A universal metamodel and its dictionary

Paolo Atzeni(1), Giorgio Gianforme(2), Paolo Cappellari(3)

(1) Università Roma Tre, Italy — atzeni@dia.uniroma3.it
(2) Università Roma Tre, Italy — giorgio.gianforme@gmail.com
(3) University of Alberta, Canada — paolo.cappellari@gmail.com

Abstract. We discuss a universal metamodel aimed at the representation of schemas in a way that is at the same time model-independent (in the sense that it allows for a uniform representation of different data models) and model-aware (in the sense that it is possible to say to whether a schema is allowed for a data model). This metamodel can be the basis for the definition of a complete model-management system. Here we illustrate the details of the metamodel and the structure of a dictionary for its representation. Exemplifications of a concrete use of the dictionary are provided, by means of the representations of the main data models, such as relational, object-relational or XSD-based. Moreover, we demonstrate how set operators can be redefined with respect to our dictionary and easily applied on it. Finally, we show how such a dictionary can be exploited to automatically produce detailed descriptions of schema and data models, in a textual (i.e. XML) or visual (i.e. UML class diagram) way.

1 Introduction

Metadata is descriptive information about data and applications. Metadata is used to specify how data is represented, stored, and transformed, or may describe interfaces and behavior of software components.

The use of metadata for data processing was reported as early as fifty years ago [22]. Since then, metadata-related tasks and applications have become truly pervasive and metadata management plays a major role in today's information systems. In fact, the majority of information system problems involve the design, integration, and maintenance of complex application artifacts, such as application programs, databases, web sites, workflow scripts, object diagrams, and user interfaces. These artifacts are represented by means of formal descriptions, called schemas or models, and, consequently, metadata. Indeed, to solve these problems we have to deal with metadata, but it is well known that applications solving metadata manipulation are complex and hard to build, because of heterogeneity and impedance mismatch. Heterogeneity arises because data sources are independently developed by different people and for different purposes and subsequently need to be integrated. The data sources may use different data models, different schemas, and different value encodings. Impedance mismatch arises because the logical schemas required by applications are different from the physical ones exposed by data sources. The manipulation includes designing mappings (which describe how two schemas are related to each other) between the schemas, generating a schema from another schema along with a mapping between them, modifying a schema or mapping, interpreting a mapping, and generating code from a mapping.

In the past, these difficulties have always been tackled in practical settings by means of ad-hoc solutions, for example by writing a program for each specific application. This is clearly very expensive, as it is laborious and hard to maintain. In order to simplify such manipulation, Bernstein et al. [11, 12, 23] proposed the idea of a *model management system*. Its goal is to factor out the similarities of the metadata problems studied in the literature and develop a set of high-level operators that can be utilized in various scenarios. Within such a system, we can treat schemas and mappings as abstractions that can be manipulated by operators that are meant to be generic in the sense that a single implementation of them is applicable to all of the data models. Incidentally, let us remark that in this paper we use the terms “schema” and “data model” as common in the database literature, though some model-management literature follows a different terminology (and uses “model” instead of “schema” and “metamodel” instead of “data model”).

The availability of a uniform and generic description of data models is a prerequisite for designing a model management system. In this paper we discuss a “universal metamodel” (called the *supermodel*), defined by means of metadata and designed to properly represent “any” possible data model, together with the structure of a dictionary for storing such metadata.

There are many proposals for dictionary structure in the literature. The use of dictionaries to handle metadata has been popular since the early database systems of the 1970’s, initially in systems that were external to those handling the database (see Allen et al. [1] for an early survey). With the advent of relational systems in the 1980’s, it became possible to have dictionaries be part of the database itself, within the same model. Today, all DBMSs have such a component. Extensive discussion was also carried out in even more general frameworks, with proposals for various kinds of dictionaries, describing various features of systems (see for example [9, 19, 21]) within the context of industrial CASE tools and research proposals. More recently, a number of metadata repositories have been developed [26]. They generally use relational databases for handling the information of interest. There are other significant recent efforts towards the description of multiple models, including the Model Driven Architecture (MDA) and, within it, the Common Warehouse Metamodel (CWM) [27], and Microsoft Repository [10]; in contrast to our approach, these do not distinguish metalevels, as the various models of interest are all specializations of a most general one, UML based.

The description of models in terms of the (meta-)constructs of a metamodel was proposed by Atzeni and Torlone [8]. But it used a sophisticated graph language, which was hard to implement. The other papers that followed the same or similar approaches [14–16, 28] also used specific structures.

We know of no literature that describes a dictionary that exposes schemas in both model-specific and model-independent ways, together with a description of models. Only portions of similar dictionaries have been proposed. None of them offer the rich interrelated structure we have here.

The contributions of this paper and its organization are the following. In Section 2 we briefly recall the metamodel approach we follow (based on the initial idea by Atzeni and Torlone [8]). In Section 3 we illustrate the organization of the dictionary we use to store our schemas and models, refining the presentation given in a previous conference paper (Atzeni et al. [3]). In Section 4 we illustrate a specific supermodel used to generalize a large set of models, some of which are also commented upon. Then, in Section 5 we discuss how some interesting operations on schemas can be specified and implemented on the basis of our approach. Section 6 is devoted to the illustration of generic reporting and visualization tools built out of the principles and structure of our dictionary. Finally, in Section 7 we summarize our results.

2 Towards a universal metamodel

In this section we summarize the overall approach towards a model-independent and model-aware representation of data models, based on an initial idea by Atzeni and Torlone [8].

The first step toward a uniform solution is the adoption of a general model to properly represent many different data models (e.g. entity-relationship, object-oriented, relational, object-relational, XML). The proposed general model is based on the idea of construct: a construct represents a "structural" concept of a data model. We find out a construct for each "structural" concept of every considered data model and, hence, a data model is completely represented by the set of its constructs. 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. "Structural" concepts for these data models are, for example, entity, attribute of entity, and binary relationship for the ER and class, field, and reference for the OO. Moreover, constructs have a name, may have properties and are related to one another.

A UML class diagram of this construct-based representation of a simple ER model with entities, attributes of entities and binary relationships is depicted in Figure 1. Construct *Entity* has no attribute and no reference; construct *AttributeOfEntity* has a boolean property to specify whether an attribute is part of the identifier of the entity it belongs to and a property type to specify the data type of the attribute itself; construct *BinaryRelationship* has two references toward the entities involved in the relationship and several properties to specify role, minimum and maximum cardinalities of the involved entities, and whether the first entity is externally identified by the relationship itself.

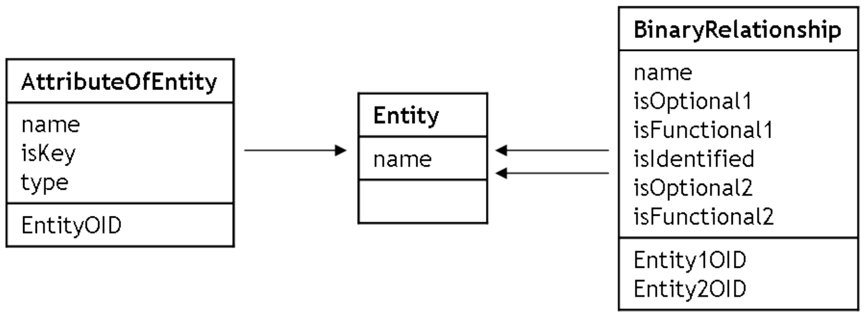


Fig. 1. A simple entity-relationship model

With similar considerations about a simple OO model with classes, simple fields (i.e. with standard type) and reference fields (i.e. a reference from a class to another) we obtain the UML class diagram of Figure 2. Construct *Class* has no attribute and no reference; construct *Field* is similar to *AttributeOfEntity* but it does not have boolean attributes, assuming that we do not want to manage explicit identifiers of objects; construct *ReferenceField* has two references toward the class owner of the reference and the class pointed by the reference itself.

