David Embley, Bernhard Thalheim

Conceptual Modelling

A Handbook

June 16, 2010

Springer

Contents

Chapter 2
UML and OCL in Conceptual Modeling

Martin Gogolla

2.1 Introduction

The development of the Entity-Relationship (ER) model is probably one of the cornerstones for conceptual modeling of information systems. The Unified Modeling Language (UML) takes up central ideas from the ER model and puts them into a broad software development context by proposing various graphical sublanguages and diagrams for specialized software development tasks. It is said that the most commonly used UML diagram form is the class diagram. Entities and relationships have their counterparts there and are called classes and associations. Additionally, UML class diagrams allow the developer to include behavior in form of operations.

The first versions of UML were developed in the mid 90s of the last century. UML has changed since then and is still under development. Since many years UML includes a textual language, the Object Constraint Language (OCL), whose main task is to enrich the UML diagrams by textual constraints which cannot be expressed otherwise. However, apart from constraining, OCL can be used for querying UML models as well.

The rest of this chapter is structured as follows. The first section will introduce the correspondence between basic ER modeling concepts and their UML counterparts. The next section will explain how more advanced conceptual modeling concepts can be formulated in UML. The following section will use OCL for features not expressible in diagrammatic form. Then we turn to the description of Relational databases with UML. Before we conclude, we will show how to metamodel conceptual modeling features with UML itself.

Martin Gogolla
Department of Computer Science, Database Systems Group, University of Bremen, 28334 Bremen, Germany, e-mail: gogolla@informatik.uni-bremen.de

2.2 Basic Conceptual Modeling Features in UML

This section introduces the central features of UML [OMG10b, RBJ05] class and object diagrams and the Object Constraint Language (OCL) [OMG10a, WK03, RG98] which is part of the UML.

2.2.1 Class and Object Diagrams

The main purpose of class diagrams within the UML is to capture the basic static structures and operations of a system. In this subsection we will shortly explain the most important features in class diagrams like classes and associations. In later sections we discuss more advanced features.

Classes: A class is a descriptor for a set of objects sharing the same structure and behavior. In the database context, we concentrate on the structural aspect, although the behavioral aspect may be represented in UML as well. Object properties can be described by attributes classified by data types like `String` or `Boolean`. Later we see that properties can also stem from roles in associations which connect classes.

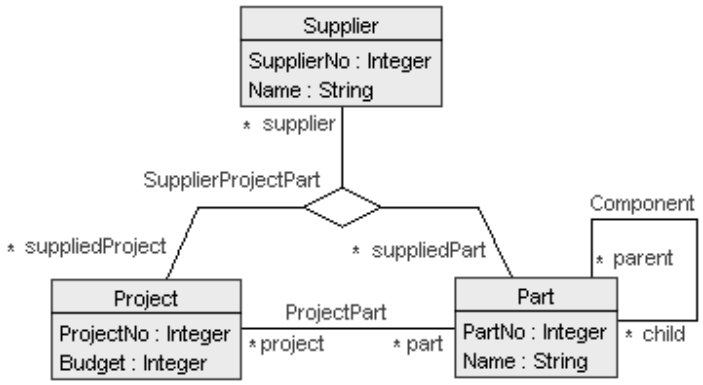

Fig. 2.1 Example UML Class Diagram 1

Example: Fig. 2.1 follows the example from Chen's original paper [Che76] on the ER model and shows the classes `Supplier`, `Project`, and `Part` together with some basic attributes including their data types, e.g., we identify `Supplier::Name:String` and `Project::Budget:Integer`. In this contribution, the general scheme for denoting properties (attributes and roles) is `Class::Property:PropertyType`. Most names for entities, relationships, and attributes are taken from Chen's original article. Our UML and OCL examples have been realized in the tool USE [GBR05, GBR07]. USE supports the de-

velopment of information systems with UML and OCL by testing, validation, and verification techniques.

Associations: An association represents a connection between a collection of classes and may be given a name. An association is manifested by a set of object connections, so-called links, sharing the same structure. A binary association can be defined between two different classes; objects of the respective classes play a particular role in the association; a binary association can also be defined on a single class; then objects of the class can play two different roles; such a binary association is called reflexive. A ternary association involves three roles. The notion n-ary association refers to a ternary or a higher-order association. Binary associations are shown with a simple line, and an n-ary association with a small rhomb-shaped polygon.

Example: In Fig. 2.1, we identify the binary association `ProjectPart` with roles `project` and `part`, the ternary association `SupplierProjectPart` with roles `supplier`, `suppliedProject`, and `suppliedPart`, and the reflexive association `Component` with roles `parent` and `child`.

Objects and Links: Structural aspects in UML can also be represented in an object diagram showing objects, links, and attribute values as manifestations of classes, associations, and attributes. An object diagram shows an instantiation of a class diagram and represents the described system in a particular state. Underlining for objects and links is used in object diagrams in order to distinguish them clearly from class diagrams.

Example: Figure 2.2 shows an object diagram for the class diagram from Fig. 2.1. Objects, links, and attribute values fit to the classes, associations, and attributes. The object identity is shown in the top part of the object rectangle to the left of the class to which the object belongs to. Formally, there is no connection between the object identity and attribute values. For the example classes `Supplier` and `Part`, we have chosen object identities which are close to but not identical with the attribute `Name`, but for the class `Project` the object identities have no connection to the attribute values. There are two `Project` objects, two `Supplier` objects and five `Part` objects. Each `Part` object represents a piece of software realizing controller (`Ctrl`) code which is responsible for a particular portion of a car. The `Component` links express part-whole relationships, for example, the `Engine Code (engineCtrl)` includes the `Battery Code (batteryCtrl)` and the `Motor Code (motorCtrl)`.

Roles: Proper roles must be specified on a class diagram in order to guarantee unique navigation, in particular in presence of reflexive associations or when two or more associations are present between two classes. Navigation in a class diagram means to fix two classes and to consider a path from the first class to the second class by using associations. The roles on the opposite side of a given class in an association determine also properties of the given class by navigating via the roles. Therefore, in UML and OCL the opposite side roles must be unique. Recall that properties can also come from attributes.

Example: On links, also the roles are captured. This is necessary in reflexive associations and in other situations, for example, if two associations are present between two given classes. For example in Fig. 2.2, if we consider the link be-

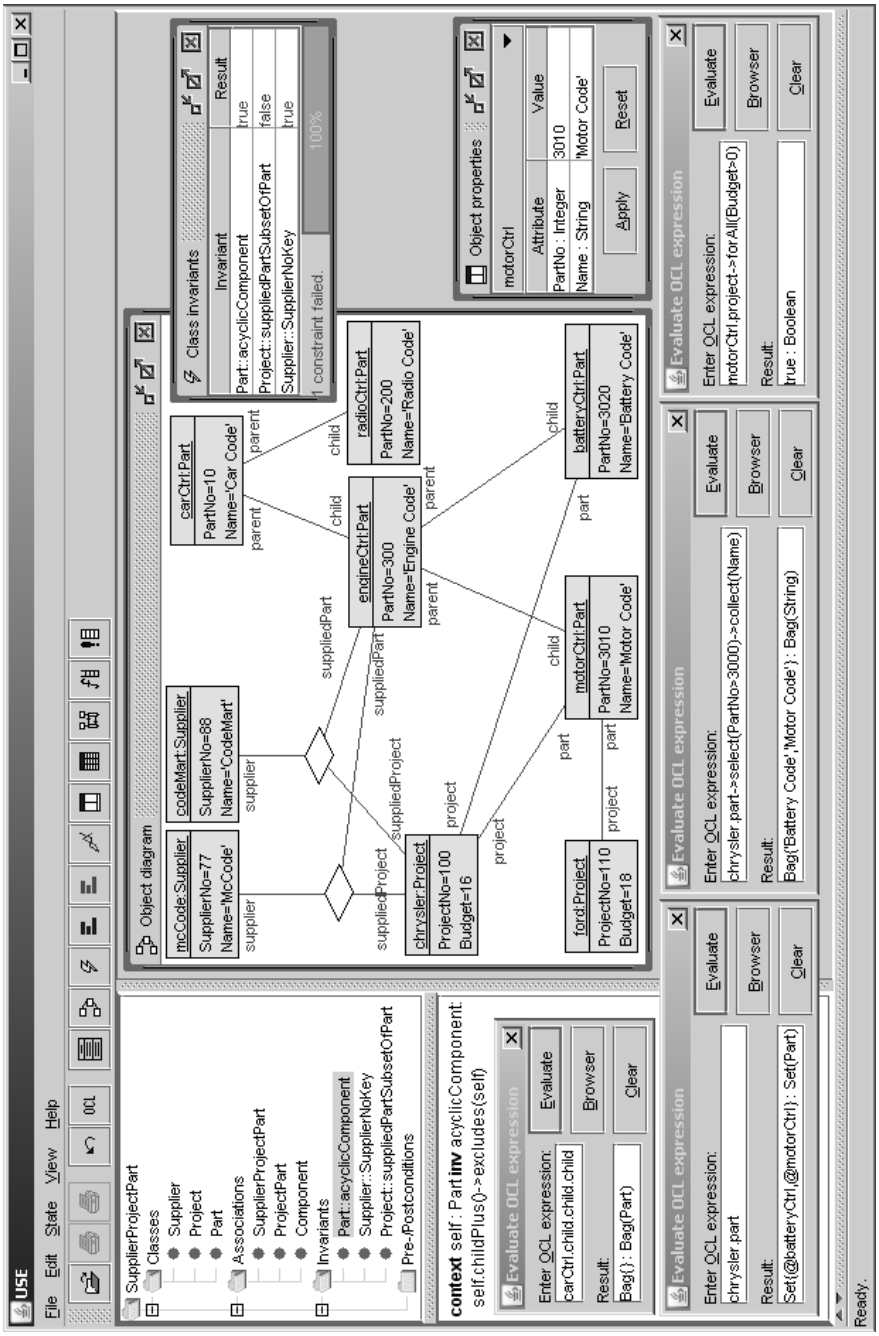

Fig. 2.2 Example Object Diagram 1 (and other USE functionality)

tween `carCtrl` and `engineCrtl`, without roles we could not tell which object plays the `parent` role and which one the `child` role. In the class diagram in Fig. 2.1, the class `Project` has two direct navigation possibilities to class `Part`: One via association `ProjectPart` and the other one via association `SupplierProjectPart`. One obtains therefore two properties in class `Project` returning `Part` objects: `Project::part:Set(Part)` from association `ProjectPart` and `Project::suppliedPart:Set(Part)` from association `SupplierProjectPart`. In the object diagram we obtain, for example, `ford.part` = `Set`{`motorCtrl`} as well as `ford.suppliedPart` = `Set`{}.

Class Diagram versus Database Schema: In the database context, it is interesting to remark that the connection between a class diagram and its object diagrams resembles the connection between a database schema and its associated database states: The class diagram induces a set of object diagrams and the database schema determines a set of database states; object diagrams and database states follow the general principles formulated in the class diagram and database schema, respectively. Because example object diagrams have to be displayed on screen or paper, they tend to show fewer information than proper, large database states. They may however explain the principles underlying a class diagram pretty well if the examples are well chosen.

2.2.2 Object Constraint Language

The UML includes a textual language that allows the developer to navigate in class diagrams and to formulate queries and restricting integrity constraints for the class diagram: The Object Constraint Language (OCL). Roughly speaking from a practical perspective, the OCL may be viewed as an object-oriented version of the Structured Query Language (SQL) originally developed for the Relational data model. Roughly speaking from a theoretical perspective, OCL may be viewed as a variant of first-order predicate logic with quantifiers on finite domains only. The central language features in OCL are: Navigation, logical connectives, collections and collection operations.

