Decomposition of Relationships through Pivoting*

Joachim Biskup[1], Ralf Menzel[1], Torsten Polle[1] and Yehoshua Sagiv[2]

[1] Institut für Informatik, Universität Dortmund, Germany
[2] Department of Computer Science, Hebrew University of Jerusalem, Israel

Abstract. In the literature there are several proposals to map entity-relationship schemas onto object-oriented schemas, but they only treat relationship sets naïvely or restrict them to binary relationship sets. We follow a different approach in the treatment of relationship sets. Our goal is to let the designer specify relationships of any arity and then to employ semantic constraints to decompose relationships into smaller fragments. The semantic constraints in use are functional constraints, which are defined in the object-oriented framework. The decomposition process guided by functional constraints is similar to the decomposition process in the relational approach with functional dependencies, but it takes advantage of the features provided by the object-oriented data model. In object-oriented schemas it is possible to enforce a certain kind of functional constraints automatically, namely unary functional constraints.

1 Introduction

The Entity-Relationship approach (ER approach) to modelling data, as proposed by Chen [7], is a simple way of representing data in the form of entities and relationships among entities. Because of its simplicity, it has found a wide spread acceptance and serves as starting point for transformations into different data models. One of the target data models is the object-oriented model (OO model) [3, 1, 14], and there are numerous proposals to map ER-schemas, their variations or different conceptual schemas onto object-oriented schemas [12, 19, 11, 20, 15, 18, 4]. The basic idea of these transformations is to map entity sets onto (entity) classes in the object-oriented data model and relationship sets, also called associations, onto (relationship) classes. This mapping seems unnatural in the object-oriented framework, when it comes to n-ary relationships. Therefore the common approach is to restrict relationships to binary relationships as, e.g. in [15], or to nest relationships [18], which clears the path to represent these relationships by means of methods. Another way is to make relationships first class citizens of the object-oriented data model as it is done in the object-relation model [21]. But even there it is claimed that in practice only binary relationships or special ternary relationships occur. The difficulties to decompose ternary or even n-ary relationships into binary relationships is discussed by Thalheim [22].

* Appeared in: Proc. 15th International Conference on Conceptual Modeling, LNCS 1157, Springer-Verlag, Berlin 1996, 28-41.

We follow a different approach in the treatment of relationship classes. Our goal is to let the designer specify relationships of any arity in the conceptual model, i.e., the ER model, and then to employ semantic constraints given by the designer to decompose the corresponding relationship classes losslessly into smaller fragments in the object-oriented data model. For this decomposition we use mainly functional dependencies, which are defined in the object-oriented data model. The difference in the decomposition in this setting to the decomposition in the relational approach is that we use the features of the object-oriented data model. The decomposition relies heavily on a transformation called pivoting. This transformation is a special case of pivoting introduced in [4], and hence called *property pivoting*. Since we do not refer to (general) pivoting in this article, we use the term pivoting meaning property pivoting.

This paper is organised as follows. In Sect. 2 a simple object-oriented data model is introduced. The transformation pivoting used at the core of the decomposition is presented in Sect. 3. We start with briefly outlining the mapping from ER-schemas to OO-schemas and then give a simple definition of pivoting and later on extend it to recursive pivoting, leading to a transformation suitable to decompose relationships. Finally, we discuss the interplay between the behaviour of pivoting and the characteristics of semantic constraints, which guide the transformation process. Here we put special emphasis on the features of object-oriented models influencing the decomposition process.

2 Object-Oriented Data Model

2.1 Overview

In this section we present the basic terms relevant to our notion of an object-oriented data model, concentrating only on the *main* concepts that are important for our transformations. This is in particular the capability to reference objects. Therefore our type system is only rudimentarily developed. The data model does not even contain inheritance, although it is needed for the transformations. The reason is that inheritance is only required for technical purpose, and therefore we decided to leave it out to facilitate the presentation. This simplification shifts our data model closer to the network model and even to the relational model, but still these lack the concept of an object identifier, which is used for the reference mechanism, and hence essential for pivoting. In Sect. 2.2 we will give an good sized example for this model.

Our idea of a database is that it consists of two parts. One part, the database schema, is relatively stable over the time. It is used to describe the structural part of applications.