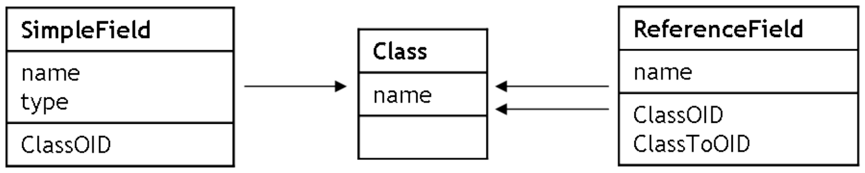


Fig. 2. A simple object-oriented model

In this way, we have uniform representations of models (in terms of constructs) but these representations are not general. This is unfeasible as the number of (variants of) models grows because it implies a corresponding rise in the number of constructs. To overcome this limit, we exploit an observation of Hull and King [20], drawn on later by Atzeni and Torlone [7]: most known models have constructs that can be classified according to a rather small set of generic (i.e. model independent) *metaconstructs*: lexical, abstract, aggregation, generalization, and function. Recalling our example, entities and classes play the same role (or, in other terms, "they have the same meaning"), and so we can define a generic metaconstruct, called *Abstract*, to represent both these concepts; the same happens for attributes of entities and of relationships and fields of classes, representable by means of a metaconstructs called *Lexical*. Conversely, relationships and references do not have the same meaning and hence one metaconstruct is not enough to properly represent both concepts (hence *BinaryAggregationOfAbstracts* and *AbstractAttribute* are both included).

Hence, each model is defined by its constructs and the metaconstructs they refer to. This representation is clearly at the same time model-independent (in the sense that it allows for a uniform representation of different data models) and model-aware (in the sense that it is possible to say to whether a schema is allowed for a data model). An even more important notion is that of *supermodel* (also called *universal metamodel* in the literature [13, 24]): 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 Figure 3: the supermodel portion is predefined, but can be extended (and we will present our recent extension later in this paper), 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.

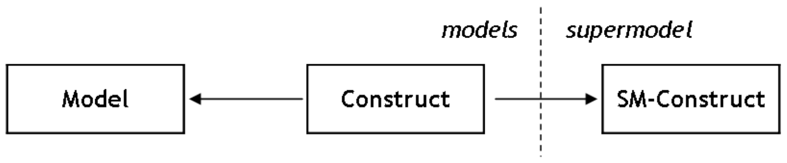


Fig. 3. A simplified conceptual view of models and constructs.

In this scenario, a schema for a certain model is a set of instances of constructs allowed in that model. Let us consider the simple ER schema depicted in Figure 4. Its construct-based representation would include two instances of *Entity*

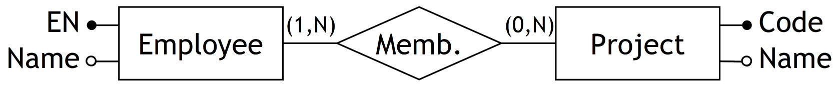


Fig. 4. A simple entity-relationship schema.

(i.e. Employee and Project), one instance of *BinaryRelationship* (i.e. Membership) and four instances of *AttributeOfEntity* (i.e. EN, Name, Code, and Name). The model-independent representation (i.e. based on metaconstructs) would include two instances of *Abstract*, one instance of *BinaryAggregationOfAbstracts* and four instances of *Lexical*. For each of these instances we have to specify values for its attributes and references, meaningful for the model. So for example, the instance of *Lexical* corresponding to EN would refer to the instance of *Abstract* of employee through its *abstractOID* reference and would have a 'true' value for its *isIdentifier* attribute. This example is illustrated in Figure 5, where we omit not relevant properties, represent references only by means of arrows, and represent links between constructs and their instances by means of dashed arrows. In the same way, we can state that a database for a certain schema is a set of instances of constructs of that schema.

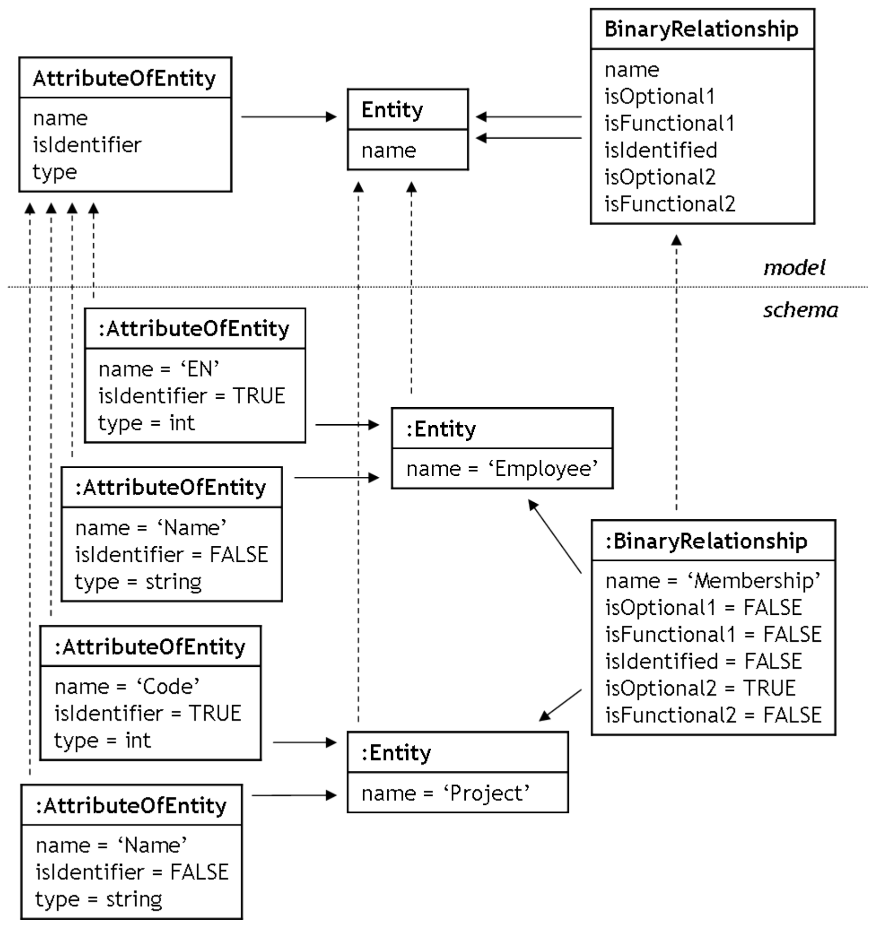


Fig. 5. A construct based representation of the schema of Figure 4.

As a second example, let us consider the simple OO schema depicted in Figure 6. Its construct-based representation would include two instances of *Class* (i.e. Employee and Department), one instance of *ReferenceField* (i.e. Membership) and five instances of *Field* (i.e. EmpNo, Name, Salary, DeptNo, and Sept-

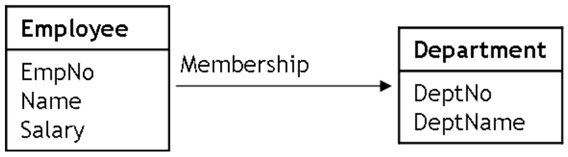


Fig. 6. A simple object-oriented schema.

Name). Alternatively, the model independent representation (i.e. based on meta-constructs) would include two instances of *Abstract*, one instance of *AbstractAttribute* and five instances of *Lexical*.

