Transforming Data Models with UML

Martin Gogolla, Arne Lindow
University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
[gogolla|lindow]@informatik.uni-bremen.de

Abstract.
This chapter studies an approach to establish a formal connection between data models, in particular between conceptual data models and implementation data models. We use metamodeling techniques based on the Meta Object Facility MOF. MOF may be regarded as a subset of the Unified Modeling Language UML. As prominent example data models, we formally describe and thereby analyze the Entity-Relationship and the Relational data model. In addition, we represent the transformation between these data models by MOF language features. Thus we describe the data models and their transformation within a single framework. All results are formally represented and validated by a MOF compliant tool. The approach presented is general enough so that it can used for other data models being important for the Semantic Web, e.g., the object-oriented data model or semi-structured data models like XML-based models.

1 Introduction

Without doubt conceptual modeling of information systems with the Entity-Relationship model is a well-accepted standard technique in software design. The area of conceptual modeling also has connections to ontology based approaches which claim to provide a semantically richer grounding of information. Thus conceptual modeling is relevant for questions concerning the Semantic Web. Nowadays, central aspects of Entity-Relationship schemata can be found in the Unified Modeling Language UML [1] under the name class diagram. Transformations of Entity-Relationship schemata into implementation-oriented data models like the Relational data model also belong to standard software development techniques.

Recently, much attention has been paid to metamodeling approaches within the software engineering community. For example, UML is described in a semi-formal way by a metamodel with language features of the so-called Meta Object Facility MOF [2]. Roughly speaking, MOF is a subset of UML using class diagrams, i.e. a description formalism originating from the Entity-Relationship model. For UML, the abstract syntax is described with class diagrams and context-sensitive conditions are formulated with the Object Constraint Language OCL [1, 3], a textual sublanguage of UML having a precise set-theoretic semantics [4].

The present chapter now links classical database models like the Entity-Relationship and the Relational model with the MOF metamodeling approach. We present a MOF model for the Entity-Relationship and the Relational model. This is similar to modeling of UML with MOF where class diagrams are specified and corresponding invariants are formulated with OCL. In addition, we present the transformation of Entity-Relationship schemata into Relational schemata with a MOF class diagram and corresponding OCL rules. All three layers, i.e. Entity-Relationship, Relational, and transformation layer, are formally specified and validated with the UML Specification Environment USE [5], an animation system for UML descriptions. USE supports also central MOF language features and we employ it as a MOF compliant tool.

The purpose of the chapter is to make assumptions of data models explicit and provide possibility for verification support. We want to formally ground and to represent well-known information system techniques by current formal methods. We do this in order to better understand underlying principles and to gain new formal insight into the fundamental nature of these data models and their transformation. Such formal MOF representations of data models are also needed in the context of newer proposals for handling data warehouses like the Common Warehouse Metamodel CWM [6]. The transformation of Entity-Relationship schemata into Relational schemata is long known. However, we are not aware of an approach explaining both data models and their transformation in a single, complete formal and logical setting. Work on transformation is usually done in an algorithmic way. This algorithmic way of describing translations has its analogy in the web area in terms of different serializations of ontology metamodels. Therefore, the purpose of this chapter is to precisely present that transformation and to show and study formal and informal transformation properties. Our approach has the advantage that it allows a very condensed representation of schemata and transformations. The central ingredients of the Entity-Relationship and the Relational data model and their transformation can be pictured on a single diagram.

Our work has connections to other approaches studying metamodels or metaobjects. Formal transformations between data models become important in the context of newer approaches [6, 7] which link metamodeling and data warehouses. Beside current work on formal foundation of the UML metamodel [8] there have been studies on the metamodel of predecessors of UML like OMT [9] or Booch [10]. The metamodel of OML has also been worked out [11]. Apart from these efforts concentrating on software engineering aspects, metamodels have also been considered in the context of information systems [12, 13]. Metamodeling has been successfully employed for repositories [14, 15], database design [16], and data warehouse design [17]. Other use of metamodels and metaobjects was made in the area of schema integration [18], engineering data warehouses [19], and Semantic Web applications [20]. Well recognized in the Semantic Web community are approaches like [21] studying also metamodels for transformations.

The structure of the rest of this chapter is as follows. Section 2 shortly introduces our metamodel for the Entity-Relationship and the Relational data model and their transformation. Section 3 explains our running example in conventional notation. Section 4 shows how the example schemata can be considered as instantiations of the defined Entity-Relationship and Relational metamodels. In Section 5 the details of restricting OCL invariants are explained. Section 6 compares properties of the Entity-Relationship and the Relational model on the basis of the metamodel representation. Section 7 closes the chapter with concluding remarks.

