Derivation Rules in Object-Oriented Conceptual Modeling Languages

Antoni Olivé

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
08034 Barcelona (Catalonia)
olive@lsi.upc.es

Abstract. We propose three new methods for the definition of derivation rules in O-O conceptual modeling languages. The first method applies to static rules, and associates each derived element with a defining operation. The specification of this operation is then the definition of the corresponding derivation rule. The second method applies to constant relationship types whose instances can be derived when the instances of one of its participant entity types are created. We propose a variant of the previous method to deal with these types. The third method deals with hybrid types, and suggests a simple way to define their partial derivation rules. We propose an adaptation of the three methods to the UML.

1 Introduction

Derived types and derivation rules have been considered fundamental components of conceptual schemas of information systems since the early eighties [17]. The rationale is that derivation rules are domain knowledge that an information system needs to know in order to be able to derive some facts and, according to the 100% (or completeness) principle [18], that knowledge must be described in the conceptual schema. The importance of derivation rules is also recognized in industry, where they are sometimes classified as a kind of business rule [6].

Many conceptual modeling languages provide specific constructs for defining derived types and derivation rules. Among the early languages we mention SDM [16], which provides a detailed analysis of derivation kinds and the constructs supporting them; and CIAM [13], which puts a strong emphasis on derived types and the temporal perspective. Among the recent languages, we mention the family of languages descendants of KL-ONE [5], called Description Logics, which provide a specific formal syntax for defining derived entity and relationship types, and a strong reasoning support [4, 8]; Chimera [9], which includes deductive rules for data derivation, and a set of mappings for the implementation of these rules in commercial database systems (triggers, views, etc.); and ORM [14], which allows informal and formal [2] versions of derivation rules. Finally, we mention the standard language UML [24], which also allows defining derived types and derivation rules.

J. Eder and M. Missikoff (Eds.): CAiSE 2003, LNCS 2681, pp. 404–420, 2003.

The approaches taken by the above languages for defining derivation rules are diverse. This fact poses naturally the question of which is the best approach, if it exists. This paper tries to provide an answer to that question in the context of object-oriented languages [22] and, in particular, in the context of the UML.

We have analyzed the possible methods for defining derivation rules in object-oriented languages, and we recommend one of them as the probably most appropriate in many practical situations. The recommended method consists in defining a query function for each derived type. The derivation rule is then the postcondition assertion of the corresponding function. We adapt the method to the case of the UML. In this language, the query functions are the query operations, and their postconditions can be specified formally in the OCL language [24, ch. 6]. We show that the recommended method simplifies the languages.

Many derivation rules are static, in the sense that they derive facts that hold in a state of the information base from other facts holding in the same state. We believe that the above results can be adapted to temporal derivation rules, although this will not be explored in this paper. However, we have identified a special kind of derivation rule which lies in between the static and the temporal ones. We call them creation-time rules, because they are evaluated when objects are created. These rules derive constant relationship types, whose value is established at the time an object is created. We have found, surprisingly, that many relationship types are derived in this way. In the paper, we characterize these rules, show how they can be defined in object-oriented languages and, in particular, in the UML.

We study also the approaches for defining hybrid entity and relationship types. A type is hybrid when it is partially base and partially derived [23]. Hybrid types can always be transformed into derived ones [1, 15], and this may explain why they have received very little attention in the conceptual modeling literature (one notable exception is [20]). However, the transformation may not be always convenient. This paper proposes a method for defining hybrid types in object-oriented languages and, in particular, in the UML.

This paper is structured as follows. The next Section reviews the basic concepts, presents the method we propose for the definition of derived types and derivation rules in object-oriented languages, and suggests an adaptation to the UML. Section 3 extends the above method to creation-time rules. Section 4 deals with hybrid types, and presents the method we propose for their definition in object-oriented languages. Section 5 summarizes the conclusions and points out future research. Throughout the paper, we include examples taken from the EU-Rent (a car rental company) case study described in [6].

2 Language Constructs for Defining Derivation Rules

In this Section, we start by reviewing the concepts of derivability and derivation rules. Then, we propose a new method for the definition of derivation rules in O-O languages. We suggest also an adaptation of the method to the UML.

2.1 Derivability and Derivation Rules

The derivability of an entity or relationship type is the way how the information system knows its population at any instant [23]. According to derivability, a type may be base, derived or hybrid. We give below a few comments on each of them. We adopt here a logical and temporal view of the information base, and assume that entities and relationships are instances of their types at particular time points. In the information base, we represent by $E(e,t)$ the fact that e is instance of entity type E at time t, and by $R(e_1,\ldots,e_n,t)$ the fact that entities $e_1,\ldots,e_n$ participate in a relationship instance of R at time t. We denote by $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ the schema of a relationship type named R with entity type participants $E_1, \ldots, E_n$, playing roles $p_1, \ldots, p_n$, respectively. When the role name is omitted, it is assumed to be the same as the corresponding entity type. In the logical representation, attributes will be considered as ordinary binary relationship types.

An entity type is *base* when its population is given directly or indirectly by the users by means of insertion and deletion events. Similarly, a relationship type is base when its instances are given directly or indirectly by the users by means of insertion, update and deletion events.

An entity type E is *derived* when its population at time t can be obtained from the facts in the information base, using a derivation rule. The general form of the rule is[1]:

$$E\ (e,t) \leftrightarrow \phi\ (e,t)$$

The expression $\phi\ (e,t)$ defines the necessary and sufficient conditions for e to be an instance of E at t. The information system may know the population at time t by using the above rule. Derived entity types can be classified into several categories. One of them, that we will use later on, is specialization. We say that E is derived by *specialization* of entity types $E_1, \ldots, E_n$ when the population of E at t is a subset of the intersection of the populations of $E_1, \ldots, E_n$ at t [23].

