abstract attributes with aggregations (of lexicals) and foreign keys.
21. Replace aggregations (of lexicals) and foreign keys with abstracts and binary aggregations of them.
22. Replace aggregations (of lexicals) and foreign keys with abstracts and foreign keys over them.
23. Replace aggregations (of lexicals) and foreign keys with abstracts and abstract attributes.

We briefly comment on the completeness of this set of rules with respect to the models used in our experiments, described by the table in Fig. 17. We need to show that Assumptions 1 and 2 are satisfied. Let us begin with Assumption 2, so we describe the minimal models. Here, it suffices

	ER	B-ER	Obj	OR	Rel	XSD
ER	-	9	9,17	9,17	9,19	9,17,11
B-ER	10	-	17	17	19	17,11
Obj	18,10	18	-	-	20	5,11
OR	18,10	18	23	-	20	11
Rel	21,10	21	23	-	-	22
XSD	4,18,10	4,18	4	1	20	-

Fig. 28 Translations between families

to list the minimal models in each family and the sequences of reductions that form a translation from the progenitor of the family to them, as follows:

- (n-ary) Entity-Relationship: we have a minimal model with no generalizations and no attributes (lexicals) on aggregations; therefore, the reduction is composed of 6 and 14 (or 12 or 13).
- Binary Entity-Relationship: one minimal model again, with no generalizations and no lexicals on binary aggregations and no many-to-many aggregations; the reduction is 7, 15 (or 12 or 13), 8.
- Object: one minimal model, with no generalizations and no structures of attributes: the reduction is 2, 16 (or 12 or 13).
- Object-Relational: here there are three minimal models, the progenitors of the relational model (with a reduction 2, 3, 16 (or 12 or 13), 20) and of the object model (4, 23) and a model with no structures of attributes in abstracts and no generalizations, for which the reduction is 2, 16 (or 12 or 13).
- Relational: here, for the sake of simplicity, we have handled just one model, so the reduction is trivial.
- XSD: one minimal model, where structures of attributes are only monovalued; the reduction is just translation 1.

With respect to Assumption 1, we need to show that for each pair of families we have a translation from one to the other, and viceversa. This is shown in the table in Fig. 28, where each cell indicates the translation or the sequence of translations needed to go from the model associated with the row to the model associated with the column. It is worth noting that, in some cases, the cell contains two or even three basic translations; then, with reference to the discussion we made in Sect. 5 after Assumption 1, we can think that the system has a composition of these translations defined as a basic one.