Navigation: The navigation features in OCL allow you to determine connected objects in the class diagram by using the dot operator '`.`'. Starting with an expression `expr` of start class `C`, one can apply a property `propC` of class `C` returning, for example, a collection of objects of class `D` by using the dot operator: `expr.propC`. The expression `expr` could be a variable or a single object, for example, or a more complicated expression. The navigation process can be repeated by writing `expr.propC.propD`, if `propD` is a property of class `D`.

Examples: Given the object diagram in Fig. 2.2, the following navigation expressions are syntactically valid in OCL and yield the stated return values and return types. OCL uses the convention that types are denoted with parentheses ( ) and values with braces { }.

```
chrysler.part =
  Set{batteryCtrl,motorCtrl}:Set(Part)          (1)
batteryCtrl.project.supplier =
  Bag{codeMart,mcCode}:Bag(Supplier)            (2)
carCtrl.child =
  Set{engineCtrl,radioCtrl}:Set(Part)           (3)
carCtrl.child.child =
  Bag{batteryCtrl,motorCtrl}:Bag(Part)          (4)
carCtrl.child.child.child =
  Bag{}:Bag(Part)                               (5)
```

Expressions (3) and (4) are similar insofar that expression (3) employs the dot in one place and expression (4) in two places. The difference in the result type, namely `Set(Part)` versus `Bag(Part)`, will be explained below.

Logical Connectives: OCL offers the usual logical connectives for conjunction `and`, disjunction `or`, and negation `not` as well as the implication `implies` and a binary exclusive or `xor`. An equality check `=`, an inequality check `<>`, and a conditional `if then else endif` is provided on all types.

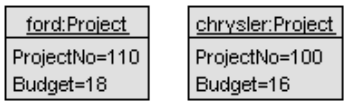

Fig. 2.3 Objects `ford` and `chrysler` from Example Object Diagram 1

Examples: If we consider the objects `ford` and `chrysler` from Fig. 2.2 being repeated in Fig. 2.3 for ease of tracing the resulting values, an OCL engine will deliver the following results.

```
ford.Budget>16 and chrysler.Budget>16 = false:Boolean
ford.Budget>16 or chrysler.Budget>16 = true:Boolean
not(ford.Budget>16) = false:Boolean
ford.Budget>16 implies chrysler.Budget>16
  = false:Boolean
ford.Budget>16 xor chrysler.Budget>16 = true:Boolean
ford=ford = true:Boolean
ford=chrysler = false:Boolean
if ford.Budget>16 then 42 else 43 endif = 42:Integer
if chrysler.Budget>16 then mcCode else codeMart endif
  = codeMart:Supplier
```

Collections: In the original OCL there were three kinds of collections: Sets, bags, and sequences. Later ordered sets were added, which we do not discuss here because they are similar to sequences; a discussion of OCL collections can be found in [BGH+10]. A possible collection element can appear at most once in a set, and

the insertion order in the set does not matter. An element can appear multiple times in a bag, and the order in the bag collection does not matter. An element can appear multiple times in a sequence in which the order is significant.

Examples: The expressions to follow state characteristic features of the OCL collections.

```
Set{11,22}        =Set{22,11}         = true
Bag{11,22}        =Bag{22,11}         = true
Sequence{11,22}   =Sequence{22,11}    = false
-----------------------------------------
Set{11,22}        =Set{11,22,11}      = true
Bag{11,22}        =Bag{11,22,11}      = false
Sequence{11,22}   =Sequence{11,22,11} = false
-----------------------------------------
Set{11,11,22}     =Set{11,22,11}      = true
Bag{11,11,22}     =Bag{11,22,11}      = true
Sequence{11,11,22}=Sequence{11,22,11} = false
```

We use terms of type `Set(Integer)` to demonstrate these features. However, we could have used terms of type `Set(Project)` as well, e.g., `Set{ford, chrysler}` instead of `Set{11, 22}`. Sets are insensitive to insertion order and insertion frequency. Bags are insensitive to insertion order, but are sensitive to insertion frequency. Sequences are sensitive to insertion order and insertion frequency.

Conversions: OCL collections can be nested and converted into each other. Bags and sequences can be converted to sets with `->asSet()`, sets and sequences to bags with `->asBag()`, and sets and bags to sequences with `->asSequence()`. The conversion to sequences assumes an order on the elements. The arrow notation will be explained in more detail below.

Examples: The following evaluations give an impressions of how the conversion work.

```
Sequence{11,22,11}->asBag() =
  Bag{1,11,22}:Bag(Integer)
Sequence{11,22,11}->asSet() =
  Set{11,22}:Set(Integer)
Bag{11,22,11}->asSet() =
  Set{11,22}:Set(Integer)
```

Special Type `OclAny`: Collection terms in OCL possess a type like in the following examples.

```
Sequence{ford,chrysler,ford}: Sequence(Project)
Set{42,41,40}: Set(Integer)
```

However, the special type `OclAny` is a supertype of all other types, and `OclAny` can be used for collections. Therefore, the following expressions are valid in OCL.

```
Set{'Talking Heads', 3.14, 42, false}: Set(OclAny)
Bag{Set{8, 9}, Set{ford, carCtrl}}: Bag(Set(OclAny))
```

Collection Operations: There is a large number of operations on collections in OCL. A lot of convenience and expressibility is based upon them. The most important operations on all collection kinds are `forAll`, `exists`, `select`, `collectNested`, `collect`, `size`, `isEmpty`, `includes`, and `including`. The table in Fig. 2.4 gives an overview on the functionality of the operations.

Operation	Functionality
`forAll`	realizes the universal quantification.
`exists`	formulates existential quantification.
`select`	filters elements with a predicate.
`collectNested`	applies a term to each collection element.
`collect`	applies a term to each collection element flattening the result.
`size`	determines the number of collection elements.
`isEmpty`	tests on emptiness.
`includes`	checks whether a possible element is included in the collection.
`including`	returns a collection which includes an element.

Fig. 2.4 Important collection operations

There are also special operations available only on particular collections, e.g., the operation `at` on sequences for retrieving an element by its position. All collection operations are applied with the arrow notation already mentioned above. Roughly speaking, the dot notation is used when a property follows, i.e., an attribute or a role follows, and the arrow notation is used when a collection operation follows.

Variables in collection operations: Most collection operations allow variables to be declared (possibly including a type specification), but the variable may be dropped if it is not needed.

Example: The following expressions are equivalent.

```
motorCtrl.project->forAll(Budget<120) = true
motorCtrl.project->forAll(p|
  p.Budget<120) = true
motorCtrl.project->forAll(p:Project|
  p.Budget<120) = true
```

Another important possibility is a feature to retrieve the finite set of all current instances of a class by appending `.allInstances` to the class name. In order to guarantee finite results `.allInstances` cannot be applied to data types like `String` or `Integer`.

Examples: With regard to collection operations, an OCL evaluator would obtain the following results in the above object diagram.

```
motorCtrl.project->forAll(Budget<120) = true:Boolean
chrysler.supplier->exists(s|s.SupplierNo=99) =
  false:Boolean
Part.allInstances->select(PartNo>=300) =
  Set{batteryCtrl,engineCtrl,motorCtrl}:Set(Part)
```

```
chrysler.part->collect(p|p.Name) =
  Bag{'Battery Code','Motor Code'} : Bag(String)
chrysler.part->collectNested(p|p.parent) =
  Bag{Set{engineCtrl},Set{engineCtrl}}:Bag(Set(Part))
chrysler.part->collect(p|p.parent) =
  Bag{engineCtrl,engineCtrl}:Bag(Part)
chrysler.part->collectNested(p|p.parent)->size() =
  2:Integer
ford.supplier->isEmpty() = true:Boolean
chrysler.part->includes(carCtrl) = false:Boolean
chrysler.part->including(carCtrl) =
  Set{batteryCtrl,carCtrl,motorCtrl}:Set(Part)
```

Argument collection	Collection operation	Result type
Set/Bag/Sequence(T)	forAll	Boolean
Set/Bag/Sequence(T)	exists	Boolean
Set/Bag/Sequence(T)	select	Set/Bag/Sequence(T)
Set/Bag/Sequence(T)	collectNested	Bag/Bag/Sequence(T')
Set/Bag/Sequence(T)	collect	Bag/Bag/Sequence(T')
Set/Bag/Sequence(T)	size	Integer
Set/Bag/Sequence(T)	isEmpty	Boolean
Set/Bag/Sequence(T)	includes	Boolean
Set/Bag/Sequence(T)	including	Set/Bag/Sequence(T)

Fig. 2.5 Result types of collection operations

Result types in collection operations: The result types of collection operations are shown in the table in Fig. 2.5. Most notably, the operation `collectNested(...)` and `collect(...)` change the kind of an argument collection `Set(T)` to a `Bag(T')` collection. The reason for this is that term inside the collect may evaluate for two different collection elements to the same result. In order to reflect that the result is captured for each collection element, the result appears as often as a respective collection element exists. This convention in OCL resembles the same approach in SQL: SQL queries with the additional keyword `distinct` return a set; plain SQL queries without `distinct` return a bag. In OCL, the convention is similar: Plain `collect(...)` expressions return a bag; using the conversion `asSet()` as in `collect(...)->asSet()` returns a set.

Example: With respect to return types in collection operations, we see the following evaluation in which `collect(...)` is applied to a set, but it properly returns a bag.

```
Set{radioCtrl,motorCtrl}->
  collect(p|p.Name.substring(7,10)) =
    Bag{'Code','Code'}:Bag(String)
```

In the above examples, we also saw this result for a `collectNested` term.

```
chrysler.part =
  Set{batteryCtrl,motorCtrl} : Set(Part)
chrysler.part->collectNested(p|p.parent) =
  Bag{Set{engineCtrl},Set{engineCtrl}}:Bag(Set(Part))
```

Thus the `collectNested(...)` operation applied to `Set(Part)` with the inner term `p.parent`, which returns `Set(Part)`, yields `Bag(Set(Part))`. In this example, a bag is needed in order to capture the result correctly.

Operation `flatten()`: In OCL, collections can be nested. For example, one can build bags whose elements are sets. In order to flatten nested collections to unnested ones, the operation `flatten()` is available. The kind of the result collection is determined by the outermost collection. For example, bags of sets of something would be flattened to bags of something. For building sequences, an implementation-dependent order is chosen.

Example: The next expressions demonstrate the effect of `flatten()`.

```
Set{Set{10,20},Set{30,40}}->flatten() =
  Set{10,20,30,40}:Set(Integer)
Set{Set{10,20},Set{20,30}}->flatten() =
  Set{10,20,30}:Set(Integer)
Bag{Bag{10,20},Bag{30,40}}->flatten() =
  Bag{10,20,30,40}:Bag(Integer)
Bag{Bag{10,20},Bag{20,30}}->flatten() =
  Bag{10,20,20,30}:Bag(Integer)
Bag{Set{10,20},Set{30,40}}->flatten() =
  Bag{10,20,30,40}:Bag(Integer)
Bag{Set{10,20},Set{20,30}}->flatten() =
  Bag{10,20,20,30}:Bag(Integer)
```

Dot shortcut: Another convenient OCL feature is the dot shortcut which allows the developer an easy navigation through a class diagram using multiple roles. Speaking technically, a property `propD` may follow a dot as in the term `expr.propC.propD`, although the left part `expr.propC` yields a collection and only a collection operation and not a property (attribute or role) would be expected. However, the term `expr.propC.propD` is understood as a shortcut for `expr.propC->collect(x|x.propD)`. The aim of this shortcut is to avoid to explicitly write calls to `collect(...)` and to simply navigate with properties (attributes or roles) as for example in `expr.propC.propD.propE`. The dot shortcut is on the one hand very convenient, because it allows the developer easy navigation through a class diagram. On the other hand it blurs the distinction between a single object and an object collection insofar that a property can be applied with the dot shortcut to a collection as if the collection was an object.

