M^2ORM^2: A model for the transparent management of relationally persistent objects

Luca Cabibbo and Roberto Porcelli

Dipartimento di Informatica e Automazione
Università degli studi Roma Tre
Via della Vasca Navale 79, 00146 Roma, Italy

Abstract. Object-oriented application development often involves storing application objects in a relational database. Sometimes it is desirable to develop the persistent classes and the relational database in an independent way, and to use an object persistent manager to connect them in a suitable way. This paper introduces M^2ORM^2, a model for describing meet-in-the-middle mappings between object schemas and relational schemas, to support the transparent management of object persistence by means of relational databases.

1 Introduction

Developing information systems involves various technologies, to be used in a combined and, possibly, synergic way. Relational database management systems provide an effective and efficient management of persistent, shared and transactional data [2]. Object-oriented tools and methods (programming languages and analysis and design methodologies) support the effective development of the application logic of complex information systems [16]. In practice, it is common to develop object applications with a layered architecture, containing at least an application logic layer and a persistence layer. *Persistent classes* are classes whose objects hold persistent data; they belong to the application logic layer, and are made persistent by means of code that connect them, in a suitable way, to the persistence layer. Object persistence can be achieved in several ways. In this paper, we will consider persistent objects managed by relational databases; that is, each application object is represented by means of tuples of a relational database.

Recently, various frameworks for the *transparent* management of object persistence have been implemented [18, 22]. By using them, the programmer manages persistent objects by means of standard API's such the ODMG ones [7], that is, the same way he would use objects in an object database. However, such frameworks can persist objects also by means of a relational database or files. Persistence is transparent to the programmer, since he does not know actual implementation details. The correspondence between objects and the underlying persistence support is managed by a software module, which can be implemented as a pre-compiler, a post-compiler, or an interpreter. Transparent persistence of

objects can be achieved in various ways. In the *R/O mapping* approach (*Relation to Object mapping*, also called *reverse engineering*), persistent classes are automatically generated from a relational database. This way, the programmer populates the database by means of creations and modifications of objects from persistent classes. Then, persistent classes propagate such creations and modification to the underlying database. In the *O/R mapping* approach (*Object to Relation mapping*, also called *forward engineering*), starting from the classes that should be made persistent, the database is automatically generated, together with the code needed to propagate object persistence to the database.

A persistence management that is completely transparent is not always useful or possible. Often, an application should be developed that accesses an already existing relational database that is shared by many applications; this restricts the use of O/R mapping. Furthermore, the application logic designer needs to use all the features of the object model, without being constrained by the relational database schema; this restricts the use of R/O mapping. Luckily, there is a further way to persist objects, which allows to manage the cases in which the application logic and the database have been developed and evolve in an independent way. The *meet-in-the-middle* approach assumes that the persistent classes and the database are designed and implemented separately. In this case, the correspondences between persistent classes and the relational database should be given, possibly described in a declarative way. These correspondences describes a "meet in the middle" between the two schemas and are used by the persistence manager to let the objects persist by means of the database. The meet-in-the-middle approach is very versatile, since modifications in persistent classes and/or in the relational database can be managed by simply redefining the correspondences.

Unfortunately, existing systems support the meet-in-the-middle approach only in a limited way. Several object persistence managers are indeed based on the O/R and R/O approaches and, essentially, they allow to manage only correspondences between similar structures (e.g., the objects of a single class with the tuples of a single relation); such systems may permit a meet between the two schemas, but only as a local tuning activity, after the schema translation from one model to the other. Limitations of existing systems are motivated by the difficulties in reasoning about complex correspondences to avoid anomalies and inconsistencies.