Definition 1. A *(database) schema* D consists of a finite set of *class schemes* of the form

$$c\{p_1 : c_1, \ldots, p_n : c_n\}F.$$

c is the name of the class scheme. Each *property name*[3] p_i is unique in a given class scheme. Its type, written $\mathrm{Ran}_D(c, p_i)$, is the name c_i of another class scheme. The set of names of class schemes in D is written $\mathrm{Class}(D)$, and the set of property names occurring in the definition of a particular class scheme with name c is written $\mathrm{Props}(c)$. The set F consists of

- *functional constraints* of the form $c(m_1 \cdots m_n \rightarrow m_{n+1} \cdots m_o)$ where $1 \leq n < o$ and where $m_i \in \mathrm{Props}(c) \uplus \{\mathrm{Id}\}$[4], for $1 \leq i \leq o$ and,
- *(range) completeness constraints* of the form $c\{m\}$ where $m \in \mathrm{Props}(c)$.

Functional constraints play a similar rôle in our data model as functional dependencies in the relational data model, where they ensure that tuples agree on the values of the attributes on the right-hand side, whenever they have the same values for the attributes on the left-hand side. So functional constraints ensure that objects agree on the values of the properties on the right-hand side, whenever they have the same values for properties on the left-hand side. Functional constraints correspond to Weddell's path functional dependencies [23], where the length of all paths is not greater than one. If the property Id occurs on the right-hand side of a functional constraint, this functional constraint is called a *key constraint*. If in the relational model a set of attributes forms a key, the values for these key attributes uniquely determine a tuple. Key constraints are used if a set of property values is to uniquely determine an object.

A (range) completeness constraint for a property states that all objects of the type of the property are referenced by an object through this property.

The other part of a database is a *database instance*. It describes the time-varying part of a database, and is used to understand formally that semantic constraints are satisfied. We found it natural and intuitive to think of objects as vertices and of property values as edges in a graph, following the approach of Beeri [3].

Definition 2. An *(database) instance* of a database schema D is a directed graph $G(V, E)$ with vertex and edge labelling as class and property names respectively. G must also satisfy the following constraints, where the class name label of a vertex v is denoted $l_{Cl}(v)$.

1. *(property value integrity)* If $u \xrightarrow{p} v \in E$, then $p \in \mathrm{Props}(l_{Cl}(u))$ and $l_{Cl}(v) = \mathrm{Ran}_D(l_{Cl}(u), p)$.
2. *(property functionality)* If $u \xrightarrow{p} v, u \xrightarrow{p} w \in E$, then $v = w$.
3. *(property value completeness)* If $u \in V$, then there exists $u \xrightarrow{p} v \in E$ for all $p \in \mathrm{Props}(l_{Cl}(u))$.

[3] We will often use the terms property and class instead of property name and class name.

[4] Id is the identity property. We assume that it does not correspond to the name of any property in D, and furthermore $\mathrm{Ran}_D(c, \mathrm{Id}) := c$ for all classes $c \in \mathrm{Class}(D)$. It is used to refer to the object itself. This is necessary for so-called *key constraints*.

Property value integrity ensures that property values are of the type given in the corresponding database schema. Property functionality ensures that properties are scalar, i.e., single-valued. Property value completeness forbids *null values*, i.e., the property value for an object must always be defined. We can construct a database instance out of a database schema. For the schema depicted in Fig. 2 this is done in Fig. 3.

If we have an object, i.e., a vertex, in an instance, and a property that is defined for the class name label of the object, we can reach another object by following the edge labelled by the property. If the property is the identity property Id, we arrive at the original object again.

Definition 3. Let $G(V, E)$ be an instance of schema D, $u \in V$ be a vertex, and $p \in \mathrm{Props}(l_{Cl}(u)) \cup \{\mathrm{Id}\}$ be a property. Then $u.p$ denotes the vertex u if $p = \mathrm{Id}$ and the vertex v, where $u \xrightarrow{p} v \in E$, if $p \in \mathrm{Props}(l_{Cl}(u))$.

Definition 4. A functional constraint $c(p_1 \cdots p_n \rightarrow p_{n+1} \cdots p_o)$ over a schema D is *satisfied* by an instance $G(V, E)$ for D iff for any pair of vertices $u, v \in V$, where $l_{Cl}(u) = l_{Cl}(v) = c$, $u.p_i = v.p_i$, $1 \leq i \leq n$ implies $u.p_j = v.p_j$, $n < j \leq o$.

Definition 5. A completeness constraint $c\{p\}$ over a schema D is *satisfied* by an instance $G(V, E)$ for D iff for any vertex v, where $l_{Cl}(v) = \mathrm{Ran}_D(c, p)$, there is an incoming edge $u \xrightarrow{p} v \in E$ and $l_{Cl}(u) = c$.

As in the relational data model, it is also possible to define the *logical consequences* of a set of functional constraints and give a sound and complete derivation system for the implication of functional and completeness constraints.