Examples: The following examples illustrate the dot shortcut and the effects of `flatten()` in context of the above object diagram.

```
chrysler.part->collectNested(p|p.parent) =
  Bag{Set{@engineCtrl},Set{@engineCtrl}}:
```

```
    Bag(Set(Part))
chrysler.part->collectNested(p|p.parent)->flatten() =
  Bag{@engineCtrl,@engineCtrl}:Bag(Part)
chrysler.part->collect(p|p.parent) =
  Bag{@engineCtrl,@engineCtrl}:Bag(Part)
chrysler.part->collectNested(p|p.parent)->flatten()->
  collect(p|p.Name) =
  Bag{'Engine Code','Engine Code'}:Bag(String)
chrysler.part.parent.Name =
  Bag{'Engine Code','Engine Code'}:Bag(String)
```

Example: Above we mentioned the term `carCtrl.child` with type `Set(Part)` and the term `carCtrl.child.child` with type `Bag(Part)`. This difference in the type is essentially a consequence from a combination of the dot shortcut and the fact that `collect` returns a bag when applied to a set: `carCtrl.child.child` is short for `carCtrl.child->collect(child)` which is a term having type `Bag(Part)`.

Operation definitions with OCL: OCL may be used to define side-effect free operations. You may associate a correctly typed OCL term with an operation name. The term may use the declared parameters. The operation definition may be recursive.

Example: In the class `Project` one could define an operation `partCompetitors()` returning type `Set(Project)`. This operation should yield the set of those projects needing at least one common part with the considered project. The OCL operation `excluding` (used below) eliminates an element from a collection.

```
Project::partCompetitors():Set(Project) =
  self.part.project->excluding(self)->asSet()
```

The operation is formulated within the class `Project`. Therefore the variable `self` references the current object on which the operations is called.

As an example for a recursive operation, we define in the class `Part` the transitive closure `childPlus()` of the role `child` with the help of an auxiliary recursive operation.

```
Part::childPlus():Set(Part)=childPlusAux(self.child)
Part::childPlusAux(aPartSet:Set(Part)):Set(Part)=
  let oneStep:Set(Part)=aPartSet.child->asSet() in
  if oneStep->exists(p|aPartSet->excludes(p))
    then childPlusAux(aPartSet->union(oneStep))
    else aPartSet endif
```

The last example uses the following OCL features not mentioned yet: `let` allows the developer to define sub-expressions to be used in various places; `union` is another collection operation with the obvious meaning. We emphasize that the operation `childPlus` defined in the above manner is well-defined and terminating for all possible object diagrams. Recall, that the class `Part` (as any other class) has only finitely many instances in each system state. Therefore, the recursion finally

terminates. The maximal set which can be computed is `Part.allInstances`. Analogously to the transitive closure `childPlus()`, one could define the transitive and reflexive closure `childStar()`.

2.3 Advanced Conceptual Schema Elements in UML

This section shows how to describe conceptual schemas in UML class diagrams without using any OCL features. In the first part, those UML class diagram features are introduced which are relevant for conceptual schema representation. In the second part, it is discussed how to represent standard ER modeling concepts with these UML features.

2.3.1 Class Diagram Features for Conceptual Schemas

The language features in UML class diagrams introduced above are: Classes, data-valued attributes, associations, and roles. We now turn to describe: Object-valued, collection-valued and compound attributes, role multiplicities, association classes, generalizations, aggregations, compositions, and invariants.

Object-valued attributes: Attributes in UML may not only be data-valued as above, but the attribute type may be a class as well which leads to object-valued attributes. Like associations, object-valued attributes also establish a connection between classes. The object-valued attribute is however only available in the class in which it is defined. The information from that attribute is not directly present in the attribute type class. Thus an object-valued attribute may be regarded as a unidirectional association without an explicit name and where only one role is available.

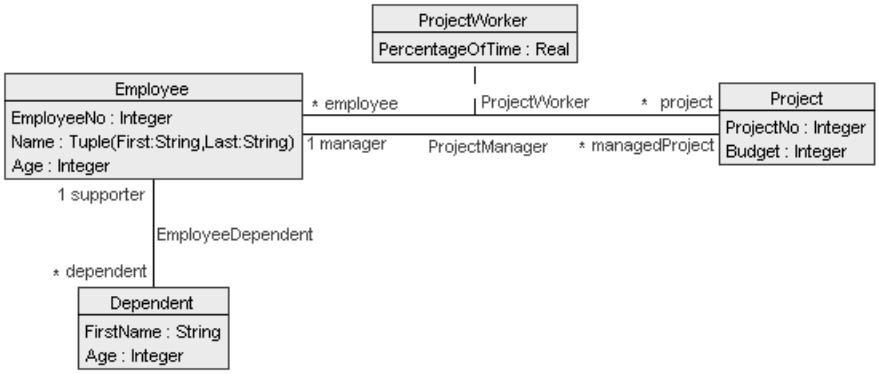

Fig. 2.6 Example UML Class Diagram 2

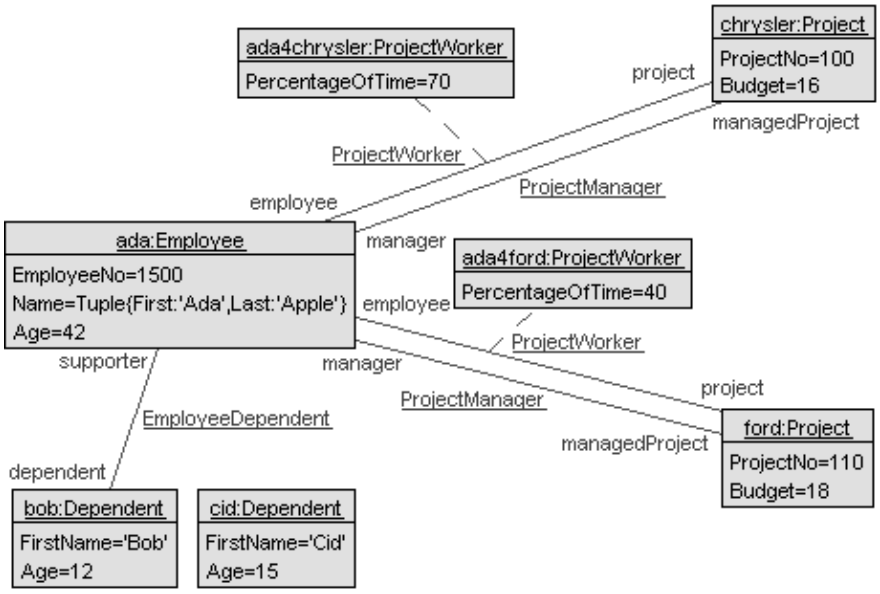

Fig. 2.7 Example UML Object Diagram 2

Examples: The examples in this section will be discussed in the context of the class diagram in Fig. 2.1 and the class diagram in Fig. 2.6 which extends the former one by introducing the new classes `Employee`, `Dependent`, and `ProjectWorker`, and the associations `EmployeeDependent`, `ProjectManager`, and `ProjectWorker`. The fact that `ProjectWorker` is mentioned as a class as well as an association will be explained below. The object diagram in Fig. 2.7 shows an example state for the class diagram from Fig. 2.6. As a forward reference we remark that we will come back later to the fact that ada's project participation sums up to 110 percent.

As an example for an object-valued attribute and as an alternative for the association `ProjectManager`, we could extend the class `Project` by an attribute `manager` with type `Employee`. This could be represented altogether as `Project::manager:Employee`.

Collection-valued attributes: We have already introduced the collection kinds set, bag, and sequence. These collection kinds can be used as type constructors on data types and classes. For building attribute types, the constructors may be nested.

Examples: An attribute could possess a type like `Set(Project)`. As an alternative for the association `ProjectManager` we could have one attribute `managedProject:Set(Project)` in the class `Employee` and another attribute `manager:Employee` in class `Project`. There is however an important difference between the model with the association `ProjectManager` including the roles `manager` and `managedProject` and the model with the two attributes `manager` and `managedProject`. In the model with the association, we would

always have that the roles `managedProject` and `manager` represent the same set of object connections, i.e., the following two OCL expressions will evaluate to true in that model:

```
Employee.allInstances->forAll(e|
  e.managedProject->forAll(p|p.manager=e))
Project.allInstances->forAll(p|
  p.manager.managedProject->includes(p))
```

This is not required to hold in the model possessing the two attributes. In this case the two attributes `managedProject` and `manager` are independent from each other and may represent different sets of object connections.

Another useful application of collection-valued types are collections over the data types like `Set(Sequence(String))`. A value for an attribute typed in that way could be for example the complex value `Set{Sequence{'Rome', 'Euro'}, Sequence{'Tokio', 'Yen'}}`.

Compound attributes: Apart from using the collection constructors `Set`, `Bag`, and `Sequence` for attributes, one can employ a tuple constructor `Tuple`. A tuple has a set of components each possessing a component descriminator and a component type. The collection constructors and the tuple constructor may be nested in an orthogonal way.

Examples: The above value for the type `Set(Sequence(String))` could be represented also with type

```
Set(Tuple(Town:String,Currency:String))
```

and with the corresponding value

```
Set{Tuple{Town:'Rome', Currency:'Euro'},
    Tuple{Town:'Tokio', Currency:'Yen'}}.
```

As a further example for a compound attribute using the `Tuple` constructor, we see in the class diagram in Fig. 2.6 the attribute `Name` in class `Employee` which is a compound attribute with type `Tuple(First:String, Last:String)`.

Role multiplicities: Associations may be restricted by specifying multiplicities. In a binary association, the multiplicity on the other side of a given class restricts the number of objects of the other class to which a given object may be connected to. In a simple form, the multiplicity is given as an integer interval `low..high` (with `low`$\leq$`high`) which expresses that every object of the given class must be connected to at least `low` objects and at most `high` objects of the opposite class. The `high` specification may be given as `*` indicating no higher bound. A single integer `i` denotes the interval `i..i` and `*` is short for `0..*`. The multiplicity specification may consist of more than one interval.

Examples: The multiplicity `1` on the role `supporter` indicates that an object of class `Dependent` must be linked to exactly one object of class `Employee` via the association `EmployeeDependent`.

Association classes: Associations may be viewed again as classes leading to the concept of an association class. Association classes are shown with a class rectangle

and are connected to the association (represented by a line or a rhomb) with a dashed line. Association classes open the possibility of assigning attributes to associations.

Examples: The association `ProjectWorker` is modeled also as a class: `ProjectWorker` is an association class. This makes it possible to assign the attribute `PercentageOfTime` to the association `ProjectWorker`. In the class diagram, `ProjectWorker` can be found redundantly as the class name and as the association name. The specification as the class name would be sufficient.

Generalizations: Generalizations [SS77] are represented in UML with directed lines having an unfilled small triangle pointing to the more general class. Usually the more specific class inherits the properties from the more general class. Generalizations are known in the database context also as ISA (IS-A) hierarchies. In the programming language context often the notion inheritance shows up. Viewed from the more general class its more specific classes are its specializations. In general, a class may have many specializations, and a class may have many generalizations. A set of generalizations may be restricted to be `disjoint` and a set of generalizations may be classified as `complete`. The classification `disjoint` means that any two specific classes are not allowed to have a common instance. The label `complete` means that every instance of the general class is also an instance of at least one more specific class. The explicit keywords `overlapping` and `incomplete` may be attached to sets of generalizations for which no respective restriction is made.