Derived relationship types are defined similarly. A relationship type $R(p_1{:}E_1, \ldots, p_n{:}E_n)$ is derived when its instances at time t can be obtained from the facts in the information base, using a derivation rule. The general form of the rule is:

$$R\ (e_1,\ldots,e_n,t)\ \leftrightarrow \varphi(e_1,\ldots,e_n,t)$$

The information system may know the population at time t by using the above rule. We also say that R is derived by specialization of relationship types $R_1, \ldots, R_m$ when the instances of R at t are a subset of the intersection of the instances of $R_1, \ldots, R_m$ at t.

Derivation rules may be static or temporal. A static derivation rule derives a fact ($E(e,t)$ or $R(e_1,\ldots,e_n,t)$) holding at t from other facts holding at the same time t. A temporal derivation rule derives a fact holding at t from one or more facts that may hold at previous times t' ($t' \leq t$). In this paper, we deal almost exclusively with static derivation rules; the only exception is Section 3.

An entity or relationship type is *hybrid* when its population is given partially by the users (insertion, update and deletion events) and partially by a derivation rule. We deal with hybrid types in Section 4.

One of the derived entity types that may be defined in the case study is *CarToBeServiced*. Cars must be serviced every three months or 10,000 miles, whichever occurs first. In logic, its derivation rule could be:

$$\text{CarToBeServiced (car, t)} \leftrightarrow \text{Car (car,t)} \wedge \text{MileageNow (car,mNow,t)} \wedge$$

[1] The free variables are assumed to be universally quantified in the front of the formula.

LastService (car,dateLast,t) ∧ MileageLastService (car,mLast,t) ∧
(t – dateLast > 3Months ∨ mNow – mLast > 10000m)

Note that *CarToBeServiced* is a specialization of *Car*. On the other hand, the above rule is static because the cars to be serviced at *t* can be determined from the existing cars, and their properties, at time *t*.

An example of derived relationship type is: CarRentalRate (Car, Money). The rental rate of a car is defined as the rate for the group that car's model belongs to. Thus, the derivation rule in logic is:

CarRentalRate (car,rentRate,t) ↔
Car (car,t) ∧ ModelOfCar (car,cModel,t) ∧
CarGroupOfModel (cModel,cGroup,t) ∧
GroupRentalRate (cGroup,rentRate,t)

Note that this rule is also static.

2.2 Defining Derivation Rules in Object-Oriented Languages

Now, we focus on how to define derivation rules in O-O conceptual modeling languages. In general, the elements of the structural parts of these languages are entity types, attributes and relationship types. These elements may be base or derived.

Structurally, we propose to define the derived elements like the base ones, with the addition of some mark to indicate that they are derived. In this way, derived elements can (and must) be referred to in any part of a conceptual schema (constraints, derivation rules, etc.) like the base ones.

We propose to define derivation rules by means of operations. All O-O languages and data models include the concept of operation [22, 7]. Our operations will be query operations (also called query functions), which return a value but do not alter the information base.

The key point of our method consists in associating each derived element with a query operation, called the *defining* operation. The only purpose of a defining operation is to specify the corresponding derivation rule. The defining operations are purely conceptual; they may or may not be part of the implementation. The defining operations are not visible to the rest of the schema and, therefore, they should not be invoked in any part of the schema.

The signature of a defining operation depends on the derived element. For a derived entity type *E*, the defining operation is a class operation of *E* without arguments:

population (): Set (E)

The intended result of the operation is the set of entities which are instance of *E*. The exact name of the operation (*population*) must be decided in the adaptation of the method to a particular language; normally it will be a predefined name.

In the case study, the defining operation corresponding to entity type *CarToBeServiced* would be a class operation of that entity type with signature:

population () : Set (CarToBeServiced)

For an attribute *A* of entity type *E* with values of type E_1, the defining operation is an instance operation in *E* without arguments:

op_A (): [E_1 | Set (E_1)]

where the result is E_1 or *Set* (E_1) depending on whether A is single valued or multivalued, respectively. The result of the operation is the value of attribute A for the corresponding instance entity. The exact name of the operation (*op_A*) must be decided in the adaptation of the method; normally it will be just A or based on A.

In the case study, the derived relationship type *CarRentalRate* would probably be defined as a single valued attribute *rentalRate* of *Car*. The defining operation is an instance operation of *Car* with signature:

rentalRate (): Money

For a binary relationship type $R(p_1:E_1, p_2:E_2)$ the defining operation may take one of the four following forms (the exact name of the operation must be decided in the adaptation of the method):

- Instance operation of E_1 with signature op_p_2 (): [E_2 | *Set* (E_2)]. The result is E_2 or *Set* (E_2) depending on the cardinalities of R. The result of the operation is the instance, or the set of instances, of E_2 related to an instance of E_1.
- Instance operation of E_2 with signature op_p_1 (): [E_1 | *Set* (E_1)]. Similar to the above.
- Instance operation of E_1 with signature op_p_2 (e_2:E_2): *Boolean*. The result is *true* if there is a relationship between *self* and the instance given in the parameter, and *false* otherwise.
- Instance operation of E_2 with signature op_p_1 (e_1:E_1): *Boolean*. Similar to the above.

From a conceptual point of view, the four options are equivalent. The place (E_1 or E_2) where the operation is defined does not imply any navigability. Conceptually, relationship types are navigable in all directions. The designer may choose the place (s)he thinks is more natural or easier to specify. A factor that may influence the decision is the possibility of redefinition, which we explain below.

