Objects Meet Relations: On the Transparent Management of Persistent Objects

Luca Cabibbo

Dipartimento di Informatica e Automazione
Università degli studi Roma Tre
cabibbo@dia.uniroma3.it

Abstract. Many information systems store their objects in a relational database. If the object schema or the relational schema of an application can change often or in an independent way, it is useful to let a persistent framework manage the connection between objects and relations. M^2ORM^2 is a model for describing meet-in-the-middle mappings between objects and relations, to support the transparent management of object persistence by means of relational databases. This paper presents M^2ORM^2 and describes how operations on objects and links can be implemented as operations on the underlying relations. It also proposes necessary conditions for the correctness of M^2ORM^2 mappings.

1 Introduction

Many information systems are developed using relational and object-oriented technologies. Relational database management systems provide an effective and efficient management of persistent, shared and transactional data [6]. Object-oriented tools and methods include programming languages, modeling languages such as UML [3], development processes such as UP [8] and XP [2], as well as analysis and design methods [12]. 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 in the application logic layer whose objects hold persistent data; they are made persistent by means of code that connect them, in a suitable way, to the persistence layer. In this paper, we assume that persistence of objects is implemented by a relational database.

Code in the persistence layer should change if either the structure of classes in the application logic layer or the relational database schema changes. If software is developed using an iterative process, such as UP or XP, those changes arise frequently. In this case, it is valuable to use a framework for the transparent management of object persistence [13, 14], rather than writing and maintaining such code directly. This way, the programmer manages persistent objects by means of standard API's, such as the ODMG ones [5], that is, the same way he would use objects in an object database. Persistence is transparent to the programmer, since he does not know actual implementation details.

Transparent persistence of objects is achieved mainly in two ways. In the *R/O mapping* approach (*Relation to Object mapping*, also called *reverse engineering*, implemented by, e.g., Torque [14]), persistent classes are automatically

A. Persson and J. Stirna (Eds.): CAiSE 2004, LNCS 3084, pp. 429–445, 2004.
© Springer-Verlag Berlin Heidelberg 2004

generated from a relational database. The programmer populates the database by means of creations and modifications of objects from persistent classes. Then, persistent classes propagate such creations and modifications to the underlying database. In the *O/R mapping* approach (*Object to Relation mapping*, also called *forward engineering*, implemented by, e.g., OJB [13]), a database is automatically generated from the classes that should be made persistent, together with the code needed to propagate object persistence to the database. These two approaches can be unsatisfactory, however, since they do not allow the persistent classes and the relational database to be created or change in an independent way. The O/R mapping does not allow using a database shared among several applications, whereas the R/O mapping prevents the developer from applying object-oriented skills in implementing the application logic layer, such as using design patterns.

The *meet-in-the-middle* approach is a further way for the transparent management of object persistence. This approach allows developers to manage the cases in which the application logic and the database have been developed and evolve in an independent way, since it 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 object schema and the relational schema, and are used by the persistence manager to let objects persist by means of the relational 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. Indeed, several object persistence managers are based on the O/R and R/O approaches. Furthermore, they manage only correspondences between similar structures (e.g., objects of a single class with tuples of a single relation). Some of them permit a meet between the object and database schemas, but only as a local tuning activity, after the schema translation from one model to the other.

This paper presents M^2ORM^2, a model to describe mappings (correspondences) between object schemas and relational schemas, to support the transparent management of object persistence based on the meet-in-the-middle approach. The goal of M^2ORM^2 is to generalize and extend the kinds of correspondences managed by currently available systems, thus allowing for more possibilities to meet schemas. Specifically, rather than considering only correspondences between single classes and single relations, as most systems do, in M^2ORM^2 it is possible to express complex correspondences between clusters of related classes (intuitively, each representing a single concept) and clusters of related relations. Furthermore, correspondences describing relationships between clusters can be expressed. With respect to other proposals, M^2ORM^2 takes into consideration specific details of the object and relational data models, together with the way in which objects and links are manipulated, to identify as many ways to meet schemas as possible.

M^2ORM^2 has been already introduced, in an informal way, in a previous paper by the same author [4]. There: (i) M^2ORM^2 has been presented in an informal way; (ii) the semantics of M^2ORM^2 has just been outlined only by means of a few examples; and (iii) the problem of identifying conditions for the correctness of M^2ORM^2 mappings has been just stated. The main contributions of this paper are: (i) the formalization of M^2ORM^2; (ii) a description of how operations on objects and links are realized as operations on the underlying relations; and (iii) the presentation of a number of necessary conditions for the correctness of M^2ORM^2 mappings.

For a comparison of M^2ORM^2 with the literature and a number of available systems for the transparent management of object persistence (e.g. [9–11, 13, 14]), we refer the reader to [4]. There, we have shown that M^2ORM^2 generalizes and extends the kinds of correspondences managed by various proposals and systems.

The paper is organized as follows. Section 2 proposes terminology and notation used to describe objects and relations. Section 3 presents M^2ORM^2, to describe mappings between schemas, together with an example. Section 4 presents the semantics of M^2ORM^2 mappings, by describing how operations on objects and links can be implemented as operations on an underlying relational database. Section 5 identifies a number of necessary conditions for the correctness of M^2ORM^2 mappings. Finally, in Sect. 6 we draw some conclusions.

