Taxonomies and Derivation Rules in Conceptual Modeling

Antoni Olivé

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Jordi Girona 1-3, C5-D207
08034 Barcelona (Catalonia)
olive@lsi.upc.es

Abstract. This paper analyzes the relationships between taxonomic constraints and derivation rules. The objectives are to see which taxonomic constraints are entailed by derivation rules and to analyze how taxonomic constraints can be satisfied in presence of derived types. We classify derived entity types into several classes. The classification reveals the taxonomic constraints entailed in each case. These constraints must be base constraints (defined in the taxonomy) or be derivable from them. We show how the base taxonomic constraints can be satisfied, either by the derivation rules (or the whole schema), or by enforcement. Our results are general and could be incorporated into many conceptual modeling environments and tools. The expected benefits are an improvement in the verification of the consistency between taxonomic constraints and derivation rules, and a guide for the determination of the taxonomic constraints that must be enforced in the final system.

1 Introduction

Taxonomies are fundamental structures used in many areas of information systems engineering and other fields [8]. In its most basic form, a taxonomy consists of a set of (entity or relationship) types and a set of *IsA* relations among them [11]. Extensionally, an *IsA* relation is a constraint between the populations of two types [3]. Usually, taxonomies include also other constraints, mainly disjointness and covering, which are needed to adequately represent domain knowledge [24].

Derived types and derivation rules have a long tradition in conceptual modeling, starting at least in the early eighties [22]. SDM [20] is recognized as one of the first languages that emphasized the need and support of derived types. Derivation rules were also included in the ISO framework [23]. CIAM was a methodology strongly based on derived types, with a temporal perspective [17]. Many other later conceptual languages include specific constructs for the definition of derivation rules. Among them, we mention the family of languages descendants of KL-ONE [11] called Description (or terminological) Logics [9]. The recent industry standard UML language [31] also allows derived types but, unfortunately, they are restricted to attributes and associations.

K.R. Dittrich, A. Geppert, M.C. Norrie (Eds.): CAiSE 2001, LNCS 2068, pp. 417–432, 2001.

There are some relationships between constraints defined in a taxonomy, that we call taxonomic constraints, and derived types. As a very simple example, assume a taxonomy with entity types *Clerk*, *Engineer*, *Employee* and *Person*, and the constraint *Clerk IsA Employee*. Once defined a taxonomy, in general we have several options concerning derivability of its entity types, and the corresponding derivation rules. For example, we can make *Employee* base and *Clerk* derived. In this case, the derivation rule of *Clerk* must entail the constraint, with a rule like "A clerk is an employee such that its category is 'c'". Another option is to make *Employee* derived and *Clerk* base. In this case, the derivation rule of *Employee* may or may not entail the constraint. If the rule is like "An employee is a person that works in some company", then it does not entail the constraint. If the rule is "An employee is a person who is a clerk or an engineer" then it entails the constraint. However, in this case the rule entails also *Engineer IsA Employee*, and this constraint must be included in the taxonomy. Thus, we see that there are relationships from taxonomic constraints to derivation rules, and the other way around. The objective of this paper is to analyze such relationships at the conceptual level.

Knowledge of the relationships between taxonomic constraints and derivation rules is important for at least two purposes: (1) During conceptual modeling verification, to ensure that derivation rules entail some constraints defined in the taxonomy, and that the taxonomy includes all constraints entailed by derivation rules; and (2) During information systems design, to determine which taxonomic constraints need to be explicitly enforced (by database checks, assertions or triggers, or transaction pre/postconditions) and which are entailed by derivation rules, and need not to be enforced.

The main contributions of this paper are: (1) a classification of derived entity types; (2) an analysis of the constraints entailed by derivation rules and that must be defined in a taxonomy; and (3) an analysis of the possible ways of satisfaction of taxonomic constraints in presence of derived types. We assume a temporal conceptual schema and information base, and allow multiple specialization and classification.