In the case study, one of the derived relationship types is:

AvailableNextDay (branchWhereAvailable:Branch, carAvailable:Car)

A car is available to a branch the next day if it is owned by that branch and currently it is in the parking lot, or it is due today from rental to that branch. In general, a branch has several cars available next day, but a car may be available next day at most to one branch. The defining operation could be defined in *Branch* with one of the signatures:

carAvailable (): Set (Car)
carAvailable (c:Car): Boolean

Or, alternatively, defined in *Car* with one of the signatures:

branchWhereAvailable (): Branch
branchWhereAvailable (b:Branch): Boolean

We proceed similarly for an n-ary relationship type $R(p_1:E_1,\ldots, p_n:E_n)$, with $n > 2$. Now we have a choice of $n(n-1)$ defining operations, each of which:

- is an instance operation of some E_i, $i \in \{1,..,n\}$,
- has $n - 2$ arguments, and
- has a return result of type E_j, $j \neq i$, $j \in \{1,..,n\}$

For $i = 1$ and $j = 2$, the general form of the instance operation is:

op_p_2 (p_3:E_3, …, p_n:E_n): [E_2| Set (E_2)]

where again the result is E_2 or *Set* (E_2) depending on the cardinalities of R. The result of the operation is the set of instances of E_2 related to an instance of E_1 and to the instances of E_3, …, E_n given in the arguments. The exact name of the operation

(*op_p2*) must be decided in the adaptation of the method. Normally, if one of the participants is a data type, we would not define the operation in it.

In the case study, we have the derived ternary relationship type:

CarsAvailableNextDay
(branchWhereAvailable:Branch, CarGroup, numberOfCars: Natural)

It is defined as the number of cars of a given car group available the next day to a given branch. In this case, it seems natural to define the defining operation in *Branch*, with signature:

numberOfCarsAvailableNextDay (cg: CarGroup): Natural

The above operations are specified formally in the style and the (sub)language most appropriate to the corresponding O-O language. In general, the preferred style could be the use of postconditions. For query operations, postconditions specify the value of the result. Note that postconditions of query operations do no have the frame problem [3].

In all O-O languages, instance operations may be redefined in subtypes. This applies also to the query operations we have defined above for the derivation rules of attributes and relationship types. The important implication of this is that, in our method, derivation rules of attributes and relationship types can be redefined.

In the case study, we have an example of redefinition in attribute:

TotalCost (ReturnedRental, amount:Money)

This attribute gives the amount to be paid by a customer when a rental is returned. That amount is defined as the sum of the insurance and rental amounts. In an O-O language, the defining operation of this attribute would be an operation in *ReturnedRental*, with signature:

totalCost (): Money

However, for *LateRental*, a subtype of *ReturnedRental* corresponding to the rentals returned after their due time, the total cost includes a late charge. We then redefine the defining operation:

totalCost (): Money

in *LateRental*. The specification of this operation will define the derivation rule of total cost for the particular case of a late rental.

2.3 Adaptation to the UML

The above method is easily adaptable to any O-O conceptual modeling language. The main decisions to be made are:

- How to specify the defining operations.
- The language of the above specification.
- How to relate a derived element with its defining operation.

We suggest here an adaptation to the UML. We have chosen to specify the defining operations by means of postconditions, using the OCL language. The relationship between a derived element and its defining operation is basically by naming conventions. We illustrate the adaptation with some examples from the EU-Rent case study.

In the UML, derived elements are marked with a slash (/) placed in front of their name. Figure 1 shows five derived elements: the entity type *CarToBeServiced*, the

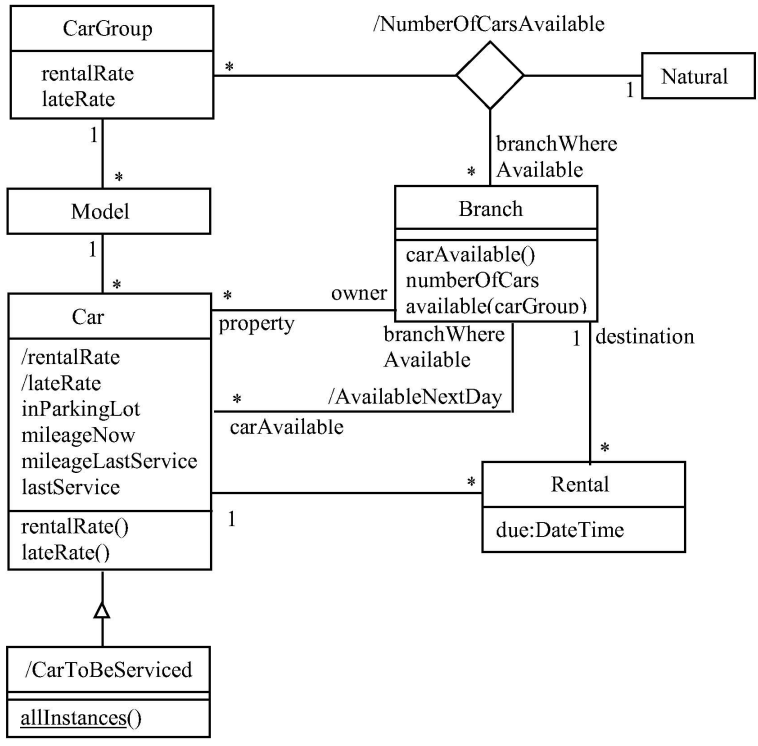

Fig. 1. Fragment of the EU-Rent case study with derived types, in the UML.

attributes *RentalRate* and *lateRate* of *Car*, the binary association (relationship type) *AvailableNextDay* and the ternary association *NumberOfCarsAvailable*.

Figure 1 shows also the defining operations corresponding to the five derived elements. Normally, these operations would not appear in the class diagrams of conceptual schemas, because they cannot be invoked. We have included them here only for illustration purposes.

In the UML, operations have, among others, the boolean attribute *isQuery* [24, p.2-27]. The value of this attribute for our defining operations is *true* (indicating that the operation leaves the state unchanged).

