Tales of ER and RE Syntax and Semantics

Martin Gogolla

July 22, 2005

Abstract

This paper explains how four model transformations between database models work: (1) An ER (Entity-Relationship) database schema is transformed into a collection of ER database states, (2) a RE (Relational) database schema into a collection of RE database states, (3) an ER database schema into a RE database schema, and (4) a collection of ER database states into a collection of RE database states. These four separate transformations may be viewed as a single transformation between the ER datamodel and the RE datamodel.

The schemas are regarded as determining the syntax of the datamodels, and the set of associated states is regarded as being the semantics of the datamodels, because the states associate meaning with the schemas. When one usually considers database models, one formally only treats syntactical aspects, i.e., schemas, and handles the semantics merely informally. Our approach allows to formally handle syntax and semantics of database models and their transformation within a single and uniform framework. The approach thus allows to precisely describe properties of the datamodels and properties of the transformation.

The method behind our approach is to divide a language into a syntax and semantics part and to describe a transformation between two languages as a direction-neutral affair. Formal properties of the languages to be transformed and formal properties of the transformation are described uniformly. Transformation properties can be properties regarding syntax and semantics. The method can be applied not only to database languages but to transformations between common computer science languages.

Contents

List of Figures

Chapter 1

Introduction

Within the database community the Entity-Relationship (ER) and the Relational (RE) database models have been studied and used for decades. Both models are subject to introductory courses in database and software engineering education. A typical course will introduce the main concepts of both database models in an informal way, explain how to translate ER schemas into Relational database schemas and will deepen the matter by practical exercises using a design tool and a database system.

Recently, metamodeling has gained much attention, in particular in connection with the Unified Modeling Language (UML) and the Model Driven Architecture (MDA). This paper proposes another metamodel formulated with UML and OCL for the ER and RE database models. In contrast to known approaches, this paper however describes with its metamodel not only the concepts of the database schemas but also gives a formal interpretation, i.e., a formal semantics, to ER and RE database schemas: The paper formally connects database states to the schemas. Furthermore, the translation between the database models is also described in a formal way, and semantical properties of the translation are expressed based on the formal semantics of the database models. We are not aware of another approach handling these two classical database models with respect to syntax and semantics and their translation in a rigorous and uniform way. In particular, we are not aware of an approach expressing the ultimate goal of the schema translation process, namely the equivalence between the database state spaces for the different models, in a formal and explicit way.

The above two paragraphs made use of the notion *model* in two different meanings: *Database model* and *metamodel*. A *database model* denotes a collection of interrelated concepts for describing certain classes of databases: The ER database model knows, for example, entities, relationships, and instances; the Relational database model talks, for example, about Relational schemas, attributes, and tuples. The *metamodel* which we develop in this paper is a collection of UML classes, associations, and OCL constraints. This *metamodel* covers the two *database models* and their *transformation*.

The rest of paper is structured as follows. Chapter 2 will give an overview on the used metamodel. First, a package diagram will state the overall metamodel structure.

Second, both database models are sketched. Third, the concepts for the transformation are introduced.

Chapter 3 and 4 show the same first seven sections: After treating common concepts from the ER and RE database models, the ER syntax and the ER semantics and afterwards the RE syntax and the RE semantics are handled; then common datamodel concerns and thereafter the transformation is studied. Chapter 3 discusses the topics in form of object diagrams. Chapter 4 discusses the same topics by considering textual constraints. Chapter 4 finally shows the operations needed in the metamodels and explains a classification of the constraints.

Chapter 5 turns to the basic idea behind the modeling method used in this paper, gives other examples and discusses the role of tool support. Chapter 6 closes the paper with a conclusion.

Chapter 2

Overview by Class and Package Diagrams

2.1 Package Diagram with Class Names

Figure 2.1 shows the packages we have used and their dependencies. The package Base contains fundamental classes, associations, and constraints with elements common to both the ER and RE datamodel. The package ErSyn includes elements describing the syntax of the ER datamodel whereas the package ErSem covers elements relevant for the semantics of the ER datamodel. Analogously the package RelSyn and RelSem are responsible for the syntax and semantics of the RE datamodel. The package DataMods contains remaining syntactical and semantical elements only expressible for both datamodels together. In DataMods further invariants but no new classes or associations are introduced. The package Er2Rel describes modeling elements relevant for the transformation of between the ER datamodel and the RE datamodel.

A closer look at the dependencies in Fig. 2.1 should help in understanding the interconnections between and the structure of the packages. The package Base contains modeling elements for ER and RE attributes and data types and describes syntactical as well as semantical aspects. Both modeling elements, attributes and data types, occur in the ER and in the RE datamodel. Therefore these elements are considered separately. In order to describe the semantics of both data models one has to know their syntax beforehand, thus the semantic packages ErSem and RelSem rely on the syntax packages ErSyn and RelSyn. The packages DataMods and Er2Rel finally unite the two different lines for the ER and RE model in order (1) to express restrictions that each apply to both datamodels together and (2) to express the transformation between the datamodels.

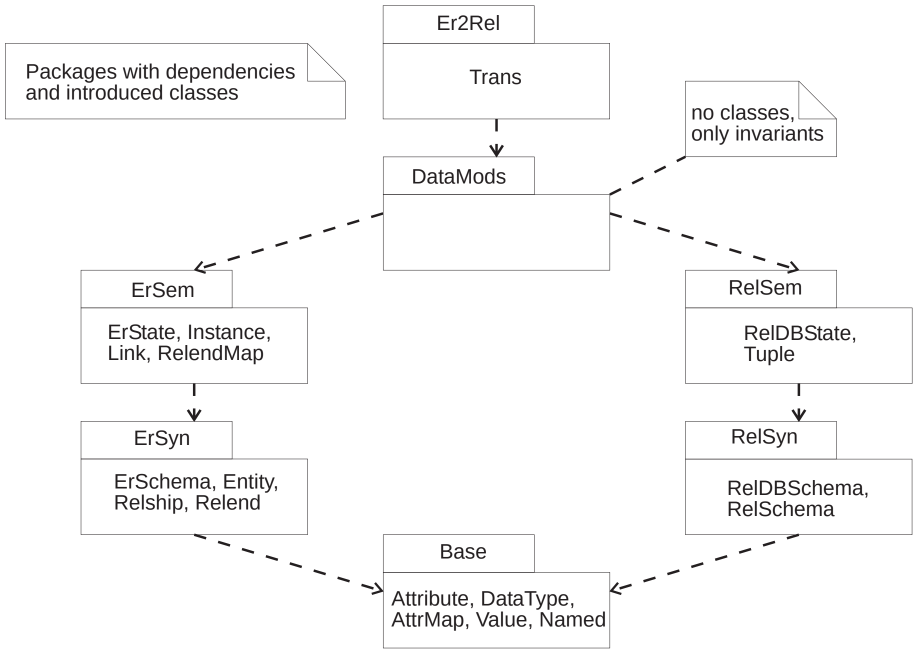

Packages with dependencies and introduced classes
Er2Rel
Trans
no classes, only invariants
DataMods
ErSem
ErState, Instance, Link, RelendMap
RelSem
RelDBState, Tuple
ErSyn
ErSchema, Entity, Relship, Relend
RelSyn
RelDBSchema, RelSchema
Base
Attribute, DataType, AttrMap, Value, Named

Figure 2.1: Package Diagram with Class Names

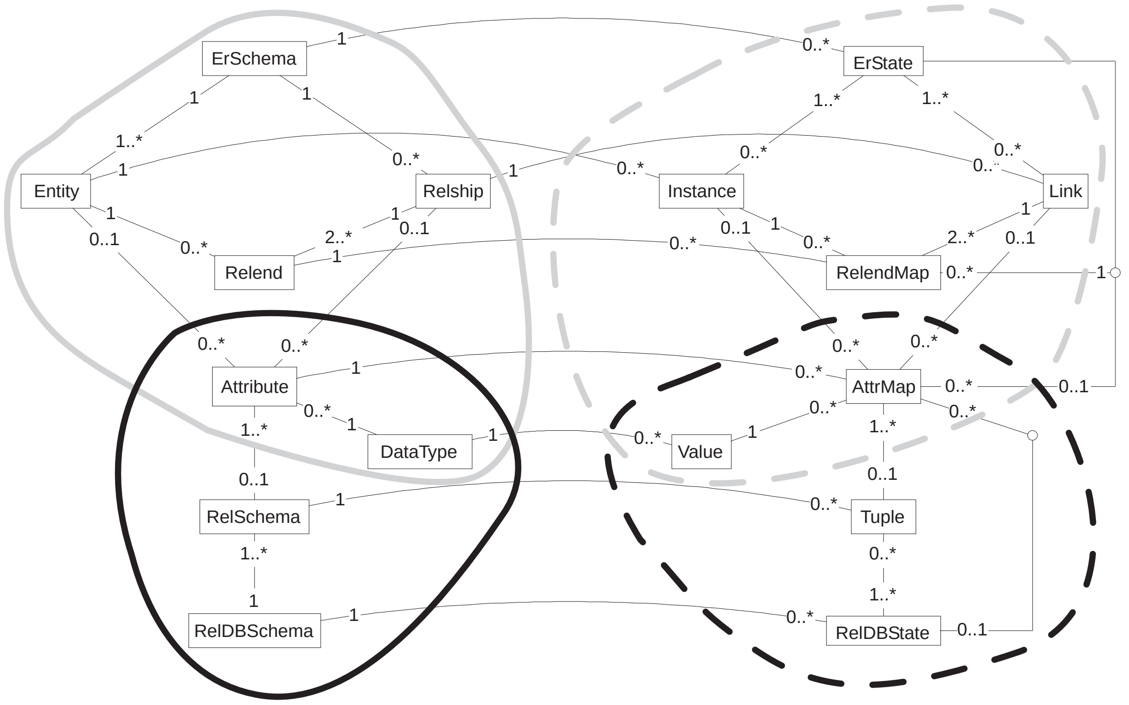


Figure 2.2: Class Diagram Modeling the ER and RE Datamodel

2.2 Modeling ER and RE Syntax and Semantics

The class diagram in Fig. 2.2 gives an overview on central classes and associations by showing four clouds: In the left part a solid grey and a solid black cloud, in the right part a dashed grey and a dashed black cloud. The two solid left clouds model the syntax of the datamodels, the two dashed right clouds the semantics. The top grey clouds describe the ER datamodel, the lower black clouds the RE datamodel. The ER and the RE datamodel share some concepts, namely the parts in the middle talking about data types, attributes and their semantics. The four clouds correspond to five packages in Fig. 2.1, namely to Base, ErSyn, ErSem, RelSyn, and RelSem. The intersection of the clouds corresponds to Base, the remaining part of the solid grey cloud to ErSyn, the remaining part of the dashed grey cloud to ErSem, the remaining part of the solid black cloud to RelSyn, and the remaining part of the dashed black cloud to RelSem. We have used only binary associations because our description is supposed to be a MOF-compliant one and MOF only knows binary associations.

Syntax of the ER datamodel: This part introduces the classes ErSchema, Entity, Relship, Relend, Attribute, and DataType. ErSchema objects consist of Entity and Relship objects which in turn may possess Attribute objects typed through DataType objects. Relend objects represent the connection points between the Relship objects and the Entity objects. All elements on the syntax side require an attribute expressing their name. Therefore, we have factored out this attribute, build an abstract class Named and made all classes on the syntax side subclasses of this class Named. The class Named is not shown in Fig. 2.2.

Semantics of the ER datamodel: In this part we set up the classes ErState, Instance, Link, RelendMap, AttrMap, and Value. The interpretation is as follows. An ErSchema object is interpreted by possibly many ErState objects. An Entity is given semantics by a set of Instance objects, and a Relship by a set of Link objects. DataType objects are given life through a set of Value objects. Relend and Attribute objects are interpreted by a set of RelendMap objects and AttrMap objects, respectively.

Syntax of the Relational datamodel: This part shows the classes RelDBSchema, RelSchema, Attribute, and DataType. RelDBSchema objects consist of RelSchema objects which possess Attribute objects typed through DataType objects. The difference in the notions RelDBSchema (Relational database schema) and RelSchema (Relational schema) is that a RelDBSchema consists of several RelSchemas or in other words that a collection of RelSchemas constitutes a RelDBSchema.

Semantics of the Relational datamodel: This last part utilizes the classes RelDBState, Tuple, AttrMap, and Value. RelDBSchema objects are interpreted by a set of RelDBState objects. Each RelDBState object consists of a set of Tuple objects. Tuple objects in turn consist of a set of AttrMap objects assigning a Value object to an Attribute within a Tuple and a RelDBState. A Tuple belongs to exactly one RelSchema as the multiplicity 1 on the association to RelSchema guarantees.

The class diagram and the resulting textual description follows certain naming and layout conventions apart from the principles *Syntax-Left-Semantics-Right* and *ER-Top-RE-Bottom*. Role names are not stated explicitly, but are always determined by the UML and OCL rule that unspecified role names consist of the respective class name with the first letter being given as a lower case letter. All associations within the syntax side are displayed with straight lines and the same holds for the associations within the semantics side, but the associations between syntax and semantics are shown with bent lines. These bent associations always have multiplicity 1 0..* and express that for every syntactical class in the left there is a semantical class in the right and that one syntactical object from the left is interpreted by a set of semantical objects from the right. These associations may be seen as a typing mechanism, because a semantical object is always associated with exactly one syntactical object, its type. Exactly as a syntax class from the left possesses a corresponding semantics class on the right, each syntax association from the left has a corresponding semantics association on the right. In addition the semantics side on the right has three associations between ErState and RelendMap, between ErState and AttrMap and between RelDBState and AttrMap which do not have corresponding syntactical parts on the left.

A further observation concerns implicitly present part-of relationships. Due to the multiplicities and constraints on the syntax side, every object on the left will be connected to a schema, and due to the multiplicities and constraints on the semantics side every object on the right will be connected to a state. This means any object originating from the left side will be connected directly or indirectly to either an ErSchema or a RelDBSchema object, and any object originating from the right side will be linked directly or indirectly to an ErState or a RelDBState object. The only exception to this rule are DataType and Value objects which may exist independently from schema objects.