This paper introduces M^2ORM^2 (an acronym for *Meet-in-the-Middle Object/Relational Mapping Model*), a model to describe mappings (that is, correspondences) between object schemas and relational schemas, to support the transparent management of object persistence based on the meet-in-the-middle approach. With respect to currently available systems, M^2ORM^2 allows for more possibilities to meet schemas. Rather than considering only correspondences between single classes and single relations, M^2ORM^2 allows to express complex correspondences between clusters of related classes (intuitively, each representing a single concept) and clusters of related relations. Furthermore, it is possible to express correspondences describing relationships between clusters. With re-

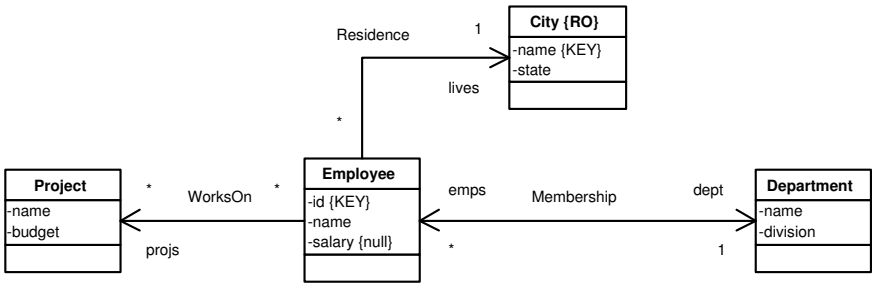

Fig. 1. An object schema

spect to other proposals, M^2ORM^2 considers specific details of the object and relational data models, to identify as many ways to meet schemas as possible.

The main contribution of this paper is the introduction of M^2ORM^2. Furthermore, a discussion is started on the correctness of mappings, by identifying some correctness conditions.

Section 2 proposes terminology and notation used to describe object schemas and relational schemas. Section 3 introduces M^2ORM^2, to describe mappings between schemas, together with an example. Section 4 discusses the problem of mapping correctness. Finally, Section 5 discusses the effectiveness of the model, by comparing it with related proposals.

2 Object schemas and relational schemas

This section presents briefly the data models (an object model and the relational model) and the terminology used in this paper.

The *object model* we consider is a non-nested semantic data model (with structural features, but without behavioral ones). We have in mind a Java-like object model, formalized as a simplified version of the ODMG model [7] and of UML [5].

At the schema level, a *class* describes a set of *objects* having the same structural properties. Each class has a set of *attributes* associated with it, each having a name and a type; in this paper we make the simplifying hypothesis that all class attributes are of a same simple type, e.g., strings. An *association* describes a binary relation between a pair of classes (in reality, between the objects of such classes). A pair of classes can be in a *generalization/specialization relationship*; this implies attribute and association inheritance from the superclass to the subclass. We consider single inheritance only. An *object schema* is a set of classes, related by generalization/specialization hierarchies, and associations among such classes. Figure 1 shows a sample object schema.

At the instance level, a class is a set of *objects*. More specifically, each object is instance of exactly one class. However, because of generalization/specialization relationships, an object may belong to more than one class: the one from which

it has been instantiated and all of its superclasses. Each object is associated with an *oid*, an unique identifier that allows to reference the object. The *state* of an object is given by the set of values that its attributes hold in a certain moment. An association is a set of *links*; each link describes a relationship between a pair of objects.

This paper takes into consideration the following integrity constraints. Class attributes can have null value; an attribute whose value cannot be null is said to be *not null*. Sometimes it is useful to search an object in a class on the basis of the value of some attributes; in this case, the attributes identifying the objects are called *key attributes* and the class is said to be *with key*; otherwise, the class is *without key*. Key attributes should be not null. A *read-only class* is a class from which it is not allowed to instantiate new persistent objects and to modify and to delete already existing objects. In an application, read-only classes are useful to access information generated by other applications that cannot be modified by this application. In Figure 1, key attributes are denoted by the constraint $\{KEY\}$, and attributes that can be null by means of the constraint $\{null\}$; constraint $\{RO\}$ denotes read-only classes.

For associations we consider multiplicity and navigability constraints. A *role* is an end of an association, that is, a class involved in the association. Roles have name, navigability, and multiplicity. The *multiplicity* of a role in an association denotes how many objects (at least and at most) of that class can be linked to each object of the class on the other role of the association. The *navigability* of a role denotes the possibility to reach the objects of that class by means of links from objects of the other role of the association.