The OCL includes, for each entity type *E*, the predefined class operation of *E*:

```
allInstances () : Set (E)
```

which gives the set of all instances of *E*, and all its subtypes, at the time the operation is evaluated [24, p.6-30]. This operation must not be specified by the designer, because its meaning is predefined. We propose, however, to use this same operation as the defining operation of derived entity types. In these cases, the designer should specify the postcondition of the operation, using the OCL.

In the case study, the OCL specification of the defining operation of *CarToBeServiced* is:

```
context CarToBeServiced :: allInstances ():
                                    Set(CarToBeServiced)
```

```
    post result = car.allInstances() -> select
                (mileageNow – mileageLastService > 10000 or
                 Now – lastService > 3Months)
```

In the postcondition, we have used a special variable, *Now*, which we assume gives the current time. In Figure 1, the name of the operation *allInstances* is underlined to indicate that it is a class operation. For space reasons we do not show the return type in the diagram.

Note that in the above example we have used the expression `car.allInstances()` and, therefore, we have invoked the operation *allInstances*. This is the only way we have in the OCL to know the population of an entity type. This operation is the only exception to the general rule that the defining operations should not be invoked in the OCL expressions corresponding to constraints or derivation rules (postconditions).

For attributes, the name of the defining operation is the same as the name of the attribute. The specification of the defining operation corresponding to attribute *rentalRate* is then:

```
context Car :: rentalRate () : Money
  post result = model.carGroup.rentalRate
```

A similar example is attribute *lateRate*, which gives the rate to be paid by rentals returned late. It is defined similarly to the *rentalRate*. The specification of its defining operation is:

```
context Car :: lateRate () : Money
  post result = model.carGroup.lateRate
```

For binary relationship types, we propose that the name of their defining operation be the same as the name of the corresponding role. In the example of Figure 1, a possible specification of the defining operation in *Branch* corresponding to association *AvailableNextDay* could be:

```
context Branch :: carAvailable () : Set (Car)
  post result = property -> select (inParkingLot = True)
              -> union (rental -> select (due = Today).car)
```

It is difficult to establish a general convention regarding the name of the defining operation for *n*-ary associations. It may be preferable to let the designer give the name (s)he thinks is more adequate. In such a case, it will be necessary to document with a note or otherwise that the operation is a defining operation of a derived association.

In the example of ternary association *NumberOfCarsAvailable*, Figure 1, we have defined an operation with the same name in *Branch*. The OCL specification would be:

```
context Branch ::
  numberOfCarsAvailable (inCarGroup : CarGroup) : Natural
  post result = carAvailable ->
              select (c: Car | c.model.carGroup =
              inCarGroup) -> size ()
```

Note the reference to *carAvailable* in this postcondition. This is an example of a derived association used like a base one in an OCL expression.

Figure 2 provides an example of derivation rule redefinition. Attribute *totalCost* of a returned rental is defined as:

```
context ReturnedRental :: totalCost () : Money
  post result = insuranceAmount + rentalAmount
```

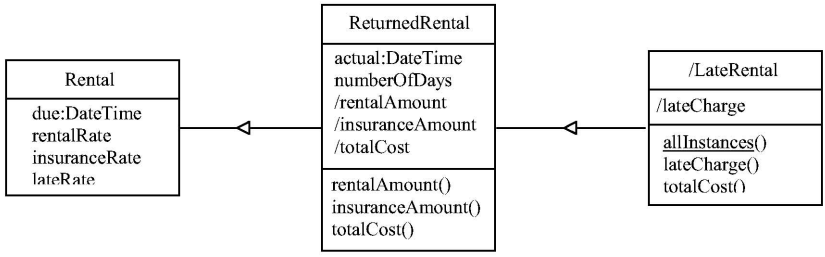

Fig. 2. Fragment of the EU-Rent case study with derived types, in the UML. (*Rental* is extended in Figure 3)

but this derivation rule is redefined in *LateRental*. This is shown by redefining the operation *totalCost*() in this type. Here, the derivation rule could be:

```
context LateRental :: totalCost () : Money
  post result = insuranceAmount + RentalAmount +
                lateCharge
```

Note that *LateRental* is a derived entity type. The specification of its defining operation could be:

```
context LateRental :: allInstances () : Set(LateRental)
  post result = ReturnedRental.allInstances() ->
                                        select(actual > due)
```

Therefore, *LateRental* is a specialization of *ReturnedRental*.

2.4 Comparison with Other Methods

We now compare our method with the methods used by other languages. Given that the focus of this paper is on the definition of derivation rules in O-O languages, here we only compare with languages that may be considered more or less object-oriented and that include, in particular, the concept of redefinable (query) operation, or similar. This explains why we do not include in the comparison other conceptual modeling languages such as Telos [9], Syntropy [10], Description Logics [8], or ORM [14].

We start the comparison with the UML. In this language, derivation rules are represented by abstraction dependencies with the stereotype <<derive>>. A dependency relates the derived element (the client) with the elements (the suppliers) from which it can be computed. The details of the computation are specified by a mapping expression written in some language, possibly the OCL [24, p. 2-19]. Graphically, the dependency is shown as a dashed arrow between those elements [24, p. 3-90+].

We believe that our method has two main advantages over the UML method. The first is the economy. We draw on a standard construct already provided by the language (operation), instead of requiring a new, specific construct (dependencies with the stereotype <<derive>>). Our approach makes the language and its supporting tools simpler, which is important in many respects. In particular, in the framework of the Model Driven Architecture (MDA) [25], our approach eases the development of mappings from conceptual schemas to platform specific models, because now derivation rules become just a particular case of operations.

The second advantage is obtained by the use of the standard mechanism of operation redefinition, with its well defined semantics. As we have seen, this mechanism may simplify the definition of some complex derivation rules. Redefinition does not apply to UML dependencies.