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 [5] and of UML [3].

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. 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. An *object schema* is a set of classes and associations among such classes. Figure 1 shows a sample object schema. (For simplicity, in this paper we do not consider generalization/specialization relationships among classes, although M^2ORM^2 is able to manage them.)

At the instance level, a class is a set of *objects*. Each object has an associated *oid*, an unique identifier for referencing 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 *non null*. A *class with key* is a class in which an object can be identified

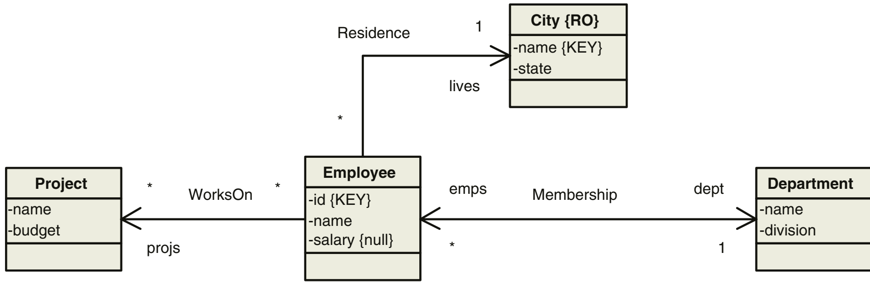

Fig. 1. An object schema

on the basis of the value of some of its attributes, called the *key attributes* of the class. Key attributes should be non null. A *read-only class* is a class from which it is not permitted to create new persistent objects and to modify or delete already existing objects. In an application, read-only classes are useful to access information generated by other applications. In Fig. 1, key attributes are denoted by constraint $\{KEY\}$ and attributes that can be null by 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, an occurrence of a class involved in the association. Roles have name, navigability, and multiplicity.

In the *relational model* [6], at the schema level a *relation* describes a set of tuples. A relation schema is a set of *attributes*. 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.

We consider the following integrity constraints. Attributes can be or not be *non null*. Each relation has a *key*. A *key attribute* is an attribute that belongs to a key; key attributes should be non 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 constraint $\{NK\}$ and those forming artificial keys by constraint $\{AK\}$. Referential constraints are denoted by arrows and implemented by means of attributes marked with constraint $\{FK\}$. Constraint $\{null\}$ denotes attributes that can be null.

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*), which allow the programmer to create persistent objects, read persistent objects (that is, performing a unique search of an object based on its key), as well as to modify and delete objects per-

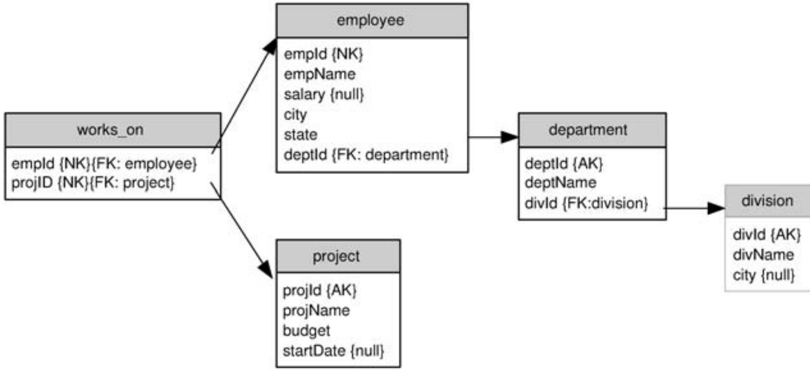

Fig. 2. A relational schema

sistently. Navigation, formation, breaking, and modification of persistent links between objects are also possible. Reading is meaningful only for classes with key. Creation, modification, and deletion is not meaningful for read-only classes. 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 operations 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 sections.

3 A Model for Object/Relational Schema Mappings

In this section we formally present M^2ORM^2, a model for describing mappings among object schemas and relational schemas. M^2ORM^2 is an acronym for *Meet-in-the-Middle Object/Relational Mapping Model*. The goal of M^2ORM^2 is to support the transparent management of relationally persistent objects based on the meet-in-the-middle approach.

Let $\mathbf{S}_r$ and $\mathbf{S}_o$ be a relational schema and an object schema, respectively. We assume that such schemas have been independently developed. In particular, both of them can be partially denormalized: the relational schema for efficiency reasons and the object schema to contain "coarse grain" objects.

A mapping $\mathcal{M}_{\mathbf{S}_r,\mathbf{S}_o}$ between $\mathbf{S}_r$ and $\mathbf{S}_o$ is a multi-graph $(\mathcal{N}, \mathcal{A})$, where $\mathcal{N}$ is a set of *nodes* and $\mathcal{A}$ is a set of *arcs* between them. Each node describes a correspondence between a group of classes and a group of relations. Each arc describes a relationship between the elements represented by a pair of nodes. Intuitively, using the Entity-Relationship terminology [1], each node represents an entity (which can be denormalized in one of the schemas) and each arc represents a binary relationship between two entities[1].