On the other side, it is possible to use the same approach based on "concepts of interest" in order to obtain a high-level description of the supermodel (i.e. of the whole set of metaconstructs). From this point of view the concepts of interest are three: construct, construct property and construct reference. In this way we have a full description of the supermodel, with constructs, properties and references, as follows. Each construct has a name and a boolean attribute (*isLexical*) that specifies whether its instances have actual, elementary values associated with (for example, this property would be true for *AttributeOfAbstract* and false for *Abstract*). Each property belongs to a construct and has a name and a type. Each reference relates two constructs and has a name. A UML class diagram of this representation is presented in Figure 7.

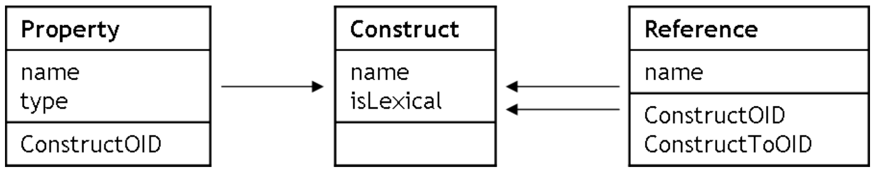


Fig. 7. A description of the supermodel

3 A multilevel dictionary

The conceptual approach to the description of models and schemas presented in Section 2, despite being very useful to introduce the approach, is not effective to actually store data and metadata. Therefore, we have developed a relational implementation of the idea, leading to a multilevel dictionary organized in four parts, which can be characterized along two coordinates: the first corresponding to whether they describe models or schemas and the second depending on whether they refer to specific models or to the supermodel. This is represented in Figure 8.

The various portions of the dictionary correspond to various UML class diagrams of Section 2. In the rest of this section, we comment on them in detail.

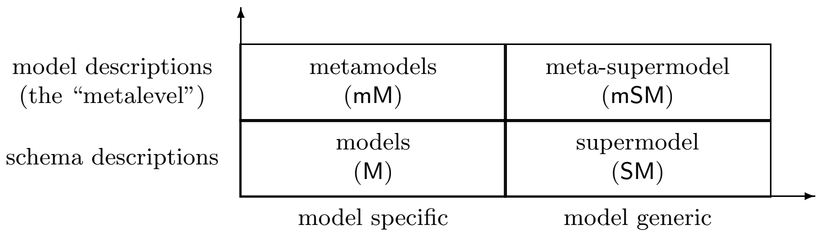


Fig. 8. The four parts of the dictionary

The meta-supermodel part of the dictionary describes the supermodel, that is, the set of constructs used for building schemas of various models. It is composed of three relations (whose names begin with MSM to recall that we are in the meta-supermodel portion), one for each "class" of the diagram of Figure 7. Every relation has one OID column and one column for each attribute and reference of the corresponding "class" of such a diagram. The relations of this part of the dictionary, with some of the data, are depicted in Figure 9. It is worth noting that these relations are rather small, because of the limited number of constructs in our supermodel.

MSM_Construct

OID	Name	IsLexical
mc1	Abstract	false
mc2	Lexical	true
mc3	Aggregation	false
mc4	BinaryAggregationOfAbstracts	false
mc5	AbstractAttribute	false
...	...	...

MSM_Property

OID	Name	Construct	Type
mp1	Name	mc1	string
mp2	Name	mc2	string
mp3	IsIdentifier	mc2	bool
mp4	IsOptional	mc2	bool
mp5	Type	mc2	string
...	...	...	...

MSM_Reference

OID	Name	Construct	ConstructTo
mr1	Abstract	mc2	mc1
mr2	Aggregation	mc2	mc3
mr3	Abstract1	mc4	mc1
mr4	Abstract2	mc4	mc1
mr5	Abstract	mc5	mc1
mr6	AbstractTo	mc5	mc1
...	...	...	...

Fig. 9. The mSM part of the dictionary

The metamodels part of the dictionary describes the individual models, that is, the set of specific constructs allowed in the various models, each one corresponding to a construct of the supermodel. It has the same structure as the meta-supermodel part with two differences: first, each relation has an extra column containing a reference towards the corresponding element of the supermodel (i.e. of the meta-supermodel part of the dictionary); second, there is an extra relation to store the names of the specific models and an extra column in the Construct relation referring to this extra relation. The relations of this part of the dictionary, with some of the data, are depicted in Figure 10.

MM_Model	
OID	Name
m1	ER
m2	OODB

MM_Construct				
OID	Name	Model	MSM-Constr.	IsLexical
co1	Entity	m1	mc1	false
co2	AttributeOfEntity	m1	mc2	true
co3	BinaryRelationship	m1	mc4	false
co4	Class	m2	mc1	false
co5	Field	m2	mc2	true
co6	ReferenceField	m2	mc5	false

MM_Property				
OID	Name	Constr.	Type	MSM-Prop.
pr1	Name	co1	string	mp1
pr2	Name	co2	string	mp2
pr3	IsKey	co2	bool	mp3
pr4	Name	co3	string	...
pr5	IsOpt.1	co3	bool	...
...	...	...	...	...
pr6	Name	co4	string	mp1
pr7	Name	co5	string	mp2
...	...	...	...	...

MM_Reference				
OID	Name	Constr.	Constr.To	MSM-Ref.
ref1	Entity	co2	co1	mr1
ref2	Entity	co3	co1	mr3
ref3	Entity	co3	co1	mr4
ref4	Class	co5	co4	mr1
ref5	Class	co6	co4	mr5
ref6	ClassTo	co6	co4	mr6

Fig. 10. The mM part of the dictionary

We refer to these first two parts as the "metalevel" of the dictionary, as it contains the description of the structure of the lower level, whose content describes schemas. The lower level is also composed of two parts, one referring to the supermodel constructs (therefore called the SM part) and the other to model-specific constructs (the M part). The structure of the schema level is, in our system, automatically generated out of the content of the metalevel: so, we can say that the dictionary is self-generating out of a small core. In detail, in the model part there is one relation for each row of MM_Construct relation. Hence each of these relations corresponds to a construct and has, besides an OID column, one column for each property and reference specified for that construct in relations MM_Property and MM_Reference, respectively. Moreover, there is a relation schema to store the name of the schemas stored in the dictionary and each relation has an extra column referring to it. Hence, in practice, there is a set of relations for each specific model, with one relation for each construct allowed in the model. This portion of the dictionary is depicted in Figure 11, where we show the data for the schemas of Figures 4 and 6.

Schema	
OID	Name
s1	ER Schema
s2	OO Schema

ER-Entity		
OID	Name	Schema
e1	Employee	s1
e2	Project	s1

ER-AttributeOfEntity					
OID	Entity	Name	Type	isKey	Schema
a1	e1	EN	int	true	s1
a2	e1	Name	string	false	s1
a3	e2	Code	int	true	s1
a4	e2	Name	string	false	s1

ER-BinaryRelationship							
OID	Name	IsOptional1	IsFunctional1	...	Entity1	Entity2	Schema
r1	Membership	false	false	...	e1	e2	s1

OO-Class		
OID	Name	Schema
cl1	Employee	s2
cl2	Department	s2

OO-Field				
OID	Class	Name	Type	Schema
f1	cl1	EmpNo	int	s2
f2	cl1	Name	string	s2
f3	cl1	Salary	int	s2
f4	cl2	DeptNo	int	s2
f5	cl2	DeptName	string	s2

OO-ReferenceField				
OID	Name	Class	ClassTo	Schema
ref1	Membership	cl1	cl2	s2

Fig. 11. The dictionary for schemas of specific models

Analogously, in the supermodel part there is one relation for each row of MSM_Construct relation; hence each one of these relations corresponds to a metaconstruct (or a construct of the supermodel) and has, besides an OID column, one column for each property and reference specified for that metaconstruct in relations MSM_Property and MSM_Reference, respectively. Again, there is a relation schema to store the name of the schemas stored in the dictionary and each relation has an extra column referring to it. Moreover, the Schema relation has an extra column referring to the specific model each schema belongs to. This portion of the dictionary is depicted in Figure 12, where we show the data for the schemas of Figures 4 and 6, and hence we show the same data presented in Figure 11. It is worth noting that Abstract contains the same data as ER-Entity and OO-Class taken together. Similarly, AttributeOfAbstract contains data in ER-AttributeOfEntity and OO-Field.