2.3 Transformation between ER and RE

The class diagram in Fig. 2.3 describes the transformation between the ER datamodel and RE datamodel and shows the constituents of the package Er2Rel. It introduces one additional class Trans which represents transformation objects. A transformation object can express with its links that one ErSchema is transformed into a RelDBSchema together with the corresponding ErState objects transformed into RelDBState objects via the association between Trans and ErState and the association between Trans and RelDBState as well as via the association between ErState and RelDBState. This addition association between ErState and RelDBState expresses which ER state correspond to which RE state. This is not already expressed in the pure set of states associated with the transformation. Indeed, this last association could also more precisely be expressed as a ternary association with Trans as an additional class. Remember however, we use MOF, which offers only binary associations for modeling, and then this situation would have to be modeled with an additional class making the scenario much more complicated than the simpler class diagram which we have chosen.

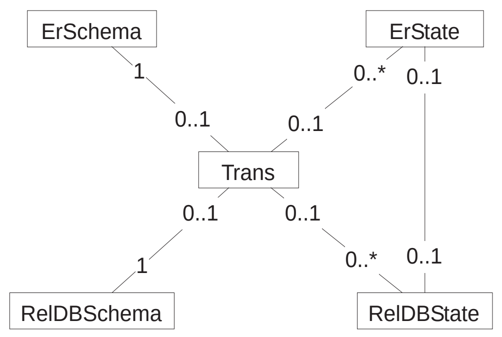


Figure 2.3: Class Diagram Modeling the Transformation between ER and RE

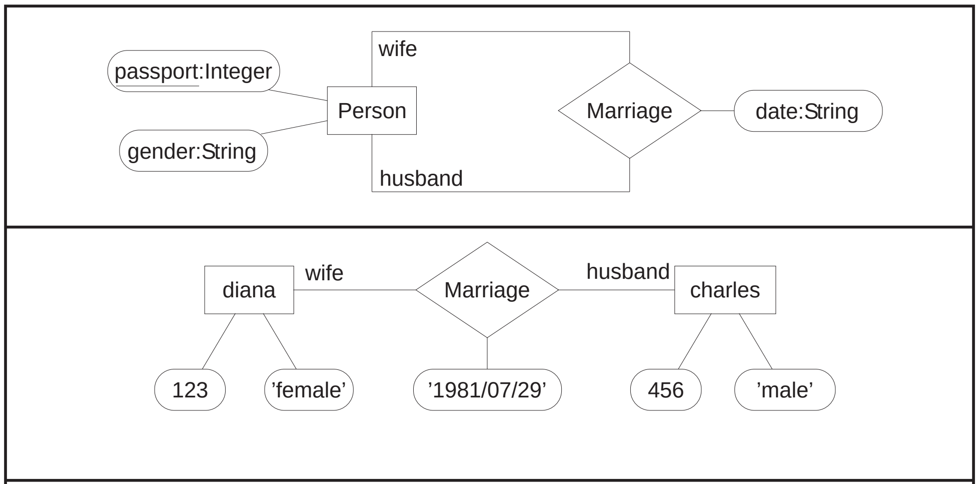
By expressing the transformation by a transformation class and intentionally not, e.g., by operations we do not introduce any direction into the transformation. One tends to think of the connection between the ER and RE model as *translating an ER schema into a RE database schema* (see our package name Er2Rel), but the opposite direction is worth to be considered for re-engineering purposes as well. Our design decision to represent transformations as classes together with appropriate associations does not impose any transformation direction. Forward and backward engineering techniques are covered in principle by this direction-neutral design.

Thus it is only one view on the class diagram in Fig. 2.3 that it represents a transformation from the top part to the lower part. Another view is that a transformation from the lower part to the top part is described. But one can view the class diagram also as describing a transformation from left part to the right part. That transformation is a relationship between syntactical elements, in this case pairs of schemas, and semantical elements, in this case pairs of states.

Chapter 3

Exploration by Object Diagrams

Figure 3.1 shows a simple scenario which we want to study in this chapter. Put in simple words, the scenario says: Diana marries Charles. The figure is structured into four parts starting on the top: The two top parts show the ER version of the scenario, the lower two parts the Relational version; for the ER version as well as the Relational version first a database schema and afterwards a state is shown.



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

```
 Person | passport | gender
--------+----------+----------
        | 123      | 'female'
        | 456      | 'male'

 Marriage | wife_passport | husband_passport | date
----------+---------------+------------------+--------------
          | 123           | 456              | '1981/07/29'
```

Figure 3.1: Example Scenario *Diana marries Charles*

The ER database schema displays one entity Person having two attributes, namely passport number and gender, with the key attribute passport being underlined and one reflexive relationship Marriage with indicated relationship end names wife and husband and one relationship attribute date. The ER state has two instances for entity Person, diana and charles, and assigns a passport and a gender value to both. The ER state also pictures a link between the two instances expressing that their marriage took place on 1981/07/29.

The Relational database schema possesses two Relational schemas originating from the translation of the ER database schema. The entity Person is described by a Relational schema Person having the same attributes as the entity. The relationship is represented by a Relational schema Marriage having in particular two attributes, one for each relationship end, where the attribute names must utilize the relationship end names and the name of the key attribute: wife_passport and husband_passport. The facts from the ER state, i.e., two instances and one link, are represented in the Relational state by three tuples.

3.1 Base

We now want to represent the four parts from Fig. 3.1 as instantiations of the meta-model we have sketched before. We start with the package Base. Since the ER database schema and the RE database schema both utilize the data types Integer and String,

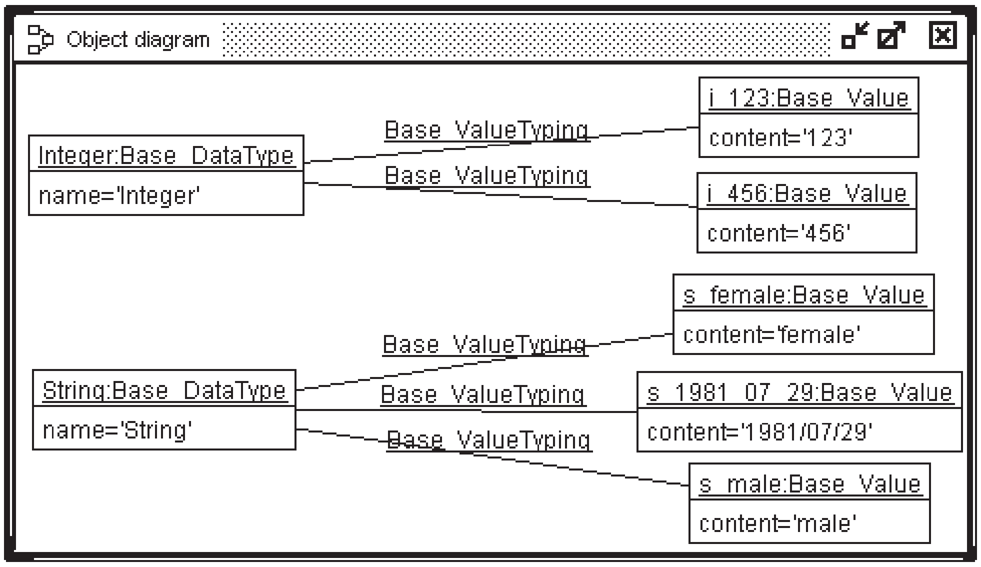


Figure 3.2: Object Diagram with Data Types and Values

we first introduce these data types. Both data types are pictured in Fig. 3.2 as objects of class DataType. In Fig. 3.2 also Value objects are shown which represent the semantical interpretation of DataType objects. One data type can have a set of values associated with it. The five Value objects have been given object identifiers which already indicate their Content attribute. Alternatively, we could have denoted these objects, for example, with object identifiers value1 to value5. Then, we had to use these object identifiers in the object rectangles before the colons instead of the object identifiers from Fig. 3.2, but the Content attribute would be as in Fig. 3.2.

Constraints have to restrict the possible object diagrams allowed by the class diagram in the package Base. For example, there will be a constraint guaranteeing a unique representation of each value. Thus it will be forbidden to have another Value object called i_123B being also linked to data type Integer and having also content='123'. In the package Base also the class Attribute and AttrMap are present. Thus, for example, the attribute passport from the ER schema or the attribute passport from the RE schema belong to this package. Concrete objects for this part will be displayed later in connection with the database schemas and database states.

3.2 ER Syntax

The ER database schema is shown as an object diagram in Fig. 3.3. All object identifiers (the part before the colon in the object rectangles) end with Er whereas the forthcoming RE database schema will show Rel as endings. The ER schema itself being

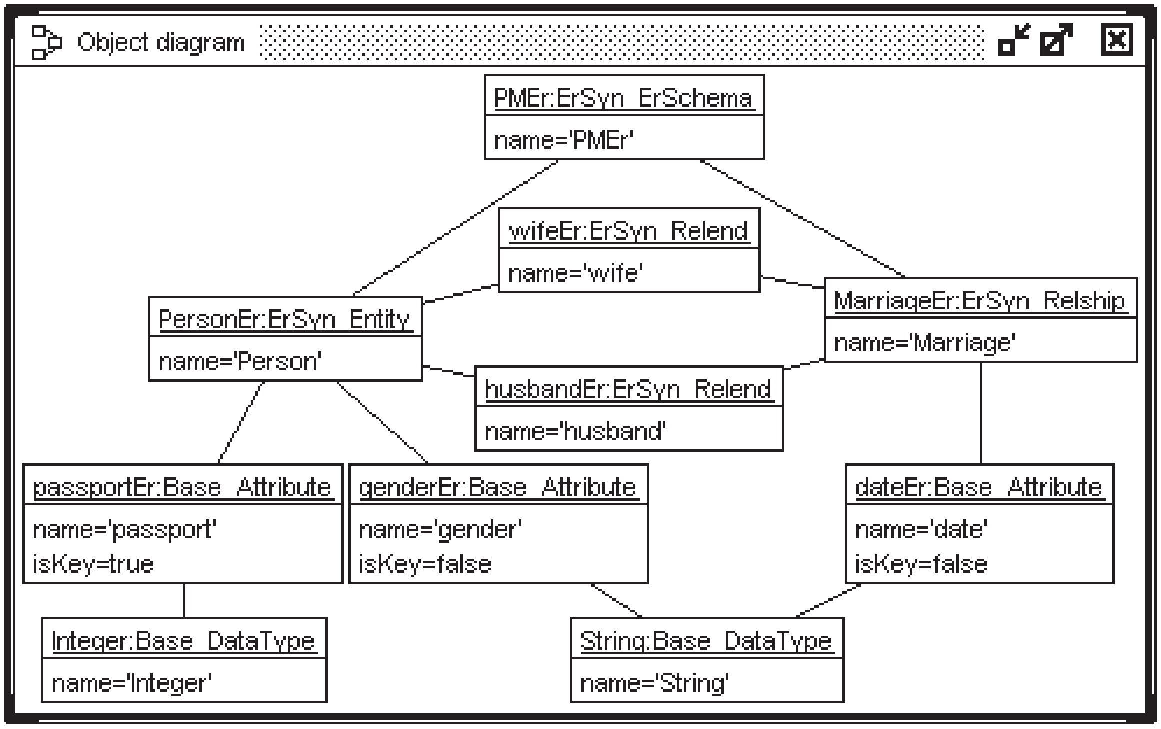


Figure 3.3: Object Diagram with ER Schema

not explicitly present in Fig. 3.1 is called PMEr as a shorthand for Person-Marriage. This object diagram introduces one ErSchema object, one Entity object, two Attribute objects for the entity, one Relship object, two Relend objects for the relationship ends, and one relationship Attribute object. Links establish proper connections so that the graphically represented ER schema is captured formally as an object diagram.

Constraints have to restrict the class diagram so that only meaningful object diagrams are allowed to be drawn. For example, with respect to the class diagram only it would be allowed to have an additional link in Fig. 3.3 between the attribute dateER and the entity PersonER. This has to be excluded.

3.3 ER Semantics

Following the overview diagrams in Fig. 3.1 we now come to the ER state being given in Fig. 3.4. First, we identify the objects: One ErState object, two Instance objects, one Link object, two RelendMap objects for the link ends, and five AttrMap objects for the attribute values. Second, links establish the proper connections.

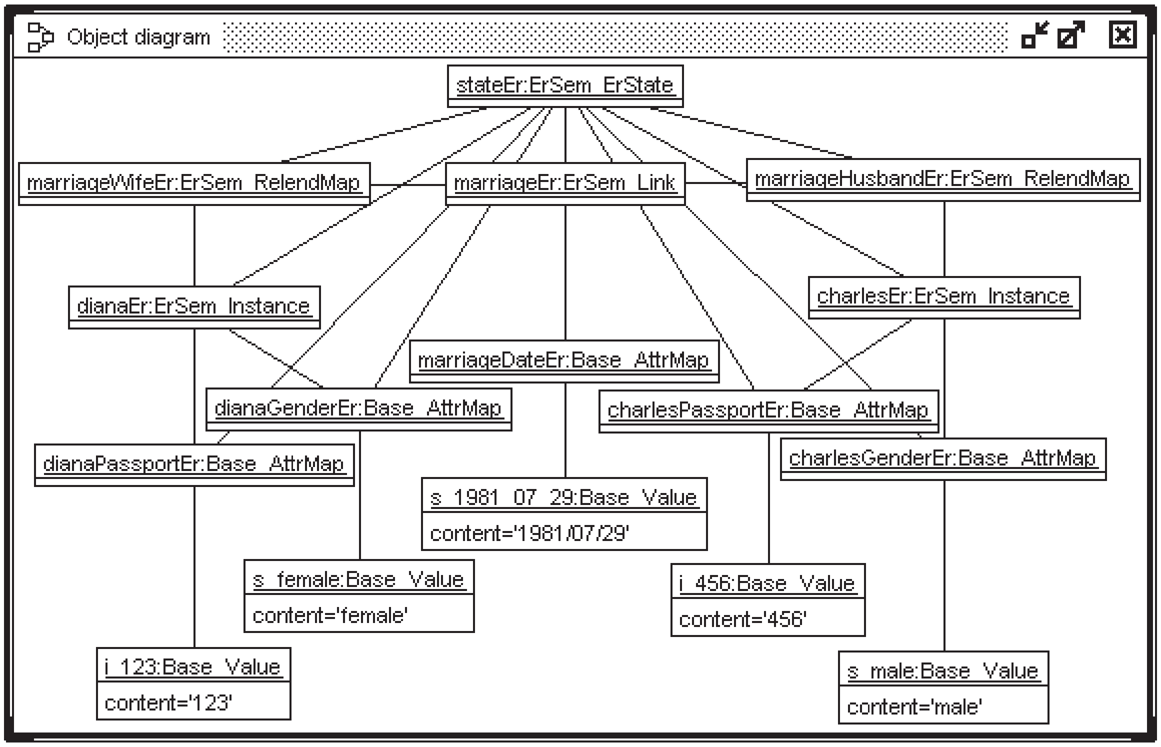


Figure 3.4: Object Diagram with ER State

As before, constraints must restrict the possible object diagrams. For example, the above object diagram would become invalid, if the link between dianaPassportEr and i_123 would be replaced by a link between dianaPassportEr and s_female: Attribute typing restrictions would be violated.