2 MOF Class Diagram

The central ingredients of the Entity-Relationship and the Relational data model and their transformation can be pictured on a single diagram. In Figure 1 we show a metamodel, a MOF class diagram, for the Entity-Relationship (ER) data model, for the Relational data

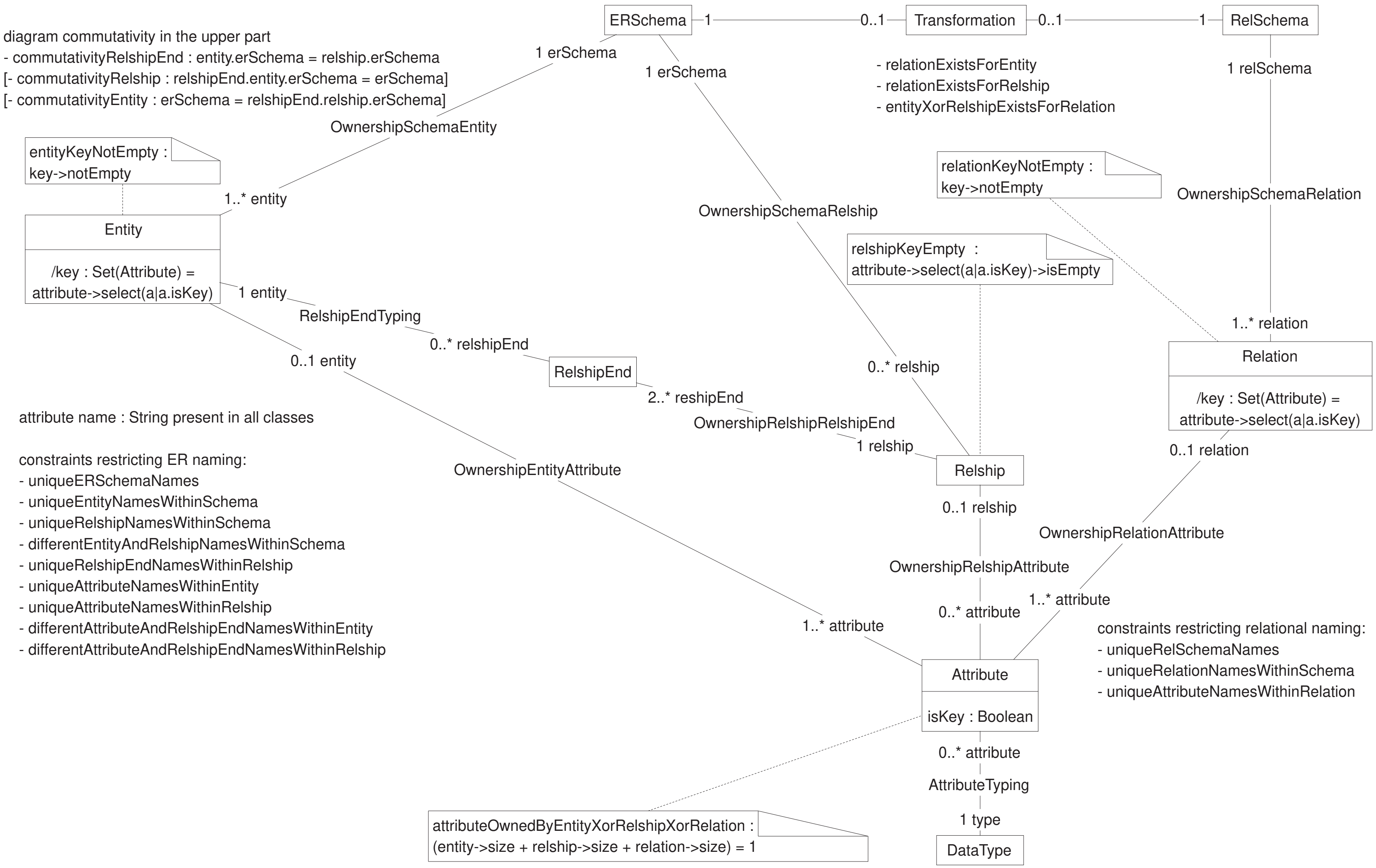
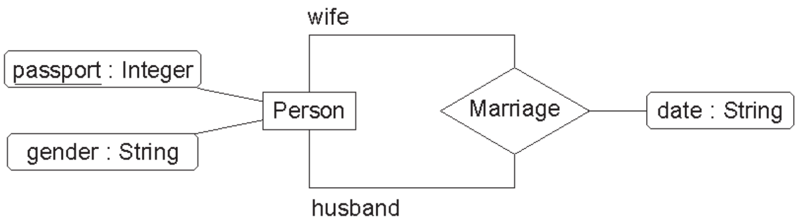


Figure 1: Metamodel for Transforming the ER into the Relational Model




```
Person(passport:Integer, gender:String)
Marriage(wife_passport:Integer, husband_passport:Integer, date:String)
```

Figure 2: Example for an Entity-Relationship and a Relational Schema

model, and the transformation from the Entity-Relationship model into the Relational model. The left part with classes ERSchema, Entity, Relship, and RelshipEnd belongs to the Entity-Relationship model, the right part with classes RelSchema and Relation represents the Relational model, the bottom part with classes Attribute and DataType is shared by both data models, and the top part with class Transformation reflects the data model transformation aspect. One main purpose of this chapter is explaining this metamodel. We do this by studying a detailed example.

But first we will have a short walk through that metamodel. The Entity-Relationship part of the metamodel expresses that an ERSchema consists of at least one Entity object and possibly zero or more Relship objects. A Relship object possesses two or more RelshipEnd objects which in turn are typed through an association to class Entity. Entity and Relship objects may possess Attribute objects which are typed through an association to class DataType.

The Relational part of the metamodel expresses that a RelSchema consists of at least one Relation which in turn may possess one or more Attribute objects. As in the Entity-Relationship part, Attribute objects are typed through an association to class DataType.

The transformation part of the metamodel consists of class Transformation together with two associations expressing that a Transformation object is linked to exactly one ERSchema object and one RelSchema object. This expresses that translation of the Entity-Relationship schema will result in the Relational schema.