In Chimera [9, 12], the derived elements may be entity types, attributes and relationship types (views). In the three cases, the corresponding derivation rules are defined by means of deductive rules. The derivation rules of attributes may be redefined, but not those of relationship types. On the other hand, entity types may have operations, which in particular can be accessor operations. The difference between derived attributes and accessor operations is very small [9, p. 199]. In the first case, the attribute can be referred to as any other (base) attribute; in the latter, it can be referred to by means of an explicit operation invocation.

Our method can be adapted to the Chimera. We would associate to each derived element an accessor operation with some predefined name. In this way, the deductive rules for data derivation would not be needed, thus making the language simpler. On the other hand, using our method the derivation rules of relationship types could be redefined.

Finally, we point out the similarity between our method and the one used by the language for deductive object-oriented databases proposed recently in [19]. In this language, derived information is defined by a method type and a method rule. Roughly, a method type defines an attribute or a relationship type. A method rule corresponds to a query operation, although defined as a deductive rule. Methods are declared within class declarations, and the methods are invoked trough instances of the classes. Like in our proposal, method rules may be redefined in subclasses.

3 Derivation Rules for Constant Relationship Types

The method described in the previous section can be extended easily to deal with a frequently used particular class of derived relationship types. These are the constant relationship types whose instances can be derived when the instances of one of its participant entity types are created. In this section, we characterize this class of relationship types and explain how they can be defined in our method.

A relationship type R (p_1:E_1, …, p_i:E_i, …, p_n:E_n) is constant with respect to p_i if the set of instances of R in which participates an instance e_i of E_i is determined when e_i is created, and remains fixed during its lifetime [11]. For example,

RentalRate (Rental, Money)

is constant with respect to *Rental*, because the rate to be paid by a rental is determined at creation time (when a rental is created), and cannot be changed later.

All the attributes shown in Figure 2 are constant relationship types with respect to *Rental*, *ReturnedRental* or *LateRental*. The only exception is attribute *due* of *Rental*, because rentals can be extended.

Formally, R (p_1:E_1, …, p_i:E_i, …, p_n:E_n) is constant with respect to p_i if:

$$R(e_1,\ldots,e_i,\ldots,e_n,t) \rightarrow \forall t'\ (E_i(e_i,t') \rightarrow R(e_1,\ldots,e_i,\ldots,e_n,t'))$$

That is, if R (e_1,…,e_i,…,e_n,t) holds at time t then it must also hold at all times t' at which e_i is instance of E_i. The adaptation to the example is:

$$\text{RentalRate}(r,\text{rentRate},t) \rightarrow \forall t'\ (\text{Rental}(r,t') \rightarrow \text{RentalRate}(r,\text{rentRate},t'))$$

Like any other, constant relationship types can be derived, and their derivation rule can be static or temporal. An example with a static rule is (Figure 2):

InsuranceAmount (ReturnedRental,Money)

which is defined as the insurance rate of the rental multiplied by the number of days of the rental. In logic:

InsuranceAmount (rr,insAmt,t) ↔
ReturnedRental (rr,t) ∧ InsuranceRate (rr,iRate,t) ∧
NumberOfDays (rr,nd,t) ∧ insAmt = nd * iRate

There are some derived constant relationship types whose derivation rule is temporal and, in principle, they cannot be defined in non-temporal languages. However, in many cases these rules have a particular form that allows their definition, even in non-temporal languages. We call them *creation-time* rules. An example is:

RentalRate (Rental,Money)

The rate of a rental is the rental rate of its car at the time *when the rental is created*. The following static derivation rule would be incorrect:

RentalRate (r,rentRate,t) ↔ Rental (r,t) ∧
CarOfRental (r,rentedCar,t) ∧ CarRentalRate (rentedCar,rentRate,t)

This rule is incorrect because the rental rate of the car *rentedCar* can change after the rental is created, but this does not imply that the rental rates of the previous rentals of that car must be changed. The correct form of the rule must define that the rental rate is determined from the rental rate of the car when the rental is created. Assuming a predicate *CreatedAt* (*Rental*,*Time*) that gives the instant when a rental is created, the rule would then be:

RentalRate (r,rentRate,t) ↔ Rental (r,t) ∧ CreatedAt (r,t_0) ∧
CarOfRental (r,rentedCar,t_0) ∧ CarRentalRate (rentedCar,rentRate,t_0)

Let R (p_1:E_1, …, p_i:E_i, …, p_n:E_n) be constant with respect to p_i. We say that R is derived with a creation-time rule if its derivation rule has the form:

$$R(e_1,\ldots,e_i,\ldots,e_n,t) \leftrightarrow E_i(e_i,t) \wedge \text{CreatedAt}(e_i,t_0) \wedge \varphi(e_1,\ldots,e_n,t_0)$$

where the literals in $\varphi(e_1,\ldots,e_n,t_0)$ can be evaluated at time t_0 with facts holding at that time. In the example above, we have:

$\varphi(e_1,\ldots,e_n,t_0) \equiv$
CarOfRental (r,rentedCar,t_0) ∧ CarRentalRate (rentedCar,rentRate,t_0)

It can be shown that all derived constant relationship types whose derivation rule is static, can be defined as derived with a creation-time rule.

Now, we focus on how to define creation-time rules in O-O languages. Most non-temporal O-O languages use a concept, *object creation*, which we will use too to define creation-time derivation rules. That concept is used, among other things, to define the initial value for attributes, which is the value an attribute takes when an object is created, if no explicit value is given. In the UML, such initial value is, in general, an expression that is meant to be evaluated at the time the object is initialized [24, p. 2–26].