The lower part of the object diagram with objects of class Attribute and Value belongs to the package Base. Remember that data types and values as well as attributes and attribute maps are also used by the RE datamodel. The object diagram shows semantical objects only, but remember that all semantical objects are typed by syntactical objects. Thus all objects in the above ER state diagram possess links not shown here, but present in the complete object diagram. This fact is emphasized in Fig. 3.5.

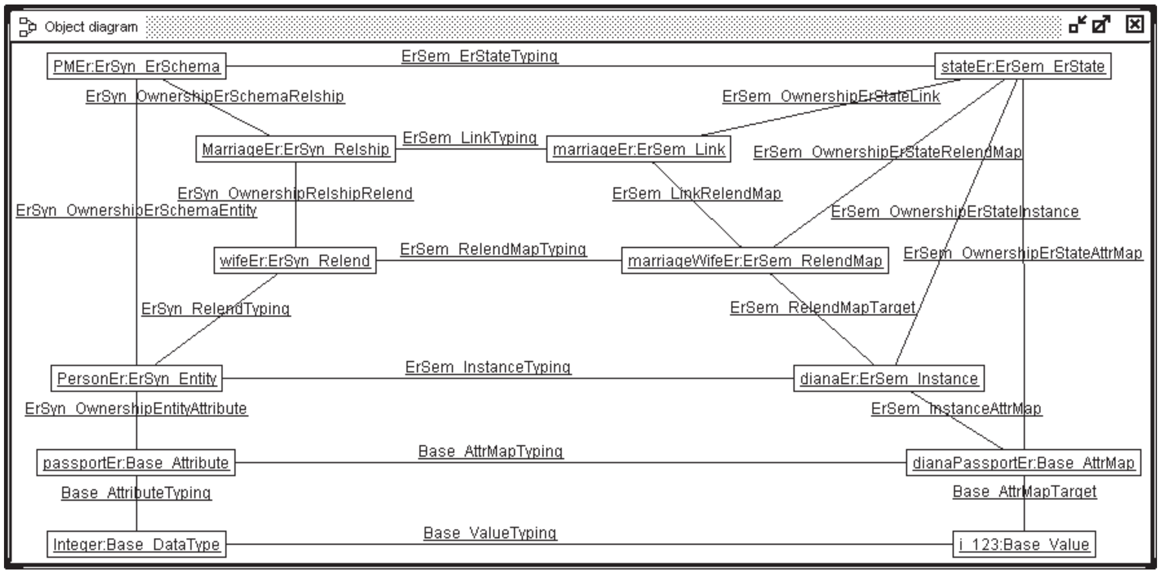


Figure 3.5: Object Diagram for Interplay between Syntax and Semantics

In the left of Fig. 3.5, part of the objects belonging to the ER schema are shown and, in the right of the figure, part of the objects belonging to the complete ER state are displayed. The links in the middle are all typing links. For example, the semantical object stateER is typed by the syntactical object PMEr, the semantical object dianaEr is typed by the syntactical object PersonEr and the semantical object i_123 is typed by the syntactical object Integer. Now this figure also allows to explain the AttrMap objects and their links: The AttrMap object dianaPassportEr with its links to stateER, dianaEr, passportER, and i_123 expresses that in the given ER state the instance diana receives with respect to the attribute passport the value 123. An analogous modeling is chosen for RelendMap objects.

Up to now we have considered scenarios where an ER schema is associated with a single state only. But in general, an ER schema may be interpreted by many ER states. Figure 3.6 shows an object diagram involving one ER schema and two ER states. The Instance object diana as well the AttrMap object assigning the name attribute are shared between the two states. This object diagram also explains why each AttrMap object must be linked to its four connected objects, i.e., an ErState, an Instance, an Attribute, and a Value. One can check: If one of these four links would be missing, one could not express the change of Diana's address properly under the assumption that a single Instance object has to represent Diana in both states.

Figure 3.7 shows a variation of the previous scenario. The ER schema is the same as before (except the object identifiers), however the two states are different. As before, the two different states share one Instance object, but both attribute values have

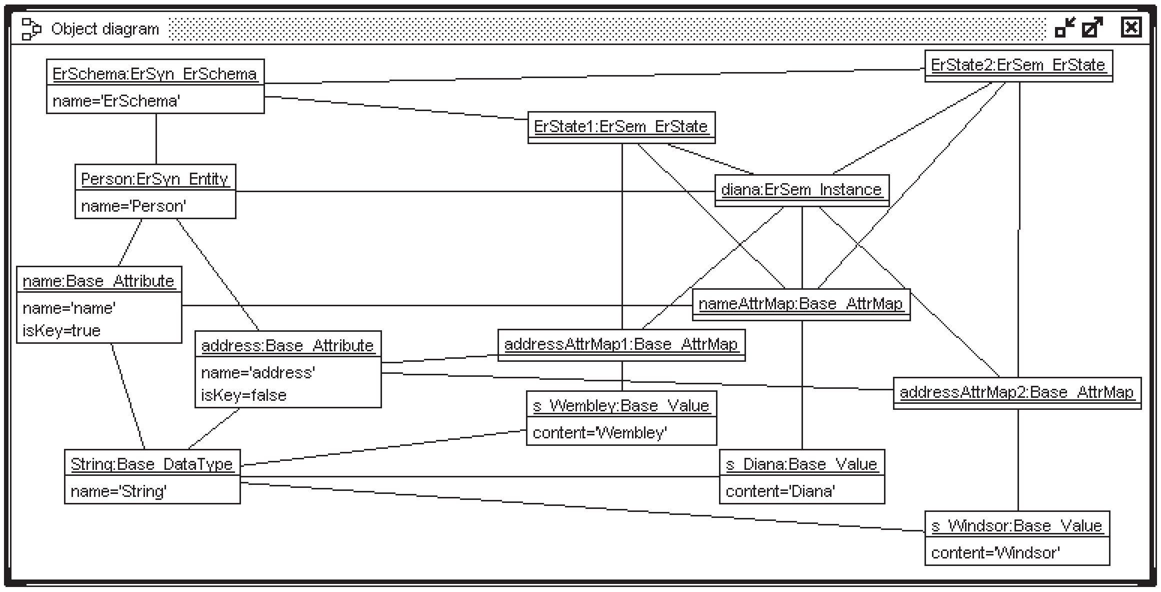


Figure 3.6: Object Diagram for *Diana moves from Wembley to Windsor*

changed between the first and the second state, even the key attribute value. Although both states show completely different values, both states share the common Instance object and thus Diana is present in both states, in the first state under the name 'Diana Spencer', in the second state under the name 'Diana Mountbatten-Windsor'.

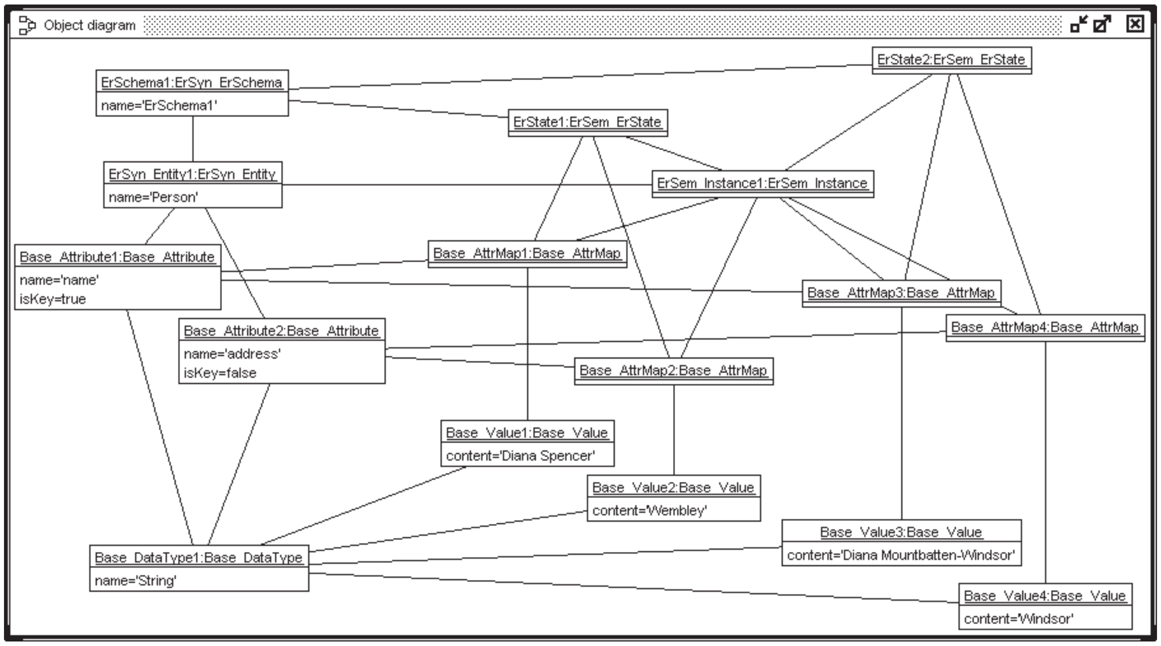


Figure 3.7: Object Diagram for *Diana moves and marries*

3.4 RE Syntax

Next we come to the Relational database schema which is pictured in Fig. 3.8. As in the case of the ER database schema respective objects are shown, in this case, one RelDBSchema object, two RelSchema objects, and five Attribute objects. Recall that a Relational database schema can consist of many Relational schemas. Proper links are established the correct object connections. All object identifiers in the RE part end with Rel in contrast to the objects in the ER part which all ended with Er.

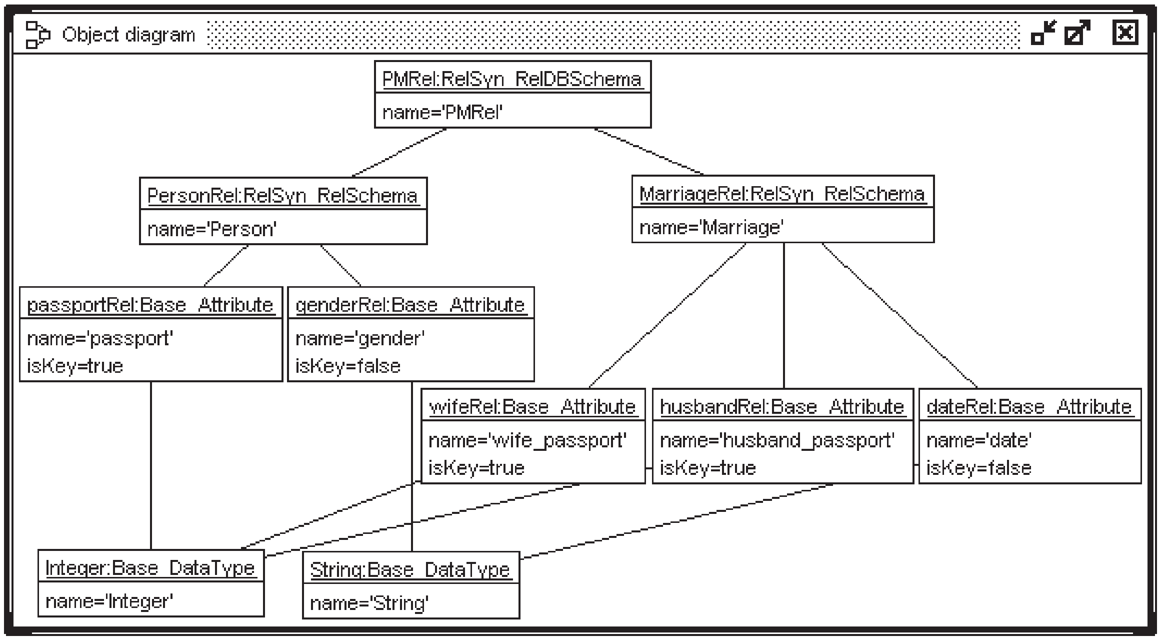


Figure 3.8: Object Diagram with RE Schema

In order to sketch an example for a needed constraint, imagine we would not have name='wife_passport' and name='husband_passport' for wifeRel and husbandRel but we would have name='passport' for both objects. Then one would have two different attributes with the same name in one RE schema. This has to be excluded.

Comparing the ER database schema and RE database schema one finds that both are represented by ten objects and twelve links, however the RE schema has a flat structure with two Relational schemas whereas the ER database schema exhibits its non-flat structure because the relationship depends on the entity.

3.5 RE Semantics

The last part in the overview diagram in Fig. 3.1 was the RE state. The corresponding screenshot showing the RE semantics is displayed in Fig. 3.9. Analogously to the flat RE schema structure, the RE state represents the two Instance objects and the single

Link object from the ER state homogeneously with three Tuple objects in a flat, hierarchical manner.