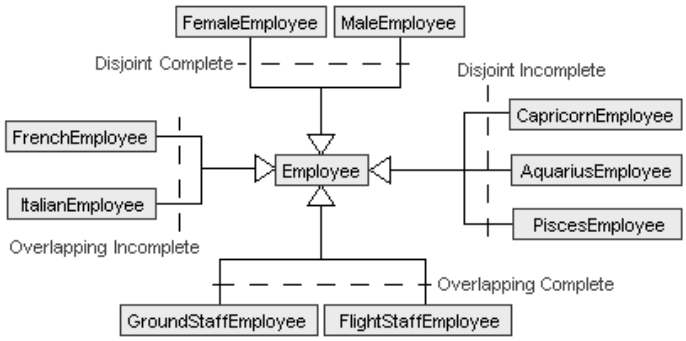

Fig. 2.8 Different Example Generalizations and Specializations in UML

Examples: Fig. 2.8 shows different specializations of the class `Employee`. The subclasses `FemaleEmployee` and `MaleEmployee` represent a disjoint and complete classification. The subclasses `CapricornEmployee`, `AquariusEmployee`, and `PiscesEmployee` classify employees according to their birthday (December 22-January 20, January 21-February 19, February 20-March 20, respectively). This classification is disjoint but incomplete. The subclasses `GroundStaffEmployee` and `FlightStaffEmployee` in the context of an airline company are labeled overlapping and complete, because each airline employee either works on the ground or during a flight and, for example, a steward

is allowed to work on the ground during boarding and of course during the flight. The subclasses `FrenchEmployee` and `ItalianEmployee` are overlapping because employees may have two citizenships, but it is incomplete because, e.g., Swiss employees are not taken into account.

Special care must be devoted to the classifications `overlapping` and `incomplete`. As already said, they represent the general case and no restriction is made by these classifications. But the wording could improperly suggest, that an overlap *must* exist and the incompletion *must* occur, although this is not the case. Altogether, `overlapping` and `incomplete` in the class diagram would accept an object diagram which is `disjoint` and `complete`, but `disjoint` and `complete` in the class diagram would not accept an object diagram being `overlapping` or `incomplete`.

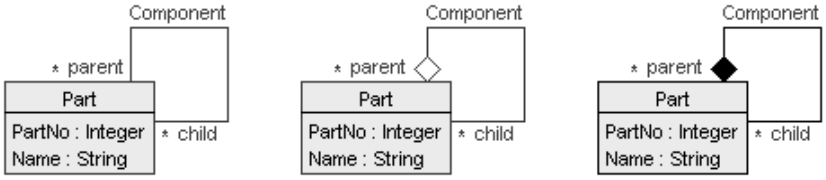

Fig. 2.9 Component as Association, Aggregation, and Composition

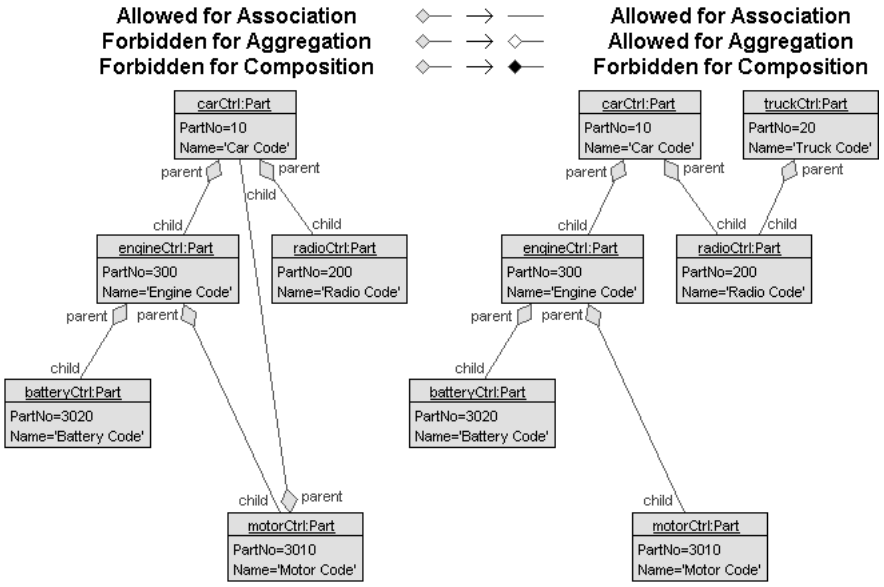

Fig. 2.10 Forbidden and Allowed Object Diagrams for Aggregation and Composition

Aggregations: Part-whole relationships [SS77] are available in UML class diagrams in two forms. The first form represents a loose binding between the part and the whole, the second form realizes a stronger binding. Both forms can be understood as binary associations with additional restrictions. The first form called aggregation is drawn with a hollow rhomb on the whole side and is often called white diamond. The second form called composition is drawn with a filled rhomb on the whole side and is often called black diamond. The links in an object diagram belonging to a class diagram with a part-whole relationship must be acyclic if one regards the links as directed edges going from the whole to the part. This embodies the idea that no part can include itself as a subpart. Such cyclic links are allowed however for arbitrary associations. Part objects from an aggregation are allowed to be shared by two whole objects whereas this is forbidden for composition.

Examples: The class diagrams in Fig. 2.9 show on the left the association `Component` already introduced and on the right two alternatives in which the association is classified as an aggregation with a white diamond and as a composition with a black diamond, respectively. Recall that roles are essential in reflexive associations and therefore in reflexive part-whole relationships. Here the `parent` objects play the whole role and the `child` objects play the part role. The two object diagrams in Fig. 2.10 explain the differences between association, aggregation, and composition. The diamonds are shown as grey diamonds, a symbol which does *not* exist in the UML. We will discuss what happens if the grey diamond is substituted by a white or black one. If the grey diamond is replaced by a white diamond, the left object diagram is forbidden, because there is a cycle in the part-whole links which would mean that the object `carCtrl` is a part of itself. This would also hold for the other two objects on the cycle. Recall that if we would have a simple association instead of the grey diamond, this object diagram would be allowed. If the grey diamond is replaced by a white diamond, the right object diagram is an allowed object diagram. Here, the object `radioCtrl` is shared by the objects `carCtrl` and `truckCtrl`. Naturally, if the grey diamond would become an association, the right object diagram is allowed as well.

Compositions: Compositions pose further restrictions on the possible links in addition to the required acyclicity. Part objects from a composition cannot be shared by two whole objects. The table in Fig. 2.11 gives an overview on the properties of associations, aggregations, and compositions.

	Acyclicity	Prohibition of sharing
Association	–	–
Aggregation	+	–
Composition	+	+

Fig. 2.11 Overview on Properties of Associations, Aggregations, and Compositions

Examples: Let us now discuss what happens in Fig. 2.10 if the grey diamond is substituted in order to represent compositions. If the grey diamond is replaced by a

black diamond, the left object diagram is again forbidden, because there is a cycle in the part-whole links. If the grey diamond is replaced by a black diamond, the right object diagram is a forbidden object diagram for compositions, because sharing of objects is not allowed in that case. To show also a positive example for composition and aggregation, we state that, if we remove the link from `motorCtrl` to `carCtrl` in the left object diagram, we get a valid object diagram for composition and aggregation.

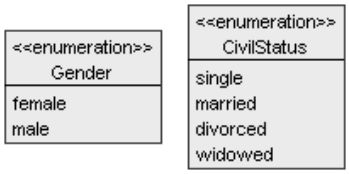

Fig. 2.12 Enumerations in UML

Data types and enumeration types: UML offers a collection of predefined data types with usual operations on them. The data types include `Integer`, `Real`, `String`, and `Boolean`. Application dependent enumeration types can also be defined in a class diagram. The enumeration type name is followed by the list of allowed enumeration literals. Enumeration types can be used as attribute, operation parameter or operation return types.

Examples: Fig. 2.12 shows two enumeration types useful in the context of our example. The type `Gender` may represent the gender of an employee and the type `CivilStatus` its civil status.

Invariants: OCL allows the developer to specify invariants, i.e., conditions which must be true during the complete lifetime of an object (or perhaps more precisely, at least, in moments when no activity in the object takes place). Such invariants are implicitly or explicitly universally quantified OCL formulas introduced with the keyword `context`.

Example: In order to require that employees have an age of at least 18, one could state the following invariant.

```
context Employee inv EmployeeAreAtLeast18: Age>=18
```

That constraint has an implicit variable `self` of type `Employee` and is equivalent to:

```
context self:Employee inv EmployeeAreAtLeast18:
  self.Age>=18
```

Instead of `self` we could have used any other name for the variable, e.g. the variable `e`. The invariant corresponds to the following OCL formula which must be true in all system states.

```
Employee.allInstances->forAll(self|self.age>=18)
```

2.3.2 Representation of Standard ER Modeling Concepts

This section explains how those basic ER modeling concepts which do not need OCL can be expressed in UML class diagrams. Some more advanced ER modeling concepts needing OCL, e.g., primary keys or computed attributes, are explained later when also OCL is used.

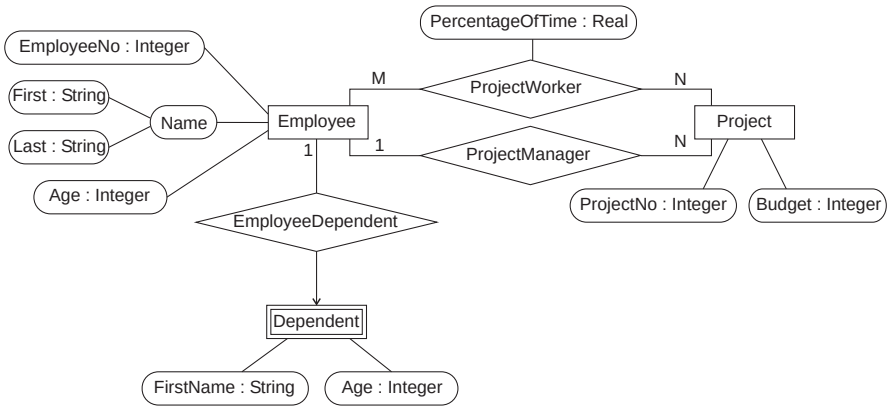

Fig. 2.13 Example ER Diagram

The main concepts from the ER model have a direct representation in UML class diagrams. The ER diagram in Fig. 2.13 shows the ER representation of what has been shown in the UML class diagram in Fig. 2.6.

- Standard entities are represented in the ER notation and in the UML as rectangles. In the ER notation, single lines are used for ordinary entities and double lines for dependent entities.
- In the ER approach, binary relationships and n-ary relationships are shown as rhombs with the relationship name within the rhomb. Binary relationships are pictured as lines in the UML. N-ary relationships in UML are shown with small rhombs. The relationship respectively association name is given close to, but not inside the rhomb.
- Simple ER cardinalities (called multiplicities in UML) as in the example diagram can equivalently be shown with the UML multiplicities `0..*` and `1`. But be warned: The ER notation with intervals as in `(0,*)` is placed differently in the ER approach and UML.
  In the context of relationships, we emphasize that relationship names are usually mandatory in the ER approach. Association names are however optional in the UML in general. This fact shows that relationships play a more important role in ER than associations in the UML. One reason for this may be seen in the fact that one generates Relational schemas from relationships and one needs names for these schemas.

- Standard attributes have an extra symbol in ER, but the attributes are integrated into the class rectangle in UML.
- Roles are shown in a similar way in ER and UML, although we have not explicitly shown them in the ER approach.
- Weak entities depicted in ER as double lined rectangles do not have an explicit notation in UML, but may be expressed with a `1..*` multiplicity. In addition, the owning entity could indicate ownership with a black diamond. Additional OCL constraints which are discussed in the next section will govern the object identification.
- ISA hierarchies [SS77] from ER may be represented in UML with generalizations and additional constraints. Union, disjoint, overlapping, and partitioned ISA hierarchies as discussed in the ER literature correspond to generalizations with constraints as shown in the table in Fig. 2.14.