There is an analogy between initial values for attributes and creation-time derivation rules, because both are meant to be evaluated when an object is created. The analogy justifies the method we propose to define creation-time rules. If a relationship type R (p_1:E_1, …, p_i:E_i, …, p_n:E_n) is constant with respect to p_i, and is derived by a creation-time derivation rule, then we define this rule as an operation of

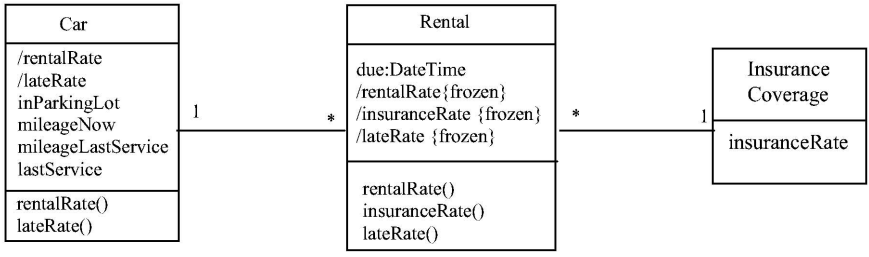

Fig. 3. Attributes *rentalRate*, *insuranceRate* and *lateRate* of *Rental* are derived with a creation-time derivation rule.

E_i, with the understanding that the rule is meant to be evaluated when the instances of E_i are created. In the example of *RentalRate*, we would define the operation in *Rental*.

Figure 3 illustrates the adaptation of our method to the UML with three examples. In this language, a constant attribute is defined with the keyword *frozen*. The same keyword is used for constant binary associations. The figure defines that *rentalRate*, *insuranceRate* and *lateRate* are constant attributes of *Rental*. The three attributes are derived. The creation-time rule for *rentalRate* is:

```
context Rental :: rentalRate () : Money
  post result = car.rentalRate
```

The rule for *lateRate* is similar:

```
context Rental :: lateRate () : Money
  post result = car.lateRate
```

The insurance rate of a rental is given by the insurance coverage of that rental. The rule, then is:

```
context Rental :: insuranceRate () : Money
  post result = insuranceCoverage.insuranceRate
```

4 Hybrid Types

In this section, we review the concept of hybrid entity and relationship types, and then we propose a method for their definition in O-O languages. We suggest also an adaptation of the method to the UML.

4.1 Definition

An entity type *E* is *hybrid* when its population is given partially by the users (insertion and deletion events) and partially by a derivation rule, which has the general form:

$$E\ (e,t) \leftarrow \phi\ (e,t)$$

Such derivation rules are partial because they define only part of the population of *E*. A hybrid entity type has an extensional part (given by the users) and an intensional one (given by the derivation rule). We will assume that these parts are disjoint, as is normally the case.

In the case study, the conceptual schema includes entity types *Car* and *ExternalCar*. An external car is a car a branch rents from a competitor to satisfy a

demand that cannot be satisfied with the available own cars. In this case, *ExternalCar* is base and *Car* is hybrid. All external cars are cars, but there are other cars (own cars). The partial derivation rule is:

Car (c,t) ← ExternalCar (c,t)

Any hybrid type *E* can be transformed easily into an equivalent derived one. The procedure for the transformation is described in [1, 15]. The idea is to define a new base entity type E_{base}, such that its instances are the extensional part of *E*. Then, *E* becomes derived, with the derivation rule:

$$E\ (e,t) \leftrightarrow \phi\ (e,t) \vee E_{base}\ (e,t)$$

In the example, we would have:

$$Car\ (c,t) \leftrightarrow ExternalCar\ (c,t) \vee Car_{base}\ (e,t)$$

Similarly, a relationship type *R* is *hybrid* when its population is given partially by the users (insertion, update and deletion events) and partially by a derivation rule, which has the general form:

$$R\ (e_1,...,e_n,t) \leftarrow \varphi\ (e_1,...,e_n,t)$$

As before, any hybrid relationship type can be transformed into an equivalent derived one.

In the case study, when a customer picks up a car (s)he must sign a contract and declare the additional drivers, if any. The drivers of a car are the signer of the contract and the declared additional drivers. Thus, we could have the relationship types:

SignerOfContract (Rental, Customer)

Driver (Rental, Customer)

In this case, *Driver* is hybrid, with the partial derivation rule:

Driver (r,cust,t) ← SignerOfContract (r,cust,t)

4.2 Defining Hybrid Types in Object-Oriented Languages

Most O-O languages (including the UML) ignore the hybrid types. The obvious solution is, then, to transform these types into their derived equivalents. Unfortunately, as we have seen, this requires the definition of a new type (E_{base} or R_{base}), which in some cases may be considered artificial.

However, we have found that the *IsA* construct (generalization or specialization links), present in all O-O languages, allows a simple method for the definition of hybrid types.

In the method we propose, we require that the partial derivation rule of an entity type *E* have the general form ($n \geq 1$):

$$E\ (e,t) \leftarrow E_1\ (e,t) \vee ... \vee E_n\ (e,t)$$

That is, the intensional population of *E* is given by the union of the populations of E_1, ..., E_n. We call these types the intensional types of *E*. To avoid circularity, an intensional type of *E* cannot be derived by specialization of *E*. The example above follows this form, with $n = 1$ and $E_1 = ExternalCar$.

All partial derivation rules ($E\ (e,t) \leftarrow \phi\ (e,t)$) can be transformed into this form by defining, if needed, a new derived entity type E_1 with derivation rule ($E_1\ (e,t) \leftrightarrow \phi\ (e,t)$), and then we have ($E\ (e,t) \leftarrow E_1\ (e,t)$). However, our method provides the best results when the original derivation rule has already the form:

$$\phi\ (e,t) \equiv E_1\ (e,t) \vee ... \vee E_n\ (e,t)$$

so that no transformation is needed.