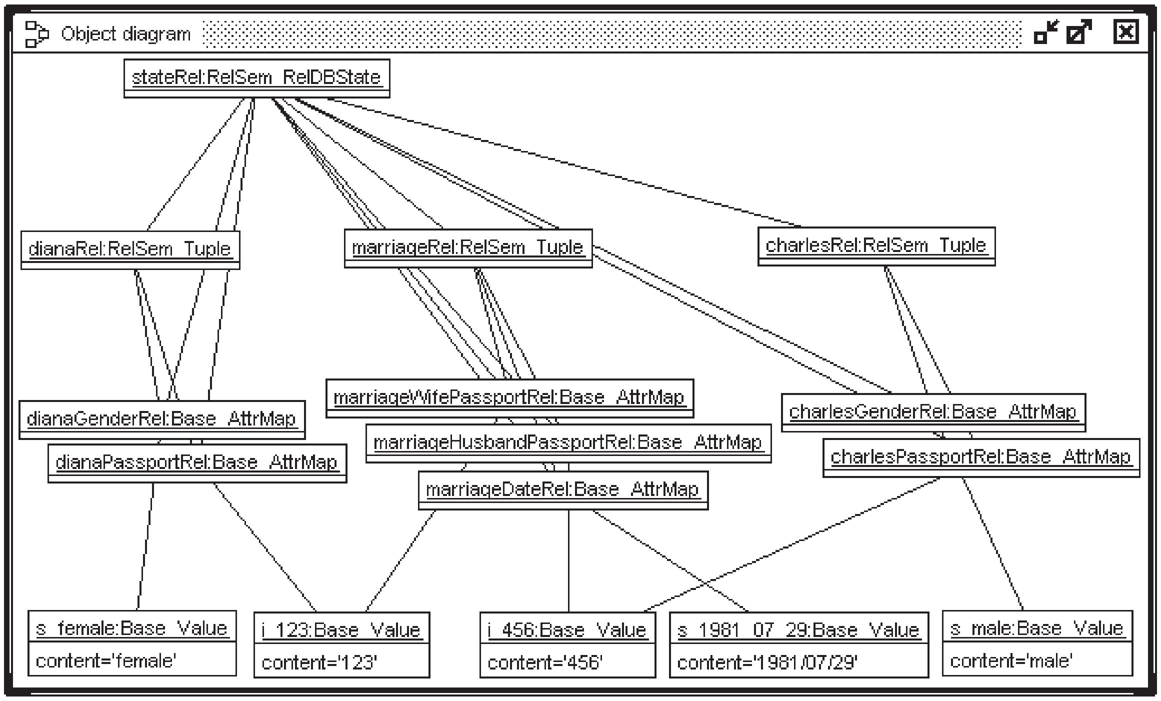


Figure 3.9: Object Diagram with RE State

To point out once again the need of textual constraints, imagine the link between charlesPassportRel and i_456 would be replaced by a link between charlesPassportRel and i_123 (different passport number for Charles). Then one would have two different Tuple objects in the same relation (Person) with the same key value. This must be avoided.

Comparing the ER state and RE database state one finds that both are represented by 16 objects and 17 links. However, the ER state has a greater degree of indirection and with this a better shelter against possible update errors and a better shelter against inconsistency: If, e.g., Charles receives a new passport number, this would be reflected in the ER state by updating a single AttrMap link, whereas this would induce the update of two AttrMap links in the RE state. This higher degree of indirection can be seen as one of the reasons for calling the ER datamodel a semantic datamodel in comparison to the RE datamodel which is classified as value-based.

3.6 Common Datamodel Concerns

The package DataMods introduces new invariants but no new classes. One of these invariants will guarantee on the schema side that an attribute either belongs to an ER entity or an ER relship or a RE schema. The other invariants will require on the state side (1) that an attribute map is either linked to an ER instance or an ER link or a RE tuple and (2) that an attribute map is either linked to an ER database state or a RE database state.

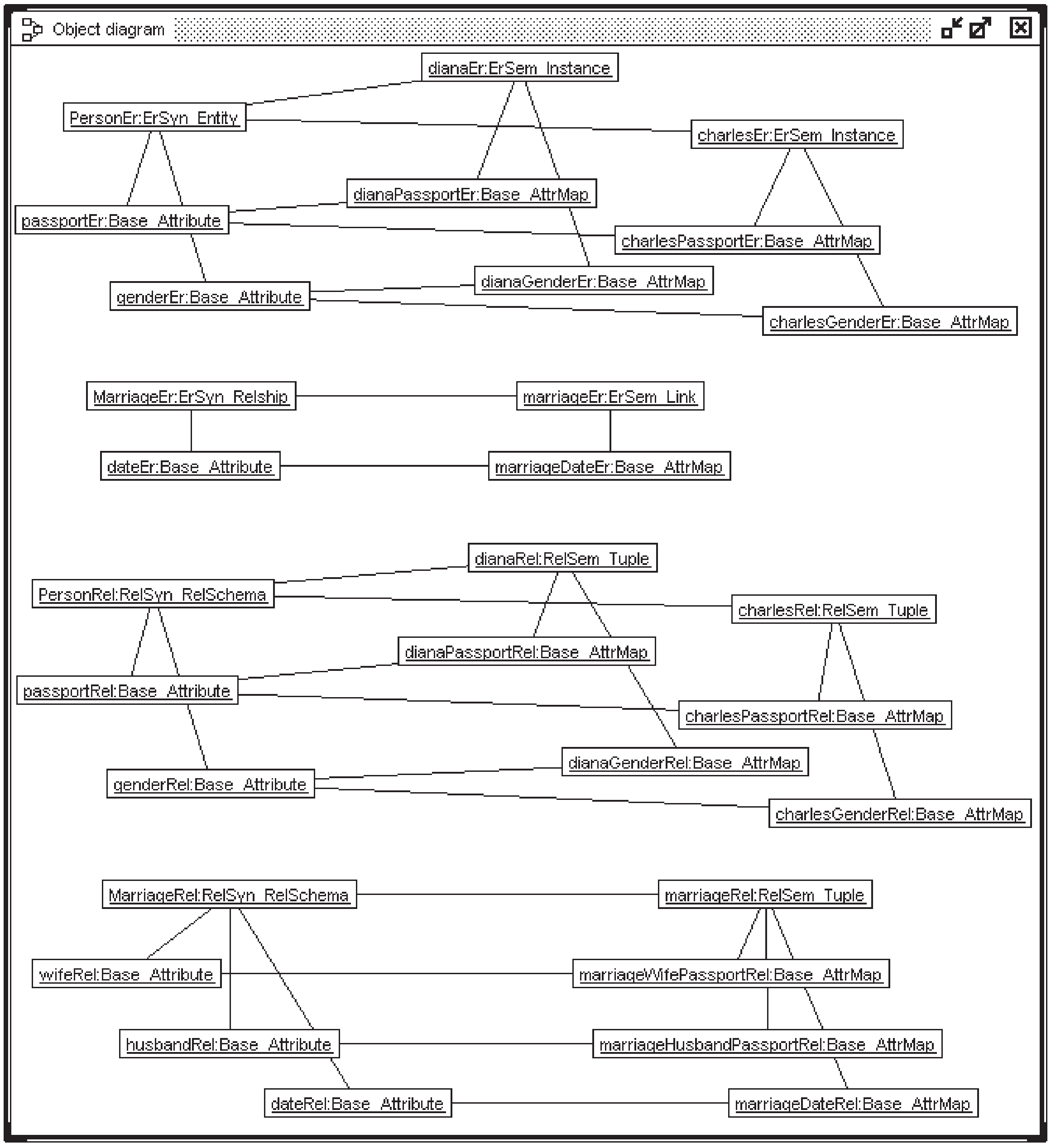

Object diagram
dianaEr:ErSem_Instance
PersonEr:ErSyn_Entity
charlesEr:ErSem_Instance
dianaPassportEr:Base_AttrMap
passportEr:Base_Attribute
charlesPassportEr:Base_AttrMap
dianaGenderEr:Base_AttrMap
genderEr:Base_Attribute
charlesGenderEr:Base_AttrMap
MarriageEr:ErSyn_Relship
marriageEr:ErSem_Link
dateEr:Base_Attribute
marriageDateEr:Base_AttrMap
dianaRel:RelSem_Tuple
PersonRel:RelSyn_RelSchema
charlesRel:RelSem_Tuple
dianaPassportRel:Base_AttrMap
passportRel:Base_Attribute
charlesPassportRel:Base_AttrMap
dianaGenderRel:Base_AttrMap
genderRel:Base_Attribute
charlesGenderRel:Base_AttrMap
MarriageRel:RelSyn_RelSchema
marriageRel:RelSem_Tuple
wifeRel:Base_Attribute
marriageWifePassportRel:Base_AttrMap
husbandRel:Base_Attribute
marriageHusbandPassportRel:Base_AttrMap
dateRel:Base_Attribute
marriageDateRel:Base_AttrMap

Figure 3.10: Object Diagram for Attribute and AttrMap Owners

Figure 3.10 captures this for the example scenario. As usual, the left side displays schema aspects whereas the right side captures state aspects, the top part covers the ER datamodel and the bottom part the RE datamodel. The left side shows for each Attribute object to which Entity, Relship, or RelSchema object it is linked to. The right side explains to which Instance, Link, or Tuple each AttrMap object is connected to. There are no attributes connected to more than one schema element, and there are no attribute maps connected to more than one state. Thus, if we would, for example, replace the link between genderEr and PersonEr by a link between genderRel and PersonEr, we would have a constraint violation, because genderRel would have two associated schema objects and genderEr would be connected to no schema object. Figure 3.10 does not show the ErState or RelDBState objects to which the AttrMap objects are connected to.

3.7 Transformation

What is now still missing is the transformation object which connects the schemas and the states by proper links. The corresponding screenshot displaying the ErSchema object, the ErState object, the RelDBSchema object, the RelDBState object, and the Trans object can be found in Fig. 3.11. In this simple scenario with only one state for the ER and RE datamodel the additional link between the states seems unnecessary. However, if more than two states were captured such links are needed.

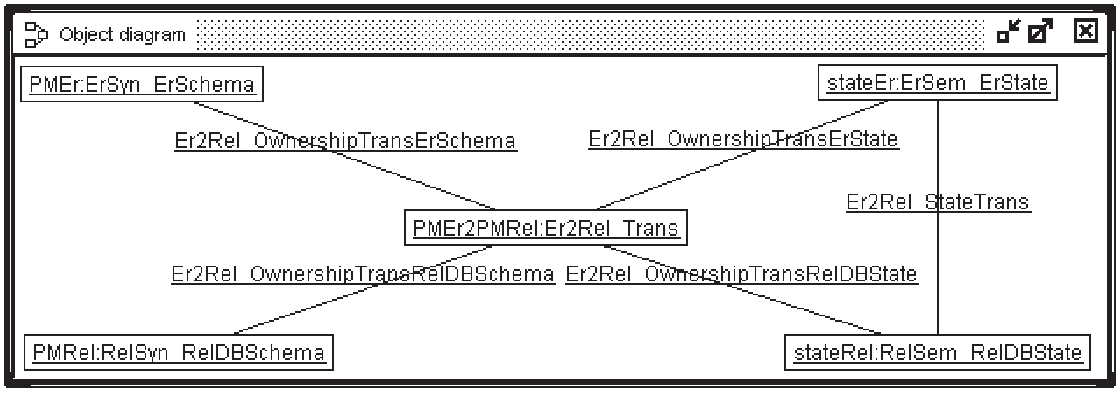

Object diagram
PMEr:ErSyn_ErSchema
stateEr:ErSem_ErState
Er2Rel_OwnershipTransErSchema
Er2Rel_OwnershipTransErState
Er2Rel_StateTrans
PMEr2PMRel:Er2Rel_Trans
Er2Rel_OwnershipTransRelDBSchema
Er2Rel_OwnershipTransRelDBState
PMRel:RelSyn_RelDBSchema
stateRel:RelSem_RelDBState

Figure 3.11: Object Diagram with Transformation

A constraint violation would occur, if we would drop, for example, the link between PMEr2PMRel and stateRel, because then there would be a RE state connected via the ER state to the transformation object, but not connected directly to the transformation object.

Chapter 4

Details of Textual Constraints

We now show the details of all constraints. Some of the constraint have been mentioned before when we discussed the object diagrams. We will explain all constraints in a systematic manner. Thus this presentation has more the character of a reference description than being regular running text.

We cover the seven packages in the order we have handled them before. On first reading, the reader may turn to the first constraint in the package only which represents a typical package constraint or to one of the following typical constraints:

1. Base::Base_Value::differentContentOrDataType
2. ErSyn::ErSyn_ErSchema::uniqueEntityNamesWithinErSchema
3. ErSem::ErSem_Instance::keyMapUnique
4. RelSyn::RelSyn_RelSchema::relSchemaKeyNotEmpty
5. RelSem::RelSem_Tuple::keyMapUnique
6. DataMods::Base_Attribute::linkedToOneOfInstanceLinkTuple
7. Er2Rel::Er2Rel_Trans::forTupleExistsOneInstanceXorLink

The naming convention is:

PackageDefiningTheConstraint::PackageDefiningTheClass_Class::Constraint.

4.1 Base

Naming restriction: Names are defined, have a non-zero length, and consist of letters, digits and the underscore.

```
context self:Base_Named inv nameOk:
  let small:Set(String)=
    Set{'a','b','c','d','e','f','g','h','i','j','k','l','m',
        'n','o','p','q','r','s','t','u','v','w','x','y','z'} in
  let capital:Set(String)=
    Set{'A','B','C','D','E','F','G','H','I','J','K','L','M',
        'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'} in
  let digit:Set(String)=
    Set{'0','1','2','3','4','5','6','7','8','9'} in
  self.name.isDefined and self.name.size>0 and
  Set{1..self.name.size}->forAll(i |
    Set{'_'}->union(small)->union(capital)->union(digit)->
      includes(self.name.substring(i,i)))
```

Distinguishability of values: Different Values have different content or are linked to different DataTypes.

```
context self:Base_Value inv differentContentOrDataType:
  Base_Value.allInstances->forAll(self2 |
    self<>self2 implies
    (self.content<>self2.content or self.dataType<>self2.dataType))
```

Commutativity restriction: The DataType of the Attribute of an AttrMap is identical to the DataType of the Value of the AttrMap.