ISA notion	UML notion
union	complete and overlapping
disjoint	complete and disjoint
overlapping	incomplete and overlapping
partitioned	incomplete and disjoint

Fig. 2.14 Correspondence between ISA hierarchies and UML constraints

- Compound and multi-valued attributes are realized in UML with the `Tuple` and collection constructors `Set`, `Bag`, and `Sequence`.
- Mandatory or optional participation in relationships is expressed in UML with multiplicities.
- Part-whole relationships [SS77] have been proposed in the ER approach with several notations. Part-whole relationships are represented in UML with the white or black diamond.

2.4 Employing OCL for Conceptual Schemas

This section will explain the use of UML extension concepts like constraints and stereotypes for standard ER concepts as keys, derived and computed attributes. The section will also show how to utilize queries which are executed on sample database states during database schema development.

2.4.1 Standard ER Concepts Expressed with OCL

Keys: An identification mechanism for objects is probably a very fundamental application of OCL within conceptual modeling. In databases, objects often possess a set of attributes which identify an object uniquely.

Example: In the running example, we assume `Employee` objects are identified by the attribute `EmployeeNo`. This is expressed in OCL as follows.

```
context e1:Employee inv EmployeeNoIsKey:
  Employee.allInstances->forAll(e2|
    e1<>e2 implies e1.EmployeeNo<>e2.EmployeeNo)
```

Alternatively and equivalently, we could state the implication the other way round.

```
context e1:Employee inv EmployeeNoIsKey:
  Employee.allInstances->forAll(e2|
    e1.EmployeeNo=e2.EmployeeNo implies e1=e2)
```

We emphasize, that within the context of an object-oriented data model like the one from UML, there is a difference between specifying no keys at all and designating the set of all attributes as the key. Assume `Part` objects are identified by the combination of the part number and the name. Recall that name and part number are the only attributes of class `Part`.

```
context p1:Part inv NamePartNoIsKey:
  Part.allInstances->forAll(p2|
    p1<>p2 implies
      (p1.Name<>p2.Name or p1.PartNo<>p2.PartNo))
```

Requiring this invariant is different from giving no key specification, because with this invariant it is not possible to have two different `Part` objects with the same `PartNo` and `Name`. But this is possible in a model where no keys are specified.

In order to represent the identification of `Dependent` objects in the spirit of the ER model, the key restriction for class `Dependent` would look as follows.

```
context d1:Dependent inv FirstNameEmployeeNoIsKey:
 Dependent.allInstances->forAll(d2| d1<>d2 implies
  (d1.FirstName<>d2.FirstName or
   d1.supporter.EmployeeNo<>d2.supporter.EmployeeNo))
```

As a variation, a similar requirement could be stated using inequality on `Employee` objects.

```
context d1:Dependent inv FirstNameEmployeeNoIsKey:
  Dependent.allInstances->forAll(d2| d1<>d2 implies
    (d1.FirstName<>d2.FirstName or
     d1.supporter<>d2.supporter))
```

Further possibilities for conceptual modeling of keys are discussed in [Gog99].

Derived or computed attributes: We have discussed compound and multi-valued attributes above. Another variation for attributes are so-called derived or computed attributes. Derived respectively computed attributes can be realized in OCL with an invariant or with a derivation rule within an operation.

Example: Assume we want to record for `Part` objects the number of direct (not indirect) children the respective `Part` object has with respect to the `Component` association. This could be realized with an invariant assuming class `Part` has an additional attribute `NumOfChildren` or as a definition for an additional operation `NumOfChildren()`:

```
context Part inv NumOfChildrenDerived:
  NumOfChildren=self.child->size()
Part::NumOfChildren()=self.child->size()
```

2.4.2 Constraints and Stereotypes

General OCL invariants may be employed for conceptual modeling in order to describe integrity constraints for a conceptual schema. UML offers to denote such invariants in explicit form or the constraints may be indicated as a shortcut by using stereotypes.

Keys as stereotypes: Because certain kind of constraints appear frequently in conceptual modeling, it makes sense to indicate this recurring structure by indicating the constraints only with stereotypes. A very good example for this are keys. At least two alternative notations for key stereotypes can be thought of: (1) Indicating for each attribute separately whether it contributes to the key, or (2) indicating the set of key constituents as a whole.

Example: For the running example, key specifications for selected classes could look as shown in Fig. 2.15. In the class `Employee` the key consists of the attribute `EmployeeNo`. For `Part`, the key is made of `PartNo` and `Name`. The key for `Dependent` consists of `FirstName` and the reference to the key of the supporter.

Alternative keys could be indicated similar to the above mentioned key stereotype notation (2) in which the complete set of alternative key attributes would be indicated as a whole. As a side remark, we mention that the underlining of attributes in UML class diagrams, which is used in some ER notations to indicate keys, has already a fixed, different meaning in UML: Underlined attributes indicate class attributes which in contrast to ordinary object attributes describe properties of the class and not properties of the single instances belonging to the class.

Stereotypes for general invariants: Due to UML's and OCL's flexibility, apart from keys various useful patterns for invariants could be provided by stereotypes, e.g., attribute restrictions, commutativity restrictions, and existence dependencies.

- Attribute restrictions could be an alternative for enumeration types with the additional advantage that respective operations would be applicable then as well.

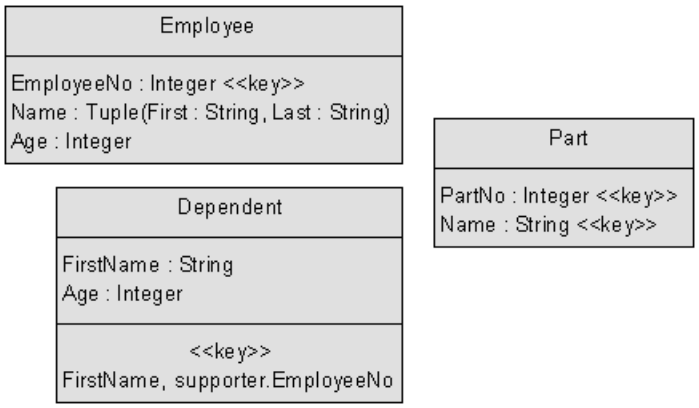

Fig. 2.15 Keys represented as UML Stereotypes

For example, an attribute `month:Integer` could be restricted by a stereotype `<<1..12>>`.

- Commutativity restrictions could indicate that two paths in the class diagram are commutative in the sense that the two paths either yield the same result or that the result of one path is included in the other. Given the context of a particular class and appropriate roles, for example, `self.role1.role2=self.role3` would require that the results of the two expressions involving the roles coincide. Instead of requiring equality, one could allow that one specifies an inclusion with the stereotype `<<subset>>`. For example, within the context of class `Employee`, the requirement `self.managedProject <<subset>> self.project` would express that a manager works on her or his projects.
- Existence dependencies, like the one for `Dependent` objects could be specified by providing a term indicating the master object the slave object depends on. E.g., within the context of class `Dependent` the term `self.supporter` within the class rectangle in a special section labeled `<<dependency>>` could indicate that for each `Dependent` object, a supporting employee must exists. In easy cases like the above one, dependencies can also be shown with a multiplicity specification.
- Apart from these application-specific constraints, the UML provides a standard constraint for requiring that two or more associations exclude each other with the keyword `xor` and a standard constraint expressing that one association is included in the other by using the keyword `subset`.

Transitive closure: By means of appropriate operations it is possible in UML and OCL to define the transitive closure as a language built-in. Any property `C::prop:Set(C)` for a class `C` can be extended to `C::propPlus:Set(C)` for yielding the transitive closure and to `C::propStar:Set(C)` for yielding the transitive and reflexive closure. One would automatically extend the model with appropriate operations as indicated below.

```
C::propPlus():Set(C)=propPlusAux(self.prop)
C::propPlusAux(aSet:Set(C)):Set(C)=
  let oneStep:Set(C)=aSet.prop->asSet() in
  if oneStep->exists(p|aSet->excludes(p)) then
    propPlusAux(aSet->union(oneStep)) else aSet endif
C::propStar():Set(C)=propPlus()->including(self)
```

This notation can be generalized to bags and sequences.

Example: The requirement that no `Part` object can be connected to itself with a chain of `Component` links in the `child` direction could be stated as follows.

```
context p:Part inv ComponentNotReflexive:
  not(p.childPlus->includes(p))
```

General constraints: Apart from constraints indicated with stereotypes, one can naturally employ the invariant mechanism of OCL and define special, application dependent invariants.

Examples: Above we have discussed what would happen if the association `ProjectManager` would be replaced by two object-valued attributes in the participating classes. In order to only allow similar object diagrams as in the model with the association, one would need then the following two invariants.

```
context Employee inv ManagerManagesOwnProjects:
  managedProject->forAll(p|p.manager=self)
context Project inv ProjectManagedByProjectManager:
  manager.managedProject->includes(self))
```

Note that in general, both directions of the constraint and not only one direction has to be state.

Another example for a general constraint concerns the attribute `Percentage-OfTime` in the relationship `ProjectWorker`. The sum of percentages for a single employee should not be more than 100 percent.

```
context Employee inv SumPercentageOfTimeLessEqual100:
  self.projectWorker.PercentageOfTime->sum()<=100
```

With respect to this constraint, the object diagram from Fig. 2.7 is invalid, because the sum of ada's project participation is 110 percent.

The above example also shows one OCL feature which we have not covered yet: In the context of association classes it is possible to navigate from a participating class to the association class and also from the association class to the participating classes. Above, the role `projectWorker` is a property within the class `Employee` having result type `Set(ProjectWorker)`.

2.4.3 Queries

OCL also supports the formulation of queries. Ordinary SQL following the select-from-where pattern would be formulated in OCL obeying an allInstances-select-collect pattern.

Example: Find employee numbers of employees having at least two dependents.

```
select EmployeeNo
from   Employee
where  exists
        (select *
         from   Dependent d1, Dependent d2
         where  d1.EmployeeNo=d2.EmployeeNo and
                d1.EmployeeNo=Employee.EmployeeNo and
                d1.FirstName<>d2.FirstName)

Employee.allInstances->
  select(dependent->
           exists(d1,d2|d1.FirstName<>d2.FirstName))->
    collect(EmployeeNo)

Employee.allInstances->
  select(e:Employee|e.dependent->
           exists(d1,d2|d1.FirstName<>d2.FirstName))->
    collect(EmployeeNo)
```

The SQL query, which is formulated on a Relational database schema, uses a subquery to filter the result and a select clause to indicate which attributes are wanted. In OCL, one starts with an `allInstances` expression, then one filters the objects with a select expression and finally obtains the desired attributes with a collect expression.

2.5 Describing Relational Schemas with UML

This section will show how Relational schemas are represented in UML. Constraints and stereotypes will represent primary keys and foreign keys.

2.5.1 Relational Schemas

Relational Schemas in UML: There are radically different alternatives for representing Relational schemas in UML: (1) One might represent each entity and each relationship from the conceptual schema as a separate class, or (2) one could use the type constructors offered by OCL (like `Tuple` and `Set`) and represent the entire database as a single complex value. There are other solutions which lie between these extreme points. We will further follow an alternative in which a Relational

schema is represented by a class, however we will shortly also explain the other extreme.