By means of this derivation system we can calculate for a class scheme the set of properties uniquely determined by a given set of properties. This set of properties is called the *closure of X under F*, denoted $\mathrm{Cl}_F(X)$, for a given set X of properties and a given set F of functional constraints.

2.2 Example

A designer should be supported to focus on the essential parts of the applications in the conceptual design phase, i.e., he has to find out what are the vital things and associations among them that constitute the application and their abstractions. He should not be burdened to deal with restrictions or to take the characteristics of further steps into account, i.e., to break relationships into smaller ones.

We give in Fig. 1 an example of a conceptual schema an experienced designer would intuitively tend to model with smaller relationships. The ER-diagram reflects the Assignment from Teachers and Assistants to Courses in combination with the Date they take place at, and Rooms and Wings they are given in. The object-oriented database schema (displayed in Fig. 2) is obtained by the mapping sketched in Sect. 3. The semantic constraints are added later in a refinement of the original conceptual schema. They can already be declared in the conceptual schema, but we refrained from doing so because it would overload the diagram.

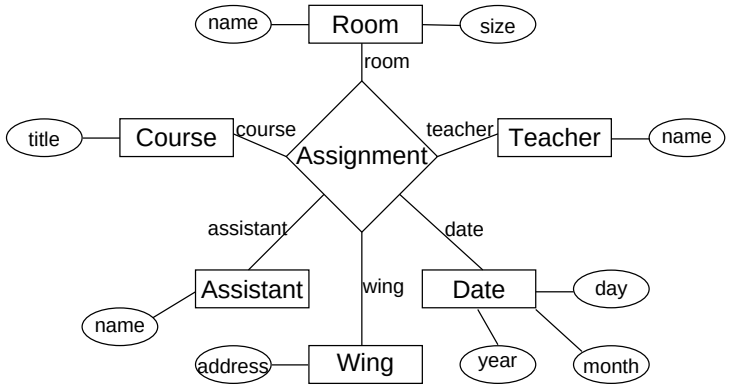

Fig. 1. ER-diagram

$$
\begin{array}{l}
\mathsf{Course}\{\mathsf{title} : \mathsf{String}\}\{\mathsf{Course}(\mathsf{title} \rightarrow \mathrm{Id})\} \\
\mathsf{Teacher}\{\mathsf{name} : \mathsf{String}, \ldots\}\{\mathsf{Teacher}(\mathsf{name} \rightarrow \mathrm{Id})\} \\
\mathsf{Assistant}\{\mathsf{name} : \mathsf{String}, \ldots\}\{\mathsf{Assistant}(\mathsf{name} \rightarrow \mathrm{Id})\} \\
\mathsf{Date}\{\mathsf{year} : \mathsf{Int}, \mathsf{month} : \mathsf{Int}, \mathsf{day} : \mathsf{Int}\}\{\mathsf{Date}(\mathsf{year\ month\ day} \rightarrow \mathrm{Id})\} \\
\mathsf{Room}\{\mathsf{size} : \mathsf{Int}\}\{\} \\
\mathsf{Wing}\{\mathsf{address} : \mathsf{String}\}\{\} \\
\mathsf{Assignment}\{\mathsf{course} : \mathsf{Course}, \mathsf{assistant} : \mathsf{Assistant}, \mathsf{date} : \mathsf{Date}, \mathsf{teacher} : \mathsf{Teacher}, \\
\qquad \mathsf{room} : \mathsf{Room}, \mathsf{wing} : \mathsf{Wing}\} \\
\qquad \{\mathsf{Assignment}(\mathsf{course} \rightarrow \mathsf{assistant\ date}), \\
\qquad\ \mathsf{Assignment}(\mathsf{teacher} \rightarrow \mathsf{room}), \\
\qquad\ \mathsf{Assignment}(\mathsf{room} \rightarrow \mathsf{wing}), \\
\qquad\ \mathsf{Assignment}(\mathsf{course\ teacher} \rightarrow \mathrm{Id})\} \\
\mathsf{Int}\{\}\{\} \quad \mathsf{String}\{\}\{\}
\end{array}
$$

Fig. 2. Database schema

The semantic constraints declared for the class scheme Assignment are of main interest. For every Course there is exactly one Assistant and it takes place at only one Date. The functional constraint Assignment(course → assistant date) enforces that this restriction is met. Imagine that in the application at hand a Teacher is assigned a fixed Room, then the semantic constraint Assignment(teacher → room) ensures just this requirement. Additionally, the constraint that a Room is situated in only one Wing is reflected in the functional constraint Assignment(room → wing).