In the *relational model* [2], at the schema level a *relation* describes a set of tuples. A relation schema is a set of *attributes*, each with a name and a type; in this paper, we assume that all relation attributes are of a simple type, e.g., strings. A *relational schema* is a set of relations. At the instance level, a relation is a set of *tuples* over the attributes of the relation.

This paper takes into consideration the following integrity constraints. Attributes can be or not be *not null*. Each relation has a *key*, that is, a non-empty set of attributes that allows the identification of the tuples of the relation. A *key attribute* is an attribute that belongs to a key; key attributes should be not null. Sometimes relations are identified by means of *artificial keys* (or *surrogates*), rather than by means of *natural keys* (that is, keys based on attributes having a natural semantics). In a relation with artificial key, the insertion of a new tuple involves the generation of a new artificial key; the DBMS is usually responsible of this generation. For *referential constraint* (or *foreign key*) we mean a non-empty set of attributes of a relation used to reference tuples of another relation.

Figure 2 shows a sample relational schema. Attribute forming natural keys are denoted by the constraint $\{NK\}$, and those forming artificial keys by the constraint $\{AK\}$. Referential constraints are denoted by arrows and implemented by means of attributes with the constraint $\{FK\}$. The constraint $\{null\}$ denotes attributes that can be null.

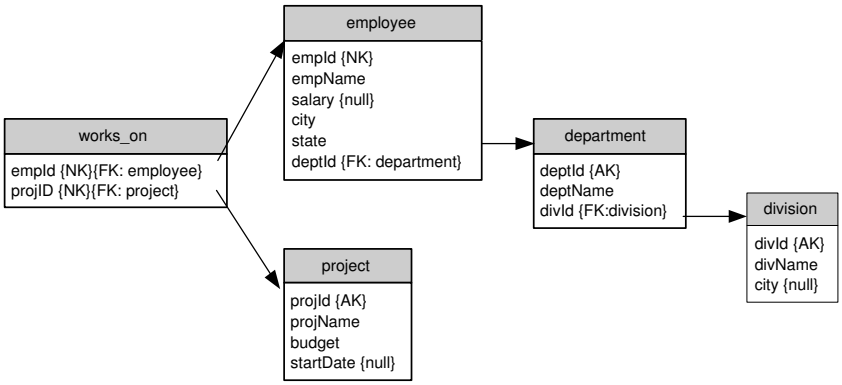

Fig. 2. A relational schema

We assume that programmers manage object schemas only in a programmatic way. In practice, objects and links are manipulated by means of *CRUD* operations (*Create*, *Read*, *Update*, *Delete*), that allow the creation of persistent objects, the reading of persistence objects (that is, the unique search of an object based on its key), as well as the persistent modification and deletion of objects. Moreover, navigation, formation, breaking, and modification of persistent links between objects is allowed. Reading is meaningful only for classes with key. Creation, modification, and deletion is not meaningful for read-only classes. For example, Figure 3 shows a possible fragment of code to increase the salary of an employee, with respect to the scheme of Figure 1.

In correspondence to such programmatic manipulations of an object schema, a meet-in-the-middle-based object persistence manager should translate CRUD operations on objects and links into operation over an underlying relational database. This translation should happen in an automatic way, on the basis of a suitable mapping between the object schema and the relational schema, as described in the next section.

```
/* increase the salary of Employee #123 */
PersistenceManager pm = new PersistenceManager(...);
Transaction tx = pm.newTransaction();
tx.begin();
Employee e = (Employee) pm.getObjectById( new EmployeeId("123") );
e.setSalary( e.getSalary()*1.1 );
tx.commit();
```

Fig. 3. Persistent objects are managed in a programmatic way

3 A model for object/relational schema mappings

M^2ORM^2 (an acronym for *Meet-in-the-Middle Object/Relational Mapping Model*) is a model for describing mappings among object schemas and relational schemas, to support the transparent management of relationally persistent objects based on the meet-in-the-middle approach. In this paper, we assume that an object schema and a relational schema have been independently developed. In particular, the relational schema could be (partially) denormalized for efficiency reasons; the object schema could be denormalized as well, that is, it may contain "coarse grain" objects.