Concerning attributes and operations of the above classes, all classes inherit from a class Named (not shown in Figure 1) so that they all possess an attribute name giving a String value for the object's name. Class Attribute additionally has a boolean attribute isKey indicating whether the respective Attribute object contributes to the key of the schema component, i.e. the entity or the relation, to which the Attribute object belongs. Classes Entity and Relation possess a derived, set-valued operation key() giving the set of key attributes of the entity and the relation, respectively. The diagram also mentions the names of constraints to be explained in detail in Section 5.

3 Running Example in Conventional Notations

Figure 2 shows an Entity-Relationship and a Relational schema which we will use as a running example. The Entity-Relationship schema pictures one entity Person with two exemplary attributes passport and gender. The underlining indicates that attribute passport is the key attribute for entity Person. The Entity-Relationship schema also includes one relationship Marriage. It is a reflexive relationship where a single entity participates twice. Two different kinds of participation are distinguished by role names wife and husband. Relationship Marriage possesses one attribute date. The Relational schema represents a direct translation of the Entity-Relationship schema into the Relational data model. Both main Entity-Relationship schema components are translated into a relation, respectively. Relation Person possesses two attributes passport and gender. Relation Marriage owns three attributes wife_passport, husband_passport, and date. Again, key attributes are underlined. Thus, the key for relation Person consists of attribute passport, and the key for relation Marriage is made of two attributes wife_passport and husband_passport. The data types of attributes are notated in both schemata after colons.

4 Running Example as MOF Object Diagram

The object diagram in Figure 3 shows three things: (1) The ER schema as well as (2) the Relational schema from Figure 2 and (3) the formal transformation between them. That object diagram belongs to metaclass diagram given in Figure 1. It is a screenshot of the USE system. The object diagram is divided into four parts: (1) The upper part with white object rectangles for Transformation, ERSchema, and RelSchema objects, (2) the middle left part with light grey rectangles for components of the Entity-Relationship schema, (3) the middle right part with dark grey rectangles for components of the Relational schema, and (4) the lower part with white rectangles for data types. We now discuss parts of that object diagram in detail. By doing this we explain how the respective data models and their transformation are represented in our metamodeling approach.

Data types: Both database schemata use two data types String and Integer. Correspondingly two DataType objects are introduced in the lower part of Figure 3. One DataType object possesses object identifier String within the USE system. This is visualized by writing down the identifier String in front of the colon in the upper part of the object rectangle. The value of the object's attribute name is also equal to String. Alternatively, we could have used a more abstract object identifier like for example dataTypeOne for that DataType object and dataTypeTwo for the other DataType object which in Figure 3 now has object identifier Integer. Then these identifiers dataTypeOne and dataTypeTwo would appear in front of colons in the upper part of object rectangles, but still these objects would represent the data types String and Integer, because the name attribute value would indicate this. We have preferred to have this strong correspondence between the name attributes values of the objects and its object identifiers but we are not forced to do so, because from a formal viewpoint there is no compelling relationship between the name attribute value and the object identifier. However, we will proceed analogously in having a naming correspondence as far as possible between object identifiers and name attribute values of objects.

Entity-Relationship schema components: The light grey objects in the left part of Figure 3 represent components of the Entity-Relationship schema. We see one Entity object PersonER with name attribute value Person and one Relship object MarriageER with name attribute value Marriage. Here, we have chosen object identifiers close to, but not