[1] In M^2ORM^2, an arc can also represent a generalization/specialization relationship between two entities. However, we ignore this aspect in this paper.

Each node $N \in \mathcal{N}$ is a triple $(\mathbf{C}, \mathbf{R}, \alpha)$, where:

- $\mathbf{C}$ is a *class cluster* (or *c-cluster*), that is, a non-empty tuple $\langle C_1, \ldots, C_n \rangle$ of classes and a set of associations among such classes[2];
- $\mathbf{R}$ is a *relation cluster* (or *r-cluster*), that is, a non-empty tuple $\langle R_1, \ldots, R_m \rangle$ of relations, together with referential constraints among them;
- α is a set of *attribute correspondences* (defined next) between $\mathbf{C}$ and $\mathbf{R}$.

In a c-cluster $\mathbf{C}$, one of the classes should be selected as the *primary class* of the c-cluster. The other classes of the c-cluster are *secondary* classes. The associations of the node should be, directly or indirectly, of type one-to-one or many-to-one from the primary class to secondary classes. Intuitively, such associations should relate, by means of navigable roles, each object of the primary class with at most an object from each of the secondary classes.

Similarly, in an r-cluster $\mathbf{R}$, one of the relations should be selected as the *primary relation* of the r-cluster. The other relations of the r-cluster, 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.

In a node $N = (\mathbf{C}, \mathbf{R}, \alpha)$, the correspondence among elements in $\mathbf{C}$ and elements in $\mathbf{R}$ is also specified by the set α of attribute correspondences. An *attribute correspondence* is a pair $(C_i.a, R_j.b)$, where C_i is a class in $\mathbf{C}$, a is an attribute of C_i, R_j is a relation in $\mathbf{R}$, and b is an attribute of R_j.

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

Figure 3 shows, in a graphical way, a $\mathrm{M}^2\mathrm{ORM}^2$ mapping between the schemas of Fig. 1 and 2. It is 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*.

More specifically, node $N_P = (\mathbf{C}_P, \mathbf{R}_P, \alpha_P)$ describes the correspondence between a class (*Project*) and a relation (*project*). The c-cluster $\mathbf{C}_P$ includes just the class *Project*, and the r-cluster $\mathbf{R}_P$ includes just the 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 Fig. 3. Node N_P describes a total correspondence between a

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

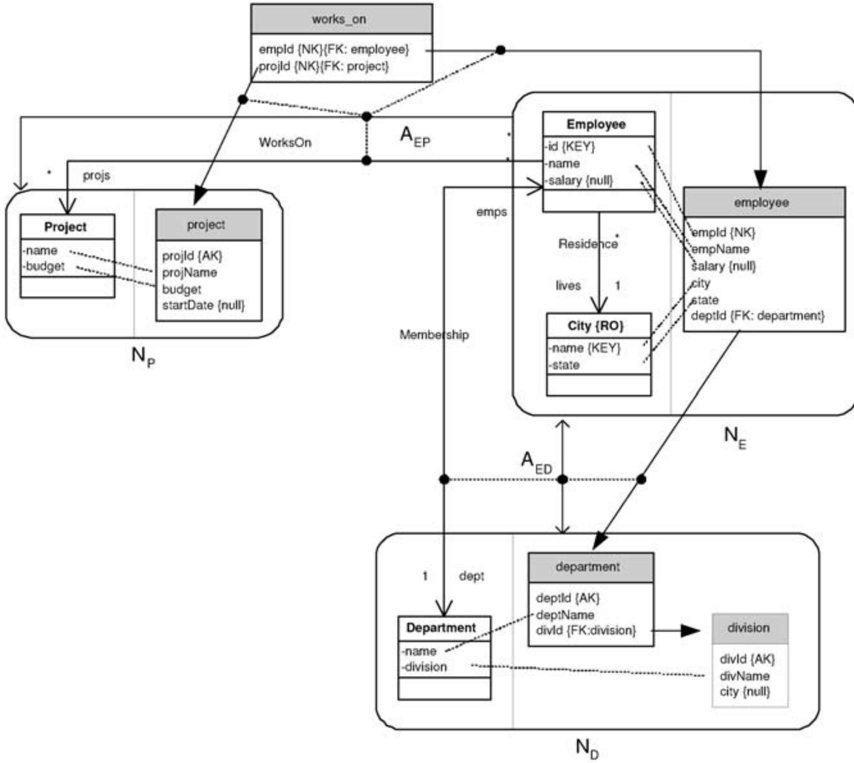

Fig. 3. A $\mathrm{M^2ORM^2}$ mapping between the schemas of Fig. 1 and 2

class and a relation: each object of the class is represented by means of a tuple of the relation.

As a more complex case, node $N_E = (\mathbf{C}_E, \mathbf{R}_E, \alpha_E)$ describes the correspondence between two classes (*Employee* and *City*) and a relation (*employee*). The primary class of the node is *Employee*. It is linked to the secondary class *City* through the navigable role *lives* of the association *Residence*. In this case, the c-cluster $\mathbf{C}_E$ includes both the classes and the association. 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 the relation *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, the attribute correspondences include the correspondence between the key attributes of the primary class and of the primary relation *(Employee.id, employee.empId)* as well as *(Employee.name, employee.empName)*, *(Employee.salary, employee.salary)*, *(City.name, employee.city)*, and *(City.state, employee.state)*.