4 A significant supermodel with models of interest

As we said, our approach is fully extensible: it is possible to add new metaconstructs to represent new data models, as well as to refine and increase precision of actual representations of models. The supermodel we have mainly experimented

Schema		
OID	Name	Model
s1	ER Schema	m1
s2	OO Schema	m2

Abstract		
OID	Name	Schema
e1	Employee	s1
e2	Project	s1
cl1	Employee	s2
cl2	Department	s2

Lexical					
OID	Abstract	Name	Type	IsIdentifier	Schema
a1	e1	EN	int	true	s1
a2	e1	Name	string	false	s1
a3	e2	Code	int	true	s1
a4	e2	Name	string	false	s1
f1	cl1	EmpNo	int	?	s2
f2	cl1	Name	string	?	s2
f3	cl1	Salary	int	?	s2
f4	cl2	DeptNo	int	?	s2
f5	cl2	DeptName	string	?	s2

AbstractAttribute				
OID	Name	Abstract	AbstractTo	Schema
ref1	Membership	cl1	cl2	s2

BinaryRelationship							
OID	Name	IsOptional1	IsFunctional1	...	Entity1	Entity2	Schema
r1	Membership	false	false	...	e1	e2	s1

Fig. 12. A portion of the SM part of the dictionary

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. Moreover, other supermodels can be used in different contexts: we have had preliminary experiences with Semantic Web models [5, 6, 18], with the management of annotations [25], and with adaptive systems [17]. In this section we discuss in detail our actual supermodel. We describe all the metaconstructs of the supermodel, describing which concepts they represent, and how they can be used to properly represent several well known data models.

A complete description of all the metaconstructs follows:

Abstract - Any autonomous concept of the scenario.

Aggregation - A collection of elements with heterogeneous components. It make no sense without its components.

StructOfAttributes - A structured element of an *Aggregation*, an *Abstract*, or another *StructOfAttributes*. It could be not always present (*isOptional*) and/or admit null values (*isNullable*). It could be multivalued or not (*isSet*).

AbstractAttribute - A reference towards an *Abstract* that could admit null values (*isNullable*). The reference may originate from an *Abstract*, an *Aggregation*, or a *StructOfAttributes*.

Generalization - It is a “structural” construct stating that an *Abstract* is a root of a hierarchy, possibly total (*isTotal*).

ChildOfGeneralization - Another “structural” construct, related to the previous one (it can not be used without *Generalization*). It is used to specify that an *Abstract* is leaf of a hierarchy.

Nest - It is a "structural" construct used to specify nesting relationship between *StructOfAttributes*.

BinaryAggregationOfAbstracts - Any binary correspondence between (two) *Abstract*s. It is possible to specify optionality (*isOptional1/2*) and functionality (*isFunctional1/2*) of the involved *Abstract*s as well as their role (*role1/2*) or whether one of the *Abstract*s is identified in some way by such a correspondence (*isIdentified*).

AggregationOfAbstracts - Any n-ary correspondence between two or more *Abstract*s.

ComponentOfAggregationOfAbstracts - It states that an *Abstract* is one of those involved in an *AggregationOfAbstracts* (and hence can not be used without *AggregationOfAbstracts*). It is possible to specify optionality (*isOptional1/2*) and functionality (*isFunctional1/2*) of the involved *Abstract* as well as whether the *Abstract* is identified in some way by such a correspondence (*isIdentified*).

Lexical - Any lexical value useful to specify features of *Abstract*, *Aggregation*, *StructOfAttributes*, *AggregationOfAbstracts*, or *BinaryAggregationOfAbstracts*. It is a typed attribute (*type*) that could admit null values, be optional, and identifier of the object it refers to (the latter is not applicable to *Lexical* of *StructOfAttributes*, *BinaryAggregationOfAbstracts*, and *AggregationOfAbstracts*).

ForeignKey - It is a "structural" construct stating the existence of some kind of referential integrity constraints between *Abstract*, *Aggregation* and/or *StructOfAttributes*, in every possible combination.

ComponentOfForeignKey - Another "structural" construct, related to the previous one (it can not be used without *ForeignKey*). It is used to specify which are the *Lexical* attributes involved (i.e. referring and referred) in a referential integrity constraint.

A UML class diagram of these (meta)constructs is presented in Figure 13.

We summarize constructs and (families of) models in Figure 14, 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, a structured attribute could be multivalued, or not, on the basis of the value of a property *isSet*.

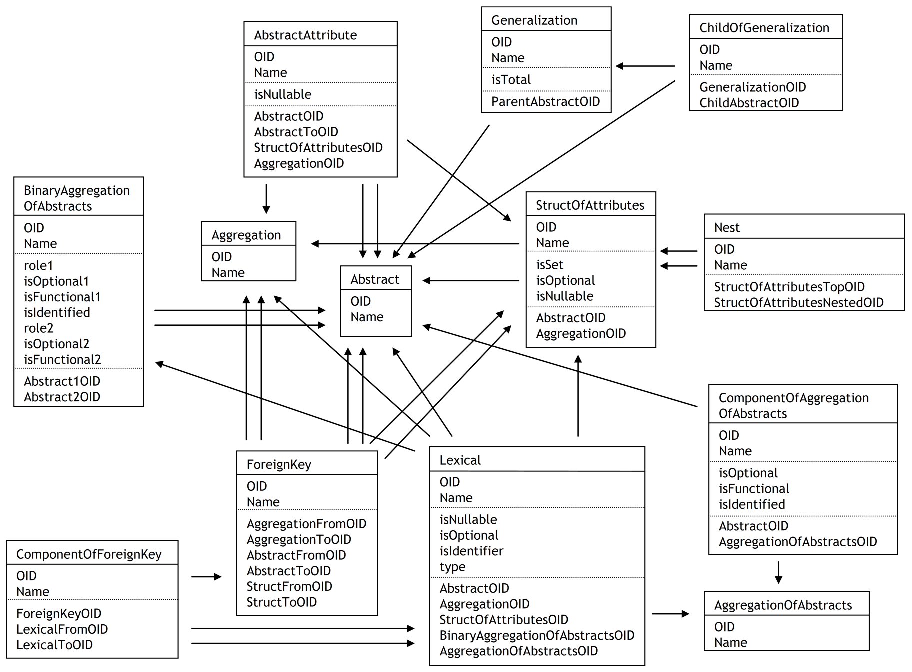


Fig. 13. The Supermodel

	Entity-Relationship	Binary Entity-Relationship	Object (UML Class Diagram)	Object-Relational	Relational	XSD
Abstract	Entity	Entity	Class	TypedTable		Root-Element
Lexical	Attribute	Attribute	Field	Column	Column	Simple-Element
Binary Aggregation OfAbstracts		Binary-Relationship				
Abstract Attribute			ReferenceField	Reference		
Aggregation OfAbstracts	Relationship					
Generalization	Generalization	Generalization	Generalization	Generalization		
Aggregation				Table	Table	
ForeignKey				ForeignKey	ForeignKey	ForeignKey
StructOf Attributes			Structured-Field	Structured-Field		Complex-Element

Fig. 14. Constructs and models

The remainder of this section is devoted to a detailed description of the various models.