Remark: All commutativity restriction constraint names will start with `c_` and will mention the visited classes in the constraint name.

```
context self:Base_AttrMap inv c_AttrMap_Attribute_Value_DataType:
  self.attribute.dataType=self.value.dataType
```

Naming restriction: Different DataTypes have different names.

```
context self:Base_DataType inv uniqueDataTypeNames:
  Base_DataType.allInstances->
    forAll(self2 | self.name=self2.name implies self=self2)
```

4.2 ER Syntax

Commutativity restriction: The ErSchema of the Entity of a Relend is identical to the ErSchema of the Relship of the Relend.

```
context self:ErSyn_Relend inv c_Relend_Entity_Relship_ErSchema:
  self.entity.erSchema=self.relship.erSchema
```

Naming restriction: Different ErSchemas have different names.

```
context self:ErSyn_ErSchema inv uniqueErSchemaNames:
  ErSyn_ErSchema.allInstances->
    forAll(self2 | self.name=self2.name implies self=self2)
```

Naming restriction: Within one ErSchema, different Entities have different names.

```
context self:ErSyn_ErSchema inv uniqueEntityNamesWithinErSchema:
  self.entity->forAll(e1,e2 | e1.name=e2.name implies e1=e2)
```

Naming restriction: Within one ErSchema, different Relships have different names.

```
context self:ErSyn_ErSchema inv uniqueRelshipNamesWithinErSchema:
  self.relship->forAll(r1,r2 | r1.name=r2.name implies r1=r2)
```

Naming restriction: Within one ErSchema, Entities and Relships have different names.

```
context self:ErSyn_ErSchema
  inv differentEntityAndRelshipNamesWithinErSchema:
  self.entity->forAll(e | self.relship->forAll(r | e.name<>r.name))
```

Naming restriction: Within one Relship, different Relends have different names.

```
context self:ErSyn_Relship inv uniqueRelendNamesWithinRelship:
  self.relend->forAll(re1,re2 | re1.name=re2.name implies re1=re2)
```

Naming restriction: Within one Entity, different Attributes have different names.

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

Naming restriction: Within one Relship, different Attributes have different names.

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

Naming restriction: Within one Entity, opposite side Relends and Attributes have different names.

```
context self:ErSyn_Entity
  inv differentOsRelendAndAttributeNamesWithinEntity:
  self.osRelend()->forAll(re | self.attribute->forAll(a |
    re.name<>a.name))
```

Naming restriction: Within one Relship, Relends and Attributes have different names.

```
context self:ErSyn_Relship
  inv differentRelendAndAttributeNamesWithinRelship:
  self.relend->forAll(re | self.attribute->forAll(a | re.name<>a.name))
```

Naming restriction: Within one Entity, different opposite side Relends have different names.

```
context self:ErSyn_Entity inv uniqueOsRelendNamesWithinEntity:
  self.osRelend()->forAll(re1,re2 | re1.name=re2.name implies re1=re2)
```

Key restriction: The set of key attributes of an Entity is not empty.

```
context self:ErSyn_Entity inv entityKeyNotEmpty:
  self.key()->notEmpty
```

Key restriction: The set of key attributes of a Relship is empty.

```
context self:ErSyn_Relship inv relshipKeyEmpty:
  self.attribute->select(a | a.isKey)->isEmpty
```

4.3 ER Semantics

The most difficult but probably also the most interesting modeling concept on the semantics side are Maps. They occur in the class diagram in Fig. 2.2 in form of the classes AttrMap and RelendMap. AttrMaps are specialized in the ER database model part to InstanceAttrMaps and to LinkAttrMaps, and in the Relational datamodel part to TupleAttrMaps. Figure 4.1 shows these specializations as subdiagrams of the original class diagram in Fig. 2.2. In contrast to the original class diagram in Fig. 2.2, Maps are shown as four-ary associations.

InstanceAttrMaps describe that an Attribute evaluates in an ErState with respect to an Instance to a particular Value.

LinkAttrMaps describe that an Attribute evaluates in an ErState with respect to a Link to a particular Value.

RelendMaps describe that a Relend evaluates in an ErState with respect to an Link to a particular Instance.

TupleAttrMaps describe that an Attribute evaluates in a RelDBState with respect to a Tuple to a particular Value.

A set of Map objects determines a function with three arguments. Each argument is represented by one arm of the relationship. The result class of the function is the class of the fourth remaining arm. We have pictured the Maps in Figure 4.1 as relationships and have indicated the result class of the underlying function with an arrow. The constraints to follow formally guarantee the functional restrictions and assure that two different Map objects differ in their argument arms. For example, two different InstanceAttrMap objects differ already in their Attribute, Instance, or ErState arm. The diagrams in Figure 4.1 have merely illustrating character. They are not valid UML class diagrams, among other reasons, due to the multiplicities on the diamond side of the association arms. It is the aim of the following constraints to realize the intent explained before.

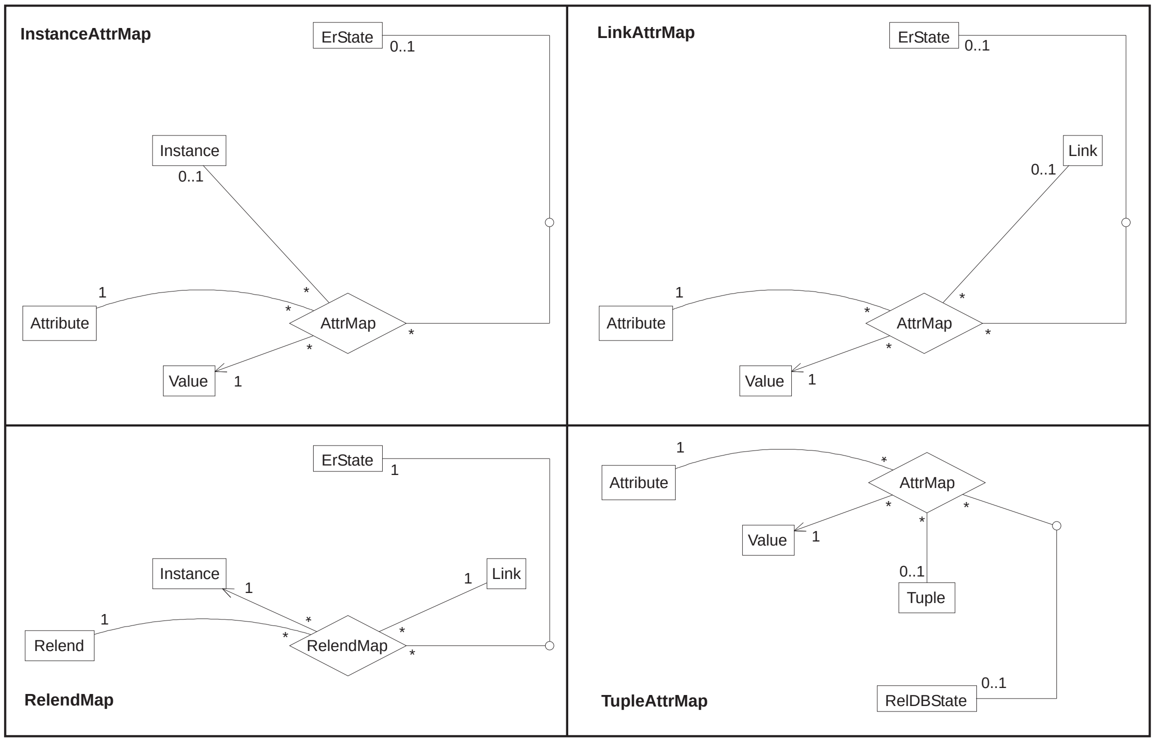


Figure 4.1: InstanceAttrMap, LinkAttrMap, RelendMap, and TupleAttrMap

Functional restriction: An InstanceAttrMap, i.e., an AttrMap being linked to an Instance, represents a non-redundant, functional assignment of a Value to an Attribute of an Entity within an ErState for the given Instance.

```
context self:Base_AttrMap inv instanceAttrMapIsFunction:
  Base_AttrMap.allInstances->forAll(self2 |
    (self<>self2 and self.instance->size=1 and self2.instance->size=1)
    implies
      ((self.attribute=self2.attribute and self.instance=self2.instance)
       implies
       (self.erState<>self2.erState and self.value<>self2.value)))
```

Functional restriction: A LinkAttrMap, i.e., an AttrMap being linked to a Link, represents a non-redundant, functional assignment of a Value to an Attribute of a Relship within an ErState for the given Link.

```
context self:Base_AttrMap inv linkAttrMapIsFunction:
  Base_AttrMap.allInstances->forAll(self2 |
    (self<>self2 and self.link->size=1 and self2.link->size=1)
    implies
    ((self.attribute=self2.attribute and self.link=self2.link)
     implies
     (self.erState<>self2.erState and self.value<>self.value)))
```

Functional restriction: A RelendMap represents a non-redundant, functional assignment of an Instance to a Relend within an ErState for a given Link.

```
context self:ErSem_RelendMap inv relendMapIsFunction:
  ErSem_RelendMap.allInstances->forAll(self2 | self<>self2 implies
    ((self.relend=self2.relend and self.link=self2.link)
     implies
     (self.erState<>self2.erState and self.instance<>self.instance)))
```

Remark: A whole bunch of commutativity restrictions follow. First, we show commutativity restrictions touching ErSem and ErSyn classes. Second, we show commutativity restrictions touching only ErSem classes.

Commutativity restriction: The Attributes of the Entity of an Instance are identical to the Attributes of the AttrMaps of the Instance; in other words, there are Attribute assignments for all Attributes of an Instance.

```
context self:ErSem_Instance inv c_Instance_Entity_AttrMap_Attribute:
  self.entity.attribute=self.attrMap.attribute->asSet
```

Commutativity restriction: The Attributes of the Relship of a Link are identical to the Attributes of the AttrMaps of the Link; in other words, there are Attribute assignments for all Attributes of a Link.

```
context self:ErSem_Link inv c_Link_Relship_AttrMap_Attribute:
  self.relship.attribute=self.attrMap.attribute->asSet
```

Commutativity restriction: The Relends of the Relship of a Link are identical to the Relends of the RelendMaps of the Link; in other words, there are Relend assignments for all Relends of a Link.

```
context self:ErSem_Link inv c_Link_Relship_RelendMap_Relend:
  self.relship.relend=self.relendMap.relend->asSet
```

Commutativity restriction: The Entity of the Relend of a RelendMap is identical to the Entity of the Instance of the RelendMap.

```
context self:ErSem_RelendMap inv c_RelendMap_Relend_Instance_Entity:
  self.relend.entity=self.instance.entity
```

Commutativity restriction: The Relship of the Relend of a RelendMap is identical to the Relship of the Link of the RelendMap.

```
context self:ErSem_RelendMap inv c_RelendMap_Relend_Link_Relship:
  self.relend.relship=self.link.relship
```

Commutativity restriction: The ErSchema of the ErState of an Instance is identical to the ErSchema of the Entity of the Instance.

```
context self:ErSem_Instance inv c_Instance_Entity_ErState_ErSchema:
  Set{self.entity.erSchema}=self.erState.erSchema->asSet
```

Commutativity restriction: The ErSchema of the ErState of a Link is identical to the ErSchema of the Relship of the Link.

```
context self:ErSem_Link inv c_Link_Relship_ErState_ErSchema:
  Set{self.relship.erSchema}=self.erState.erSchema->asSet
```

Commutativity restriction: The Entity of the Instance of an AttrMap being an InstanceAttrMap is identical to the Entity of the Attribute of the AttrMap.

```
context self:Base_AttrMap inv c_AttrMap_Attribute_Instance_Entity:
  self.attribute.entity=self.instance.entity
```

Commutativity restriction: The Relship of the Link of an AttrMap being a LinkAttrMap is identical to the Relship of the Attribute of the AttrMap.

```
context self:Base_AttrMap inv c_AttrMap_Attribute_Link_Relship:
  self.attribute.relship=self.link.relship
```

Remark: The commutativity restrictions touching only ErSem classes follow.

Commutativity restriction: The ErStates of the Instance of an AttrMap being an InstanceAttrMap include the ErStates of the AttrMap.

```
context self:Base_AttrMap inv c_AttrMap_Instance_ErState:
  self.instance->size=1 implies
  self.instance.erState->includesAll(self.erState)
```

Commutativity restriction: The ErStates of the Link of an AttrMap being a LinkAttrMap include the ErStates of the AttrMap.

```
context self:Base_AttrMap inv c_AttrMap_Link_ErState:
  self.link->size=1 implies
  self.link.erState->includesAll(self.erState)
```

Commutativity restriction: The ErStates of an Instance of a RelendMap include the ErStates of the RelendMap.

```
context self:ErSem_RelendMap inv c_RelendMap_Instance_ErState:
  self.instance.erState->includesAll(self.erState)
```

Commutativity restriction: The ErStates of a Link of a RelendMap include the ErStates of the RelendMap.

```
context self:ErSem_RelendMap inv c_RelendMap_Link_ErState:
  self.link.erState->includesAll(self.erState)
```

Commutativity restriction: The ErStates of an Instance of a RelendMap include the ErStates of the Link of the Relendmap.

```
context self:ErSem_RelendMap inv c_RelendMap_Instance_Link_ErState:
  self.instance.erState->includesAll(self.link.erState)
```

Remark: This ends the list of commutativity restrictions. Uniqueness restrictions for keys follow.