We then propose to define a hybrid entity type E with intensional entity types E_1, …, E_n by:

- defining the n *IsA* relationships: E_1 *IsA* E, …, E_n *IsA* E.
- defining that E_1,…, E_n are the intensional types of E. The exact form of this definition will depend on the particular O-O language.

Note that we use the semantics of *IsA* to define implicitly that:

$$E\ (e,t) \leftarrow E_1\ (e,t) \vee ... \vee E_n\ (e,t)$$

In the example, we would just define:

ExternalCar IsA Car

and that *ExternalCar* is the intensional type of *Car*.

Similarly, we propose to define a hybrid relationship type R with intensional relationship types R_1, …, R_n by:

- defining the n *IsA* relationships: R_1 *IsA* R, …, R_n *IsA* R. We require that the language allows defining *IsA* relationships between relationship types.
- defining that R_1,…, R_n are the intensional types of R. The exact form of this definition will depend on the particular O-O language.

In the example, we would just define:

SignerOfContract IsA Driver

and that *SignerOfContract* is the intensional type of *Driver*.

If the O-O language allows *IsA* relationships between attributes, then hybrid attributes can be defined in the same way as hybrid relationship types. If this is not the case, then the attribute must be transformed first into an equivalent relationship type.

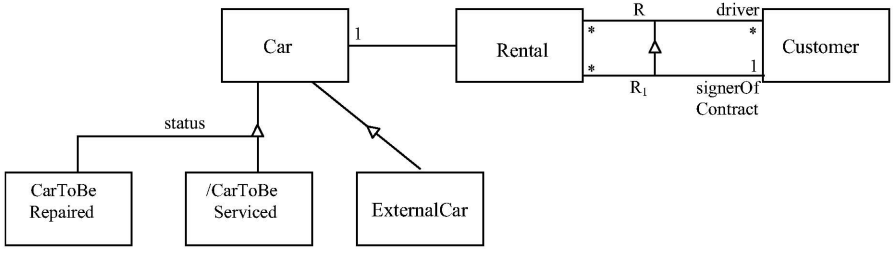

Fig. 4. Examples of hybrid entity (*Car*) and relationship (*R*) types.

4.3 Adaptation to the UML

The above method is easily adaptable to any O-O conceptual modeling language. We suggest here a possible adaptation to the UML.

The adaptation of the above definition of hybrid entity types is almost trivial. The only problem is how to define that E_1, …, E_n are the intensional entity types of E. We assume multiple classification and, therefore, in the general case E may be also a supertype of other types.

The solution we propose is to use the discriminator attribute that UML defines in generalization links [24, p. 2-45]. The discriminator is a string (including the empty string) that allows grouping a set of generalizations with the same supertype. In the general case, the same entity type may be the supertype of several sets of

generalizations. We propose then to use a particular name as the discriminator that groups the generalizations links of the intensional entity types. Here we will use the empty string. Naturally, other names are possible.

Therefore, in our method an entity type E is hybrid with intensional types E_1, …, E_n if:

- E is not defined as derived.
- E_1, …, E_n are not derived by specialization of E.
- E_1, …, E_n are subtypes of E with the empty discriminator.

Figure 4 shows the example in the UML. *Car* is hybrid with intensional type *ExternalCar* because:

- *Car* is not defined as derived.
- *ExternalCar* is not derived by specialization of *Car*.
- *ExternalCar* is subtype of *Car* with the empty discriminator.

CarToBeRepaired and *CarToBeServiced* are not intensional types of *Car* because their generalization links have discriminator *status*. On the other hand, *CarToBeServiced* is derived by specialization of *Car*. In this example, we assume that external cars may also be cars to be repaired and/or serviced.

Note that the set of generalization links defining the intensional types of E cannot be subjected to the constraint {complete}[24, p. 2-45] because then E would not be hybrid, but derived. That set must have the constraint {incomplete}, which we assume as default.

The same adaptation applies to hybrid relationship types. UML allows generalization links between relationship types (associations). Figure 4 shows also an example of this case. R is hybrid with intensional type R_1 because:

- R is not defined as derived.
- R_1 is not derived by specialization of R.
- R_1 is subtype of R with the empty discriminator.

The UML does not allow *IsA* relationships between attributes. Therefore, we must transform first the hybrid attributes into hybrid associations and then proceed as in the previous case.

It is interesting to observe that, in Figure 4, the generalization between *Car* and *ExternalCar* is not a taxonomic constraint that must be enforced at run-time [23]. According to the usual interpretation, the constraint is automatically enforced by the schema. Analogously, the same interpretation applies to the generalization between R and R_1.

4.4 Comparison with Other Approaches

To the best of our knowledge, only [20, p. 57] has made a proposal for the definition of hybrid relationship types in conceptual schemas. The proposal consists in using a special graphical symbol to denote a hybrid relationship type, and to attach to it the partial derivation rule (written in some unspecified language).

The main advantages of our method are that we do not require a new symbol, we draw on the semantics of the *IsA* relationship to define the intensional population of a hybrid type, and we do not need to write the partial derivation rule. On the other hand, we deal also with hybrid entity types.

5 Conclusions

We have proposed three related methods for the definition of derivation rules in O-O conceptual modeling languages. The first method applies to static rules, and associates each derived element with a defining operation. The specification of this operation is then the definition of the corresponding derivation rule. The second method applies to constant relationship types whose instances can be derived when the instances of one of its participant entity types are created. We have seen that the derivation rules of these types have usually a common form, that we have called creation-time rule. We have described a variant of the previous method to deal with these rules. The third method deals with hybrid types, and suggests a simple way to define their partial derivation rules.

The three methods have been formalized in logic and described independently of any particular O-O language. We have proposed also adaptations of the three methods to the UML. The methods are fully compatible with the UML-based CASE tools, and thus they can be adopted in industrial projects, if it is felt appropriate.