4.1 Relational

We consider a relational model with tables composed of columns of a specified type; each column could allow null value or be part of the primary key of the table. Moreover we can specify foreign keys between tables involving one or more columns. Figure 15 shows a UML class diagram of the constructs allowed in the

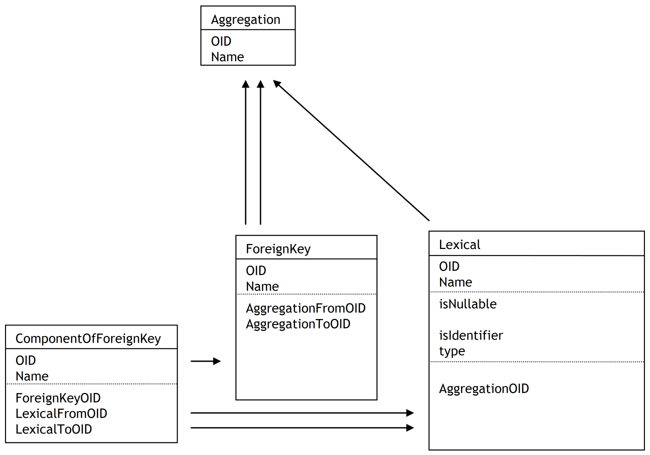


Fig. 15. The Relational model

relational model with the following correspondences:

Table - *Aggregation*.

Column - *Lexical*. We can specify the data type of the column (*type*) and whether it is part of the primary key (*isIdentifier*) or it allows null value (*isNullable*). It has a reference toward an AGGREGATION.

Foreign Key - *ForeignKey* and *ComponentOfForeignKey*. With the first construct (referencing two *Aggregation*s) we specify the existence of a foreign key between two tables; with the second construct (referencing one *ForeignKey* and two *Lexical*s) we specify the columns involved in a foreign key.

4.2 Binary ER

We consider a binary ER model with entities and relationships together with their attributes and generalizations (total or not). Each attribute could be optional or part of the identifier of an entity. For each relationship we specify minimum and maximum cardinality and whether an entity is externally identified by it. Figure 16 shows a UML class diagram of the constructs allowed in the model with the following correspondences:

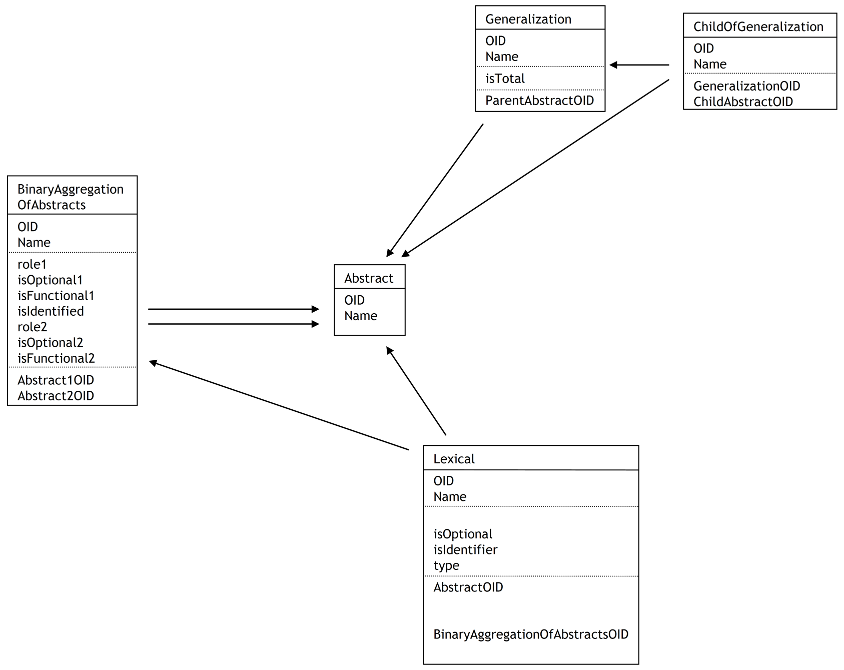


Fig. 16. The binary ER model

Entity - *Abstract*.

Attribute of Entity - *Lexical*. We can specify the data type of the attribute (*type*) and whether it is part of the identifier (*isIdentifier*) or it is optional (*isOptional*). It refers to an *Abstract*.

Relationship - *BinaryAggregationOfAbstracts*. We can specify minimum (0 or 1 with the property *isOptional*) and maximum (1 or N with the property *isFunctional*) cardinality of the involved entities (referenced by the construct). Moreover we can specify the role (*role*) of the involved entities and whether the first entity is externally identified by the relationship (*IsIdentified*).

Attribute of Relationship - *Lexical*. We can specify the data type of the attribute (*type*) and whether it is optional (*isOptional*). It refers to a *BinaryAggregationOfAbstracts*

Generalization - *Generalization* and *ChildOfGeneralization*. With the first construct (referencing an *Abstract*) we specify the existence of a generalization rooted in the referenced Entity; with the second construct (referencing one *Generalization* and one *Abstract*) we specify the childs of the generalization. We can specify whether the generalization is total or not (*isTotal*).

4.3 N-ary ER

We consider an n-ary ER model with the same features of the aforementioned binary ER. Figure 17 shows a UML class diagram of the constructs allowed in the

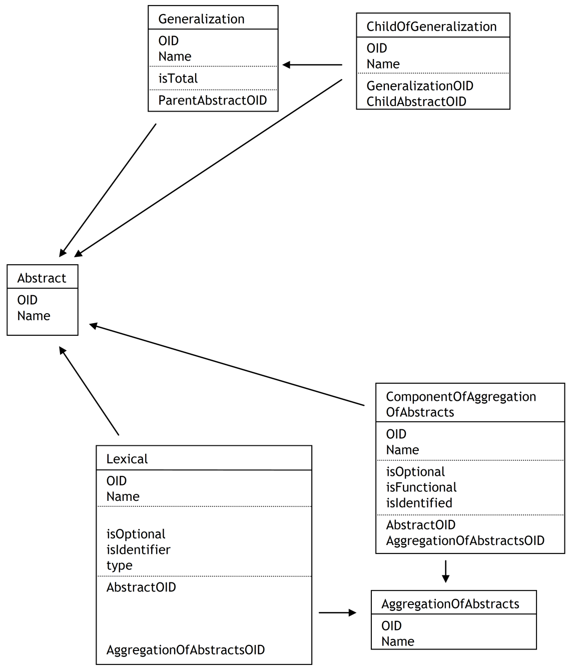


Fig. 17. The n-ary ER model

model with the following correspondences (we omit details already explained):

Entity - *Abstract*.
Attribute of Entity - *Lexical*.
Relationship - *AggregationOfAbstracts* and *ComponentOfAggregationOfAbstracts*. With the first construct we specify the existence of a relationship; with the second construct (referencing an *AggregationOfAbstracts* and an *Abstract*) we specify the entities involved in such relationship. We can specify minimum (0 or 1 with the property *isOptional*) and maximum (1 or N with the property *isFunctional*) cardinality of the involved entities. Moreover we can specify whether an entity is externally identified by the relationship (*IsIdentified*).
Attribute of Relationship - *Lexical*. It refers to an *AggregationOfAbstracts*.
Generalization - *Generalization* and *ChildOfGeneralization*.