Uniqueness restriction for keys: Two different Instances of one Entity can be distinguished in every ErState where both Instances occur by a key Attribute of the Entity.

```
context self:ErSem_Instance inv keyMapUnique:
  ErSem_Instance.allInstances->forAll(self2 |
    self<>self2 and self.entity=self2.entity
    implies
    self.erState->intersection(self2.erState)->forAll(s |
      self.entity.key()->exists(ka |
        self.applyAttr(s,ka)<>self2.applyAttr(s,ka))))
```

Uniqueness restriction for Relend maps: Two different Links of one Relship can be distinguished in every ErState where both Links occur by a Relend of the Relship.

```
context self:ErSem_Link inv relendMapUnique:
  ErSem_Link.allInstances->forAll(self2 |
    self<>self2 and self.relship=self2.relship
    implies
    self.erState->intersection(self2.erState)->forAll(s |
      self.relship.relend->exists(re |
        self.applyRelend(s,re)<>self2.applyRelend(s,re))))
```

4.4 RE Syntax

Name restriction: Different RelDBSchemas have different names.

```
context self:RelSyn_RelDBSchema inv uniqueRelDBSchemaNames:
  RelSyn_RelDBSchema.allInstances->forAll(self2 |
    self.name=self2.name implies self=self2)
```

Name restriction: Within one RelDBSchema, different RelSchemas have different names.

```
context self:RelSyn_RelDBSchema
  inv uniqueRelSchemaNamesWithinRelDBSchema:
  self.relSchema->forAll(r1,r2 | r1.name=r2.name implies r1=r2)
```

Name restriction: Within one RelSchema, different Attributes have different names.

```
context self:RelSyn_RelSchema inv uniqueAttributeNamesWithinRelSchema:
  self.attribute->forAll(a1,a2 | a1.name=a2.name implies a1=a2)
```

Key restriction: The set of key Attributes of a RelSchema is not empty.

```
context self:RelSyn_RelSchema inv relSchemaKeyNotEmpty:
  self.key()->notEmpty
```

4.5 RE Semantics

Functional Restriction: A TupleAttrMap, i.e., an AttrMap being linked to a Tuple, represents a non-redundant, functional assignment of a Value to an Attribute of a RelSchema within an ErState for the given Tuple.

```
context self:Base_AttrMap inv tupleAttrMapIsFunction:
  Base_AttrMap.allInstances->forAll(self2 |
    (self<>self2 and self.tuple->size=1 and self2.tuple->size=1)
    implies
    ((self.attribute=self2.attribute and self.tuple=self2.tuple)
     implies
     (self.relDBState<>self2.relDBState and self.value<>self2.value)))
```

Remark: A whole bunch of commutativity restrictions follow. First, we show commutativity restrictions touching RelSem and RelSyn classes. Second, we show commutativity restrictions touching only RelSem classes.

Commutativity restriction: The Attributes of the RelSchema of a Tuple are identical to the Attributes of the AttrMaps of the Tuple; in other words, there are Attribute assignments for all Attributes of a Tuple.

```
context self:RelSem_Tuple inv c_Tuple_RelSchema_AttrMap_Attribute:
  self.relSchema.attribute=self.attrMap.attribute->asSet
```

Commutativity restriction: The RelDBSchema of the RelDBState of a Tuple is identical to the RelDBSchema of the RelSchema of the Tuple.

```
context self:RelSem_Tuple inv c_Tuple_RelSchema_RelDBState_RelDBSchema:
  Set{self.relSchema.relDBSchema}=self.relDBState.relDBSchema->asSet
```

Commutativity restriction: The RelSchema of the Tuple of an AttrMap being a TupleAttrMap is identical to the RelSchema of the Attribute of the Tuple.

```
context self:Base_AttrMap inv c_AttrMap_Attribute_Tuple_RelSchema:
  self.attribute.relSchema=self.tuple.relSchema
```

Remark: One commutativity restriction touching only RelSem classes follows.

Commutativity restriction: The RelDBStates of the Tuple of an AttrMap being a TupleAttrMap include the RelDBStates of the AttrMap.

```
context self:Base_AttrMap inv c_AttrMap_Tuple_RelDBState:
  self.tuple->size=1 implies
  self.tuple.relDBState->includesAll(self.relDBState)
```

Key restriction: Two different Tuples of one RelSchema can be distinguished in every RelDBState where both Tuples occur by a key Attribute of the RelSchema.

```
context self:RelSem_Tuple inv keyMapUnique:
  RelSem_Tuple.allInstances->forAll(self2 |
    self<>self2 and self.relSchema=self2.relSchema
    implies
    self.relDBState->intersection(self2.relDBState)->forAll(s |
      self.relSchema.key()->exists(ka |
        self.applyAttr(s,ka)<>self2.applyAttr(s,ka))))
```

4.6 Common Datamodel Concerns

Figure 4.2 visualizes the three invariants of the package DataMods. The formal constraints are given below. The figure indicates that these constraints are xor constraints between associations. In contrast to the UML convention that xor constraints are shown as straight lines, the layout in the class diagram requires to draw them by curved lines.

Ownership restriction: An Attribute is either an Entity Attribute or a Relship Attribute or a RelSchema Attribute.

```
context self:Base_Attribute inv linkedToOneOfEntityRelshipRelSchema:
  (self.entity->size)+(self.relship->size)+(self.relSchema->size)=1
```

Ownership restriction: An AttrMap belongs to either an Instance or a Link or a Tuple, i.e., is an InstanceAttrMap or a LinkAttrMap or a TupleAttrMap.

```
context self:Base_AttrMap inv linkedToOneOfInstanceLinkTuple:
  (self.instance->size)+(self.link->size)+(self.tuple->size)=1
```

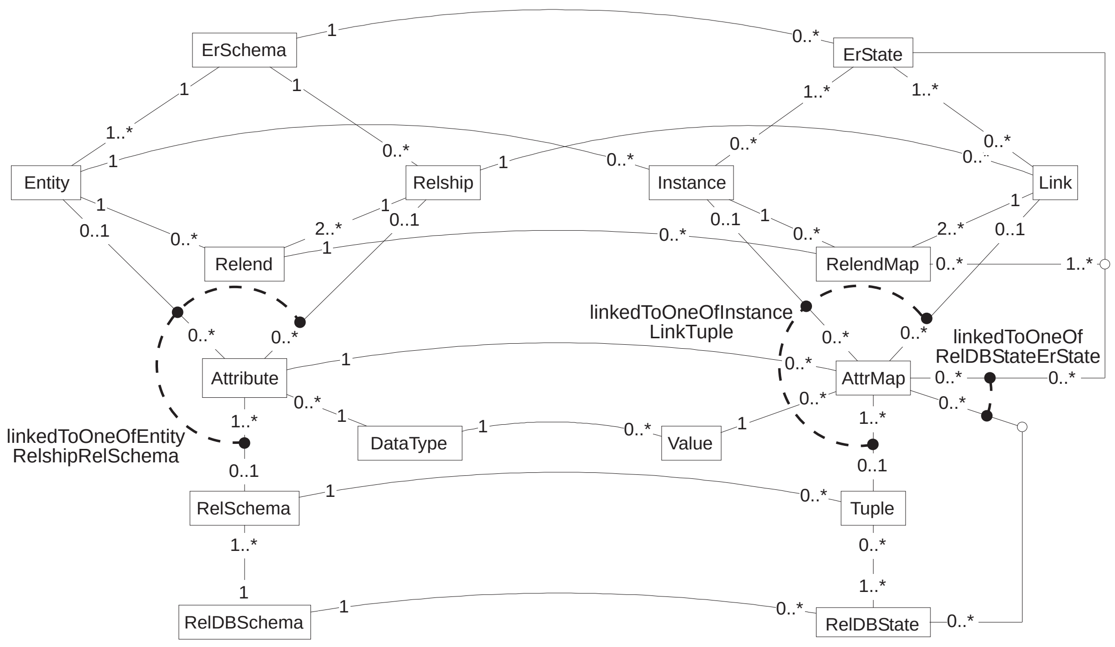


Figure 4.2: Overview on DataMods Constraints

Ownership restriction: An AttrMap lives either in a RelDBState or an ErState.

```
context self:Base_AttrMap inv linkedToOneOfRelDBStateErState:
  self.relDBState->size>0 xor self.erState->size>0
```

4.7 Transformation

Figure 4.3 gives an overview on the invariants in the package Trans. This overview classifies the constraints according to datamodel and whether syntax or semantics is touched. The classification also involves a direction. For example, the two constraints forEntityExistsOneRelSchema and forRelshipExistsOneRelSchema are visualized with an arrow from ErSyn to RelSyn because the constraints assures that an ER schema element requires a Relational database schema element to be present.

Remark: The first collection of constraints concerns the syntax part of the transformation, i.e., these constraints touch ErSyn and RelSyn classes only.

Exists constraint: For every Entity in the ErSchema there is a RelSchema having the same name and Attributes with the same properties, i.e., name, DataType, and key property.

```
context self:Er2Rel_Trans inv forEntityExistsOneRelSchema:
  self.erSchema.entity->forAll(e |
    self.relDBSchema.relSchema->one(rl |
      e.name=rl.name and
```

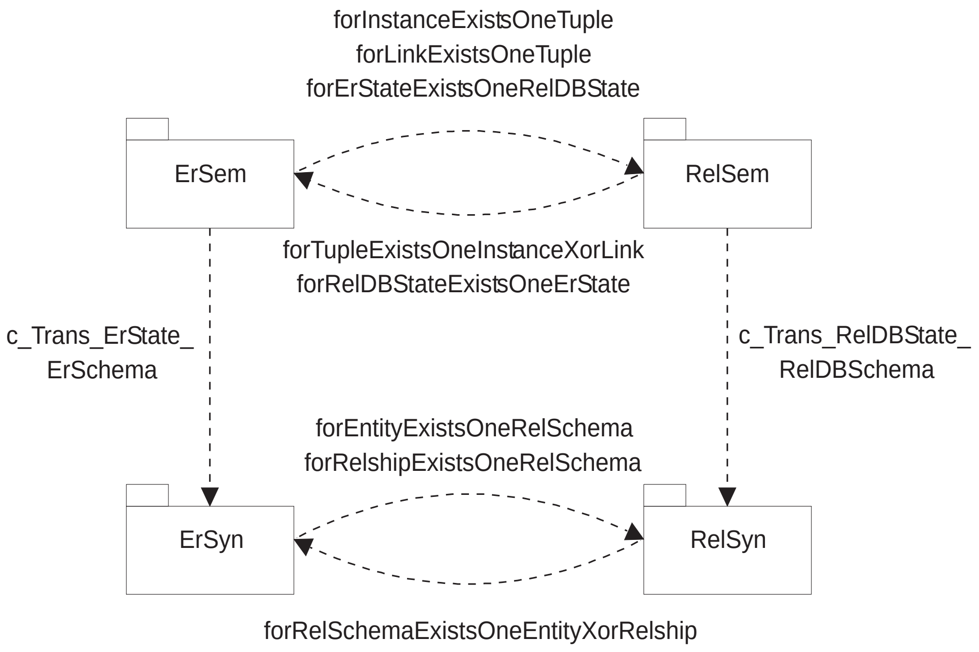

forInstanceExistsOneTuple
forLinkExistsOneTuple
forErStateExistsOneRelDBState
ErSem
RelSem
forTupleExistsOneInstanceXorLink
forRelDBStateExistsOneErState
c_Trans_ErState_ ErSchema
c_Trans_RelDBState_ RelDBSchema
forEntityExistsOneRelSchema
forRelshipExistsOneRelSchema
ErSyn
RelSyn
forRelSchemaExistsOneEntityXorRelship

Figure 4.3: Overview on Transformation Constraints

```
      e.attribute->forAll(ea |
        rl.attribute->one(ra |
          ea.name=ra.name and ea.dataType=ra.dataType and
          ea.isKey=ra.isKey))))
```

Exists constraint: For every Relship in the ErSchema there is a RelSchema having the same name, Relends representing the arms of the relationship, and Attributes with the same properties, i.e., name, DataType, and key property.

```
context self:Er2Rel_Trans inv forRelshipExistsOneRelSchema:
  self.erSchema.relship->forAll(rs |
    self.relDBSchema.relSchema->one(rl |
      rs.name=rl.name and
      rs.relend->forAll(re | re.entity.key()->forAll(rek |
        rl.attribute->one(ra |
          re.name.concat('_').concat(rek.name)=ra.name and
          rek.dataType=ra.dataType and ra.isKey))) and
      rs.attribute->forAll(rsa |
        rl.attribute->one(ra |
          rsa.name=ra.name and rsa.dataType=ra.dataType and
          ra.isKey=false))))
```

Exists constraint: For every RelSchema there is either an Entity or a Relship with the same properties and name; if the RelSchema corresponds to an Entity, both have Attributes with the same names, DataTypes, and key properties; if the RelSchema corresponds to a Relship, the RelSchema has Attributes corresponding to the arms of the Relship and both have Attributes with the same properties.

```
context self:Er2Rel_Trans inv forRelSchemaExistsOneEntityXorRelship:
  self.relDBSchema.relSchema->forAll(rl |
    self.erSchema.entity->one(e |
      rl.name=e.name and
      rl.attribute->forAll(ra |
        e.attribute->one(ea |
          ra.name=ea.name and ea.dataType=ra.dataType and
          ra.isKey=ea.isKey)))
    xor
    self.erSchema.relship->one(rs |
      rl.name=rs.name and
      rl.attribute->forAll(ra |
        rs.relend->one(re |
          re.entity.key()->one(rek |
            ra.name=re.name.concat('_').concat(rek.name) and
            ra.dataType=rek.dataType and ra.isKey))
        xor
        rs.attribute->one(rsa |
          ra.name=rsa.name and ra.dataType=rsa.dataType and
          ra.isKey=false))))
```

In Fig. 4.4 we have pictured classified the attributes of a RE database schema as alpha, beta, and gamma attributes: alpha attributes are attributes of RE schemas representing entities, beta attributes are attributes in RE schemas for ordinary relationship attributes, and gamma attributes are attributes in RE schemas representing relationship arms. These classifications will appear in the constraints to follow.

Remark: The second collection of constraints concerns the semantics part of the transformation, i.e., these constraints touch also ErSem and RelSem classes.

Exists constraint: For every Instance in a ErState there is exactly one Tuple in one RelDBState such that for every AttrMap of the Instance there is exactly one AttrMap of the Tuple having the same attribute name and Value.