In this paper, a M^2ORM^2 mapping is described by means of a multi-graph (that is, by a set of nodes and a set of arcs, with the observation that several arcs can connect a same pair of nodes). Each node describes a correspondence between a set of classes and a set of relations. Each arc describes a relationship between the correspondences described by means of a pair of nodes. Intuitively, using the Entity-Relationship terminology [3], each node represents an entity (which can be denormalized in one of the schemas) and each arc represents a binary relationship or a generalization/specialization relationship between two entities. In practice, we envision the possibility to describe M^2ORM^2 mappings as structured (e.g., XML) text files, as it happens in current systems.

A *class cluster* (or *c-cluster*) comprises a non-empty set of classes and a set of associations and/or generalization/specialization relationships among such classes.[1] In a c-cluster, one of the classes should be selected as *primary class* of the c-cluster; the other classes are *secondary* ones. The associations of the node should be, directly or indirectly, all of type one-to-one or one-to-many from the primary class to secondary classes. Intuitively, such associations should relate each object of the primary class with at most an object from each of the secondary classes.

A *relation cluster* (or *r-cluster*) comprises a non-empty set of relations, together with referential constraints among them. In an r-cluster, one of the relations should be selected as the *primary relation* of the r-cluster; the other relations, called *secondary*, should be referenced, directly or indirectly, from the primary relation of the r-cluster. Intuitively, each tuple of the primary relation should be associated with at most a tuple in each of the secondary relations through the referential constraints of the r-cluster.

Each *node* of a mapping describes the correspondence between a c-cluster and an r-cluster, by means of information about correspondences among their elements (classes, relations, attributes, associations, and generalization/specialization relationships). Specifically, in a node the correspondence among c-cluster elements and r-cluster ones are described by means of *attribute correspondences*, each of which involves an attribute of a class and an attribute of a relation. In

[1] More precisely, each element of a c-cluster is associated with a class in the object schema. Therefore, it is possible that a class takes part to more c-clusters, or that it takes part more the once in a same c-cluster. A similar consideration applies to r-clusters, that will be introduced shortly.

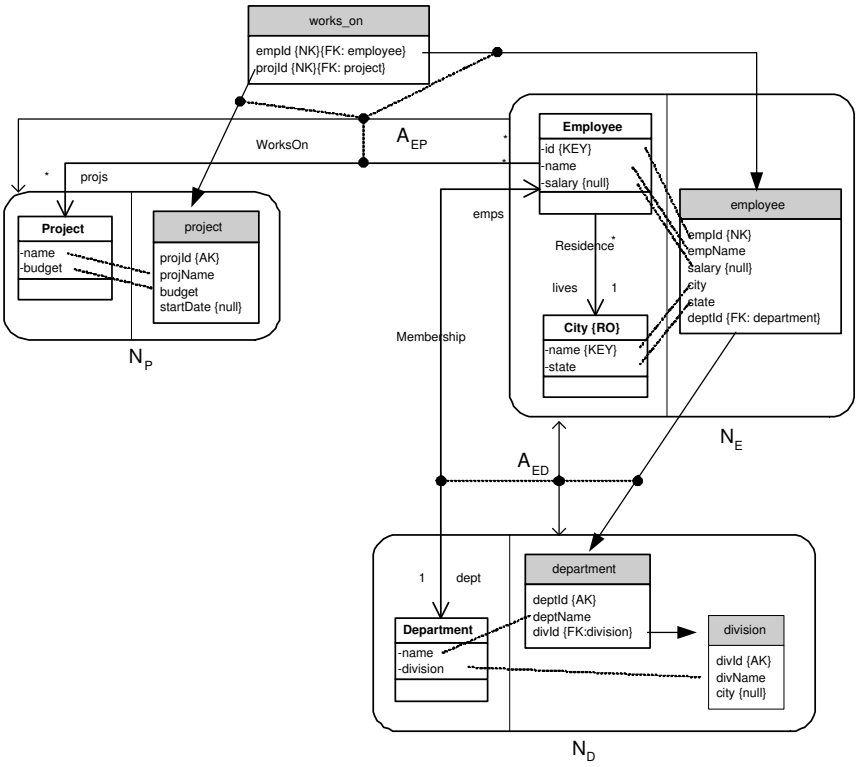

Fig. 4. A M^2ORM^2 mapping between the schemas of Figures 1 and 2

M^2ORM^2, there are three kinds of nodes, to let a class correspond with a relation, a class with multiple relations, and multiple classes with a relation. Currently, the model does not permit that multiple classes correspond with multiple relations; however, most complex correspondences of this kind can be represented by means of arcs.