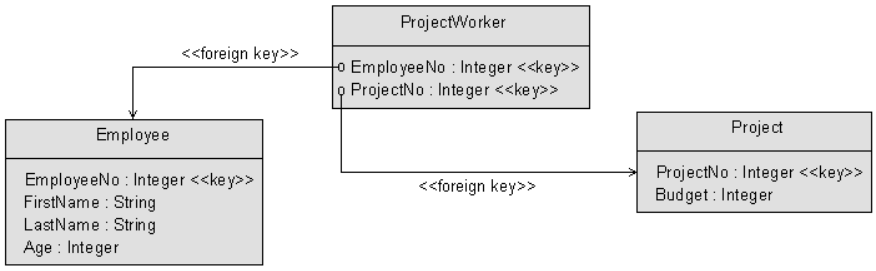

Fig. 2.16 Foreign Keys represented graphically with UML Stereotypes

Example: Let us consider only the two entities `Employee` and `Project` together with their relationship `ProjectWorker`, and let us further assume that we translate this into three Relational schemas. If we give a separate class for each entity and each relationship, we achieve the representation in Fig. 2.16. If we represent the three Relational schemas with a complex value, we achieve the structure in Fig. 2.17. Primary and foreign keys would have to be formulated additionally as OCL invariants.

```
DB:Tuple(Employee:Set(Tuple(EmployeeNo:Integer,
                            FirstName:String,
                            LastName:String,
                            Age:Integer)),
         Project:Set(Tuple(ProjectNo:Integer,
                           Budget:Integer)),
         ProjectWorker:Set(Tuple(EmployeeNo:Integer,
                                 ProjectNo:Integer)))
```

Fig. 2.17 Relational schemas as complex value

2.5.2 Constraints for Primary and Foreign Keys

Representing Primary Keys and Foreign Keys: Primary keys in the Relational schema can be shown with a stereotype as primary keys in the conceptual schema. For the representation of foreign keys there are again two alternatives, a graphical one and a textual one. (1) In the graphical solution, the Relational schema possessing the foreign key would point to the Relational schema in which the referenced

primary key occurs. Technically, this *pointing to* would be a UML dependency pictured in graphical form using a stereotype. (2) In the textual solution, the Relational schema possessing the foreign key would indicate the Relational schema in which the referenced primary key occurs. On the technical level, this would again be a UML dependency but this time displayed in textual form.

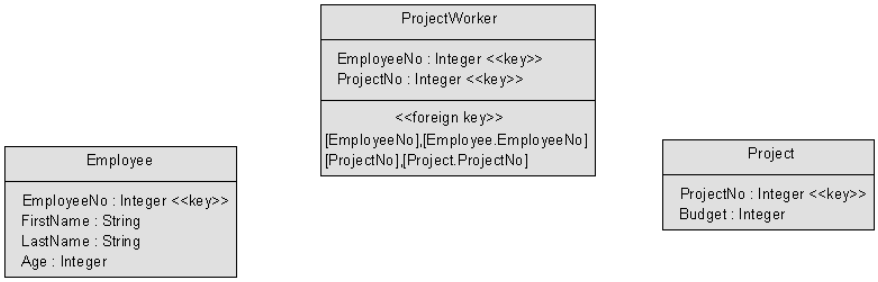

Fig. 2.18 Foreign Keys represented textually with UML Stereotypes

Example: Figures 2.16 and 2.18 show the graphical and textual alternative for the example. The graphical alternative has the advantage of visually showing the connection between the Relational schemas. But the graphical representation has also the disadvantage that it becomes more complicated and even not understandable, if the foreign key consists of more than one attribute and if additionally the foreign key references attributes in the same Relational schema.

Stereotypes for primary keys and foreign keys are only shortcuts for more involved OCL invariants not explicitly shown, but being present behind the visual representation. In our example, we would have that the stereotypes are shortcuts for the following OCL invariants.

```
context e1:Employee inv EmployeeNoIsKey:
  Employee.allInstances->forAll(e2 |
    e1<>e2 implies e1.EmployeeNo<>e2.EmployeeNo)
context p1:Project inv ProjectNoIsKey:
  Project.allInstances->forAll(p2 |
    p1<>p2 implies p1.ProjectNo<>p2.ProjectNo)
context pw1:ProjectWorker inv EmployeeProjectNoIsKey:
  ProjectWorker.allInstances->forAll(pw2 |
    pw1<>pw2 implies
      (pw1.EmployeeNo<>pw2.EmployeeNo or
       pw1.ProjectNo<>pw2.ProjectNo))
context pw:ProjectWorker inv EmployeeNoIsForeignKey:
  Employee.allInstances->exists(e |
    pw.EmployeeNo=e.EmployeeNo)
context pw:ProjectWorker inv ProjectNoIsForeignKey:
  Project.allInstances->exists(p |
```

```
pw.ProjectNo=p.ProjectNo)
```

As a final remark we emphasize that foreign keys are *not* associations, because an association would imply that it will be manifested by links which is not true for foreign keys. Foreign keys are dependencies and can be represented with stereotypes. We also emphasize that we represent Relational schemas as classes. In this UML representation, there are no associations or relationships, but only dependencies.

2.6 Metamodeling Data Models with UML

This section studies a UML metamodel for the Entity Relationship (ER) and the Relational data model. UML is well-suited for the description of metamodels. We start by describing the syntax of the ER data model through the introduction of classes for ER schemas, entities, and relationships. We also describe the semantics of the ER data model by introducing classes for ER states, instances, and links. The connection between syntax and semantics is established by associations explaining that syntactical objects are interpreted by corresponding semantical objects. Analogously this is done for the Relational data model. The CWM metamodel from [OMG03] is to a certain extent comparable to our approach. However there, only the syntax of data models is treated, not the interpretation of database schemas as in our approach.

2.6.1 Class Diagram

Consider the class diagram in Fig. 2.19. It shows four *packages*: In the left part a solid grey and a solid black package, in the right part a dashed grey and a dashed black package. The two solid left packages model the syntax of the data models, the two dashed right packages the semantics; the upper two packages describe the ER data model, the lower two packages the Relational data model. The ER and the Relational data model share some concepts, namely the parts in the middle specifying data types, attributes and their semantics. We have indicated the multiplicities in the class diagram. All role names are identical to the respective class with the first letter of the class name converted to a lower case letter, e.g., we have a role names `dataType` and `relDBSchema`. The various parts of this class diagram will be explained below with the scenario from Fig. 2.20 and the object diagrams in Figs. 2.21, 2.22, 2.23, and 2.24.

Syntax of the ER data model: 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.

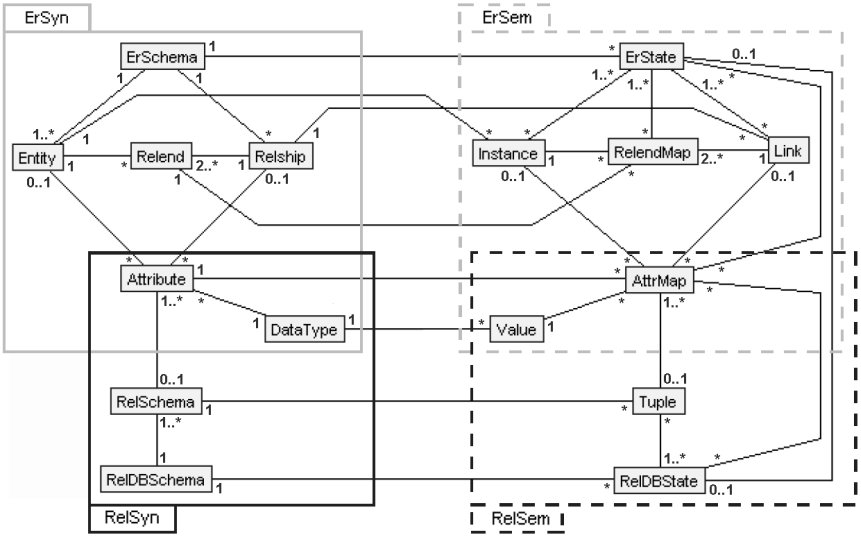

Fig. 2.19 Class Diagram Metamodeling the ER and Relational Data Model

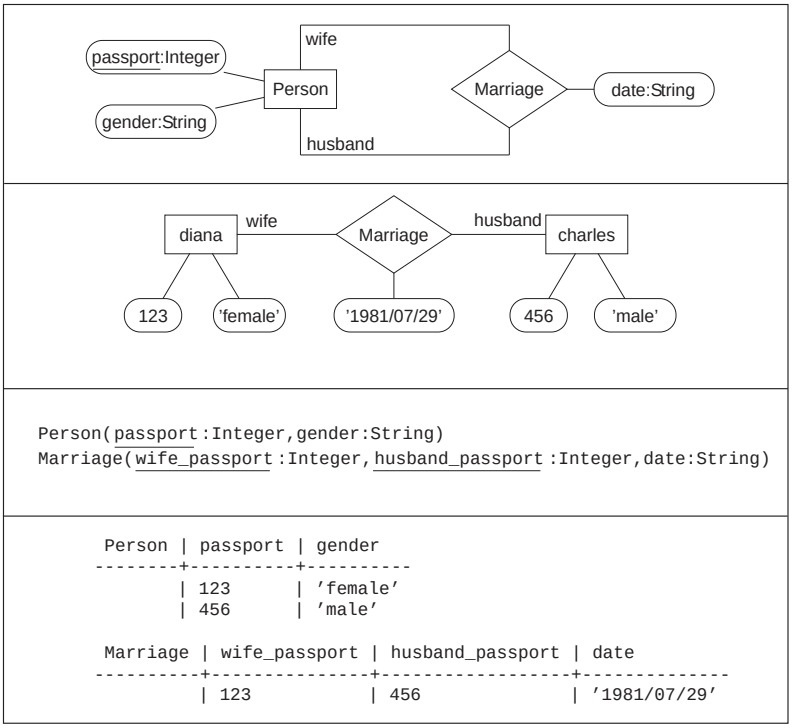

Fig. 2.20 Content of Example Scenario

Semantics of the ER data model: 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` object, respectively.

Syntax of the Relational data model: Here the classes `RelDBSchema`, `RelSchema`, `Attribute`, and `DataType` are needed. `RelDBSchema` objects consist of `RelSchema` objects which possess `Attribute` objects typed through `DataType` objects.

Semantics of the Relational data model: The 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 which are typed by a `RelSchema`. `Tuple` objects in turn consist of a set of `AttrMap` objects assigning a `Value` object to an `Attribute` within the `Tuple`.