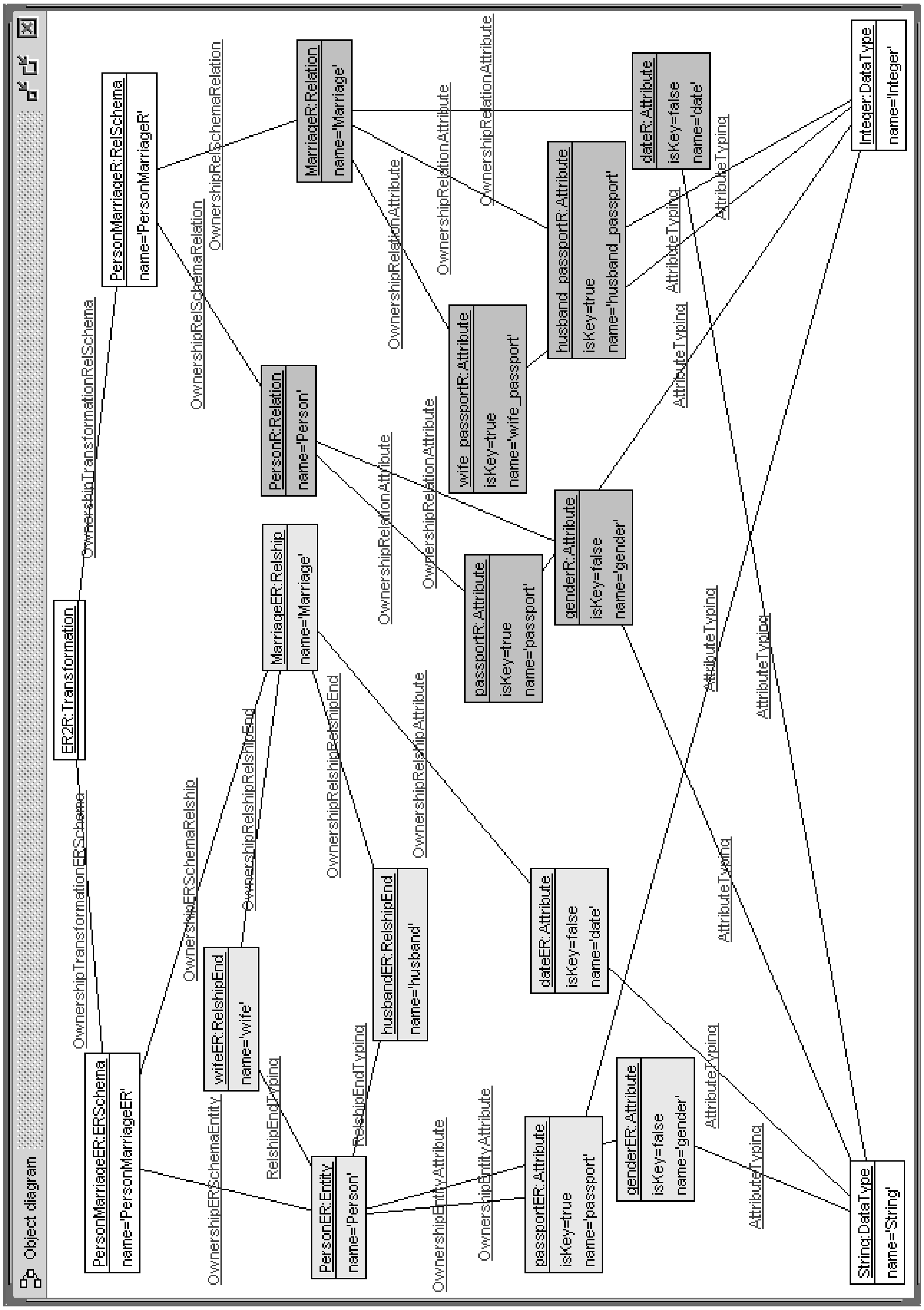


Figure 3: Example for the Formal Transformation

identical to name attribute values, because in the Relational schema to be explained below we will have components with the same name attribute values but this time being objects of class Relation. They will have object identifiers PersonR and MarriageR. Object identifiers in the Entity-Relationship part will always end with ER and those in the Relational part with R. Entity and Relship objects are connected by two RelshipEnd objects wifeER and husbandER representing role names of the relationship. In addition, three Attribute objects passportER, genderER, and dateER represent the two attributes of the Entity object and the single attribute of the Relship object, respectively. The isKey attribute value for passportER indicates that this attribute is a key attribute. Links to the above explained DataType objects constitute the attribute types.

Relational schema components: The dark grey objects in the right part of Figure 3 constitute components of the Relational schema. This Relational schema has two Relation objects with object identifiers PersonR and MarriageR and name attribute values Person and Marriage, respectively. The relation with name Person possesses two attributes with names passport and gender, and the relation with name Marriage has three attributes with names wife_passport, husband_passport, and date. As indicated by the isKey attribute values, the key for the relation named Person consists of the passport attribute, and the key of the relation named Marriage is made of attributes named wife_passport and husband_passport. As in the Entity-Relationship schema, the attribute typing is provided by links to DataType objects.

Transformation, ER and Relational schema: In the upper part of Figure 3, two white schema object rectangles represent the ER and the Relational schema. Both are connected with links to its direct components: The ERSchema object is linked to Entity and Relship objects and the RelSchema object to two Relation objects. The Transformation object is connected to both of these schema objects.

5 Constraints

As a central description ingredient, the metamodel is restricted by a number of OCL constraints in order to establish correct system states, for example, to guarantee that Entity-Relationship and Relational schemata are represented correctly. Without such constraints undesirable system states would be allowed. To mention only one example, without constraints it would be possible that an Entity object is connected to a RelshipEnd and to an Attribute object where all three objects have the same value for their name attribute.

5.1 Name Constraints

One category of constraints concerns naming of objects, i.e. these constraints restrict values of the name attribute. There is a distinction between globally unique names and locally unique names. For example, names of Entity-Relationship schemata must be globally unique which means that two distinct Entity-Relationship schemata also generally must have distinct name attribute values. Formally, this is notated as:

```
context ERSchema inv uniqueERSchemaNames:
  ERSchema.allInstances->forAll(s1,s2|
    s1.name=s2.name implies s1=s2)
```

The implication could have alternatively been written the other way round emphasizing distinctness and using inequations instead of equations.

```
context ERSchema inv uniqueERSchemaNames:
  ERSchema.allInstances->forAll(s1,s2|
    s1<>s2 implies s1.name<>s2.name)
```

Locally unique names mean that object names are unique within a certain local namespace or context. This is required, for example, for entities within an Entity-Relationship schema. It means that two distinct entities belonging to the same Entity-Relationship schema must have distinct names, but in contrast to globally unique names it is allowed that two distinct entities with the same name exist if they come from two distinct Entity-Relationship schemata. Formally this is expressed as:

```
context ERSchema inv uniqueEntityNamesWithinSchema:
  entity->forAll(e1,e2|e1.name=e2.name implies e1=e2)
```

Constraints concerning globally unique names use the OCL construct `allInstances` retrieving for a class all current objects of that class. The other constraint for globally unique names is:

```
context RelSchema inv uniqueRelSchemaNames:
  RelSchema.allInstances->forAll(s1,s2|
    s1.name=s2.name implies s1=s2)
```