Condensed, informal version:

```
forAll(erSt | one(relSt |
  forAll(i | one(t |
    forAll(amEr | one(amRel | equiv(amEr,amRel)))))) -- alpha
```

Formal constraint:

```
context self:Er2Rel_Trans inv forInstanceExistsOneTuple:
```

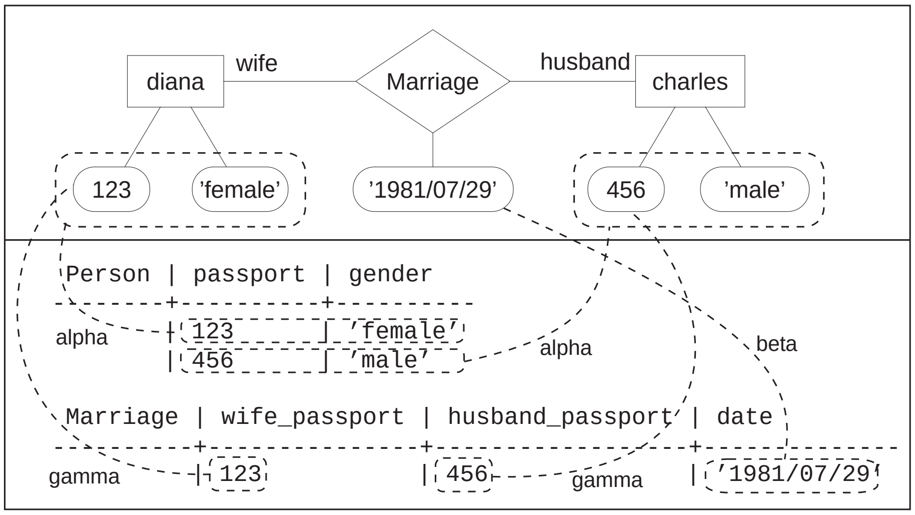


Figure 4.4: alpha, beta, and gamma Attributes

```
  self.erState->forAll(erSt | self.relDBState->one(relSt |
    erSt.instance->forAll(i | relSt.tuple->one(t |
      i.attrMap->forAll(amEr | -- alpha
        t.attrMap->one(amRel |
          amEr.attribute.name=amRel.attribute.name and
          amEr.value=amRel.value))))))
```

Exists constraint: For every Link in a ErState there is exactly one Tuple in one RelDBState such that (A) for every AttrMap of the Link there is exactly one AttrMap of the Tuple having the same attribute name and Value and (B) for every RelendMap of the Link and every AttrMap of a key Attribute of the Instance referred to in the Link there is exactly one AttrMap of the Tuple having a corresponding attribute name and Value.

Condensed, informal version:

```
forAll(erSt | one(relSt |
  forAll(l | one(t |
    forAll(amEr | one(amRel | equiv(amEr,amRel))) -- beta
    and
    forAll(rm | forAll(amEr | -- gamma
      one(amRel | equiv(amEr,amRel)))))))
```

Formal constraint:

```
context self:Er2Rel_Trans inv forLinkExistsOneTuple:
  self.erState->forAll(erSt | self.relDBState->one(relSt |
    erSt.link->forAll(l | relSt.tuple->one(t |
      l.attrMap->forAll(amEr | -- beta
        t.attrMap->one(amRel |
          amEr.attribute.name=amRel.attribute.name and
          amEr.value=amRel.value))
      and
      l.relendMap->forAll(rm | -- gamma
        rm.instance.attrMap->
        select(amEr | amEr.attribute.isKey)->forAll(amEr |
          t.attrMap->select(amRel | amRel.attribute.isKey)->one(amRel |
            amRel.attribute.name =
              rm.relend.name.concat('_').concat(amEr.attribute.name) and
            amRel.value=amEr.value)))))))
```

Exists constraint: For every Tuple in a RelDBState (1) there is either exactly one Instance such that for every attrMap of the Tuple there is exactly one attrMap in the Instance holding the same information or (2) there is exactly one link such that for every attrMap of Tuple the following holds: (A) if the attrMap belongs not to a key Attribute, there is exactly one attrMap in the Link holding the same information, and (B) if the attrMap belongs to a key Attribute, there is exactly one RelendMap in the Link and exactly one attrMap of the RelendMap such that the attrMap from the Tuple and the attrMap from the Link hold the same information.

Condensed, informal version:

```
forAll(relSt | one(erSt |
  forAll(t |
    one(i | -- alpha
      forAll(amRel | one(amEr | equiv(amRel,amEr))))
    xor
    one(l | forAll(amRel |
      ( amRel.notKey implies -- beta
        one(amEr | equiv(amRel,amEr) ) )
      and
      ( amRel.isKey implies -- gamma
        one(rm | one(amEr | equiv(amRel,amEr) ) ) ))))))
```

Formal constraint:

```
context self:Er2Rel_Trans inv forTupleExistsOneInstanceXorLink:
  self.relDBState->forAll(relSt | self.erState->one(erSt |
    relSt.tuple->forAll(t |
      erSt.instance->one(i | -- alpha
        t.attrMap->forAll(amRel |
          i.attrMap->one(amEr |
            amEr.attribute.name=amRel.attribute.name and
            amEr.value=amRel.value)))
      xor
      erSt.link->one(l |
        t.attrMap->forAll(amRel |
          ( amRel.attribute.isKey=false implies -- beta
            l.attrMap->one(amEr |
              amEr.attribute.name=amRel.attribute.name and
              amEr.value=amRel.value) )
          and
          ( amRel.attribute.isKey=true implies -- gamma
            l.relendMap->one(rm |
              rm.instance.attrMap->select(amEr | amEr.attribute.isKey)->
              one(amEr |
                amRel.attribute.name =
                  rm.relend.name.concat('_').concat(amEr.attribute.name)
                and amRel.value=amEr.value)))))))
```

Exists constraint: In a transformation, there is exactly one RelDBState for every ERState.

```
context self:Er2Rel_Trans inv forErStateExistsOneRelDBState:
  self.erState->forAll(erSt |
    self.relDBState->one(relSt | erSt.relDBState=relSt))
```

Exists constraint: In a transformation, there is exactly one ErState for every RelDBState.

```
context self:Er2Rel_Trans inv forRelDBStateExistsOneErState:
  self.relDBState->forAll(relSt |
    self.erState->one(erSt | relSt.erState=erSt))
```

Commutativity constraint: The ErSchemas of the translated ErStates of a transformation are identical to the ErSchema of the transformation.

```
context self:Er2Rel_Trans inv c_Trans_ErState_ErSchema:
  self.erState->notEmpty implies
  self.erState.erSchema->asSet=Set{self.erSchema}
```

Commutativity constraint: The RelDBSchemas of the translated RelDBStates of a transformation are identical to the RelDBSchema of the transformation.

```
context self:Er2Rel_Trans inv c_Trans_RelDBState_RelDBSchema:
  self.relDBState->notEmpty implies
  self.relDBState.relDBSchema->asSet=Set{self.relDBSchema}
```

4.8 Operation Definitions

The operation ErSyn_Entity::key() determines the set of key attributes of an entity.

```
ErSyn_Entity::key():Set(Base_Attribute) = -- key attributes
  self.attribute->select(a | a.isKey)
```

The operation ErSyn_Entity::osRelend() yields for an entity the set of relationship ends lying on the opposite side of an association in which the entity participates, i.e., the set of all relationship ends which may be applied to an instance of the entity.

```
ErSyn_Entity::osRelend():Set(ErSyn_Relend) = -- other side relends
  self.relend->collect(re | re.relship.relend->excluding(re))->
    flatten->asSet
```

The operation ErSem_ErState::findInstance(...):ErSem_Instance detects in an ER state the instance which is determined by the key values and the entity name given as parameters.

```
ErSem_ErState::
findInstance(keyAttrs:Sequence(String),entName:String):ErSem_Instance=
  let theEntity:ErSyn_Entity=
    self.erSchema.entity->select(name=entName)->any(true) in
  self.instance->select(entity=theEntity)->
  select(i:ErSem_Instance|Sequence{1..keyAttrs->size div(2)}->
    forAll(j:Integer|
      let attrName:String=keyAttrs->at(2*(j-1)+1) in
      let attrValue:String=keyAttrs->at(2*(j-1)+2) in
      let attr:Base_Attribute=theEntity.attribute->
            select(name=attrName)->any(true) in
      i.applyAttr(self,attr)=attrValue))->any(true)
```

Remark: The use of any is in this case and the following occurrences a deterministic use. The constructions are always done in such a way that any is applied to a collection with exactly one element. The given constraints will guarantee this. Thus the use of any here can be seen as a type cast from a singleton set to the type of the element yielding undefined if the set is empty or has more than one element.

The operation ErSem_Instance::applyAttr(...):String evaluates in a given state an attribute of an instance.

```
ErSem_Instance::
applyAttr(aState:ErSem_ErState,anAttr:Base_Attribute):String=
  self.attrMap->
    select(am|am.erState->includes(aState) and am.attribute=anAttr)->
      any(true).value.content
```

The operation ErSem_Link::applyRelend(...):ErSem_Instance evaluates in a given state a relationship end of a link.

```
ErSem_Link::
applyRelend(aState:ErSem_ErState,aRelend:ErSyn_Relend):ErSem_Instance=
  self.relendMap->
    select(rm | rm.erState->includes(aState) and rm.relend=aRelend)->
      any(true).instance
```

The operation ErSem_Link::applyAttr(...):String evaluates in a given state an attribute of an link.

```
ErSem_Link::
applyAttr(aState:ErSem_ErState,anAttr:Base_Attribute):String=
  self.attrMap->
    select(am|am.erState->includes(aState) and am.attribute=anAttr)->
      any(true).value.content
```

The operation RelSyn_RelSchema::key():Set(Base_Attribute) determines the set of key attributes of an RE schema.

```
RelSyn_RelSchema::key():Set(Base_Attribute) =
  self.attribute->select(a | a.isKey)
```

The operation RelSem_Tuple::applyAttr(...):String evaluates in a given state an attribute of a tuple.

```
RelSem_Tuple::
applyAttr(aState:RelSem_RelDBState,anAttr:Base_Attribute):String=
  self.attrMap->
    select(am | am.relDBState->includes(aState) and
                am.attribute=anAttr)->any(true).value.content
```

4.9 Classification of Constraints

The table in Fig. 4.5 gives an overview on the constraints we have discussed so far in the order we have mentioned them. The table has one line for each constraint. In the left of the table we see six classification categories labelled N, C, K, F, L, and E. Afterwards the defining package of the constraint, the context class of the constraint, and finally the constraint name are shown. A filled circle in a line classifies the constraint. The six categories have the following meaning according to their first letters.

N	C	K	F	L	E			
●						Base	Base_Named	nameOk
–	–	–	–	–	–	4	Base_Value	differentContentOrDataType
	●						Base_AttrMap	c_AttrMap_Attribute_Value_DataType
●							Base_DataType	uniqueDataTypeNames
	●					ErSyn	ErSyn_Relend	c_Relend_Entity_Relship_ErSchema
●						13	ErSyn_ErSchema	uniqueErSchemaNames
●							ErSyn_ErSchema	uniqueEntityNamesWithinErSchema
●							ErSyn_ErSchema	uniqueRelshipNamesWithinErSchema
●							ErSyn_ErSchema	differentEntityAndRelshipNamesWithinErSchema
●							ErSyn_Relship	uniqueRelendNamesWithinRelship
●							ErSyn_Entity	uniqueAttributeNamesWithinEntity
●							ErSyn_Relship	uniqueAttributeNamesWithinRelship
●							ErSyn_Entity	differentOsRelendAndAttributeNamesWithinEntity
●							ErSyn_Relship	differentRelendAndAttributeNamesWithinRelship
●							ErSyn_Entity	uniqueOsRelendNamesWithinEntity
		●					ErSyn_Entity	entityKeyNotEmpty
		●					ErSyn_Relship	relshipKeyEmpty
			●			ErSem	Base_AttrMap	instanceAttrMapIsFunction
			●			19	Base_AttrMap	linkAttrMapIsFunction
			●				ErSem_RelendMap	relendMapIsFunction
	●						ErSem_Instance	c_Instance_Entity_AttrMap_Attribute
	●						ErSem_Link	c_Link_Relship_AttrMap_Attribute
	●						ErSem_Link	c_Link_Relship_RelendMap_Relend
	●						ErSem_RelendMap	c_RelendMap_Relend_Instance_Entity
	●						ErSem_RelendMap	c_RelendMap_Relend_Link_Relship
	●						ErSem_Instance	c_Instance_Entity_ErState_ErSchema
	●						ErSem_Link	c_Link_Relship_ErState_ErSchema
	●						Base_AttrMap	c_AttrMap_Attribute_Instance_Entity
	●						Base_AttrMap	c_AttrMap_Attribute_Link_Relship
	●						Base_AttrMap	c_AttrMap_Instance_ErState
	●						Base_AttrMap	c_AttrMap_Link_ErState
	●						ErSem_RelendMap	c_RelendMap_Instance_ErState
	●						ErSem_RelendMap	c_RelendMap_Link_ErState
	●						ErSem_RelendMap	c_RelendMap_Instance_Link_ErState
		●					ErSem_Instance	keyMapUnique
		●					ErSem_Link	relendMapUnique
●						RelSyn	RelSyn_RelDBSchema	uniqueRelDBSchemaNames
●						4	RelSyn_RelDBSchema	uniqueRelSchemaNamesWithinRelDBSchema
●							RelSyn_RelSchema	uniqueAttributeNamesWithinRelSchema
		●					RelSyn_RelSchema	relSchemaKeyNotEmpty
			●			RelSem	Base_AttrMap	tupleAttrMapIsFunction
	●					6	RelSem_Tuple	c_Tuple_RelSchema_AttrMap_Attribute
	●						RelSem_Tuple	c_Tuple_RelSchema_RelDBState_RelDBSchema
	●						Base_AttrMap	c_AttrMap_Attribute_Tuple_RelSchema
	●						Base_AttrMap	c_AttrMap_Tuple_RelDBState
		●					RelSem_Tuple	keyMapUnique
				●		DataMods	Base_Attribute	linkedToOneOfEntityRelshipRelSchema
				●		3	Base_AttrMap	linkedToOneOfInstanceLinkTuple
				●			Base_AttrMap	linkedToOneOfRelDBStateErState
					●	Er2Rel	Er2Rel_Trans	forEntityExistsOneRelSchema
					●	10	Er2Rel_Trans	forRelshipExistsOneRelSchema
					●		Er2Rel_Trans	forRelSchemaExistsOneEntityXorRelship
					●		Er2Rel_Trans	forInstanceExistsOneTuple
					●		Er2Rel_Trans	forLinkExistsOneTuple
					●		Er2Rel_Trans	forTupleExistsOneInstanceXorLink
					●		Er2Rel_Trans	forErStateExistsOneRelDBState
					●		Er2Rel_Trans	forRelDBStateExistsOneErState
	●						Er2Rel_Trans	c_Trans_ErState_ErSchema
	●						Er2Rel_Trans	c_Trans_RelDBState_RelDBSchema
15	22	6	4	3	8	59		

Figure 4.5: Overview on Classification of Constraints

Naming: A naming constraints is used in a syntax package in order to restrict the attribute name. Both global and local naming restrictions are covered.