4.4 Object-Oriented

We consider an Object-Oriented model with classes, simple and reference fields. We can also specify generalizations of classes. Figure 18 shows a UML class diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

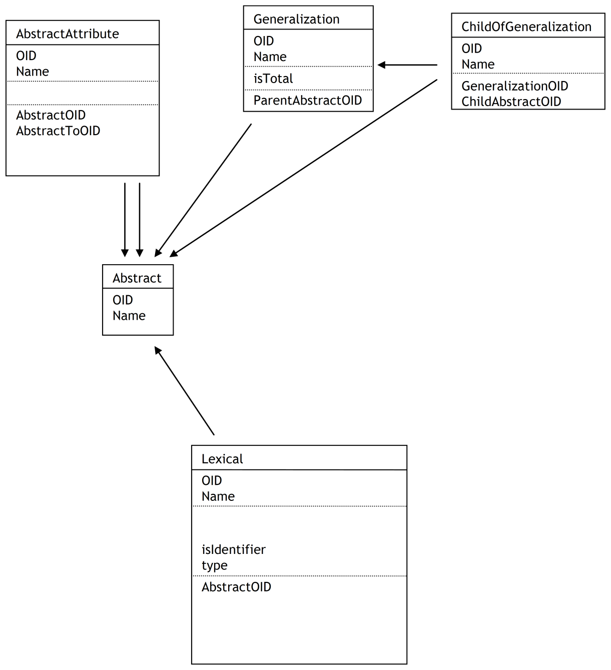


Fig. 18. The OO model

Class - *Abstract*.
Field - *Lexical*.
Reference Field - *AbstractAttribute*. It has two references toward the referencing *Abstract* and the referenced one.
Generalization - *Generalization* and *ChildOfGeneralization*.

4.5 Object-Relational

We consider a simplified version of the Object-Relational model. We merge the constructs of our Relational and OO model, where we have typed-tables rather than classes. Moreover we consider structured columns of tables (typed or not) that can be nested. Reference columns must be toward a typed table but can be part of a table (typed or not) or of a structured column. Foreign keys can involve also typed tables and structured columns. Finally, we can specify generalizations that can involve only typed tables. Figure 19 shows a UML class diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

Table - *Aggregation*.
Typed Table - *Abstract*.
Structured Column - *StructOfAttributes* and *Nest*. The structured column, represented by a *StructOfAttributes* can allow null values or not (*isNullable*)

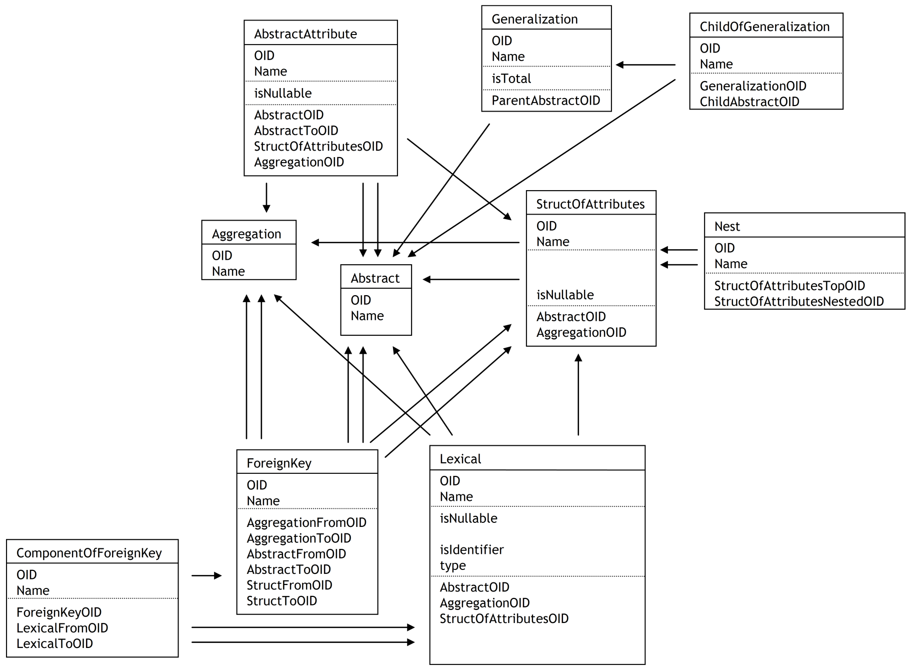


Fig. 19. The Object-Relational model

and can be part of a simple table or of a typed table (this is specified by its references toward *Abstract* and *Aggregation*. We can specify nesting relationships between structured columns by means of *Nest*, that has two references toward the top *StructOfAttributes* and the nested one.

Column - *Lexical*. It can be part of (i.e. refer to) a simple table, a typed table or a structured column.

Reference Column - *AbstractAttribute*. It may be part of a table (typed or not) and of a structured column (specified by a reference) and must refer to a typed table (i.e. it has a reference toward an *Abstract*).

Foreign Key - *ForeignKey* and *ComponentOfForeignKey*. With the first construct (referencing two tables, typed or not, and a structured column) we specify the existence of a foreign key between tables (typed or not) and structured column; with the second construct (referencing one *ForeignKey* and two *Lexical*s) we specify the columns involved in a foreign key.

Generalization - *Generalization* and *ChildOfGeneralization*.

4.6 XSD as a data model

XSD is a very powerful technique for organizing documents and data, described by a very long specification. We consider a simplified version of the XSD language. We are only interested in documents that can be used to store large amount of data. Indeed we consider documents with at least one top element unbounded. Then we deal with elements that can be simple or complex (i.e. structured). For these elements we can specify whether they are optional or whether they can be null (*nillable* according to the syntax and terminology of XSD). Simple elements could be part of the key of the element they belong to and have an associated type. Moreover we allow the definition of foreign keys (*key* and *keyref* according to XSD terminology). Clearly, this representation is highly simplified but, as we said, it could be extended with other features if there were interest in them.

Figure 20 shows a UML class diagram of the constructs allowed in the model with the following correspondences (we omit details already explained):

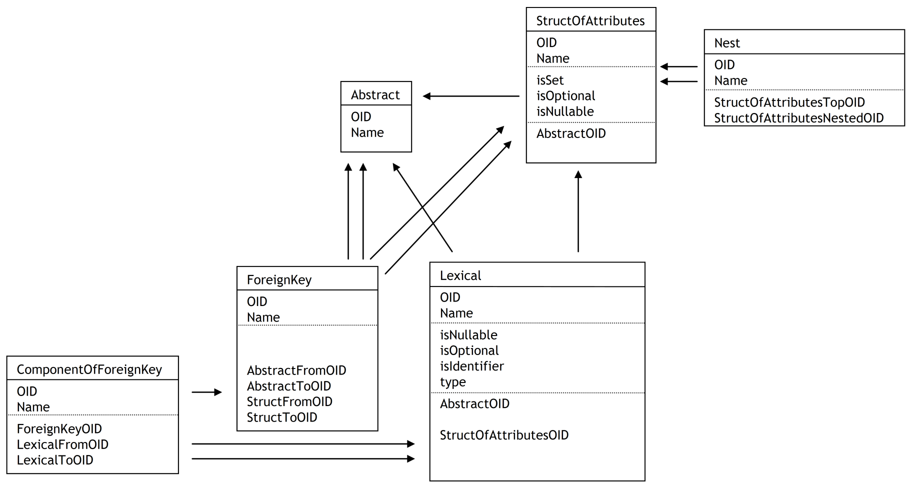


Fig. 20. The XSD language

Root Element - *Abstract*.

Complex Element - *StructOfAttributes* and *Nest*. The first construct represent structured elements that can be unbounded or not (*isSet*), can allow null values or not (*isNullable*) and can be optional (*isOptional*). We can specify nesting relationships between complex elements by means of *Nest*, that has two references toward the top *StructOfAttributes* and the nested one.

Simple Element - *Lexical*. It can be part of (i.e. refer to) a root element or a complex one.

Foreign Key - *ForeignKey* and *ComponentOfForeignKey*.