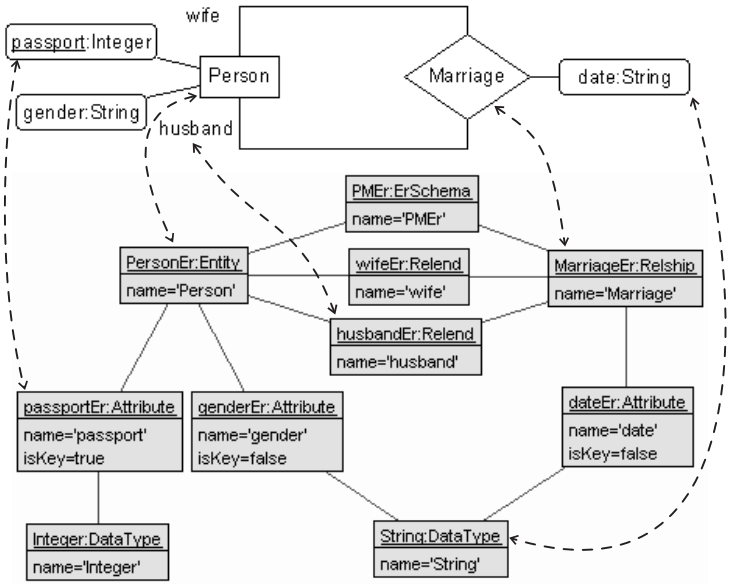

Fig. 2.21 Viewing the Example Scenario as an ER Schema

Let us shortly mention the attributes and operations relevant for the class diagram but being not displayed. All classes in the (left) syntax part possess an attribute `name` of data type `String`. The class `Attribute` has an additional boolean-

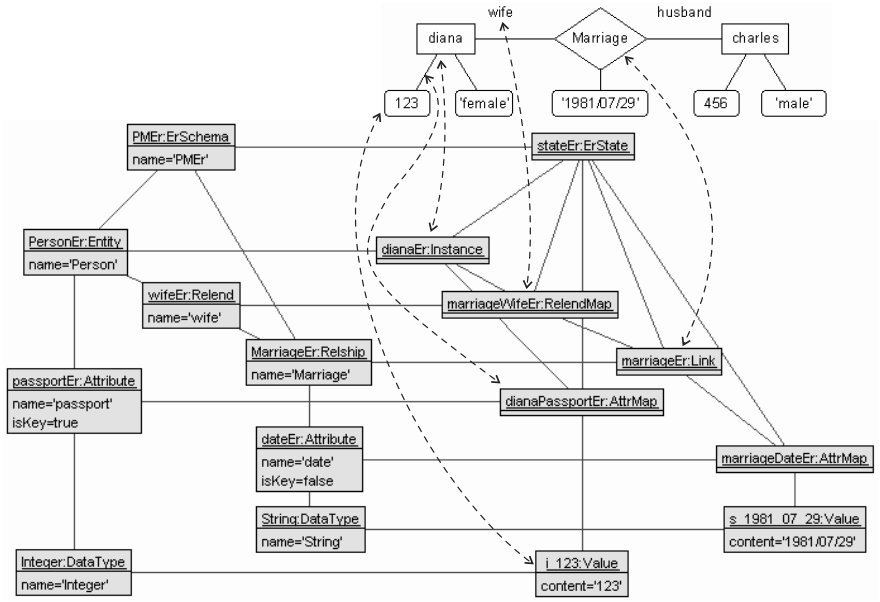

Fig. 2.22 Viewing the Example Scenario as an ER State

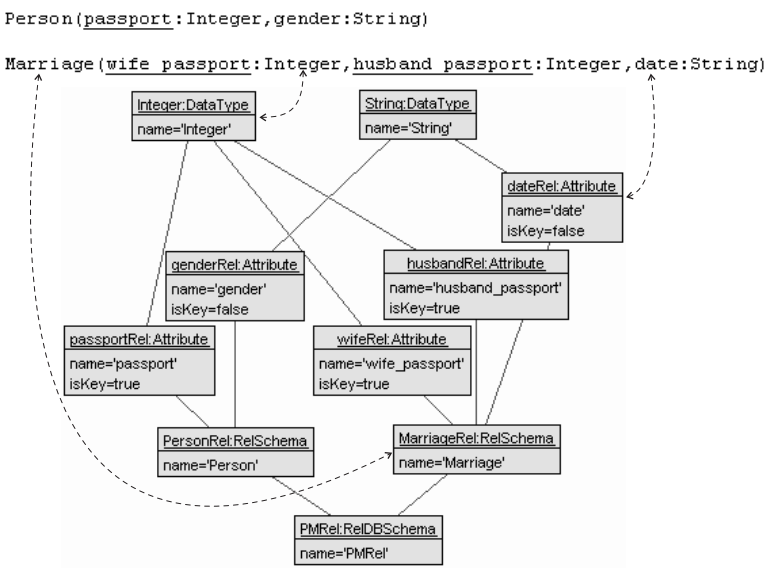

Fig. 2.23 Viewing the Example Scenario as a Relational Schema

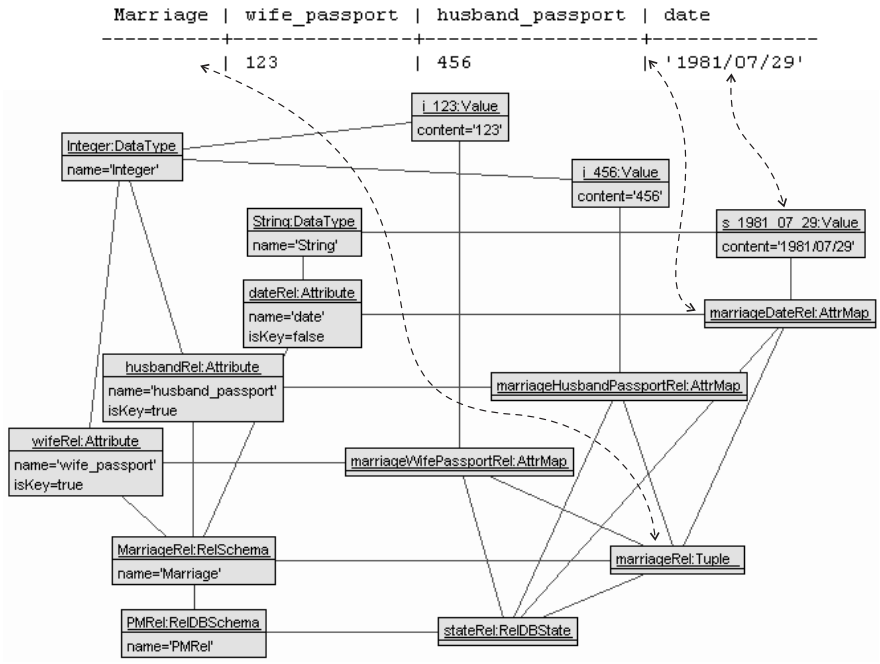

Fig. 2.24 Viewing the Example Scenario as a Relational State

valued attribute `isKey` indicating whether this attribute contributes to the key of the `Entity` or the `RelSchema`. The class `Value` possesses the attribute `content` of data type `String` indicating the actual content of the `Value` object.

Concerning operations, the classes `Instance`, `Link`, and `Tuple` have an operation `applyAttr()` with a `State` and an `Attribute` parameter returning the actual `Value` object of the `Attribute`. The class `Link` has an operation `applyRelend()` with an `ErState` and a `Relend` parameter returning the actual `Instance` of the `Relend`. The classes `Entity` and `RelSchema` possess an operation `key()` returning the set of its key attributes.

2.6.2 Object Diagrams

The modeling is best explained by an example. Figure 2.20 shows an example scenario which is represented in Fig. 2.21 as an ER schema, in Fig. 2.22 as an ER state, in Fig. 2.23 as a Relational schema, and in Fig. 2.24 as a Relational state.

Syntax of the ER data model: Fig. 2.21 shows the metamodel representation of the example ER schema. There is one `ErSchema` object connected to one `Entity` and one `Relship` object. The two `Relend` objects connect the

`Relship` with the `Entity`. The three attributes stand in connection with the `Entity` resp. `Relship` on which they are defined and with the `DataType` of the respective attribute. We regard the upper representation as the concrete syntax of the ER schema and the lower representation in form of an object diagram as the abstract syntax.

Semantics of the ER data model: Fig. 2.22 displays on the left a part of the ER schema and on the right semantical objects instantiating the objects from the ER schema on the left. The semantical objects are typed by horizontal links going to the left: The `ErState` is typed by an `ErSchema`, the `Instance` by an `Entity`, the `Link` by a `Relship`, the `RelendMap` object by a `Relend` object, each `AttrMap` object by an `Attribute` object, and each `Value` object by a `DataType` object. In order to be comprehensible, this left part does not show the complete ER state, but only a part of the ER state.

Syntax of the Relational data model: Fig. 2.23 represents the Relational database schema with two Relational schemas. The first Relational schema has two attributes, and the second one three attributes. All five attributes are typed by appropriate data types.

Semantics of the Relational data model: Fig. 2.24 gives a part from the Relational database state. Only one tuple with three components, i.e., with three `AttrMap` objects, is shown. The three `Value` objects are typed with links into the left syntax part. For example, the two `Value` objects `i_123` and `i_456` are connected to the `DataType` object `Integer`.

2.6.3 Constraints

The multiplicities in the class diagram constrain the valid object diagrams and are so-called model inherent constraints. Apart from these constraints, all parts in the class diagram must be restricted by appropriate explicit constraints. In the total we obtain about fifty constraints. We do not go into the details here, which can be found in [Gog05], but show only one typical example from each of the four parts.

Syntax of the ER data model: Within one `Entity`, different `Attributes` have different names.

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

Thus we would obtain an invalid ER schema, if we change the `name` attribute of the `genderEr` object from `'gender'` to `'passport'` in Fig. 2.21.

Semantics of the ER data model: 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:Instance inv keyMapUnique:
  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))))
```

One would achieve an invalid ER state, if we change the `content` attribute of the `i_123` object from `'123'` to `'456'`, because there is another `Instance` object (not shown in Fig. 2.22), namely `charlesEr`, with `passport` number `'456'` and `passport` is the only key attribute in the example ER schema.

Syntax of the Relational data model: The set of key `Attributes` of a `RelSchema` is not empty.

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

We would get an invalid Relational schema, if we change the `isKey` attribute of the `passportRel` object from `true` to `false`, because then the Relational schema named `Person` would not have any key attributes.

Fig. 2.25 Excerpt from Metamodel Class Diagram Explaining `commutativityAttribute`

Semantics of the Relational data model: As shown in Fig. 2.25, the `Attributes` connected to the `RelSchema` of a `Tuple` are identical to the `Attributes` connected to the `AttrMap` of the `Tuple`. In other words, there are attribute assignments for all `Attributes` of a `Tuple` (and for only those).