Commutativity: A commutativity constraint expresses that two paths in the class diagram commute, i.e., that two different expressions having the same source and target class evaluate to the same result. A commutativity constraint is either a 2-2 or a 2-1 constraint. In a 2-2 constraint both of the two expression navigate over two associations, in a 2-1 constraint one expressions navigates over two associations, the other expression navigates over one association.

Key: A key constraint involves key attributes. A key constraint either touches syntactical issues (the set of key attributes is empty or not empty) or semantical issues (the value of the key attribute must be unique).

Function: A function constraint expresses that an element which has to be modeled with MOF as a class together with multiple associations must be interpreted as an n-ary association which represents a function. Recall that there are no n-ary associations in MOF.

Link: A link constraint assures that each attribute or attribute map is linked to exactly one owning element, either a schema element in the syntax or a state element in the semantics.

Exists: An exists constraint requires in the transformation package that for each element of a certain domain a unique element of another domain exists.

Each constraint is categorized at most once. There is one with respect to the given six items non-classifiable constraint: The second constraint. The table in Fig. 4.5 shows also the number of constraints found in each package and in each category. About two thirds of the constraints (15+27=37 of totally 59 constraints) handle standard issues, i.e., naming and commutativity issues.

Although the description of the constraints showed many complicated details, the table in Fig. 4.5 reveals that many constraints have a similar purpose and with this a similar structure.

Chapter 5

Modeling Method

5.1 Basic Modeling Method

As indicated with the package diagram in Fig. 5.1, the method behind our approach is to divide a language into a syntax and semantics part and to describe a transformation between two languages as a direction-neutral affair. All packages will in general

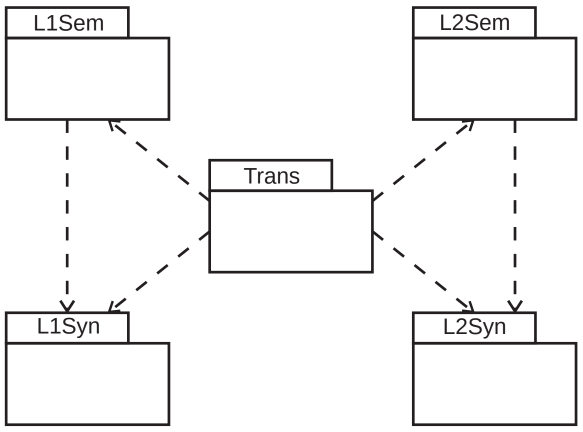


Figure 5.1: Package Structure of Underlying Modeling Method

include classes, associations, and constraints. We expect that the semantics of the languages will depend on their syntax. The transformation will rely on the syntax and the semantics of the two languages. Transformation properties could be further distinguished into syntax and semantics properties. We regard it as important that formal properties of

- the syntax of languages to be transformed,
- the semantics of the languages to be transformed and,
- the transformation itself are described in a uniform way within one language.

Only with an explicit formulation of the semantics, a transformation can make requirements about the relationship between the semantics of the first language, the semantics of the second language and the transformation. Apart from a better understanding of the underlying domain, a uniform description opens the possibility of a coherent reasoning mechanism for languages and their transformation.

Above, we have coined the notion that a transformation is a direction-neutral affair. We expect that transformations are described with the same modeling features as the languages, i.e., with classes, association, and constraints. Our transformation between ER and RE was insofar a special case as we needed only a single class in the transformation package. In general, a transformation package may involve more classes and associations on its own. We think that in order to be general, it is a good idea to start with transformation classes and to describe properties of transformations as invariants. The advantage we see in this is that no direction is imposed for the transformation. In a later reification step, the transformations may be turned into appropriate operations.

Our above transformation between ER and RE was also in another respect a special case of the general situation: Our aim for ER and RE was an equivalence between the two database state spaces, i.e., between the two semantic domains, and our constraints exactly required such an equivalence. However, with respect to the transformation properties, the approach is flexible. If we would like to do so, we could, for example, require that the semantics of the second language can be embedded in the semantics of the first language but not necessarily the other way round. The transformation properties merely depend on what is stated in the transformation constraints.

5.2 Other Examples

The method which we propose can be applied not only to database languages but to transformations between common computer science languages. We will study some examples from the modeling area as well as classical compliler technology.

A very popular sub-language of the UML are statecharts. Statecharts are similar to classical automata, but allow to structure behavior descriptions with states and transitions where states may possess sub-states in order to compose complex systems into manageable pieces in a hierarchical fashion. Hierarchical statecharts may be translated into non-hierarchical, flat statechart without changing the accepted language. This is captured in Fig. 5.2. The package diagram introduces a package SC-Hier for the syntax of hierarchical statecharts and a package AccLanHier for describing the semantics, in this case for describing the accepted languages of the hierarchical statecharts. The second language is the language of flat statecharts, again described with a syntax description SC-Flat and a semantics description AccLanFlat. The transformation package Trans could then model the equivalence between hierarchical and flat statecharts.

Figure 5.3 shows classical compliler technology in the setting we propose. The syntax packages ProgLang and ASM would describe the syntax of the programming language

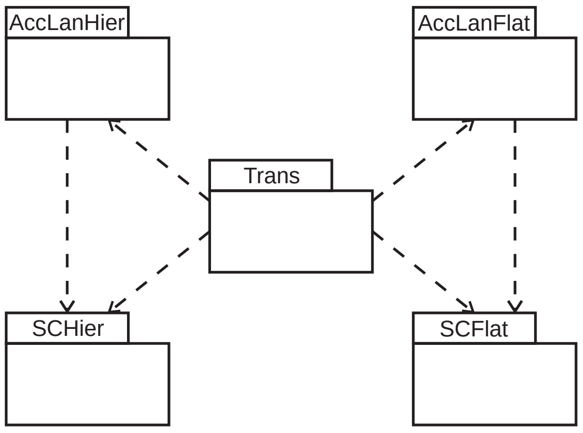


Figure 5.2: Transformation of Hierarchical Statecharts into Flat Statecharts

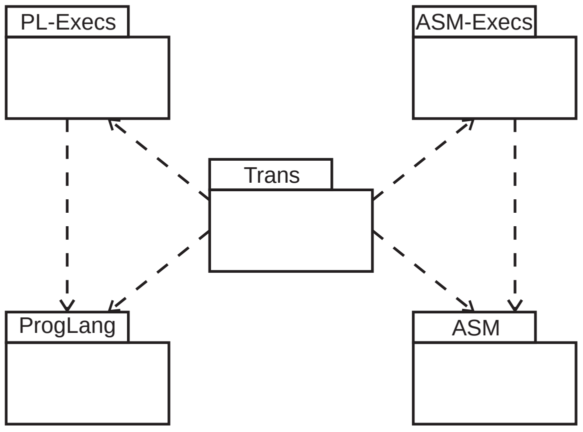


Figure 5.3: Compilation of a Programming Language into Assembler

and the assembler. The semantics packages PL-Execs and ASM-Execs model the execution traces for the programming language and the assembler. The package Trans would correspond to a classical compiler, where the syntax part would represent the pure translation and the semantics would state requirements for the compiler correctness.

5.3 Tool support

The last important ingredient for our method is tool support. It is necessary to validate the underlying class diagram including all constraints. We have done this with our OCL tool USE. Complex models like the one we have presented cannot be developed in a paper-and-pencil fashion. There are too many right or wrong design alternatives which can be taken and there are too many possible dead ends in the development.

You constantly need feedback about the modeling decisions which you have taken. Our tool (like others) gives support for this in that design decisions can be checked immediately through instantiation by building an object diagram and checking in particular whether the object diagram meets the constraints. The object diagram has not necessarily to be a complete one, it suffices to build an instantiation for the part you are currently working on. Building such object diagrams gives spontaneous feedback about the classes, associations, and constraints and helps in error detection and recovery, i.e., it helps in model debugging. Having tool support in this sense allows you to trace abstract design decisions down to concrete objects which you can handle and check on your system.

Chapter 6

Conclusion

In this paper have employed UML respectively MOF as a description language for two classical database approaches and its transformation. We have formally characterized the syntax of the datamodels, i.e., the database schemas, and we have covered the semantics of the datamodels, i.e., the database state spaces. We have also formally described the transformation and required as a correctness criterion the equivalence of the described database state spaces.

The method behind this concrete model transformation example is general. For a precise transformation of languages one has to know the syntax and semantics of both languages. Only if one knows these four ingredients, one can state criteria about the properties and the correctness of the transformation.

Topics for future research include:

- Formulation of this model transformation example with operations having pre- and postconditions.
- Formulation of the Relational datamodel as a specialization of the ER datamodel.
- Study of other database models, for example, historical database models like the Hierarchical or Network database model and newer approaches like the Object-Oriented datamodel, Semi-Structured, XML-like datamodels or Ontology-Based database models a la RDF and OWL.
- Incorporation of query features into the metamodel, e.g., incorporation of Relational algebra and Relational calculi.
- Instantiating the method with further examples, e.g., metamodeling a classical compiler for a small imperative language.
- Advanced tool support for metamodeling.

Bibliography

[AK02] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[AK03] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003.

[AKP03] D.H. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.

[AT91] P. Atzeni and R. Torlone. Management of Multiple Models: A Metamodel for Conceptual Models. In P.C. Kanellakis and J.W. Schmidt, editors, *DBPL*, pages 169–181. Morgan Kaufmann, 1991.

[CEK01] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In H. Hußmann, editor, *FASE*, volume 2029 of *LNCS*, pages 17–31. Springer, 2001.

[CESW04] T. Clark, A. Evans, P. Sammut, and J.S. Willans. Transformation Language Design: A Metamodelling Foundation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *ICGT*, volume 3256 of *LNCS*, pages 13–21. Springer, 2004.

[GL03] M. Gogolla and A. Lindow. Transforming Data Models with UML. In B. Omelayenko and M. Klein, editors, *Knowledge Transformation for the Semantic Web*, pages 18–33. IOS Press, Amsterdam, 2003.

[GLRZ02] M. Gogolla, A. Lindow, M. Richters, and P. Ziemann. Metamodel Transformation of Data Models. In J. Bezivin and R. France, editors, *Proc. UML'2002 Workshop in Software Model Engineering (WiSME 2002).* `http://www.metamodel.com/wisme-2002`, 2002.

[Gog94] M. Gogolla. *An Extended Entity-Relationship Model - Fundamentals and Pragmatics.* Springer, Berlin, LNCS 767, 1994.

[Gog95] M. Gogolla. Towards Schema Queries for Semantic Data Models. In N. Revell and A.M. Tjoa, editors, *Proc. 6th Int. Conf. and Workshop on Database and Expert Systems Applications (DEXA'95)*, pages 274–283. ONMIPRESS, San Mateo, 1995.

[Gog04] M. Gogolla. (An Example for) Metamodeling Syntax and Semantics of Two Languages, their Transformation, and a Correctness Criterion. In J. Bezivin and R. Heckel, editors, *Proc. Dagstuhl Seminar on Language Engineering for Model-Driven Software Development*. Schloss Dagstuhl Int. Conf. and Research Center for Computer Science, `http://www.dagstuhl.de/04101/`, 2004.

[Hal04] T.A. Halpin. Comparing Metamodels for ER, ORM and UML Data Models. In K. Siau, editor, *Advanced Topics in Database Research, Vol. 3*, pages 23–44. Idea Group, 2004.

[HR04] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"?. *IEEE Computer*, 37(10):64–72, 2004.

[JHPS03] A. Jodlowski, P. Habela, J. Plodzien, and K. Subieta. Extending OO Metamodels towards Dynamic Object Roles. In R. Meersman, Z. Tari, and D.C. Schmidt, editors, *CoopIS/DOA/ODBASE*, volume 2888 of *LNCS*, pages 1032–1047. Springer, 2003.

[JJ95] M.A. Jeusfeld and U.A. Johnen. An Executable Meta Model for Re-Engineering of Database Schemas. *Int. J. Cooperative Inf. Syst.*, 4(2-3):237–258, 1995.

[LGR01] A. Lindow, M. Gogolla, and M. Richters. Ein formal validiertes Metamodell für die Transformation von Schemata in Informationssystemen. In K. Bauknecht, W. Brauer, and T. Mück, editors, *Proc. GI Jahrestagung (GI'2001), Band 1, Workshop Integrating Diagrammatic and Formal Specification Techniques*, pages 662–669. Austrian Computer Society, Wien, 2001.

[MM97] V.B. Misic and S. Moser. A Formal Approach to Metamodeling: A Generic Object-Oriented Perspective. In D.W. Embley and R.C. Goldstein, editors, *ER*, volume 1331 of *LNCS*, pages 243–256. Springer, 1997.

[TS92] M.T. Tresch and M. H. Scholl. Meta Object Management and its Application to Database Evolution. In G. Pernul and A. M. Tjoa, editors, *11th International Conference on the Entity-Relationship Approach*, volume 645 of *LNCS*, pages 299–321, Karlsruhe, Germany, October 1992. Springer-Verlag.

[VP03] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (The Mathematics of Metamodeling is Metamodeling Mathematics). *Software and System Modeling*, 2(3):187–210, 2003.

[WE93] T. Welzer and J. Eder. Meta data model for database design. *LNCS*, 720:677–692, 1993.