We hope that our methods will ease the definition of derivation rules. The need of including derived types and their (formal) derivation rules in conceptual schemas has always been considered important, but nowadays, in the framework of the emerging Model Driven Architecture, it is becoming mandatory. On the other hand, we hope that our methods will ease the implementation of derivation rules.

Our work can be continued in several directions. We mention two of them here. The first is the adaptation of our methods to other languages. The second direction is to develop mappings for transforming derived types and our derivation rules to the appropriate constructs of other platform independent and/or platform specific models. The mappings should take into account whether derived types are materialized or computed when needed and, in many aspects, could draw upon the available knowledge [9, 27].

Acknowledgements. I would like to thank Jordi Cabot, Dolors Costal, Cristina Gómez, Maria Ribera Sancho and Ernest Teniente for their many useful comments to previous drafts of this paper. This work has been partially supported by the Ministerio de Ciencia y Tecnologia and FEDER under project TIC2002-00744.

References

1. Bancilhon, F.; Ramakrishnan, R. “An Amateur’s Introduction to Recursive Query Processing Strategies”. Proc. ACM SIGMOD Int. Conf. On Management of Data, May 1986, pp. 16–52.
2. Bloesch, A.C.; Halpin, T.A. “Conceptual Queries Using ConQuer-II”. Proc. ER’97, LNCS 1331, pp. 113–126.
3. Borgida, A.; Mylopoulos, J.; Reiter, R. "...And Nothing Else Changes: The Frame Problem in Procedure Specifications”. Proc. ICSE 1993, pp. 303–314.
4. Borgida, A. “Description Logics in Data Management”. IEEE Transactions on Knowledge and Data Engineering, 7(5), 1995, pp. 671–682.
5. Brachman, R.J.; Schmolze, J.G. “An Overview of the KL-ONE Knowledge Representation System”. Cognitive Science 9, 1985, pp. 171–216.

6. Business Rules Group. “Defining Business Rules – What Are They Really?”. Final Report, July 2000, http://www.businessrulesgroup.org/first_paper/br01c0.htm
7. Cattell, R.G.G.; Barry, D.K. (Eds) The Object Data Standard: ODMG 3.0. Morgan Kaufmann Pubs., 280 p.
8. Calvanese, D.; Lenzerini, M.; Nardi, D. “Description Logics for Conceptual Data Modeling”. In Chomicki, J.; Saake, G. (eds). Logics for Databases and Information Systems, Kluwer, 1998, pp. 229–263.
9. Ceri, S.; Fraternali, P. Designing Database Applications with Objects and Rules. The IDEA Methodology. Addison-Wesley, 1997, 579 p.
10. Cook, S.; Daniels, J. Designing Object Systems. Object-Oriented Modelling with Syntropy. Prentice Hall, 389 p.
11. Costal, D.; Olivé, A.; Sancho, M-R. “Temporal Features of Class Populations and Attributes in Conceptual Models”. Proc. ER 1997, LNCS 1331, pp. 57–70
12. Guerrini, G.; Bertino, E.; Bal, R. “A Formal Definition of the Chimera Object-Oriented Data Model”. Journal of Intelligent Information Systems, 11(1), pp. 5–40.
13. Gustaffsson, M.R.; Karlsson, T.; Bubenko, J.A. jr. “A Declarative Approach to Conceptual Information Modelling”. In Olle, T.W.; Sol, H.G.; Verrijn-Stuart, A.A. (eds.) Information systems design methodologies: A Comparative Review. North-Holland, 1982, pp. 93–142.
14. Halpin, T. Information Modeling and Relational Databases. From Conceptual Analysis to Logical Design. Morgan Kaufmann Pub., 2001, 761 p.
15. Halpin, T. “UML Data Models from an ORM Perspective (Part 9)”. Journal of Conceptual Modeling, June 1999, Issue 9.
16. Hammer, M.; McLeod, D. “Database Description with SDM: A Semantic Database Model”, ACM TODS, 6(3), 1981, pp. 351–386.
17. Hull, R.; King, R. “Semantic Database Modeling: Survey, Applications, and Research Issues”, ACM Computing Surveys, 19(3), 1987, pp. 201–260.
18. ISO/TC97/SC5/WG3. “Concepts and Terminology for the Conceptual Schema and the Information Base”, J.J. van Griethuysen (ed.), March 1982.
19. Liu, M.; Dobbie, G; Ling, T.W. “A Logical Foundation for Deductive Object-Oriented Databases”, ACM TODS 27(1), pp. 117–151.
20. Martin, J. Odell, J. Object-Oriented Methods: A Foundation. Prentice Hall, 1995, 412p.
21. Mylopoulos, J.; Borgida, A.; Jarke, M.; Koubarakis, M. “Telos: a language for representing knowledge about information systems”. ACM TOIS, 8(4), pp. 327–362.
22. Meyer, B. Object-Oriented Software Construction. Prentice Hall, 1997, 1254 p.
23. Olivé, A. Teniente, E. “Derived types and taxonomic constraints in conceptual modeling”, Information Systems 27 (2002), pp. 391–409.
24. OMG. “Unified Modeling Language Specification”, Version 1.4, September 2001, http://www.omg.org/technology/documents/formal/uml.htm.
25. OMG. “Model Driven Architecture (MDA)”, Doc. number ORMSC/2001-07-01, http://cgi.omg.org/docs/ormsc/01-07-01.pdf
26. OMG. “Meta-Object Facility ((MOF™))”. Version 1.4, April 2002, http://www.omg.org/technology/documents/formal/mof.htm
27. Winter, R. “Design and implementation of derivation rules in information systems”. Data & Knowledge Eng., 26, 1998, pp. 225–241.