An instance for the schema in Fig. 2 is depicted in Fig. 3. Here we took the approach to view the schema as an instance. This is quite easily accomplished by

making every class scheme name an object of itself, yielding an abstract instance. Then for every property in a class scheme we introduce an edge labelled with the property going from the vertex corresponding to the class scheme to the vertex corresponding to the property type. Viewing schemas as instances helps us to display the schema as a graph. Therefore we will subsequently present schemas as instances in this paper. To simplify the representation further, we displayed some of the vertices several times instead of once, e.g., the vertex with label String.

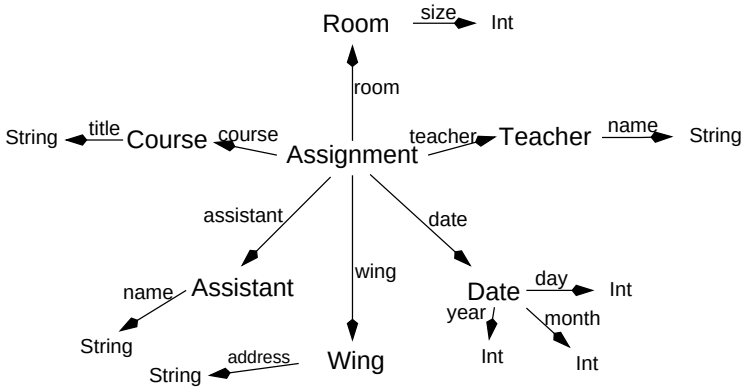

Fig. 3. Database instance

3 Object-Oriented Formalisation of Relationships

3.1 Mapping from ER to OO

For the design of object-oriented schemas, entity sets are simply formalised as (entity) classes the basic types of which are determined by the pertinent properties of the entities. In order to formalise a relationship set, we can always canonically simulate the relational approach [12, 11, 18, 21]. For every individual relationship we construct an object that is counterpart to the corresponding tuple in the relational approach. Then these objects are understood as instances of a (relationship) class. As objects correspond in this case to tuples we also need *canonical semantic constraints* to ensure that they behave as such. These constraints require that the property values of a relationship object uniquely represent the relationship, i.e., there is at most one object for any value combination. This kind of constraint is formalised as a key constraint for the class. The constraint usually accommodates all properties for this class. For example, the ER diagram that leads to the database schema in Fig. 2 is depicted in

Fig. 1. In this example the key constraint for the relationship class Assignment is Assignment(course teacher → Id). Here the application-dependent constraints allowed a reduction of the left-hand side of the constraint to the properties course and teacher.

3.2 Pivoting

Our goal is to decompose (relationship) classes into smaller fragments. Roughly speaking, we cut properties from (relationship) classes (e.g., introduced by the canonical formalisation) and graft these properties onto (entity) classes participating in the relationship.

The effect of the transformation can be redundancy reducing and that constraints are implicitly enforced. We call this implicit enforcement *natural enforcement of constraints*. The classes chosen to receive new properties are called *pivot classes*.

Now we present (property) pivoting in detail, i.e., we define how we can obtain the target schema and transformation rules from a given source schema. The effect of pivoting is concentrated in one class scheme. So we suppose we consider a class scheme with the form

$$\mathrm{RelCl}\{\mathrm{PivPr} : \mathrm{PivCl},\ \underbrace{p_1 : c_1, \ldots, p_n : c_n}_{\substack{\text{pivoted prop.}\\ \text{with types}}},\ \ldots\}F_{\mathrm{RelCl}}$$

where PivPr is called the *pivot property*, PivCl is the *pivot class* and properties p_i are the *pivoted properties*.

- We add now the class scheme $p\{\mathrm{RelCl\_PivPr\_}p_1 : c_1, \ldots, \mathrm{RelCl\_PivPr\_}p_n : c_n\}$ to the database schema, remove $p_1 : c_1, \ldots, p_n : c_n$ from the class scheme RelCl, replace PivCl with p in the class scheme RelCl and finally make the class p a subclass of class PivCl. In this process we introduce new property names for the class scheme p, assuming that this prevents name clashes with already existing property names.
- Then we adjust the semantic constraints of the classes RelCl and p to the new class schemes. Basically, this means for class scheme RelCl that we project the semantic constraints on the altered set of property names, and for class p this is a kind of projection, too, taking the newly introduced property names into account. Finally, we add a completeness constraint RelCl{PivPr} for the class scheme RelCl.