Figure 4 shows a M^2ORM^2 mapping between the schemas of Figures 1 and 2, based on three nodes: a node N_P, describing the correspondence between class *Project* and relation *project*; a node N_E, describing the correspondence between classes *Employee* and *City* and relation *employee*; and a node N_D, describing the correspondence between class *Department* and relations *department* and *division*.

Node N_P describes the correspondence between a class (*Project*) and a relation (*project*); it is clear that both the class and the relation are primary in the node. The correspondence between them is based on the attribute correspondences *(Project.name, project.projName)* and *(Project.budget, project.budget)*; these are depicted as dotted lines in Figure 4. Node N_P describes a *total* correspondence between a class and a relation: each object of the class is represented by means of a tuple of the relation. This correspondence involves a class without

key and a relation with artificial key. All the attributes of the class are involved in the correspondence; however, not all the attributes of the relation are involved in it. This correspondence allows to manage, for example, the creation of a new persistent *Project* object by means of the insertion of a new tuple in *project*; in this tuple, the values for *project.projName* and *project.budget* come from the object attributes *Project.name* and *Project.budget*, on the basis of attribute correspondences; moreover, the value for *project.projId* is automatically generated and the value for *project.startDate* remains null.

Node N_E describes the correspondence between two classes (*Employee* and *City*) linked by an association (*Residence*) and a relation (*employee*). In this node, *Employee* is the primary class. Intuitively, each object of the primary class *Employee* can be associated (by means of the association *Residence*) with an object of the secondary class *City*. Each tuple of *employee* represents an *Employee* object together with the *City* object related to it, and also the link of type *Residence* between these two objects. In this node, in which the primary class *Employee* is with key, the correspondence is based on the correspondence between the key attributes of the primary class and of the primary relation *(Employee.id, employee.empId)* together with the attribute correspondences *(Employee.name, employee.empName)*, *(Employee.salary, employee.salary)*, *(City.name, employee.city)*, and *(City.state, employee.state)*. For example, the creation of a new *Employee* object, linked to an existing *City* object, is managed by the insertion of a new tuple in *employee*. (The management of the membership of the employee in a department is described later.)

Finally, node N_D describes the correspondence between a class (*Department*) and two relations (*department* and *division*); the primary relation is *department*. Intuitively, each *Department* object is represented by a tuple of *department* and a tuple of *division*, related by means of a referential constraint. The correspondence is based on the attribute correspondences *(Department.name, department.deptName)* and *(Department.division, division.divName)*. The latter correspondence is meaningful with respect to the referential constraint between the primary relation *department* and the secondary relation *division* of the r-cluster. Indeed, this node allows to represent in the mapping the two relations together with the referential constraint between them. For example, the creation of a new *Department* object is managed by the insertion of a new tuple in *division* (with the generation of an artificial key) and of a new tuple in *department* (with the generation of another artificial key).

In general, not every mapping is correct. For example, if the primary class of a node is with key, then it is necessary that its key attributes correspond to the key attributes of the primary relation of the node. However, if the primary class is without key, then the primary relation should be with artificial key. Correctness of mappings will be discussed briefly in Section 4.

A M^2ORM^2 mapping can contain arcs. Intuitively, arcs allow to represent correspondences and elements which cannot be represented by means of nodes; specifically, further associations, generalization/specialization relationships, referential constraints, and some further relations. Each arc describes a relationship

between a pair of nodes, and can be a binary relationship (of type one-to-one, one-to-many, or many-to-many) or a generalization/specialization relationship.