Furthermore, 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 represents in the mapping the two relations together with the referential constraint between them.

A M^2ORM^2 mapping can also contain arcs. Intuitively, arcs represent correspondences and elements which cannot be represented by means of nodes; specifically, further associations, referential constraints, and some further relations. Each arc describes a relationship between two nodes, and can be of type *one-to-one*, *one-to-many*, or *many-to-many*.

Each arc $A \in \mathcal{A}$ is a tuple (N_1, N_2, γ, ρ), where:

- N_1 and N_2 are the nodes connected by the arc;
- γ is a *class correspondence* (defined next), which describes the correspondence between the primary classes of the nodes N_1 and N_2 connected by the arc;
- ρ is a *relation correspondence* (defined next), which describes the correspondence between the primary relations of the nodes N_1 and N_2 connected by the arc.

A *class correspondence* is based on the navigable roles of an association between the classes. In practice, at the programming level, roles are implemented by means of reference variables (for to-one navigability) and/or variables of collection types (for to-many navigability). If an association is unidirectional, then a single role is involved; otherwise, if it is bidirectional, there are two roles involved. A *relation correspondence* is based on the 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 a many-to-many binary relationship between the instances represented by a pair of nodes.

For example, the mapping shown in Fig. 3 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 $A_{ED} = (N_E, N_D, \gamma_{ED}, \rho_{ED})$ is of type one-to-many, and connects the nodes N_E and N_D to describe a one-to-many relationship between employees and departments. The class correspondence γ_{ED} is the pair *[Employee.dept, Department.emps]* between the classes *Employee* and *Department*, whereas the relation correspondence ρ_{ED} is the attribute *[employee.deptId]*, implementing a referential constraint between the relations *employee* and *department*. Intuitively, this arc lets the association *Membership* between *Employee* and *Department* correspond with the referential constraint between *employee* and *department*.

Arc A_{EP} between nodes N_E and N_P describes a many-to-many relationship between employees and projects. This arc is based on the (unidirectional) class correspondence *[Employee.projs]* between *Employee* and *Project* together

with the relation correspondence *[works_on.empId, works_on.projId]* between *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 tuples of the relation *works_on*.

4 Management of Operations

In the previous section, $\mathrm{M^2ORM^2}$ has been used as a syntactical tool to represent mappings. This section provides the semantics of $\mathrm{M^2ORM^2}$ mappings, by explaining how CRUD operations on an object schema can be implemented as operations on a relational schema, with respect to a $\mathrm{M^2ORM^2}$ mapping between the two schemas.

We consider sequences 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 object schema are satisfied. It is important to note that not every mapping is correct, that is, it is possible that some operation cannot be translated in a correct way. This happens, for example, if an integrity constraint over the relational schema is violated. Correctness of mappings will be discussed in Sect. 5.

We describe the management of operations with respect to the various kinds of nodes and arcs we can have in a $\mathrm{M^2ORM^2}$ mapping.

4.1 Operations on Nodes

Node Mapping a Class to a Relation. We first consider the simplest case, that is, a node N mapping a single class (say, C_1) to a single relation (say, R_1) by means of a set α of attribute correspondences. An example is node N_P. In this case, each object of the primary class C_1 is *represented by* a tuple of the primary relation R_1.

The *creation* of a new object o_1 of the class C_1 is implemented by the insertion of a new tuple t_1 in R_1, computed as follows:

- the attributes of t_1 involved in the attribute correspondences α take their value from the corresponding attributes of o_1, that is, $t_1.b = o_1.a$ if $(C_1.a, R_1.b) \in \alpha$;
- if R_1 has an artificial key, then a new key for t_1 is generated by the system;
- the other attributes of t_1 defaults to *null*.

If the class C_1 has a key, then the *reading* of an object from C_1 by means of a value k for its key is implemented by a query searching for a single tuple t_1 in R_1. The selection condition of the query equates the attributes of the key of R_1 with the corresponding values in k. If there is already an object in memory representing this tuple, then such object is returned. Otherwise, a new object o_1 of the class C_1 is created in memory; the value for the attributes of o_1 are computed as follows:

- the attributes of o_1 involved in the set α of attribute correspondences take their value from the corresponding attributes of t_1, that is, $o_1.a = t_1.b$ if $(C_1.a, R_1.b) \in \alpha$;
- the other attributes of o_1 defaults to *null*.

The *update* of a non-key attribute a of an object o_1 of the class C_1 is implemented as follows, with respect to the tuple t_1 that represents o_1 in R_1:

- if the attribute a is *transient*, that is, if a is not involved in the attribute correspondences, then t_1 is not changed;
- otherwise, if the attribute a is involved in the attribute correspondences, the attributes that correspond to a are modified in the tuple t_1.

The *deletion* of an object o_1 of the class C_1 is implemented as the deletion of the tuple t_1 that represents o_1 in R_1.