The relationship between the original schema and the transformed schema is described by a notion of *schema equivalence* [2, 5, 13]. It is based on transformations on instances, i.e., an instance of one schema is transformed into an instance of a different schema. These transformations are defined by so-called *transformation rules*. For example to transform instances of a schema into instances of its pivoted schema, we give the following transformation rules.

- If u is an object of class RelCl ($l_{Cl}(u) = \text{RelCl}$), the property value for the pivot property is v ($u \overset{\text{PivPr}}{\rightarrow} v \in E$) and the property value for some pivoted property p_i is w ($u \overset{p_i}{\rightarrow} w \in E$), we add the property value w for property RelCl_PivPr_p_i to the object v and make v an element of class p ($l_{Cl}(v) := p$).
- Finally we remove the property value v for property p_i from object u.

We say two schemas are *equivalent* if there are transformation rules, such that the transformation rules define a one-to-one and onto function that maps instances of one schema onto instances of the other.

Now we can make the following observation.

Theorem 6. *A database schema and its pivoted schema are equivalent in the above sense iff*

1. *each pivoted property is uniquely determined by the pivot property, i.e, the property values for the pivoted properties agree for two objects, whenever they hold the same property value for the pivot property, and*
2. *for all property sets M for* RelCl, $M \subset \text{Props(RelCl)}$, *the following conditions hold with $\mathcal{M} := \{p_1, \ldots, p_n\}$ the set of pivoted properties:*
   (a) $\text{Cl}_{F_{\text{RelCl}}}(M) = \text{Cl}_{F_{\text{RelCl}}}(M \backslash \mathcal{M}) \cup \text{Cl}_{F_{\text{RelCl}}}(M \cap \mathcal{M})$,
   (b) $\text{Cl}_{F_{\text{RelCl}}}(M \backslash \mathcal{M}) \subset \text{Props(RelCl)} \backslash \mathcal{M}$ *or* $\text{PivPr} \in \text{Cl}_{F_{\text{RelCl}}}(M \backslash \mathcal{M})$, *and*
   (c) $\text{Cl}_{F_{\text{RelCl}}}(M \cap \mathcal{M}) \subset \mathcal{M}$.

We merely give a brief motivation for the above conditions omitting the formal proof. The first condition is necessary because the transformation shifts properties from the relationship class to the pivot class. This means on the instance level that we cut the property values from a relationship object and graft them onto the property value of the corresponding pivot property. Thus the pivot property value must uniquely determine the pivoted property values otherwise we get a violation of property functionality for instances. So what has to be enforced in the original schema by means of functional constraints is naturally enforced in the pivoted schema due to the property functionality. Therefore we speak of a *natural enforcement* of functional constraints in the pivoted schema. The reason for the second condition is that, although we are interested in dropping specific constraints, namely those enforced naturally, this condition is necessary to preserve the effect of semantic constraints. Condition 2a ensures that the effect of constraints of the form that the left-hand side has attributes both in $\mathcal{M}$ and in $M \backslash \mathcal{M}$ are not lost because such constraints are dropped in the transformation process. The same line of argumentation is used for the conditions 2b and 2c. Basically, they ensure that there is not a constraint whose left-hand side is a subset of $\mathcal{M}$ or $M \backslash \mathcal{M}$ and the right-hand side is a subset of $M \backslash \mathcal{M}$ or $\mathcal{M}$ respectively. The reason for the more complex treatment of condition 2b is that sets of attributes that have the pivot attribute PivPr in their closure have to be dealt with in a special way.

That these conditions are sufficient indeed can be shown be lifting the whole consideration onto a level where we look solely at the semantic constraints.

In Fig. 4 the pivoted instance of the instance in Fig. 3 is presented. In the transformation property course was used as pivot property and properties assistant and date played the rôle of pivoted properties. To simplify the presentation we did not introduce new cryptic method names rather reusing the old ones and we dispense with the introduction of a subclass of Course. The graphical display of the pivoted instance in Fig. 4 lacks the representation of semantic constraints, which are part of a schema. Pivoting alters merely the semantic constraints for the relationship class and the pivot class, so it suffices to exhibit them. As the semantic constraint Assignment(course → assistant date) is naturally enforced in the pivoted schema, the semantic constraints for the pivot class Course are not affected by the transformation. This means that the semantic constraints remaining for class scheme Assignment are Assignment(teacher → room), Assignment(room → wing) and Assignment(course teacher → Id).