An arc involves a class correspondence and a relation correspondence. A *class correspondence* describes the correspondence between the primary classes of the c-clusters of the nodes connected by the arc, and it is based either on the navigable roles of an association between the classes or on a generalization/specialization relationship. In practice, the roles are implemented by means of reference attributes (for the to-one navigability) and/or collections of reference attributes (for the to-many navigability). If the association is unidirectional, then a single attribute is involved; otherwise, if it is bidirectional, there are two attributes involved. A *relation correspondence* describes the correspondence between the primary relations of the r-clusters of the nodes connected by the arc, and it is based on the set of attributes that implement the relationship between the two relations by means of referential constraints; further relations can be involved. An arc groups a class correspondence and a relation correspondence, representing a one-to-one, one-to-many, or many-to-many binary relationship or a generalization/specialization relationship between the instances represented by a pair of nodes.

For example, the mapping shown in Figure 4 contains two arcs: an arc A_{ED} for the one-to-many association between *Employee* and *Department* and an arc A_{EP} for the many-to-many association between *Employee* and *Project*.

Arc A_{ED} between nodes N_E and N_D describes the one-to-many relationship between employees and departments. This relationship is represented by means of the correspondence *[Employee.dept, Department.emps]* between the classes *Employee* and *Department* and by means of the correspondence *[employee.deptId]* between the relation *employee* and *department*. Intuitively, this arc lets the association *Membership* between *Employee* and *Department* correspond with the referential constraint between *employee* and *department*. For example, if an *Employee* object e changes its membership (that is, its *dept* reference attribute) to a *Department* object d, then the *deptId* attribute of the *employee* tuple representing e is updated to reference the *department* tuple representing d.

Arc A_{EP} between nodes N_E and N_P describes the many-to-many relationship between employees and projects. This arc is based on the (unidirectional) correspondence *[Employee.projs]* between classes *Employee* and *Project* and on the correspondence *[works_on.empId, works_on.projId]* between relations *employee* and *projects*. In practice, this arc lets the association *WorksOn* between *Employee* and *Project* correspond with the referential constraints between *employee* and *project* stored in the tuple of the relation *works_on*. For example, if a *WorksOn* link between an *Employee* object e and a *Project* object p is broken, then the *works_on* tuple representing this link is deleted.

To discuss arcs representing generalization/specialization relationships, consider the schemas shown in Figure 5. A mapping between them can be defined by means of two nodes (for persons and students) and a generalization/specialization arc connecting them. In the arc, the generalization/specialization between

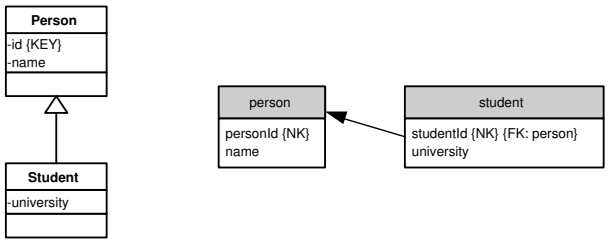

Fig. 5. A schema involving a generalization/specialization relationship

the two classes corresponds with the referential constraint from the primary key of *student* to *person*.

In practice, there are various ways to represent a generalization/specialization hierarchy by means of a relational database [3]. Until now, M^2ORM^2 can only represent the one that essentially maps each class of the hierarchy with a different relation.

4 Correctness of mappings

In the previous section, M^2ORM^2 has been used as a syntactical tool to represent mappings. In reality, mappings have semantics as well, to describe how CRUD operations on objects and links of the object schema can be realized by means of operations on the relations of the relational schema. However, not every mapping that can be described using M^2ORM^2 is correct. Intuitively, a mapping is *correct* if it supports, in an effective way, the management of CRUD operations on objects and links by means of the relational schema. Otherwise, a mapping is *incorrect* if operations on objects and links can give rise to anomalies. In the following of this section we identify classes of anomalies caused by incorrect mappings, and some categories of conditions suitable for studying the correctness of M^2ORM^2 mappings.