As a convention, we let names of constraints concerning locally unique names always include the word `Within` and indicate the namespace after `Within`. Other constraints for locally unique names are:

```
context ERSchema inv uniqueRelshipNamesWithinSchema:
  relship->forAll(r1,r2|r1.name=r2.name implies r1=r2)

context Relship inv uniqueRelshipEndNamesWithinRelship:
  relshipEnd->forAll(re1,re2|
    re1.name=re2.name implies re1=re2)

context Entity inv uniqueAttributeNamesWithinEntity:
  attribute->forAll(a1,a2|a1.name=a2.name implies a1=a2)

context Relship inv uniqueAttributeNamesWithinRelship:
  attribute->forAll(a1,a2|a1.name=a2.name implies a1=a2)

context RelSchema inv uniqueRelationNamesWithinSchema:
  relation->forAll(r1,r2|r1.name=r2.name implies r1=r2)

context Relation inv uniqueAttributesNamesWithinRelation:
  attribute->forAll(a1,a2|a1.name=a2.name implies a1=a2)
```

A third group of constraints concerning naming requires that components within one schema but different in nature must have different names. For example, entities and relationships within one Entity-Relationship schema must possess different names.

```
context ERSchema
  inv differentEntityAndRelshipNamesWithinSchema:
  entity->forAll(e|relship->forAll(r|e.name<>r.name))

context Entity
  inv differentAttributeAndRelshipEndNamesWithinEntity:
  relshipEnd->forAll(re|attribute->forAll(a|re.name<>a.name))

context Relship
  inv differentAttributeAndRelshipEndNamesWithinRelship:
  relshipEnd->forAll(re|attribute->forAll(a|re.name<>a.name))
```

5.2 Constraints Concerning Keys

This group of constraints concerns requirements for isKey attribute values and key() operation values. Entity and Relation objects must have a non-empty key. Relship objects are not allowed to have key attributes.

```
context Entity inv entityKeyNotEmpty:
  key()->notEmpty

context Relship inv relshipKeyEmpty:
  attribute->select(a|a.isKey)->isEmpty

context Relation inv relationKeyNotEmpty:
  key()->notEmpty
```

5.3 Attribute Ownership and Commutativity Constraints

Class Attribute participates in three associations OwnershipEntityAttribute, OwnershipRelshipAttribute, and OwnershipRelationAttribute. But an Attribute object must belong to either an Entity object or a Relship object or a Relation object, i.e. an Attribute object must belong to exactly one of these three schema components.

```
context Attribute
  inv attributeOwnedByEntityXorRelshipXorRelation:
  (entity->size + relship->size + relation->size) = 1
```

The next constraint concerns a commutativity property in the Entity-Relationship part of the schema. One must require that, if navigating from a RelshipEnd object via association RelshipEndTyping to the Entity class and from there via association OwnershipSchemaEntity to the the ERSchema class, this results in the same ERSchema object as when going the other way via association OwnershipRelshipRelshipEnd and association OwnershipSchemaRelship through class Relship.

```
context RelshipEnd inv commutativityRelshipEnd:
  entity.erSchema=relship.erSchema
```

This constraint commutativityRelshipEnd implies that other commutativity requirements also hold. Thus we have not to require explicitly that the following invariants will be true.

```
context Entity inv commutativityEntity:
  -- erSchema=relshipEnd.relship.erSchema
  relshipEnd->forAll(re|re.relship.erSchema=erSchema)
context Relship inv commutativityRelship:
  -- erSchema=relshipEnd.entity.erSchema
  relshipEnd->forAll(re|re.entity.erSchema=erSchema)
```

5.4 Transformation Constraints

The transformation invariants are the most challenging expressions in the specification. These invariants can be used in the first place to verify a transformation, but it is in principle also possible to try to systematically derive concrete transformation classes or transformation operations from these constraints. This is subject to future work.

Recall that transformation of Entity-Relationship schemata into Relational schemata is represented by class Transformation. This class has no attributes, but participates in two associations: One Transformation object is linked to exactly one ERSchema object and one RelSchema object. The purpose of Transformation invariants is to guarantee that the RelSchema object is the result of transforming the ERSchema object into the Relational data model. The transformation is restricted by these three invariants: `relationExistsForEntity`, `relationExistsForRelship`, and `entityXorRelshipExistsForRelation`. As the names indicate, the first two are responsible for the task that for each entity and each relationship in the Entity-Relationship schema there exists a corresponding relation in the Relational schema, and the last invariant assures that for each relation there is an entity or a relationship. The context for all invariants is a transformation, i.e. one Entity-Relationship schema and one Relational schema.

```
context Transformation inv relationExistsForEntity:
   erSchema.entity->forAll(e| relSchema.relation->exists(rl|
(A)   e.name=rl.name and
(B)   e.attribute->forAll(ea| rl.attribute->exists(ra|
         ea.name=ra.name and
         ea.type=ra.type and
         ea.isKey=ra.isKey))))
```

Invariant `relationExistsForEntity` requires that for each entity `e` in the considered Entity-Relationship schema there is a relation `rl` in the Relational schema such that (A) entity `e` and relation `rl` have the same name and (B) for each attribute `ea` of the entity there is an attribute `ra` in the relation having the same name, the same type, and the same key property. As a forward reference, we remark that the requirement that relation `rl` has no more attributes will be guaranteed by the invariant `entityXorRelshipExistsForRelation` together with naming restriction that an attribute name is unique within a relation.

```
context Transformation inv relationExistsForRelship:
  erSchema.relship->forAll(rs|
    relSchema.relation->exists(rl|
(A)   rs.name=rl.name and
(B)   rs.relshipEnd->forAll(rse|
        rse.entity.key()->forAll(rsek|
         rl.attribute->exists(ra|
           rse.name.concat('_').concat(rsek.name)=ra.name and
           rsek.type=ra.type and
           ra.isKey=true))) and
(C)   rs.attribute->forAll(rsa| rl.attribute->exists(ra|
        rsa.name=ra.name and
        rsa.type=ra.type and
        ra.isKey=false))))
```