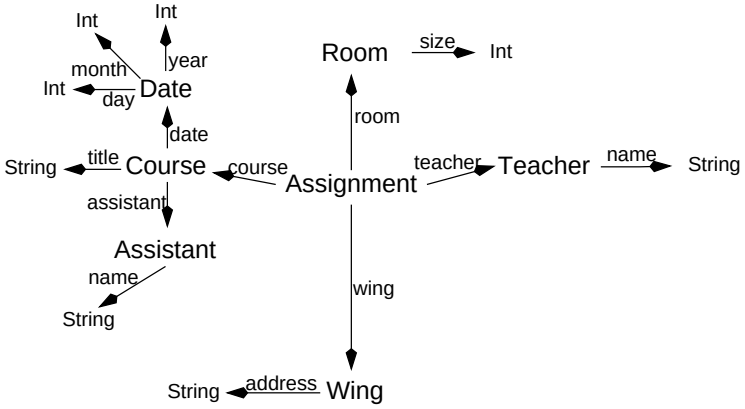

Fig. 4. Pivoted instance

Before we continue in our task to define a decomposition for relationships, we concern ourselves with special properties and their subsequent treatment, i.e., properties the closures of which are equal. The crucial point is that they are not treated equally by pivoting, namely if we want to make one a pivoted property for the other, we get a violation of condition 2c. This can be remedied by introducing a new property for the type of the pivoted property and stating that the pivoted property and the newly introduced property are inverses of each other [6]. Therefore we assume in the sequel that for all properties occurring in a class scheme the closures are different.

3.3 Natural Enforcement of Functional Constraints

Our goal is to transform a schema such that all original functional constraints are naturally enforced except for functional constraints being key constraints. In order to reach this goal, we have to consider two things. First of all, it is in general impossible to discard all functional constraints in one transformation step. This leads to a recursive application of the transformation as shown in Fig. 5. We first chose teacher as pivot property with room and wing as pivoted properties. Then we performed pivoting on the resulting schema with pivot property room and pivoted property wing. We get the same outcome if we take first room as pivot property and wing as pivoted property and afterwards choose teacher as pivot property and room as only pivoted property.

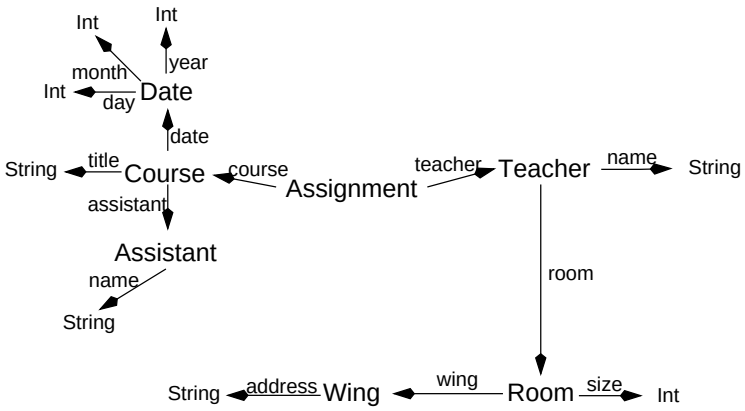

Fig. 5. Recursive pivoted instance

This example indicates that the outcome of recursive pivoting is in a sense independent of the order in which the single pivoting steps are performed. This interesting feature of recursive pivoting is captured in the following theorem.

Theorem 7. *If we choose pivot properties with appropriate pivoted properties, such that the application of pivoting with each of the pivot properties leads to an equivalent pivoted schema, we can apply pivoting recursively and the outcome is independent of the order in which we chose the pivot properties*[5].

Secondly, not all kinds of functional constraints can be naturally enforced and the second condition of Theorem 6 imposes further restriction on the set of semantic

[5] In the recursive pivoting the originally chosen set of pivoted properties has to be adapted to the new context, namely the properties for the class in which the pivot property is declared in.

constraints. The natural enforcement of functional constraints works only for those the left-hand side of which is a singleton because pivoting can be applied only with one pivot property at a time. Functional dependencies of this form are called *unary functional dependencies* [16]. We follow this notation and call functional constraints the left-hand side of which are singletons *unary functional constraints*.