References

1. Atzeni, P., Cappellari, P., Bernstein, P.A.: Model-independent schema and data translation. In: EDBT Conference, LNCS, vol. 3896, pp. 368–385. Springer, Berlin (2006)
2. Atzeni, P., Cappellari, P., Gianforme, G.: MIDST: model independent schema and data translation. In: SIGMOD Conference, pp. 1134–1136. ACM, New York (2007)
3. Atzeni, P., Del Nostro, P.: Management of heterogeneity in the Semantic Web. In: ICDE Workshops, p. 60. IEEE Computer Society (2006)
4. Atzeni, P., Gianforme, G., Cappellari, P.: Reasoning on data models in schema translation. In: FOIKS Symposium, LNCS, vol. 4932, pp. 158–177. Springer, Berlin (2008)
5. Atzeni, P., Torlone, R.: A metamodel approach for the management of multiple models and translation of schemes. Inf. Syst. **18**(6), 349–362 (1993)
6. Atzeni, P., Torlone, R.: Management of multiple models in an extensible database design tool. In: EDBT Conference, LNCS, vol. 1057, pp. 79–95. Springer, Berlin (1996)
7. Barbosa, D., Freire, J., Mendelzon, A.O.: Information preservation in XML-to-relational mappings. In: XSym Workshop, LNCS, vol. 3186, pp. 66–81. Springer (2004)
8. Barbosa, D., Freire, J., Mendelzon, A.O.: Designing information-preserving mapping schemes for XML. In: VLDB, pp. 109–120 (2005)
9. Batini, C., Ceri, S., Navathe, S.: Database Design with the Entity-Relationship Model. Benjamin and Cummings Publ. Co., Menlo Park, CA (1992)
10. Batini, C., Lenzerini, M.: A methodology for data schema integration in the entity relationship model. IEEE Trans. Software Eng. **10**(6), 650–664 (1984)
11. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR Conference, pp. 209–220 (2003)
12. Bernstein, P.A., Halevy, A.Y., Pottinger, R.: A vision of management of complex models. SIGMOD Record **29**(4), 55–63 (2000)
13. Bernstein, P.A., Melnik, S., Mork, P.: Interactive schema translation with instance-level mappings. In: VLDB, pp. 1283–1286 (2005)
14. Bézivin, J., Breton, E., Dupé, G., Valduriez, P.: The ATL transformation-based model management framework. Research Report 03.08, IRIN, Université de Nantes (2003)
15. Bohannon, P., Fan, W., Flaster, M., Narayan, P.P.S.: Information preserving XML schema embedding. In: VLDB, pp. 85–96 (2005)
16. Bowers, S., Delcambre, L.M.L.: The Uni-level description: a uniform framework for representing information in multiple data models. In: ER Conference, LNCS, vol. 2813, pp. 45–58. Springer, Berlin (2003)
17. Boyd, M., McBrien, P.: Comparing and transforming between data models via an intermediate hypergraph data model. J. Data Semantics IV pp. 69–109 (2005)
18. Claypool, K.T., Rundensteiner, E.A.: Sangam: A transformation modeling framework. In: DASFAA Conference, pp. 47–54 (2003)
19. Claypool, K.T., Rundensteiner, E.A., Zhang, X., Su, H., Kuno, H.A., Lee, W.C., Mitchell, G.: Sangam—a solution to support multiple data models, their mappings and maintenance. In: SIGMOD Conference, p. 606 (2001)
20. Cluet, S., Delobel, C., Siméon, J., Smaga, K.: Your mediators need data conversion! In: SIGMOD Conference, pp. 177–188 (1998)
21. De Virgilio, R., Torlone, R.: Modeling heterogeneous context information in adaptive Web based applications. In: ICWE Conference, pp. 56–63. ACM, New York (2006)
22. Hainaut, J.L.: Specification preservation in schema transformations—application to semantics and statistics. Data Knowl. Eng. **19**(2), 99–134 (1996)
23. Hainaut, J.L.: The transformational approach to database engineering. In: GTTSE, LNCS. vol. 4143, pp. 95–143. Springer, Berlin (2006)
24. Hull, R.: Relative information capacity of simple relational schemata. SIAM J. Comput. **15**(3), 856–886 (1986)

25. Hull, R.: Managing semantic heterogeneity in databases: a theoretical perspective. In: PODS Symposium, pp. 51–61. ACM, New York (1997)
26. Hull, R., King, R.: Semantic database modelling: survey, applications and research issues. ACM Comput. Surv. **19**(3), 201–260 (1987)
27. McBrien, P., Poulovassilis, A.: A uniform approach to inter-model transformations. In: CAiSE Conference, LNCS, vol. 1626, pp. 333–348 (1999)
28. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: The use of information capacity in schema integration and translation. In: VLDB, pp. 120–133 (1993)
29. Miller, R.J., Ioannidis, Y.E., Ramakrishnan, R.: Schema equivalence in heterogeneous systems: bridging theory and practice. Inf. Syst. **19**(1), 3–31 (1994)
30. Milo, T., Zohar, S.: Using schema matching to simplify heterogeneous data translation. In: VLDB, pp. 122–133 (1998)
31. Mork, P., Bernstein, P.A., Melnik, S.: Teaching a schema translator to produce O/R views. In: ER Conference, LNCS, vol. 4801, pp. 102–119. Springer, Berlin (2007)
32. Paolozzi, S., Atzeni, P.: Interoperability for semantic annotations. In: DEXA Workshops, pp. 445–449. IEEE Computer Society (2007)
33. Papotti, P., Torlone, R.: Heterogeneous data translation through XML conversion. J. Web Eng. **4**(3), 189–204 (2005)
34. Popa, L., Velegrakis, Y., Miller, R.J., Hernández, M.A., Fagin, R.: Translating Web data. In: VLDB, pp. 598–609 (2002)
35. Poulovassilis, A., McBrien, P.: A general formal framework for schema transformation. Data Knowl. Eng. **28**(1), 47–71 (1998)
36. Shu, N.C., Housel, B.C., Taylor, R.W., Ghosh, S.P., Lum, V.Y.: Express: a data extraction, processing, amd restructuring system. ACM Trans. Database Syst. **2**(2), 134–174 (1977)
37. Song, G., Zhang, K., Wong, R.: Model management though graph transformations. In: IEEE Symposium on Visual Languages and Human Centric Computing, pp. 75–82 (2004)
38. Ullman, J.D., Widom, J.: A First Course in Database Systems. Prentice-Hall, Englewood Cliffs, NJ (1997)