We expect our contribution to be useful for the two purposes mentioned above. In particular, we want to stress the applicability to the enforcement of constraints. Current methods and techniques for the enforcement of taxonomic constraints do not take into account derived types [4, 27]. An illustrative example is [21], which details a wide range of techniques to express any kind of taxonomic constraints into standard DBMS constructs, but it is assumed that all entity types in the taxonomy are base. These methods could be extended easily to schemas that include derived types.

To the best of our knowledge, the relationships between taxonomic constraints and derivation rules have been studied, in conceptual modeling, only in the context of Description Logics (DL) [7, 9, 12] or similar [14]. This paper differs from most of that work in several aspects: (1) we deal with taxonomies of entity types in temporal conceptual schemas; (2) we assume a clearer separation between taxonomy (with taxonomic constraints) and derivability (with derivation rules); (3) we deal with derivation rules written in the FOL language, instead of a particular DL language [10]; (4) even if we use the FOL language, here we focus more on defining the relationships than on determining automatically them; we hope this will help in adapting and using the results reported here in several conceptual modeling languages; and (5) we give a special treatment to the determination of the taxonomic constraints that must be enforced by the designer.

The structure of the paper is as follows. Section 2 reviews taxonomic constraints and introduces the notation we will use. Section 3 develops a classification of derived entity types, based mainly on an analysis of their derivation rules. That analysis also allows us to determine some taxonomic constraints entailed by the rules. We discuss how these constraints must be reflected in the taxonomy. Section 4 analyzes how the constraints defined in a taxonomy can be satisfied, in presence of derived entity types. Section 5 summarizes the results and points out future work.

2 Taxonomies

A taxonomy consists of a set of concepts and their specialization (subsumption) relationships [11]. In a conceptual schema, we have a taxonomy of entity types and one or more of relationship types. In this paper we deal only with the former. Specializations have an intensional and an extensional aspect. Here, we focus on the extensional aspect, where a specialization is a constraint between the population of two entity types. Other constraints related to taxonomies are disjointness and covering. We call taxonomic constraints the set of specialization, disjointness and covering constraints defined in a schema. We introduce below the terminology and notation that will be used throughout the paper.

We adopt 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 entity e is instance of entity type E at time t. Similarly, we represent by $R(e_1,...,e_n,t)$ the fact that entities $e_1,...,e_n$ participate in a relationship instance of relationship type R at time t [17, 6, 30]. We say that $E(e,t)$ and $R(e_1,...,e_n,t)$ are entity and relationship facts, respectively.

2.1 Taxonomic Constraints

We denote by $S = E'$ *IsA* E a *specialization* constraint between entity types E' and E. For example, *Man IsA Person*. E' is said to be a subtype of E, and E a supertype of E'. An entity type may be supertype of several entity types, and it may be a subtype of several entity types at the same time. As usual, we require that an entity type may not be a direct or indirect subtype of itself. Informally, a specialization means that if an entity e is instance of E' at time t, then it must also be instance of E at t. In the language of first order logic, the meaning of a specialization $S = E'$ *IsA* E is given by the formula[1]:

$$E'(e,t) \rightarrow E(e,t)$$

We denote by $D = E_1$ *disjoint* E_2 a *disjointness* constraint (sometimes called *IsNotA*) between entity types E_1 and E_2. For example, *Man disjoint Woman*. The informal meaning is that the populations of E_1 and E_2 at any time are disjoint. Naturally, E_1 *disjoint* E_2 is equivalent to E_2 *disjoint* E_1. The formal meaning is given by the formula:

$$E_1(e,t) \rightarrow \neg E_2(e,t)$$

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

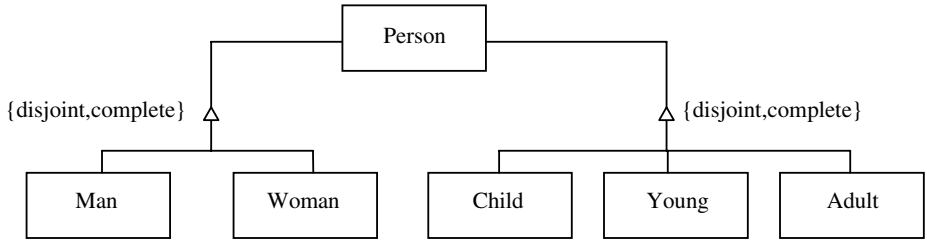

Fig. 1. Graphical representation in UML of *Person Gens Man, Woman*, and *Person Gens Child, Young, Adult*.