Invariant `relationExistsForRelship` asserts that for each relationship `rs` in the Entity-Relationship schema there is a relation `rl` in the Relational schema such that (A) relationship `rs` and relation `rl` have identical names, (B) for each relationship end there are appropriate attributes in the relation, and (C) for each relationship attribute there is an attribute in the relation. Subrequirement (B) means, that for each relationship end `rse`, any key attribute `rsek` of the referenced entity appears in the relation, and the name of the attribute in the relation consists of the relationship end name followed by underscore followed by the Entity-Relationship name of the key attribute. Subrequirement (C) says, that each attribute `rsa` of the relationship appears under the same name and the same type as a non-key attribute in the relation.

```
context Transformation inv entityXorRelshipExistsForRelation:
  relSchema.relation->forAll(rl|
    erSchema.entity->exists(e|
(A)   rl.name=e.name and
(B)   rl.attribute->forAll(ra| e.attribute->exists(ea|
        ra.name=ea.name and
        ea.type=ra.type and
        ra.isKey=ea.isKey)))
    xor
    erSchema.relship->exists(rs|
(C)   rl.name=rs.name and
      rl.attribute->forAll(ra|
(D)     rs.relshipEnd->exists(rse|
         rse.entity.key()->exists(rsek|
(E)        ra.name=rse.name.concat('_').concat(rsek.name) and
(F)        ra.type=rsek.type and
(G)        ra.isKey=true))
        xor
(H)     rs.attribute->exists(rsa|
          ra.name=rsa.name and
          ra.type=rsa.type and
          ra.isKey=false))))
```

Invariant `entityXorRelshipExistsForRelation` assures that each relation `rl` in the Relational schema either has a corresponding entity or relationship in the Entity-Relationship schema but not both. If a relation `rl` corresponds to an entity `e`, then (A) the name of the relation and entity coincide and (B) for each relation attribute `ra` there is a corresponding entity attribute `ea` with the same name, type, and key property. If a relation `rl` corresponds to a relationship `rs`, then (C) both have the same name and *EITHER* (D) a relation attribute `ra` comes from a relationship end `rse` such that (E) the name of the attribute `ra` is equal to the name of the relationship end `rse` followed by underscore followed by the name of a key attribute `rsek` of the referenced entity of the relationship end, (F) the attribute `ra` has the same type as key attribute `rsek`, and (G) attribute `ra` is part of the key of relation `rl`, *OR* (H) a relation attribute `ra` comes from a relationship attribute `rsa` with the same name, type, and attribute `ra` is a non-key attribute in the relation.

The above invariants can be used in the first place to verify a transformation. Similar work on verification of integrating ER diagrams has been done in [22]. But in principle, it should also be possible to systematically derive concrete transformation classes or transformation operations from these constraints.

5.5 USE Screenshot on Invariants

Figure 4 shows a screenshot from a USE session. It displays the explained 20 invariants. That screenshot was taken after the complete schemata and the transformation object have been established. We see that all invariants are satisfied, i.e. the result of evaluating each invariant is true. This means that all invariants are valid in the current system state. Or in other words, the Relational schema is the correct transformation of the Entity-Relationship schema. As a side remark concerning functionality of USE, we note that, if an invariant is false, USE supports failure analysis by an evaluation browser window which allows to trace which part of the current system state and which part of the respective invariant contribute to the undesired result.

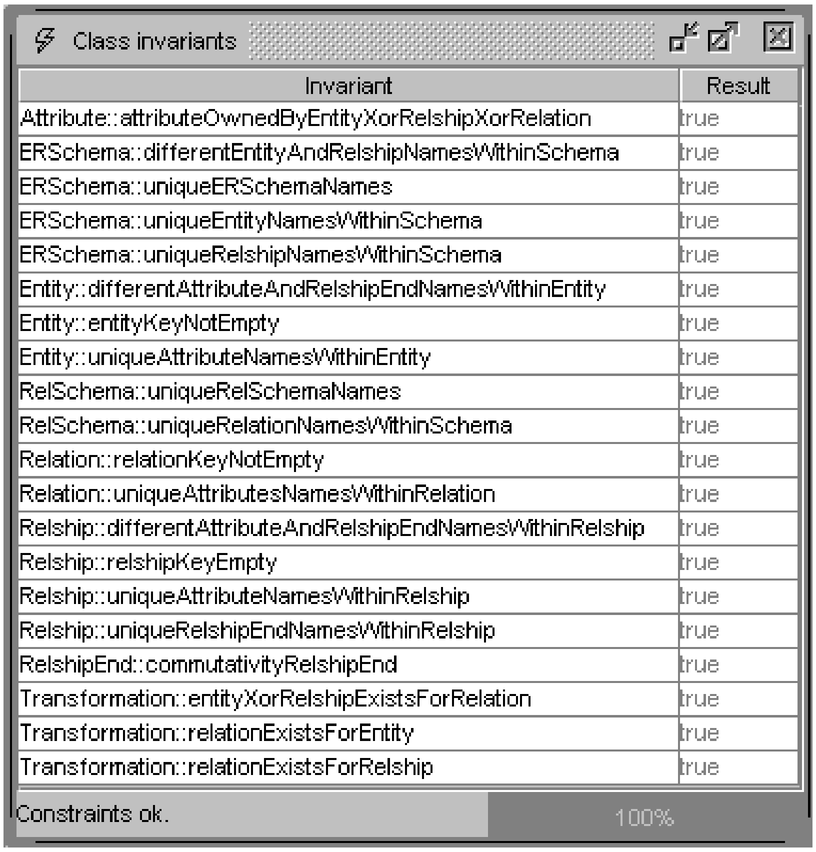
6 Comparison of ER and Relational Data Model