```
context self:Tuple inv commutativityAttribute:
  self.relSchema.attribute=self.attrMap.attribute->asSet
```

We would obtain an invalid Relational state, if we would delete the `marriage-WifePassportRel` object. Then there would exist an `Attribute` with name `wife_passport` which is present in the Relational schema named `Marriage`, but one tuple for this Relational schema would miss the attribute assignment for the attribute `wife_passport`, i.e., there would be no corresponding `AttrMap` object.

Our example scenario included only one ER state, namely an ER state where two entities and one relationship connection are present. The metamodel is however more general in the sense that not only one ER state can be described, but it is possible to link several ER states to a single ER schema. For example, the three ER

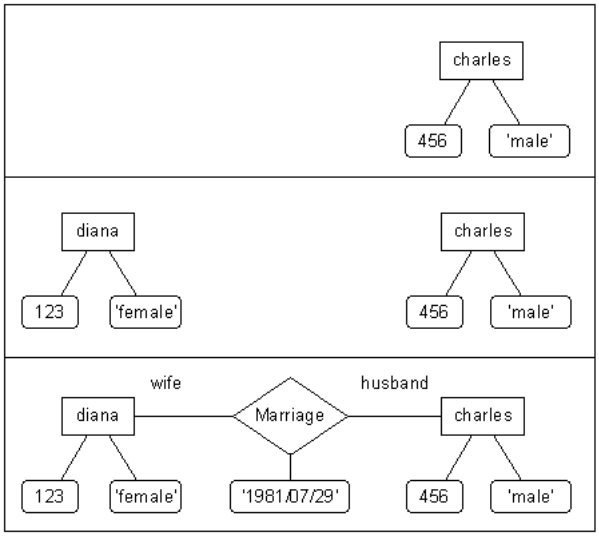

Fig. 2.26 Three consecutive ER states

states displayed in Fig.2.26 together with the corresponding ER schema could be represented as a single object diagram in the metamodel.

Apart from describing the data models, it is also possible to give a metamodel for the transformation of ER schemas into Relational database schemas. We will not go into details here but only refer to the detailed metamodel which can be found in [Gog05]. By characterizing the syntax and semantics of the data models and also the transformation itself within the same (meta-)modeling language, one can include equivalence criteria on the syntactical and on the semantical level for the transformation. In particular, one can give a semantical equivalence criterion requiring that the ER states and the corresponding Relational states carry the same information.

2.7 Further Related Work

Relevant related work has been mentioned already in the respective chapters. In addition, we want to point to the following books and papers relating on the one hand UML and conceptual modeling and on the other hand UML and constraint development. Further relevant literature can be found by using the 'Complete Search' facility on DBLP by searching with 'Conceptual UML Model' or 'UML Database Design', for example.

An early approach for developing databases with object-oriented design techniques is given in [BP98]. Comparisons between designing (database) schemas and class diagrams with UML and with ORM are discussed in [Hal02, HB99]. Object-

oriented and object-relational schemas described with UML and other object-oriented techniques are studied in [MVC03, UD03, APP06]. The work in [Amb09] proposes a UML profile for database design, whereas in [ZT07] a UML profile for conceptual modeling in connection with data warehouses is worked out.

Constraints and OCL have been used for conceptual modeling since the early days of UML. [DHL01] treats the transformation of OCL constraints into Relational database requirements. The text book [Oli07] radically uses OCL and UML for all facets of conceptual modeling. [RO08] discusses the impact of MOF to developing database schemas. [CGQ+08] is a further approach using OCL for conceptual modeling which proposes special treatment of typical, schematic integrity constraints. [CT09] emphasizes incremental development of OCL constraints.

2.8 Conclusions

This contribution has explained how UML can be employed for conceptual modeling of information systems. UML supports on the one hand all classical features of the ER model, and on the other hand also more advanced features like part-whole relationships are expressible as well. Within UML, the textual constraint and query language OCL is available. OCL has many similarities to SQL.

However, support for conceptual modeling within UML can be improved in a number of directions. There are proposals around for a UML profile for data modeling, but an overall accepted solution is still missing. Such a profile should take into account data modeling on various abstraction levels, e.g., the conceptual, the logical, and the physical level. Complete metamodeling of these data models respecting syntactical and semantical aspects is another open issue. One reason for the success of the Relational model is probably the well-studied relationship between descriptive languages like tuple or domain calculus and operationally effective languages like Relational algebra. OCL as a central ingredient for conceptual modeling and as a descriptive language within UML would benefit from a clear relationship to an operationally effective UML execution language.

References

[Amb09] Scott W. Ambler. A UML Profile for Data Modeling. Technical report, AgileData.Org, 2009. `www.agiledata.org/essays/umlDataModelingProfile.html`.

[APP06] Silvia Mara Abrahão, Geert Poels, and Oscar Pastor. A Functional Size Measurement Method for Object-Oriented Conceptual Schemas: Design and Evaluation Issues. *Software and System Modeling*, 5(1):48–71, 2006.

[BGH+10] Fabian Büttner, Martin Gogolla, Lars Hamann, Mirco Kuhlmann, and Arne Lindow. On Better Understanding OCL Collections *or* An OCL Ordered Set is not an OCL Set. In Sudipto Ghosh, editor, *Workshops and Symposia at 12th Int. Conf. Model Driven Engineering Languages and Systems (MODELS'2009)*, pages 276–290. Springer, Berlin, LNCS 6002, 2010.

[BP98] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Prentice Hall, 1998.

[CGQ+08] Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Improving the Definition of General Constraints in UML. *Software and System Modeling*, 7(4):469–486, 2008.

[Che76] Peter P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions Database Systems*, 1(1):9–36, 1976.

[CT09] Jordi Cabot and Ernest Teniente. Incremental Integrity Checking of UML/OCL Conceptual Schemas. *Journal of Systems and Software*, 82(9):1459–1478, 2009.

[DHL01] Birgit Demuth, Heinrich Hussmann, and Sten Loecher. OCL as a Specification Language for Business Rules in Database Applications. In Martin Gogolla and Cris Kobryn, editors, *Proc. 4th Int. Conf. Unified Modeling Language (UML'2001)*, pages 104–117. Springer, LNCS 2185, 2001.

[GBR05] Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[Gog99] Martin Gogolla. Identifying Objects by Declarative Queries. In Mike P. Papazoglou, Stefano Spaccapietra, and Zahir Tari, editors, *Advances in Object-Oriented Data Modeling*, pages 255–277. MIT Press, 1999.

[Gog05] Martin Gogolla. Tales of ER and RE Syntax and Semantics. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*. IBFI, Schloss Dagstuhl, Germany, 2005. Dagstuhl Seminar Proceedings 05161. 51 pages.

[Hal02] Terry A. Halpin. Metaschemas for ER, ORM and UML Data Models: A Comparison. *Jounal of Database Managagement*, 13(2):20–30, 2002.

[HB99] Terry A. Halpin and Anthony C. Bloesch. Data Modeling in UML and ORM: A Comparison. *Journal of Database Managagement*, 10(4):4–13, 1999.

[MVC03] Esperanza Marcos, Belén Vela, and José María Cavero. A Methodological Approach for Object-Relational Database Design using UML. *Software and System Modeling*, 2(1):59–75, 2003.

[Oli07] Antoni Olive. *Conceptual Modeling of Information Systems*. Springer, 2007.

[OMG03] OMG, editor. *Common Warehouse Metamodel (CWM) Specification V 1.1*. OMG, 2003. `www.omg.org`.

[OMG10a] OMG, editor. *Object Constraint Language (OCL) Specification V 2.2*. OMG, 2010. `www.omg.org`.

[OMG10b] OMG, editor. *Unified Modeling Language (UML) Specification V 2.3*. OMG, 2010. `www.omg.org`.

[RBJ05] James Rumbaugh, Grady Booch, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, 2005. 2nd Edition.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.

[RO08] Ruth Raventós and Antoni Olive. An Object-Oriented Operation-Based Approach to Translation between MOF Metaschemas. *Data Knowledge Engineering*, 67(3):444–462, 2008.

[SS77] John Miles Smith and Diane C. P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions Database Systems*, 2(2):105–133, 1977.

[UD03] Susan Darling Urban and Suzanne W. Dietrich. Using UML Class Diagrams for a Comparative Analysis of Relational, Object-Oriented, and Object-Relational Database Mappings. In Scott Grissom, Deborah Knox, Dan Joyce, and Wanda Dann, editors,

Proc. 34th SIGCSE Technical Symp. Computer Science Education (2003), pages 21–25. ACM, 2003.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

[ZT07] José Jacobo Zubcoff and Juan Trujillo. A UML 2.0 Profile to Design Association Rule Mining Models in the Multidimensional Conceptual Modeling of Data Warehouses. *Data Knowledge Engineering*, 63(1):44–62, 2007.

Appendix A: Original ER Diagram from Chen's Paper

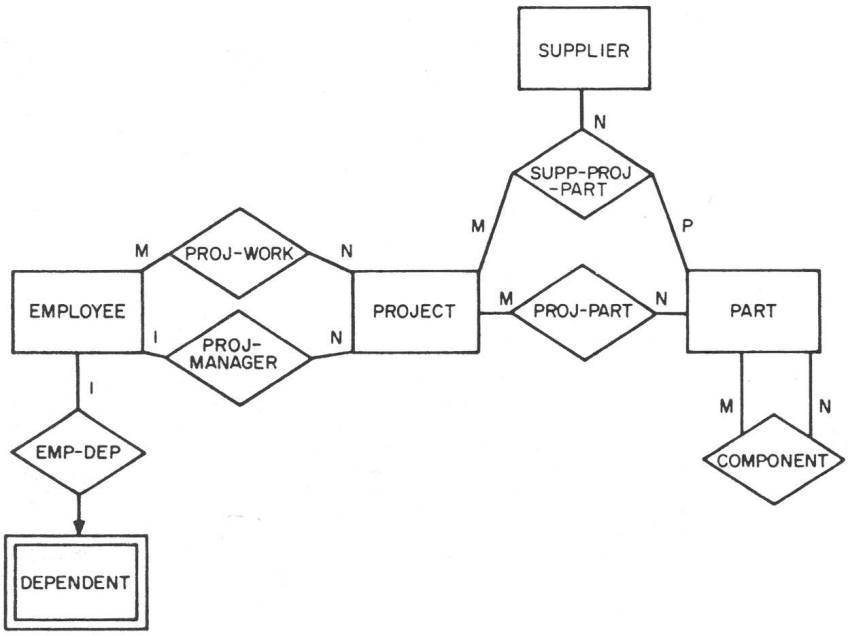

Fig. 2.27 Original ER Diagram from Chen's Paper

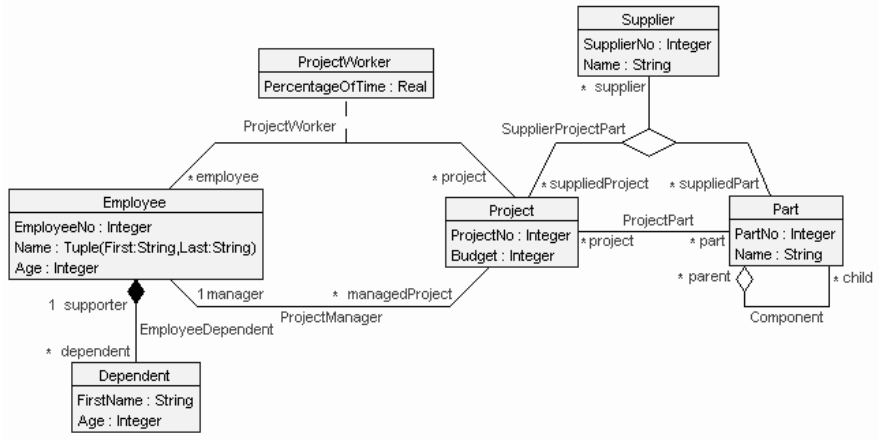

Fig. 2.28 Plain UML Class Diagram Corresponding to Fig. 2.27

```
context e1:Employee inv EmployeeNoIsKey:
  Employee.allInstances->forAll(e2|
    e1<>e2 implies e1.EmployeeNo<>e2.EmployeeNo)
-- above invariant analogously for other classes
context d1:Dependent inv FirstNameEmployeeNoIsKey:
 Dependent.allInstances->forAll(d2| d1<>d2 implies
  (d1.FirstName<>d2.FirstName or
   d1.supporter.EmployeeNo<>d2.supporter.EmployeeNo))
```

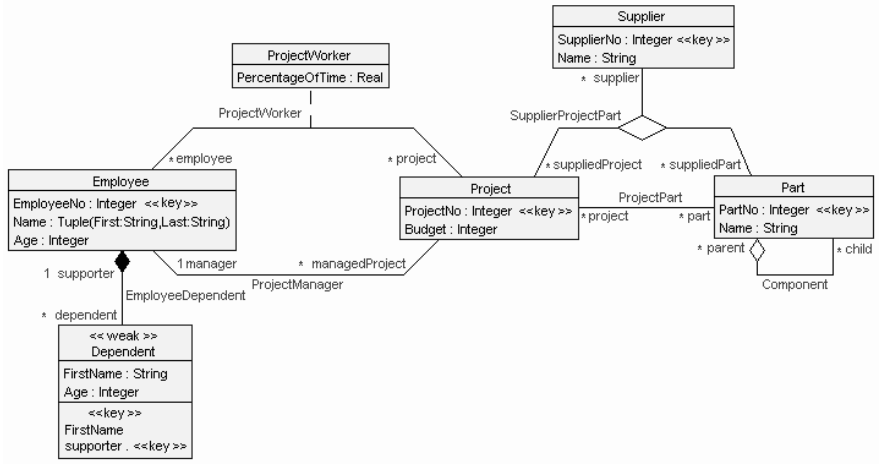

Fig. 2.29 Stereotyped UML Class Diagram Corresponding to Fig. 2.27