Finally, a *covering* constraint between an entity type E' and a set of entity types $E_1,...,E_n$, will be denoted by $C = E'$ *covered* $E_1,...,E_n$. For example, *Person covered Man, Woman*. The informal meaning is that if e is instance of E' at time t, it must be also instance of at least one E_i at t. Formally:

$$E'(e,t) \rightarrow E_1(e,t) \vee ... \vee E_n(e,t)$$

Notice that, by itself, E' *covered* $E_1,...,E_n$ does not imply E_i *IsA* E'.

A specialization constraint is a covering constraint with only one covering type: E' *IsA* $E \equiv E'$ *covered* E [24]. However, we will treat them separately, because both are widely used in practice.

We call *base* the taxonomic constraints explicitly defined in a schema. From the base constraints other may be *derived* using a set of inference rules. We are interested in derived constraints because derivation rules may entail them and, in this case, it will be necessary to check that such constraints may be derived from the base ones. We deal with derived constraints in Subsection 3.3.

All modern conceptual models and languages allow defining taxonomies, usually structured in generalizations [22, 4, 6]. A generalization $G = E$ *Gens* $E_1,...,E_n$ corresponds to a set of specialization constraints E_i *IsA* E, for $i = 1,..,n$, with a common supertype E. *Gens* is a shorthand for *Generalizes*. A generalization is *disjoint* if their subtypes are mutually disjoint; otherwise, it is *overlapping*. A generalization is *complete* if the supertype E is covered by the subtypes $E_1,...,E_n$; otherwise it is *incomplete*. Many conceptual languages provide a graphical representation of generalizations. Figure 1 shows an example of two generalizations of *Person*, in the UML language [31].

2.2 Partitions

A partition is a modeling construct that allows us to define in a succinct way a set of taxonomic constraints. A *partition* is a generalization that is both disjoint and complete. We denote by $P = E$ *Partd* $E_1,...,E_n$ a partition of entity type E into entity types $E_1,...,E_n$. *Partd* is shorthand for *Partitioned*. For example, *Person Partd Man, Woman*. A partition $P = E$ *Partd* $E_1,...,E_n$ corresponds to the constraints:

$S_i = E_i$ IsA E, for $i = 1,..,n$

$C = E$ covered $E_1,...,E_n$

$D_{i,j} = E_i$ disjoint E_j, for $i,j = 1,..,n$, $i \neq j$.

We do not require here to structure a taxonomy in partitions. However, we give a special treatment to them due to their importance in conceptual modeling [26, 29].

3 Derivation Rules

An information system may know the population of an entity type in three distinct ways, which are reviewed below. Then, we classify derived entity types according to the form of their derivation rule, and show the entailed constraints in each case.

3.1 Derivability of Entity Types

The *derivability* of an entity type is the way how the information system knows the population of that entity type at any instant. According to derivability, an entity type may be *base*, *derived* or *hybrid* [5, 26]. We give below a few comments on each of them:

- *Base*. An entity type *E* is base when the population of *E* at time *t* is given directly or indirectly by the users by means of insertion and deletion events. An insertion event of entity *e* in *E* at *t* means that *e* is instance of *E* from *t*. A deletion event of *e* in *E* at *t* means that *e* ceases to be instance of *E* at *t*. The information system knows the population of a base entity type *E* at *t* by using a general *persistence* (or frame) *axiom*: the population is the set of entities that have been inserted at time $t_1 \leq t$ and have not been deleted between t_1 and *t*.
- *Derived*. An entity type *E* is derived when the population of *E* at *t* is given by a formal derivation rule, which has the general form:

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

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

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

- We call such derivation rules *partial*, because they define only part of the population of *E*. The information system may know the population of *E* at any time by using the persistence axiom and the above rule.