Node Mapping Many Classes to a Relation. We now consider the case of a node mapping two or more classes (say, $\langle C_1, \ldots, C_n \rangle$), together with a number of associations among them, to a single relation (say, R_1), on the basis of a set α of attribute correspondences. We assume that C_1 is the primary class of the node. In such a node, each object o_1 of the primary class C_1 identifies, by means of the associations in the node, at most an object in each secondary class of the node; let us call $o_2, \ldots, o_n$ these objects, where each object o_i belongs to class C_i. In this case, the tuple of objects $\langle o_1, \ldots, o_n \rangle$ is *represented by* a single tuple t_1 of the relation R_1. An example is node N_E; each tuple of the relation *employee* represents an object o_e of *Employee* and, possibly, an object o_c of *City* and a link of type *Residence* from o_e to o_c.

More specifically, the associations in the node should have, directly or indirectly, multiplicity 1 (one, mandatory) or 0..1 (one, optional) from the primary class to secondary classes. We denote by $\mathit{mandatory}(C, N)$ the set of classes that can be reached, directly or indirectly, from a class C by means of the navigable roles of the associations in the node N that have multiplicity 1 (not 0..1). Furthermore, given an object o, we denote by $\mathit{reachable}(o, N)$ the set of objects that can be effectively reached, directly or indirectly, from the object o by means of the navigable roles of the associations in the node N; we assume that $o \in \mathit{reachable}(o, N)$ as well.

The *creation* of a new object o_1 in the primary class C_1, linked to the already existing objects $\mathit{reachable}(o_1, N)$, is implemented by the insertion of a new tuple t_1 in R_1, computed as follows:

- the attributes of t_1 involved in the set α of attribute correspondences take their value from the corresponding attributes of $\mathit{reachable}(o_1, N)$, that is, if $(C_i.a, R_1.b) \in \alpha$ and there is an object o_i of the class C_i in $\mathit{reachable}(o_1, N)$, then $t_1.b = o_i.a$;
- if R_1 has an artificial key, then a new key for t_1 is generated by the system;
- the other attributes of t_1 defaults to *null*.

The creation of new objects in secondary classes of the node does not require the modification of the relation R_1. Indeed, these new objects are considered *transient*, at least until they will not be connected to an object of the primary class of the node. This possibility will be considered later, in the context of operations on links.

If the primary class C_1 has a key, then the *reading* of an object from C_1 by means of a value k for its key is implemented by a query searching for a single tuple t_1 in R_1, as in the case for a node mapping a class to a relation. The selection condition of the query equates the attributes of the key of R_1 with the corresponding values from k. If we retrieve a tuple t_1, then it represents an object o_1 of class C_1 together with the set $\mathit{reachable}(o_1, N)$ of objects reachable from o_1. Then, if they do not already exists in memory, we should create such objects in memory, together with the links implied among them. Objects for the classes in $\mathit{mandatory}(C_1, N)$ are always created. Objects in other classes participating to the node are created only if the corresponding attributes in t_1 are not null.

More specifically, if there are no objects in memory representing the various parts of the tuple t_1, then the following new objects are created in memory, together with the value for their attributes:

- a new object o_i in a class C_i in $\mathit{mandatory}(C_1, N)$; the attributes of o_i involved in the attribute correspondences α take their value from the corresponding attributes of t_1, that is, $o_i.a = t_1.b$ if $(C_i.a, R_1.b) \in \alpha$; the other attributes of o_i defaults to *null*;
- a new object o_j in a class C_j that does not belong to $\mathit{mandatory}(C_1, N)$, if all the non null attributes of C_j in the attribute correspondences α have a non null value in the corresponding attributes of t_1; in this case, such attributes take their value from the corresponding attributes of t_1, that is, $o_j.a = t_1.b$ if $(C_j.a, R_1.b) \in \alpha$; the other attributes of o_j defaults to *null*.

Furthermore, links for the associations in the node are formed in memory among the objects involved in this operation.

Let $o_1, \ldots, o_n$ be a tuple of objects represented by a tuple t_1 in R_1, as described by this node. Then, the *update* of a non-key attribute a of an object o_i of a class C_i that can be reached from an object o_1 is managed by modifying, if it is the case, the attributes corresponding to a of the tuple t_1 that represents the object o_1 and the associated objects $o_2, \ldots, o_n$, as follows:

- if the attribute a is transient (that is, is not involved in the attribute correspondences), then the tuple t_1 is not modified;
- otherwise, if the attribute a is involved in the attribute correspondences, the attributes that correspond to a are modified in the tuple t_1.

Let $o_1, \ldots, o_n$ be a tuple of objects represented by a tuple t_1 in R_1, as described by this node. The *deletion* of the object o_1 of the primary class C_1 is implemented as the deletion of the tuple t_1 that represents o_1 in R_1. Note that some of the objects $o_2, \ldots, o_n$ may become transient.