Unfortunately the restriction to unary functional constraints is not sufficient in order to eliminate all functional constraints by recursive pivoting. To achieve that we further have to make the set of pivoted properties comprise the complete closure of the pivot property in each transformation step. If we select as pivoted properties the whole closure of the pivot property, we call the underlying pivoting *maximal pivoting*. Now what thwarts maximal pivoting? The obstacle is a possible violation of the conditions given in Theorem 6 referring to the equivalence of schemas. The first condition is fulfilled due to the confinement to the closure of the pivot property. The second condition has to be investigated in more depth. Conditions 2a and 2c are satisfied since we limit the use to unary functional constraints. Having only sets of unary functional constraints means that their closures are *topological* [9]. Therefore the equation

$$\mathrm{Cl}_F(X) = \bigcup_{A \in X} \mathrm{Cl}_F(A) \tag{1}$$

holds for sets F of unary functional constraints, ensuing the fulfilment of conditions 2a and 2c.

Condition 2b is harder to deal with. Here we consider a selection of a pivot property PivPr with a corresponding set $\mathcal{M}$ of pivoted properties such that the selection violates condition 2b. This means that there is a set of properties or to be more precise due to equation (1) a property $m \in \mathrm{Props(RelCl)} \backslash \mathcal{M}$ such that $\mathrm{Cl}_{F_{\mathrm{RelCl}}}(m) \not\subset \mathrm{Props(RelCl)} \backslash \mathcal{M}$ and $\mathrm{PivPr} \notin \mathrm{Cl}_{F_{\mathrm{RelCl}}}(m)$. To describe this situation in a better way, we build a *constraint graph* for the set F_{RelCl} of functional constraints. The set of vertices is the set of properties occurring in F_{RelCl}. For each $L \to R_1 \cdots R_n \in F_{\mathrm{RelCl}}$ we add the edges (L, R_i) to the graph. An example for this graph can be found in Fig. 6, which uses the functional constraints of the class scheme Assignment in Fig. 2.

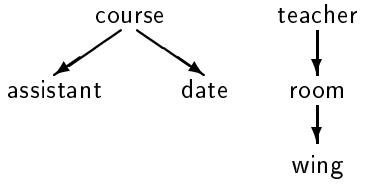

Fig. 6. Functional constraint graph

The graph describing the situation with the violation of condition 2b above is as depicted in Fig. 7. There is a path from m to a property $m' \in \mathcal{M}$ and due to the fact that $m' \in \mathcal{M}$ there is a path from PivPr to m'. In addition there is no path from m to PivPr and vice versa. This kind of structure can be forbidden if we say that the graph has to form a forest, i.e., whenever there is one vertex reachable from two other nodes, one of these two nodes must be reachable by the other.

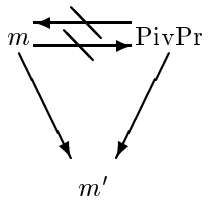

Fig. 7. Constraint graph describing the violation of condition 2b

We can make the following observation.

Theorem 8. *The two statements below concerning a class scheme are equivalent.*

- *The set of functional constraints consists merely of unary functional constraints and the corresponding graph forms a forest.*
- *Maximal recursive pivoting leads to a natural enforcement of all functional constraints occurring in the class scheme.*

4 Conclusion

In this paper we introduced a method to decompose relationship classes in an object-oriented data model that stem originally from a relationships in the ER model, and thereby improving already existing mappings from ER models to OO models. Therefore we support a user in the design process since he can concentrate on identifying the essential things of the application and their associations (later on modelled as relationships), without burdening with the task to break associations into smaller ones at this phase of the design process.

Comparing pivoting with the decomposition of a relational scheme into Boyce-Codd normal form based on the work of Delobel and Casey [8], we find a strong resemblance between both transformations. This is not astonishing as both transformation consider mainly sets of attributes and sets of functional dependencies. In fact we can even simulate pivoting in the relational model. Then it comes really close to the decomposition into Boyce-Codd normal form.

In this case we use foreign keys in relations that represent relationship sets in order to access represented entities participating in a relationship. A subtle difference between both transformations is that pivoting uses object identifiers for the reference mechanism whereas the relational model uses foreign keys, which are value oriented. Often, in the modelling process using the relational model, foreign keys are introduced that comprise only one attribute, e.g. a student number to uniquely identify a student. This can be seen as an attempt to simulate object identifiers. Using this approach throughout the modelling process shifts pivoting even closer to the decomposition into Boyce-Codd normal form. Now Theorem 6 gives conditions for pivoting to be *lossless* and *dependency preserving* [17]. Condition 2a guarantees the lossless property and conditions 2b and 2c guarantee dependency preservation. Theorem 7 gives as result that dependency preservation leads to the fact that the transformation process is independent of the order in which pivot attributes are chosen. Theorem 8 underlines the importance of unary constraints as these constraints can be naturally supported. As a by-product we know that if a set of functional dependencies consists only of unary functional dependencies, the corresponding Armstrong relation can be found in polynomial time [16].