We consider a sequence of CRUD operations on objects and links that, globally, transforms a valid instance of the object schema into another valid instance, that is, in which all the integrity constraints imposed by the schema are satisfied. An incorrect mapping can give rise to the following anomalies during the execution of such sequence of operations:

- *creation anomalies* happen when the relational schema is not able to manage the creation of a new object in some class (not a read-only class); for example, a creation anomaly happens if the class is not represented in the mapping or if some of its attributes are not in a suitable correspondence with attributes of the relational schema (including erroneous correspondences involving not null attributes and key attributes);
- *reading anomalies* are related to the inability to identify uniquely an object from a class with key; for example, a reading anomaly happens if there are erroneous correspondences involving key attributes of classes and relations;

– *update anomalies* happen if it is not possible to manage the modification of attributes of an object (belonging to non read-only class);
– *deletion anomalies* happen if it is not possible to manage the deletion of an object (belonging to non read-only class).

Similarly, there are anomalies related to the formation of links, and also to their navigation, modification, and breaking.

There are two main causes for anomalies: *incorrect correspondences between elements* and *incorrect representation of integrity constraints*.

An example of the former case happens if the correspondences among elements are incomplete, that is, if there are elements of the object schema (classes, associations, attributes) that are not in correspondence with elements of the relational schema. Indeed, each class must belong to at least one node, and the attributes of classes should occur in at least one attribute correspondence. However, relational elements must obey to different conditions; for example, it is possible that a relation or some of its attributes do not partecipate in the mapping.

Inconsistencies in the representation of integrity constraints can happen in several different ways. For example, with respect to key constraints, it is necessary that key attributes of primary classes with key are in correspondence with key attributes of primary relations with natural key. Furthermore, primary classes without key should correspond to relations with artificial key. As a further example, concerning referential constraints, in a node that lets a class correspond with more than a relation, non-key attributes of the class should correspond to non-key attributes of relations, and referential constraints towards secondary relations should be implemented by means of artificial keys.

The conditions we have just described are *necessary* conditions for the correctness of M^2ORM^2 mappings. However, in practice (i.e., for the implementation of a persistence manager based on M^2ORM^2) *sufficient* conditions should be fixed, to manage mappings in an effective way. Current systems suffer from several limitations, since conditions they are based on are very restrictive. One of the main goal of this research is to identify conditions that are as permissive as possible with respect to which mappings are semantically meaningful.

5 Discussion

The "professional" literature on databases and on pattern languages is rich of works on how to manage persistent classes by means of relational databases [1, 6, 10, 15]. Most of these works describes the O/R mapping, a kind of logical design [3] starting from an object schema rather than from an entity-relationship schema. But, as we already said in the Introduction, O/R mapping assists the definition of a database supporting a single application, and not *shared* among several applications, being this one of the main motivations for using a DBMS as the manager of persistent data.

This topic has been investigated by the scientific literature of the database community as well. Persistence [14] is a system supporting the O/R mapping.

Table 1. Comparison with other models and tools

Feature	JDO	OJB	JRELAY	JDX	M^2ORM^2
O/R mapping (forward engineering)	yes	yes	yes	yes	no
R/O mapping (reverse engineering)	n/a	yes	yes	yes	no
Meet in the middle	n/a	yes	yes	yes	yes
One class/one relation	n/a	yes	yes	yes	yes
One class/many relations with referential constraints	n/a	yes[a]	yes	no	yes
One class/many relations with vertical partitioning	n/a	no	no	no	yes[b]
Many classes/one relation	n/a	no	no	yes	yes
Many classes/many relations	n/a	no	no	no	no
Classes with key	yes	yes	yes	yes	yes
Classes without key	yes	no	yes	no	yes
Read-only classes	no	no	no	no	yes
Attribute of a class mapped into several relations	n/a	no	no	no	yes
One-to-one relationship	n/a	yes	yes	yes	yes
One-to-many relationship	n/a	yes	yes	yes	yes
Many-to-many relationship without attributes	n/a	yes	yes	no	yes
Many-to-many relationship with attributes	n/a	no	no	no	yes[b]
Many-to-many relationship with selective attributes	n/a	no	no	no	yes[b]
Generalization/specialization hierarchies	yes	yes	yes	yes	yes[c]

[a] Yes, but not implemented.
[b] Yes, but not discussed in this paper.
[c] Yes, but discussed only partially in this paper.