The modification of links of associations involved in the node should be considered as well. For example, the formation of a new link outgoing from an object

o_i of a class C_i involved in the node, whose type is an association involved in the node, may increase the number of objects reachable from o_i and, transitively, those reachable from an object o_1 of the primary class C_1 of the node. In this case, the tuple t_1 representing o_1 should be modified accordingly (by modifying null attributes to the new values owned by the new reachable objects). Conversely, if a link outgoing from o_i is broken, the number of objects reachable for o_1 may decrease. In this case, the tuple t_1 representing o_1 should be modified accordingly (by modifying to null attributes representing objects that are no more reachable). In practice, the tuple t_1 representing o_1 should be modified as the object o_1 would have been created after the link formation/modification/breaking.

Node Mapping a Class to Many Relations. We now consider the case of a node mapping a class (say, C_1) to two or more relations (say, $\langle R_1, \ldots, R_m \rangle$), together with referential constraints among them, on the basis of a set α of attribute correspondences. We assume that R_1 is the primary relation of the node. In such a node, each tuple t_1 of the primary relation R_1 identifies, by means of the referential constraints in the node, at most a tuple in each secondary relation of the node; let us call $t_2, \ldots, t_M$ these tuples, where each tuple t_i belongs to the relation R_i. In this case, the tuple of tuples $\langle t_1, \ldots, t_m \rangle$ *represents* an object o_1 of the class C_1. An example is node N_D; each object of the class *Department* is represented by a tuple t_{dept} of the relation *department* and, possibly, a tuple t_{div} of the relation *division*, where the tuple t_{dept} references the tuple t_{div}.

More specifically, the node contains referential constraints that, directly or indirectly, reference the secondary relations from the primary relations. A referential constraint is mandatory if the attribute implementing it cannot be null, but is optional if such attribute can be null. We denote by $\mathit{mandatory}(R, N)$ the set of relations that can be reached, directly or indirectly, from a relation R by means of the mandatory referential constraints in the node N. Furthermore, given a tuple t, we denote by $\mathit{reachable}(t, N)$ the set of tuples that can be effectively reached, directly or indirectly, from the tuple t by means of referential constraints in the node N; we assume that $t \in \mathit{reachable}(t, N)$ as well.

The *creation* of a new object o_1 in the class C_1 is implemented by the insertion of the tuples $t_1, \ldots, t_m$ in the relations $R_1, \ldots, R_m$, as follows:

- a tuple t_i in a relation R_i in $\mathit{mandatory}(R_1, N)$; the attributes of t_i involved in the attribute correspondences α take their value from the corresponding attributes of o_1, that is, $t_i.b = o_1.a$ if $(C_1.a, R_i.b) \in \alpha$; if R_i has an artificial key, then a new key for t_i is generated by the system;
- a tuple t_j in a relation R_j that does not belong to $\mathit{mandatory}(R_1, N)$, if all the non null attributes of R_j in the attribute correspondences α of the node have a non null value in the corresponding attributes of o_1; in this case, such attributes take their value from the corresponding attributes of o_1, that is, $t_j.b = o_1.a$ if $(C_1.a, R_j.b) \in \alpha$; if R_j has an artificial key, then a new key for t_j is generated by the system;
- if a tuple t_i should reference a tuple t_h of a secondary relation R_h by means of an attribute b, then $t_i.b$ equals the key of the tuple t_h;
- the other attributes of the tuples defaults to *null*.

If the primary class C_1 has a key, then the *reading* of an object from C_1 by means of a value k for its key is implemented by a query searching for a single tuple t_1 in the primary relation R_1, as in the case for a node mapping a class to a relation. The selection condition of the query equates the attributes of the key of R_1 with the corresponding values from k. Then, the retrieval of a tuple t_1 in R_1 is followed by the lookup of the tuples in the secondary relations that are reachable from t_1, so that all the tuples $reachable(t_1, N)$ representing an object of the class C_1 are retrieved. If there is no object in memory representing these tuples, then a new objects o_1 of the class C_1 is created in memory; the value for the attributes of o_1 are computed as follows:

- the attributes of o_1 involved in the attribute correspondences α take their value from the corresponding attributes of the tuples in $reachable(t_1, N)$, that is, $o_1.a = t_i.b$ if $(C_1.a, R_i.b) \in \alpha$;
- the other attributes of o_1 defaults to *null*.

Let o_1 be an object of the class C_1 represented by the tuples $t_1, \ldots, t_m$, as described by this node. Then, the *update* of a non-key attribute a of the object o_1 is managed as follows:

- if the attribute a is not involved in the attribute correspondences (i.e, it is transient), then no tuple is not modified;
- if the attribute a is involved in the attribute correspondences and is related to attributes of the primary relation R_1, then the attributes that correspond to a are modified in the tuple t_1;
- otherwise, if the attribute a is involved in the attribute correspondences and is related to attributes of a secondary relation R_i, then a new set of tuples is computed, as for the creation of a new object, to represent the object o_1, and, if it is the case, such tuples are added to the secondary relations of the node and the tuple t_1 is modified accordingly.

Let o_1 be an object of the class C_1 represented by the tuples $t_1, \ldots, t_m$, as described by this node. The *deletion* of the object o_1 is implemented as the deletion of the tuple t_1 that represents o_1 in the primary relation R_1. Note that the other tuples $t_2, \ldots, t_m$ may become useless; in such a case, they can be deleted as well.