5 Operators over schema and models

The model-independent and model-aware representation of data models and schemas can be the basis for many fruitful applications. Our fist major application has been the development of a model-independent approach for schema and data translation [4] (a generic implementation of the *modelgen* operator, according to Bernstein's model management [11]). We are currently working on additional applications, towards a more general model management system [2], the most interesting of which is related to set operators (i.e. union, difference, intersection). In this section we discuss the redefinition of these operators against our construct-based representation. Let us concentrate on models first. The starting point is clearly the definition of an equality function between constructs. Two constructs belonging to two models are equal if and only if they correspond to the same metaconstruct, have the same properties with the same values, and, if they have references, they have the same references with the same values (i.e. the same number of references, towards constructs proved to be equal). Two main observations are needed. First, we can refer to supermodel constructs without loss of generality, as every construct of every specific model corresponds to a (meta)construct of the supermodel, as we said in Section 2. Second, the definition is recursive but well defined as well, since the graph of the supermodel (i.e. with constructs as nodes and references between constructs as edges) is acyclic; this implies that a partial order on the constructs can be found, and all the equality check between constructs can be performed traversing the graph accordingly to such a partial order.

The union of two models is trivial, as we have simply to include in the result the constructs of both the involved models. For difference and intersection, we need the aforementioned definition of equality between constructs. When one of these operators is applied, for each construct of the first model, we look for an equal construct in the second model. If the operator is the difference, the result is composed by all the constructs of the first model that has not an equal construct in the second model; if the operator is the intersection, the result is composed only by the constructs of the first model that has an equal construct in the second model.

A very similar approach can be followed for set operators on schemas, which are usually called *merge* and *diff* [11], but we can implement in terms of union and difference, provided they are supported by a suitable notion of equivalence. Some care is needed to consider details, but the basic idea is that the operators can be implemented by executing the set operations on the constructs of the various types, where the metalevel is used to see which are the involved types, those that are used in the model at hand.

6 Reporting

In this section we focus on another interesting application of our approach, namely the possibility of producing reports for models and schemas, again in a manner that is both model-independent and model-aware. Reports can be rendered as detailed textual documentations of data organization, in a readable and machine-processable way, or as diagrams in a graphical user interface. Again, this is possible because of the supermodel: we visualize supermodel constructs together with their properties, and relate them each other by means of their references.

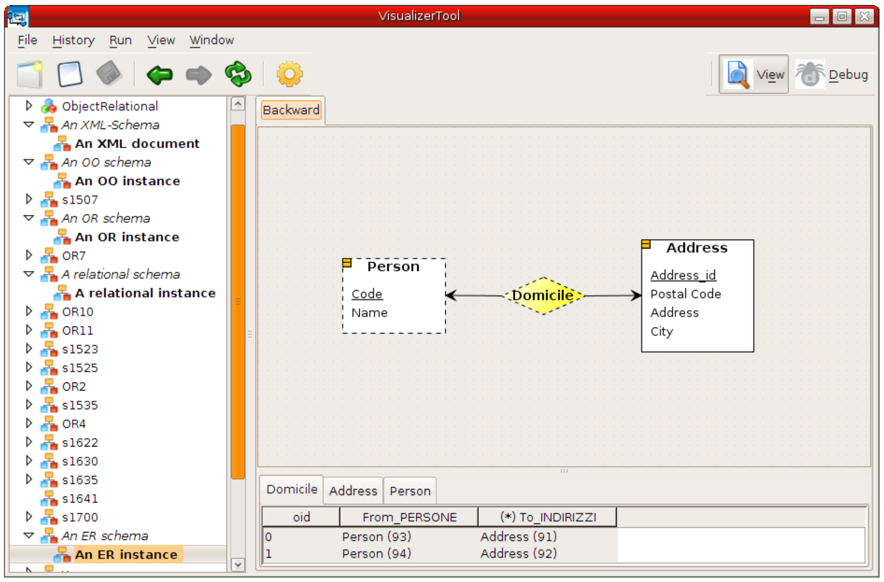
In this way, we could obtain a "flat" report of a model, which does not distinguish between type of references; so, for example, the references between a *ForeignKey* and the two *Aggregation*s involved in it would be represented as a reference from a *Lexical* towards an *Abstract*. This is clearly not satisfactory. The core idea is to classify the references in two classes: strong and weak. Instances of constructs related by means of a strong reference (e.g. an *Abstract* with its *Lexical*s) are presented together, while those having a weak relationship (e.g. a *ForeignKey* with the *Aggregation*s involved in it) are presented in different elements.

In rendering reports as text, we adopt the XML format. The main advantage of XML reports is that they are both self-documenting and machine processable if needed. Constructs and their instances can be presented according to a partial order on the constructs that can be found since, as we already said in the previous section, the graph of the supermodel (i.e. with constructs as nodes and references between constructs as edges) is acyclic.

As we said in Section 2, a schema (as well as a model) is completely represented by the set of its constructs. Hence, a report for a schema would include a set of construct elements. In order to produce a report for a schema S we can consider its constructs following a total order, $C_1, C_2, ..., C_n$, for supermodel constructs (obtained serializing a partial order of them). For each construct C_i, we consider its occurrences in S, and for each of them not yet inserted in the report, we add a construct element named C_i with all its properties as XML attributes. Let us consider an occurrence o_{i_j} of C_i. If o_{i_j} is pointed by any strong reference, we add a set of component elements nested in the corresponding construct element: the set would have a component element for each occurrence of a construct with a strong reference toward o_{i_j}. If o_{i_j} has any weak reference towards another occurrence of a construct, we add a set of reference elements: each element of this set correspond to a weak reference and has OID and name properties of the pointed occurrence as XML attributes. As an example, the textual report of the ER schema of figure 4 would be as follows:

```
<schema name="ERsimple" model="binaryER">
 <constructs>
  <ER-Entity OID="e1" name="Employee">
   <components>
    <ER-AttributeOfEntity OID="a1" name="EN" isKey="true"
                     type="int"></ER-AttributeOfEntity>
    <ER-AttributeOfEntity OID="a2" name="Name" isKey="false"
                     type="string"></ER-AttributeOfEntity>
   </components>
   <ER-Entity OID="e2" name="Project">
    <components>
     ...
    </components>
   <ER-BinaryRelationship OID="r1" name="Membership"
                isOptional1="false" isFunctional1"false" ...\>
    <references>
     <entity1 OID="e1" name="Employee"/>
     <entity2 OID="e2" name="Project"/>
    </references>
  </constructs>
</schema>
```

As we already said, a second option for report rendering is through a visual graph. A few examples, for different models are shown in Figures 21, 22, and 23.

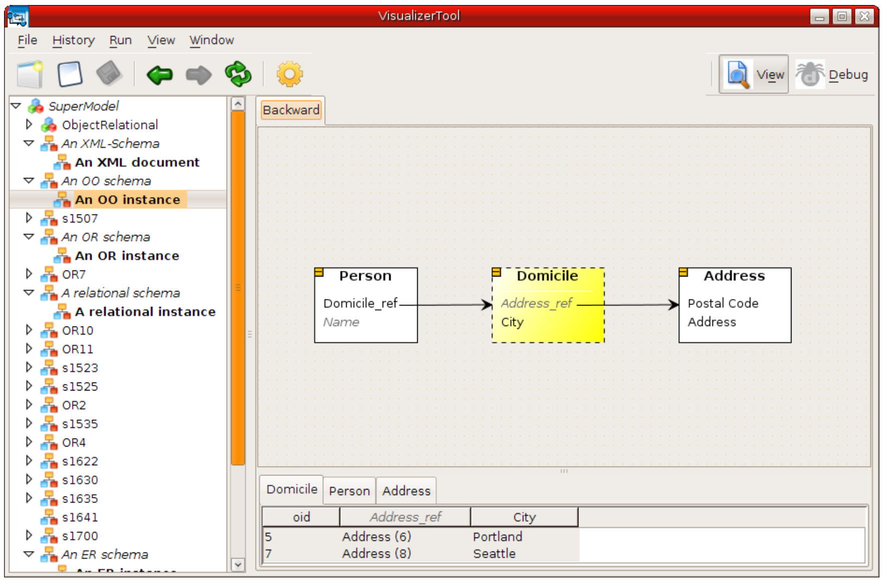


oid	From_PERSONE	(*) To_INDIRIZZI
0	Person (93)	Address (91)
1	Person (94)	Address (92)