The exemplary object diagram in Figure 3 also gives rise to a short, but substantial comparison of the Entity-Relationship and Relational data model. Both the Entity-Relationship and the Relational schema consist of seven objects:

1 Entity + 1 Relship + 2 RelshipEnds + 3 Attributes = 7 Objects (ER)

2 Relations + 5 Attributes = 7 Objects (Rel)

The single Entity object and the single Relship object correspond to two Relation objects. The two RelshipEnd and the three Entity-Relationship Attribute objects correspond to the five Relational Attribute objects. Figure 5 gives an overview by showing the data model concepts, their associations within the respective data model, and their transformation. The simple enumeration shows in exemplary form richness of concepts in the ER model with four concepts, namely Entity, Relship, RelshipEnd, and Attribute, in comparison to the Relational model with only two concepts, namely Relation and Attribute. The example also points out that Entity and Relship objects are transformed into Relation objects, as well as that RelshipEnd and Entity-Relationship Attribute objects are transformed into Relational Attribute objects.



Class invariants

Invariant	Result
Attribute::attributeOwnedByEntityXorRelshipXorRelation	true
ERSchema::differentEntityAndRelshipNamesWithinSchema	true
ERSchema::uniqueERSchemaNames	true
ERSchema::uniqueEntityNamesWithinSchema	true
ERSchema::uniqueRelshipNamesWithinSchema	true
Entity::differentAttributeAndRelshipEndNamesWithinEntity	true
Entity::entityKeyNotEmpty	true
Entity::uniqueAttributeNamesWithinEntity	true
RelSchema::uniqueRelSchemaNames	true
RelSchema::uniqueRelationNamesWithinSchema	true
Relation::relationKeyNotEmpty	true
Relation::uniqueAttributesNamesWithinRelation	true
Relship::differentAttributeAndRelshipEndNamesWithinRelship	true
Relship::relshipKeyEmpty	true
Relship::uniqueAttributeNamesWithinRelship	true
Relship::uniqueRelshipEndNamesWithinRelship	true
RelshipEnd::commutativityRelshipEnd	true
Transformation::entityXorRelshipExistsForRelation	true
Transformation::relationExistsForEntity	true
Transformation::relationExistsForRelship	true

Constraints ok. 100%

Figure 4: USE Screenshot with Invariant Evaluation

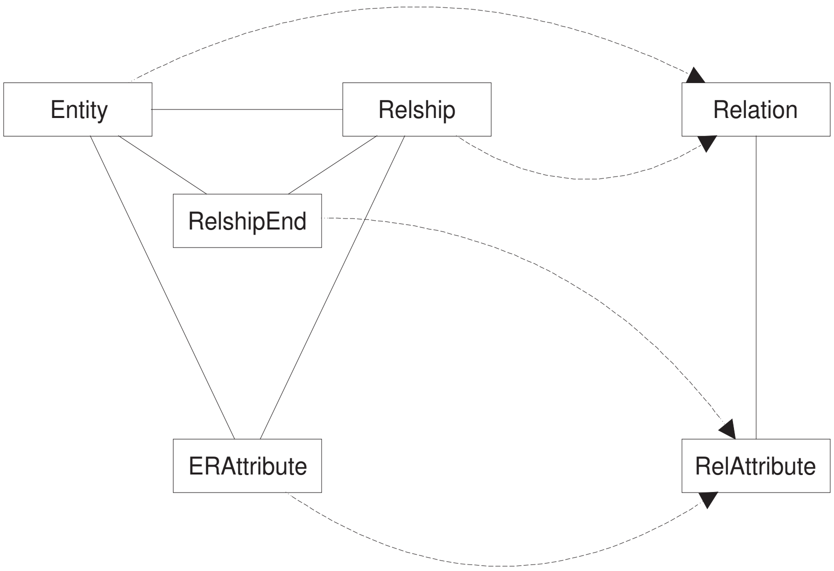


Figure 5: Transformation of Entity-Relationship Concepts into Relational Concepts

The comparison in Figure 5 also indicates how an equivalent alternative to the present transformation could look like. One could divide the general Transformation class into specialized Transformation classes. Then one would introduce Transformation classes establishing (Entity,Relation)-, (Relship,Relation)-, (RelshipEnd,Attribute)-, and (Attribute,Attribute)-pairs. The general Transformation class could contain these specialized Transformations as components. The difference to the present solution is that then the concept mapping (which in the present solution is expressed in the `exists` and `forAll` quantifiers in the OCL expressions) becomes explicit.

Yet another alternative being also equivalent would be to have an operation with name transform2rel():RelSchema in class ERSchema instead of class Transformation. The requirements which are now stated as invariants for class Transformation would become postconditions for that operation. We have chosen an approach with invariants in order to emphasize that the transformation can be given in a purely descriptive way.