4.2 Operations on Arcs

Arcs are used to represent binary relationships between nodes. In the object schema, each arc represents one or both the roles of an association between the primary classes of the two nodes. In the relation schema, each arc represents one or two referential constraints between the primary relations of the two nodes. The referential constraints may be either embedded in the relations that are already involved in the nodes or they may involve a different relation.

Suppose that A_{xy} is an arc from a node N_x to a node N_y, representing the roles r_y and r_x of an association a_{xy} from the primary class C_x of N_x to the primary class C_y of N_y (r_x references an object of the class C_y and r_y references an object of the class C_x).

Arc Embedded in Already Involved Relations. If arc A_{xy} is embedded in already involved relations, then the two roles r_x and r_y of the association correspond to two attributes (say, b_x and b_y) of the primary relations R_x of N_x and R_y of N_y, each of which references the primary relation of the other node.

The *creation* of a new link of the association a_{xy} from an object o_x of the class C_x to an object o_y of the class C_y is managed as follows:

– if the multiplicity of r_x is 0..1, then the attribute b_y of the tuple t_y of the relation R_y that represents o_y takes its value from the key of the tuple t_x of the relation R_x that represents o_x;
– if the multiplicity of r_y is 0..1, then the attribute b_x of the tuple t_x of the relation R_x that represents o_x takes its value from the key of the tuple t_y of the relation R_y that represents o_y.

The *breaking* of a link of the association a_{xy} from an object o_x of the class C_x to an object o_y of the class C_y is managed as follows:

– if the multiplicity of r_x is 0..1, then the attribute b_y of the tuple t_y of the relation R_y that represents o_y becomes null;
– if the multiplicity of r_y is 0..1, then the attribute b_x of the tuple t_x of the relation R_x that represents o_x becomes null.

The *modification* of a link of the association a_{xy} starting from an object o_x of the class C_x, from the object o_y to the object o'_y of the class C_y is managed as follows:

– if the multiplicity of r_x is 0..1, then the attribute b_y of the tuple t_y of the relation R_y that represents o_y becomes null;
– if the multiplicity of r_y is 0..1, then the attribute b_x of the tuple t_x of the relation R_x that represents o_x takes its value from the key of the tuple t'_y of the relation R_y that represents o'_y.

Arc Represented by a Different Relation. If arc A_{xy} is represented by a relation different from the primary relations of the nodes, then the two roles of the association correspond to two attributes (say, b_x and b_y) of a different relation (say, R_{xy}); these attributes reference the primary relations R_y and R_x of the two nodes N_y and N_x, respectively.

The *creation* of a new link of the association a_{xy} from an object o_x of the class C_x to an object o_y of the class C_y is managed as the insertion of a tuple t_{xy} in the relation R_{xy}, computed as follows:

– the attribute b_x of the tuple t_{xy} takes its value from the key of the tuple t_y of the relation R_y that represents o_y;
– the attribute b_y of the tuple t_{xy} takes its value from the key of the tuple t_x of the relation R_x that represents o_x;
– the other attributes of the tuple t_{xy} defaults to *null*.

The *breaking* of a link of the association a_{xy} from an object o_x of the class C_x to an object o_y of the class C_y is managed as the deletion of the tuple t_{xy} that represents the link between the two objects.

The *modification* of a link of the association a_{xy} starting from an object o_x of the class C_x, from the object o_y to the object o'_y of the class C_y is managed as the modification of the tuple t_{xy} in the relation R_{xy} that represents the link between o_x and o_y, as follows:

– the attribute b_x of the tuple t_{xy} takes its value from the key of the tuple t'_y of the relation R_y that represents o'_y.

5 Correctness of Mappings

It turns out that not every mapping that can be described using $\mathrm{M}^2\mathrm{ORM}^2$ is correct. Intuitively, a $\mathrm{M}^2\mathrm{ORM}^2$ mapping is *correct* if it supports, in an effective way, the management of CRUD operations on objects and links by means of the underlying relational schema. Otherwise, a mapping is *incorrect* if operations on objects and links can give rise to anomalies. There are various kinds of anomalies, but only two main causes for them: (i) incorrect correspondences between elements and (ii) incorrect representation of integrity constraints.

We now present a number of conditions for the correctness of a $\mathrm{M}^2\mathrm{ORM}^2$ mapping. These conditions are *necessary*: they should hold whenever the corresponding operations have to be managed on the basis of the mapping. The need for these conditions can be verified by means of counter examples and by referring to the semantics of $\mathrm{M}^2\mathrm{ORM}^2$ described in Sect. 4.

In what follows, consider a node $N = (\mathbf{C}, \mathbf{R}, \alpha)$, where $\mathbf{C} = \langle C_1, \ldots, C_n \rangle$, $\mathbf{R} = \langle R_1, \ldots, R_m \rangle$, C_1 is the primary class of the node, and R_1 is the primary relation of the node.

To implement the creation of a new object in the primary class C_1 of the node correctly, the following conditions are necessary:

[C1] If the primary relation R_1 of the node has a natural key, then it corresponds to the key of the primary class C_1 of the node; if R_1 has an artificial key, then it is not involved in the attribute correspondence α.