Fig. 21. An ER schema

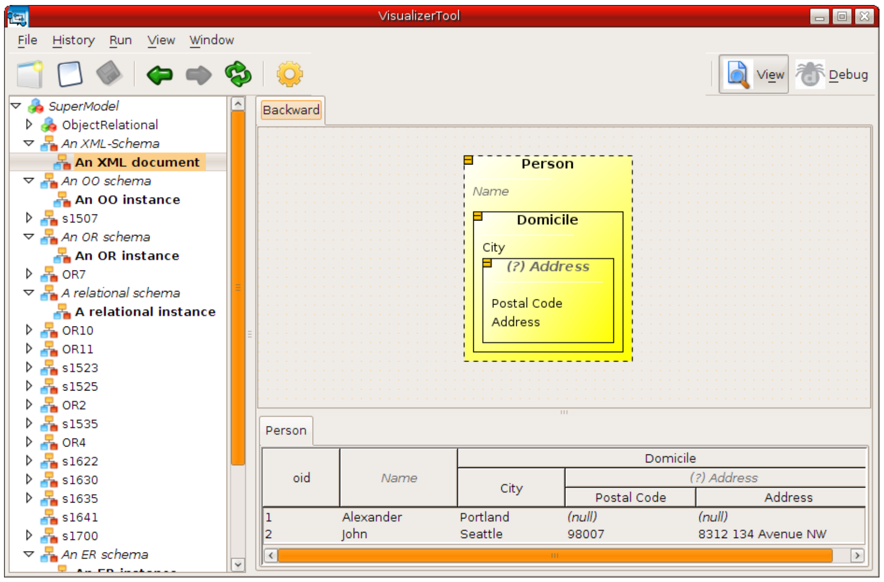
The rationale is the same as for textual reports:

– visualization is model independent as it is defined for all schemas of all models in the same way: strong references lead to embedding the “component” construct within the “containing” one, whereas weak references lead to separate graphical objects, connected by means of arrows;



oid	Address_ref	City
5	Address (6)	Portland
7	Address (8)	Seattle

Fig. 22. An OO schema



oid	Name	Domicile		
		City	(?) Address	
			Postal Code	Address
1	Alexander	Portland	(null)	(null)
2	John	Seattle	98007	8312 134 Avenue NW

Fig. 23. An XML-Schema

– visualization is model aware, in two sense: first of all, as usual the specific features of each model are taken into account; second, and more important, for each family of models it is possible to associate a specific shape with each construct, thus following the usual representation for the model (see for example the usual notation for relationships in the ER model in Figure 21.

An extra feature of the graphical visualization is the possibility to represent instances of schemas also by means of a "relational" representation that follows straightforward our construct-based modeling.

7 Conclusions

We have shown how a metamodel approach can be a the basis for a number model-generic and model-aware techniques for the solution of interesting problems. We have shown a dictionary we use to store our schemas and models, a specific supermodel (a data model that generalizes all models of interest modulo construct renaming). This is the bases for the specification and implementation of interesting high-level operations, such as schema translation as well as set-theoretic union and difference. Another interesting application is the development of generic visualization and reporting features.

Acknowledgement

We would like to thank Phil Bernstein for many useful discussions during the preliminary development of this work.

References

1. F. W. Allen, M. E. S. Loomis, and M. V. Mannino. The integrated dictionary/directory system. *ACM Comput. Surv.*, 14(2):245–286, 1982.
2. P. Atzeni, L. Bellomarini, F. Bugiotti, and G. Gianforme. From schema and model translation to a model management system. In *Sharing Data, Information and Knowledge, BNCOD 25, LNCS 5071*, pages 227–240, 2008.
3. P. Atzeni, P. Cappellari, and P. A. Bernstein. A multilevel dictionary for model management. In *ER Conference, LNCS 3716*, pages 160–175, 2005.
4. P. Atzeni, P. Cappellari, R. Torlone, P. A. Bernstein, and G. Gianforme. Model-independent schema translation. *VLDB J.*, 17(6):1347–1370, 2008.
5. P. Atzeni and P. Del Nostro. Management of heterogeneity in the semantic web. In *ICDE Workshops*, page 60. IEEE Computer Society, 2006.
6. P. Atzeni, S. Paolozzi, and P. D. Nostro. Ontologies and databases: Going back and forth. In *ODBIS (VLDB Workshop)*, pages 9–16, 2008.
7. P. Atzeni and R. Torlone. A metamodel approach for the management of multiple models and translation of schemes. *Information Systems*, 18(6):349–362, 1993.
8. P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *EDBT Conference, LNCS 1057*, pages 79–95. Springer, 1996.

9. C. Batini, G. D. Battista, and G. Santucci. Structuring primitives for a dictionary of entity relationship data schemas. *IEEE Trans. Software Eng.*, 19(4):344–365, 1993.
10. P. Bernstein, T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt. Microsoft repository version 2 and the open information model. *Information Systems*, 22(4):71–98, 1999.
11. P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR Conference*, pages 209–220, 2003.
12. P. A. Bernstein, A. Y. Halevy, and R. Pottinger. A vision of management of complex models. *SIGMOD Record*, 29(4):55–63, 2000.
13. P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD Conference*, pages 1–12, 2007.
14. J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The ATL transformation-based model management framework. Research Report 03.08, IRIN, Université de Nantes, 2003.
15. K. T. Claypool and E. A. Rundensteiner. Sangam: A framework for modeling heterogeneous database transformations. In *ICEIS (1)*, pages 219–224, 2003.
16. K. T. Claypool, E. A. Rundensteiner, X. Zhang, H. Su, H. A. Kuno, W.-C. Lee, and G. Mitchell. Sangam - a solution to support multiple data models, their mappings and maintenance. In *SIGMOD Conference*, page 606, 2001.
17. R. De Virgilio and R. Torlone. Modeling heterogeneous context information in adaptive web based applications. In *ICWE Conference*, pages 56–63. ACM, 2006.
18. G. Gianforme, R. D. Virgilio, S. Paolozzi, P. D. Nostro, and D. Avola. A novel approach for practical semantic web data management. In *KES (2), LNCS 5178 Springer*, pages 650–655, 2008.
19. C. Hsu, M. Bouziane, L. Rattner, and L. Yee. Information resources management in heterogeneous, distributed environments: A metadatabase approach. *IEEE Trans. Software Eng.*, 17(6):604–625, 1991.
20. R. Hull and R. King. Semantic database modelling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, Sept. 1987.
21. B. K. Kahn and E. W. Lumsden. A user-oriented framework for data dictionary systems. *DATA BASE*, 15(1):28–36, 1983.
22. W. C. McGee. Generalization: Key to successful electronic data processing. *J. ACM*, 6(1):1–23, 1959.
23. S. Melnik. *Generic Model Management: Concepts and Algorithms*. Springer-Verlag, 2004.
24. P. Mork, P. A. Bernstein, and S. Melnik. Teaching a schema translator to produce O/R views. In *ER Conference, LNCS 4801*, pages 102–119. Springer, 2007.
25. S. Paolozzi and P. Atzeni. Interoperability for semantic annotations. In *DEXA Workshops*, pages 445–449. IEEE Computer Society, 2007.
26. E. Rahm and H. Do. On metadata interoperability in data warehouses. Technical report, University of Leipzig, 2000.
27. R. Soley and the OMG Staff Strategy Group. Model driven architecture. White paper, draft 3.2, Object Management Group, November 2000.
28. G. Song, K. Zhang, and R. Wong. Model management though graph transformations. In *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 75–82, 2004.