The effect of the decomposition is not only to break relationship classes into smaller fragments but also to discard a certain kind of functional constraints, so-called unary functional constraints. What remains to be investigated is the trade-off between discarding a functional constraint and introducing a new completeness constraint with respect to costs for updates.

References

1. S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proc. 1989 ACM SIGMOD Int. Conf. Management of Data*, pages 159–173, 1989.
2. P. Atzeni, G. Ausiello, C. Batini, and M. Moscarini. Inclusion and equivalence between relational database schemata. *Theoretical Comput. Sci.*, 19:267–285, 1982.
3. C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. 1st Deductive and Object-Oriented Databases (DOOD '89)*, pages 405–430, Kyoto, Japan, 1989. Elsevier Science Publishers (North-Holland).
4. J. Biskup, R. Menzel, and T. Polle. Transforming an entity-relationship schema into object-oriented database schemas. In J. Eder and L. A. Kalinichenko, editors, *Adv. in Databases and Inf. Syst., Moscow 95*, Workshops in Computing, pages 109–136. Springer, 1996.
5. J. Biskup and U. Räsch. The equivalence problem for relational database schemes. In J. Biskup, J. Demetrovics, J. Paredaens, and B. Thalheim, editors, *Proc. 1st Symp. Mathematical Fundamentals of Database Syst.*, number 305 in LNCS, pages 42–70. Springer, 1988.
6. R. G. G. Cattell and T. Atwood, editors. *The object database standard: ODMG-93; release 1.1.* Morgan Kaufmann, 1994.
7. P. P.-S. Chen. The entity-relationship-model — towards a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

8. C. Delobel and R. G. Casey. Decomposition of a data base and the theory of boolean switching functions. *IBM J. Res. Dev.*, 17(5):374–386, 1973.
9. J. Demetrovics, L. O. Libkin, and I. B. Muchnik. Functional dependencies in relational databases: a lattice point of view. *Discrete Applied Mathematics*, 40:155–185, 1992.
10. R. A. Elmasri, V. Kouramajian, and B. Thalheim, editors. *Proc. 12th Int. Conf. on Entity-Relationship Approach*, Arlington, Texas, USA, 1993.
11. M. Gogolla, R. Herzig, S. Conrad, G. Denker, and N. Vlachantonis. Integrating the ER approach in an OO environment. In Elmasri et al. [10], pages 376–389.
12. R. Herzig and M. Gogolla. Transforming conceptual data models into an object model. In G. Pernul and A. M. Tjoa, editors, *Proc. 11th Int. Conf. on Entity-Relationship Approach*, number 645 in LNCS, pages 280–298, Karlsruhe, Germany, 1992. Springer.
13. R. Hull. Relative information capacity of simple relational database schemata. *SIAM J. Comput.*, 15(3):856–886, 1986.
14. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.
15. Y. Kornatzky and P. Shoval. Conceptual design of object-oriented schemes using the binary-relationship model. *Data & Knowledge Eng.*, 14(3):265–288, 1995.
16. H. Mannila and K.-J. Räihä. Practical algorithms for finding prime attributes and testing normal forms. In *Proc. Eighth ACM PODS*, pages 128–133, 1989.
17. H. Mannila and K.-J. Räihä. *The Design of Relational Databases.* Addison-Wesley, Wokingham, England, 1992.
18. R. Missaoui, J.-M. Gagnon, and R. Godin. Mapping an extended entity-relationship schema into a schema of complex objects. In M. P. Papazoglou, editor, *Proc. 14th Int. Conf. on Object-Oriented and Entity Relationship Modelling*, pages 205–215, Brisbane, Australia, 1995.
19. B. Narasimhan, S. B. Navathe, and S. Jayaraman. On mapping ER and relational models into OO schemas. In Elmasri et al. [10], pages 403–413.
20. P. Poncelet, M. Teisseire, R. Cicchetti, and L. Lakhal. Towards a formal approach for object database design. In R. Agrawal, editor, *Proc. 19th Int. Conf. on Very Large Data Bases*, pages 278–289, Dublin, Irland, 1993.
21. J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In N. Meyrowitz, editor, *OOPSLA'87*, pages 462–481, Orlando, Florida, 1987. acm Press.
22. B. Thalheim. *Fundamentals of Entity-Relationship Modeling.* Springer, 1996.
23. G. E. Weddell. Reasoning about functional dependencies generalized for semantic data models. *ACM Trans. Database Syst.*, 17(1):32–64, Mar. 1992.

This article was processed using the LaTeX macro package with LLNCS style