[C2] There are no two attributes of the classes in the node corresponding to a same attribute of a relation R_i in the node.

[C3] If an attribute of a relation R_i that is the primary relation of the node or that belongs to $mandatory(R_1, N)$ (where R_1 is the primary relation of the node) cannot be null, then either it is the artificial key of R_i, or it is a reference to another relation $R_j \in mandatory(R_1, N)$, or it is involved in an attribute correspondence in α and corresponds to an attribute of a class C_j of the node that cannot be null, where C_j is the primary class of the node or belongs to $mandatory(C_1, N)$ (where C_1 is the primary class of the node).

[C4] Each secondary relation R_i of the node has an artificial key, which is not involved in the attribute correspondence α.

Condition **[C1]** ensures that the key of the primary relation R_1 of the node has a unique value. Condition **[C2]** ensures that each attribute of a relation R_i in the node has at most one value. Condition **[C3]** ensures that each non null attribute of a mandatory relation R_i in the node has a non null value.

To implement the reading of an object of the primary class of the node C_1 correctly, the following conditions are necessary:

- **[R1]** The primary relation R_1 of the node has a natural key, and the attribute correspondences let it correspond to the key of the primary class C_1.
- **[R2]** There are no two attributes of the relations in the node corresponding to a same attribute of a class C_i in the node.
- **[R3]** If an attribute of a class C_i that is the primary class of the node or that belongs to $mandatory(C_1, N)$ (where C_1 is the primary class of the node) cannot be null, then it is involved in an attribute correspondence in α and corresponds to an attribute of a relation R_j that cannot be null, where R_j is the primary relation of the node or belongs to $mandatory(R_1, N)$ (where R_1 is the primary relation of the node).

Condition **[R1]** ensures that at most one tuple is retrieved from the primary relation R_1. Condition **[R2]** ensures that each attribute of a class C_i in the node has at most one value. Condition **[R3]** ensures that each non null attribute of a mandatory class C_i in the node has a non null value.

We now consider the update of a non transient attribute a of a class C_i of the node; we suppose that the attribute a of the primary class C_1 is related to an attribute b of a relation R_i of the node by the attribute correspondences α. We have the following necessary conditions:

- **[U1]** Attribute b is not part of the key of the primary relation R_1 of the node.
- **[U2]** If R_i is the primary class of the node or it belongs to $mandatory(R_1, N)$ (where R_1 is the primary class of the node) and b cannot be null, then a cannot be null as well.
- **[U3]** There cannot be another attribute a' of a class C_j in the node related to b by the attribute correspondences α.

If the node N is of type one class/many relations, then the conditions for the creation of a new object (**[C1]**, **[C2]**, **[C3]**, and **[C4]**) are also required.

Condition **[U1]** avoids that the primary key of a relation is changed. Condition **[U2]** deals with null values. Condition **[U3]** prevents side effects on the attributes of the objects.

To implement the deletion of an object of the primary class C_1 of the node, no additional necessary conditions are required. Indeed, if the object has been created or read, it can also be safely deleted.

Necessary conditions for a correct management of the modification of a link of an association involved in a node are the same for the creation of a new object of the primary class of the node.

Before concluding the section, it is worth noting that, in practice (that is, for eventually implementing a persistence manager based on $\mathrm{M^2ORM^2}$) *sufficient*

conditions should be fixed as well, to manage mappings in an effective way. Current systems suffer from several limitations, since the 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 — permissive with respect to significance of mappings. Please note that we do not state sufficient conditions in this paper.

6 Discussion

In this paper we have presented M^2ORM^2, a model to describe correspondences between object schemas and relational schemas. The goal of M^2ORM^2 is supporting the transparent management of object persistence based on the meet-in-the-middle approach.

As future work, we plan to implement a Java framework for the management of persistent objects based on M^2ORM^2. To this end, a number of object-oriented techniques should be used, e.g., those in [7].

From a theoretical perspective, we plan to study several extensions to the model, e.g., the management of generalization/specialization hierarchies, transient classes and attributes, and of further integrity constraints. We also plan to study in a more precise way the correctness of M^2ORM^2 mappings, and specifically to state sufficient conditions that are as permissive as possible.

References

1. C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach*. Benjamin-Cummings, 1992.
2. K. Beck. *EXtreme Programming EXplained: Embrace Change*. Addison-Wesley, 1999.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
4. L. Cabibbo and R. Porcelli. M^2ORM^2: A Model for the Transparent Management of Relationally Persistent Objects. In *International Workshop on Database Programming Languages (DBPL)*, pages 166–178, 2003.
5. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
6. R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2003.
7. M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003.
8. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
9. Java Data Objects. http://www.jdocentral.com
10. JDX. http://www.softwaretree.com/
11. JRELAY. http://www.objectindustries.com/
12. C. Larman. *Applying UML and Patterns. An introduction to object-oriented analysis and design and the Unified Process*. Prentice Hall PTR, 2002.
13. ObJect relational Bridge. http://db.apache.org/ojb/
14. Torque. http://db.apache.org/torque/