7 Conclusion

In this chapter we have introduced a formal MOF modeling for the Entity-Relationship and the Relational model as well as the transformation between these data models. All steps have been formally specified and validated with the USE system which has been employed here as a MOF compliant tool. Our approach allows to concentrate the central ideas of the Entity-Relationship and the Relational data model and their transformation into a single diagram having a precise formal semantics.

Other language features of the Entity-Relationship and the Relational data model like, for example, foreign keys have to be studied further. We feel however that such extensions can be handled within the MOF and OCL language elements due to the flexible and powerful possibilities offered. We will also consider other data models like object-oriented or semi-structured models and data models important in the context of the Semantic Web. Due to the modeling power of UML including OCL, these data models and their translation can be handled in the same way as we did here. Thus the treatment of the Entity-Relationship and Relational model can be considered as a feasibility study for the general approach which involves the description of data models and their transformation in an object-oriented and descriptive way. Further attention will also be paid to include in addition information system states already on the metamodel layer. First steps in this direction have already been studied in [23].

Acknowledgements

Thanks to Heiner Stuckenschmidt for commenting a draft version of this chapter. The contribution of Mark Richters and Paul Ziemann in other papers on similar topics is gratefully acknowledged.

References

[1] OMG, ed.: OMG Unified Modeling Language Specification, Version 1.4. OMG (2001) OMG Document `formal/01-09-68` through `formal/01-09-80`, `www.omg.org`.

[2] OMG, ed.: Meta Object Facility (1.3). OMG (1999) `www.omg.org`.

[3] Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (1998)

[4] Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. In Ling, T.W., Ram, S., Lee, M.L., eds.: Proc. 17th Int. Conf. Conceptual Modeling (ER'98), Springer, Berlin, LNCS 1507 (1998) 449–464

[5] Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. In Evans, A., Kent, S., eds.: Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000), Springer, Berlin, LNCS 1939 (2000) 265–277

[6] OMG, ed.: The Common Warehouse Metamodel Specification. OMG (2000) `www.omg.org`.

[7] Vetterli, T., Vaduva, A., Staudt, M.: Metadata Standards for Data Warehousing: Open Information Model vs. Common Warehouse Metamodel. SIGMOD Record **29** (2000) 68–75

[8] Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In Hussmann, H., ed.: Proc. Int. Conf. FASE, Springer, LNCS (2001) 17–31

[9] Ebert, J., Süttenbach, R.: An OMT Metamodel. Technical report, University Koblenz-Landau, Fachbericht Informatik 13-97 (1997)

[10] Ebert, J., Süttenbach, R.: A Booch Metamodel. Technical report, University Koblenz-Landau, Fachbericht Informatik 5-97 (1997)

[11] Henderson-Sellers, B., Firesmith, D., Graham, I.M.: OML Metamodel: Relationships and State Modeling. Journal Of Object-Oriented Programming **10** (1997) 47–51

[12] Atzeni, P., Torlone, R.: A metamodel approach for the management of multiple models and translation of schemes. Information Systems **18** (1993) 349–362

[13] Misic, V.B., Moser, S.: A Formal Approach to Metamodeling: A Generic Object-Oriented Perspective. In: Int. Conf. Entity-Relationship Approach (ER'97). (1997) 243–256

[14] Iyengar, S.: Distributed object repositories: Concepts and standards. In Embley, D.W., Goldstein, R.C., eds.: Proc. 16th Int. Conf. Conceptual Modeling (ER'97), Springer, Berlin, LNCS 1331 (1997) 85–101

[15] Blaha, M., LaPlant, D., Marvak, E.: Requirements for repository software. In: Working Conf. on Reverse Engineering, IEEE Computer Society (1998) 164–173

[16] Terrasse, M.N., Savonnet, M., Becker, G.: A UML-based Metamodeling Architecture for Database Design. In: Proc. IDEAS'2001. (2001) 231–236

[17] Miura, T., Matsumoto, W., Shioya, I.: Managing Meta Objects for Design of Warehouse Data. In: Proc. DaWaK'1999. (1999) 33–40

[18] Tan, J., Zaslavsky, A.B., Bond, A.: Meta Object Approach to Database Schema Integration. In: Proc. DOA'2000. (2000) 145–154

[19] McClatchey, R., Kovacs, Z., Estrella, F., Goff, J.M.L., Varga, L., Zsenei, M.: The Role of Meta-Objects and Self-Description in an Engineering Data Warehouse. In: Proc. IDEAS'1999. (1999) 342–350

[20] Cranefield, S.: UML and the Semantic Web. In: Proc. Int. Semantic Web Working Symposium (SWWS). (2001)

[21] Bowers, S., Delcambre, L.: Representing and transforming model-based information. In: Proc. Int. Workshop on the Semantic Web at the 4th European Conference on Research and Advanced Technology for Digital Libraries (SemWeb). (2000)

[22] Franconi, E., Ng, G.: The i.com Tool for Intelligent Conceptual Modelling. In: Proc. 7th Intl. Workshop on Knowledge Representation meets Databases (KRDB'00). (2000)

[23] Gogolla, M., Lindow, A., Richters, M., Ziemann, P.: Metamodel Transformation of Data Models. In Bezivin, J., France, R., eds.: Proc. UML'2002 Workshop in Software Model Engineering (WiSME 2002). Technical Report, University of Nantes, http://www.metamodel.com/wisme-2002 (2002)