Gateway [19] is a system for managing persistent objects by means of a relational database based on the meet-in-the-middle approach; however, the paper does not give details on the mapping model used. EBO [20] is an R/O mapping system, featuring some functionalities of meet-in-the-middle; also in this case, the mapping model has not been described.

The problem of describing and analyzing schema mappings is also of interest in the areas of data transformations [8] and data integration [9]. The notion of mapping used in this paper is inspired from the one proposed in the context of model management [4].

Table 1 compares M^2ORM^2 with some standards and commercial and/or open source systems for the transparent management of persistent objects. Java Data Objects (JDO) [11] is a standard for the transparent management of Java objects. OJB [18] is an Apache open source project, implementing the ODMG and the JDO API's; in practice, it allows to manage persistent objects as they were stored in an ODMG object database [7]. JDX [12] features O/R mapping functionalities similar to OJB ones; however, persistent objects are manipulated

by means of proprietary API's. JRELAY [13] is a JDO implementaion, implementing the three mapping approaches, but with significative limitations. From the table, it is possible to see that M^2ORM^2 provides the same features offered by current systems, plus more.

The management of generalization/specialization hierarchies needs a special comment. Until now, M^2ORM^2 supports them only in a limited way, but still comparable to the support given by other systems. In future, we plan to manage generalization/specialization relationships in a more complete way.

As future work, we plan the implementation of a framework for the management of persistent Java objects based on M^2ORM^2. From a theoretical perspective, we plan to study several extensions to the model (specifically, the management of generalization/specialization hierarchies, transient classes and attributes, further integrity constraints, and schematic heterogeneities [17]) but also to give more precise characterizations of correct M^2ORM^2 mappings.

References

1. S.W. Ambler. The fundamentals of mapping objects to relational databases. White Paper, *http://www.agiledata.org*, 2003.
2. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems. Concepts, Languages and Architectures*. McGraw-Hill, 1999.
3. C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin-Cummings, 1992.
4. P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for the management of complex models. *ACM Sigmod Record*, 29(4):55–63, 2000.
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
6. K. Brown and B.G. Whitenack. Crossing Chasms: A pattern language for object-RDBMS integration. In *Pattern Languages of Program Design 2*, 1996.
7. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
8. Data Transformations. S.ı. of the *IEEE Bull. on Data Engineering*, 22(1), 1999.
9. Integration management. S.ı. of the *IEEE Bull. on Data Engineering*, 25(3), 2002.
10. M.L. Fussell. Foundations of object relational mapping. White Paper, *http://www.chimu.com*, 1997.
11. Java Data Objects. *http://www.jdocentral.com*.
12. JDX. *http://www.softwaretree.com/*.
13. JRELAY. *http://www.objectindustries.com/*.
14. A.M. Keller, R. Jensen, and S. Agrawal. Persistence Software: Bridging object-oriented programming and relational databases. In *ACM SIGMOD International Conf. on Management of Data*, pages 523–528, 1993.
15. W. Keller. Mapping object to tables: A pattern language. In *European Conf. on Pattern Languages of Programming*, 1997.
16. C. Larman. *Applying UML and Patterns. An introduction to object-oriented analysis and design and the Unified Process*. Prentice Hall PTR, 2002.
17. R.J. Miller. Using schematically heterogeneous structures. In *ACM SIGMOD International Conf. on Management of Data*, pages 189–200, 1998.
18. ObJect relational Bridge. *http://db.apache.org/ojb/*.

19. J.A. Orenstein and D.N. Kamber. Accessing a relational database through an object-oriented database interface. In *21st Int. Conf. on VLDB*, 702–705, 1995.
20. J.A. Orenstein. Supporting retrievals and updates in an object/relational mapping system. *IEEE Bull. on Data Engineering*, 20(1):50–54, 1999.
21. E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.
22. Torque. *http://db.apache.org/torque/*.