Any hybrid entity type can be transformed easily into an equivalent derived one. This allows us to simplify our analysis, since we only have to deal with base and derived types. The procedure for the transformation is described in [5]. Here we show it by means of an example. Assume that *Building* is hybrid, with the partial derivation rule:

$$\text{Building}(b,t) \leftarrow \text{House}(b,t)$$

This means that houses are buildings, but that there may be other buildings besides houses. We define a new base entity type $Building_{ext}$, such that its population at time *t* is the set of entities that users have defined explicitly as being buildings. Usually, we will require the constraint $Building_{ext}$ *disjoint House*, but this is not mandatory. Now, *Building* is derived, with the derivation rule:

$$\text{Building}(b,t) \leftrightarrow \text{House}(b,t) \vee \text{Building}_{\text{ext}}(b,t)$$

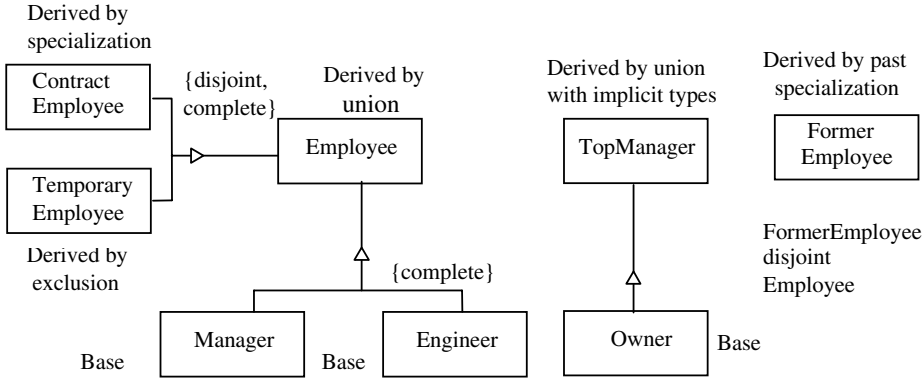

Fig. 2. Example of entity types with their derivability, and the taxonomic constraints entailed by their derivation rules.

3.2 Classification of Derived Entity Types

We are going to see that derived entity types can be classified into derived by: specialization, exclusion, past specialization, union and union with implicit types.

This classification is important to see the taxonomic constraints entailed by derivation rules. On the other hand, we think that the classification is useful for reasoning and implementation purposes (especially if entity types are materialized), in the same way as the classification of integrity constraints (key, referential, cardinalities, ...) eases reasoning about them and allows their efficient implementation.

For generality, we assume that derivation rules are written in the FOL language. However, the results shown below could be of interest to many conceptual modeling languages.

To illustrate the classification, we use as an example the following derived entity types and derivation rules:

DR1: $\text{Employee}(e,t) \leftrightarrow \text{Manager}(e,t) \vee \text{Engineer}(e,t)$
DR2: $\text{ContractEmployee}(e,t) \leftrightarrow \text{HasContract}(e,\text{contract},t)$
DR3: $\text{TemporaryEmployee}(e,t) \leftrightarrow \text{Employee}(e,t) \wedge \neg \text{ContractEmployee}(e,t)$
DR4: $\text{TopManager}(m,t) \leftrightarrow (\text{Manager}(m,t) \wedge \neg\exists \text{super HasSubordinate}(\text{super},m,t)) \vee \text{Owner}(m,t)$
DR5: $\text{FormerEmployee}(e,t) \leftrightarrow \text{Employee}(e,t1) \wedge \text{Time}(t) \wedge t1 < t \wedge \neg \text{Employee}(e,t)$

In DR5, *Time*(*t*) is a special predicate whose facts are instants of the temporal domain. Figure 2 summarizes the classification of the above entity types and the taxonomic constraints (in UML) entailed by the derivation rules. Both results are obtained by the procedure described below.

The procedure is a simple four-step transformation of each derivation rule into an equivalent set of (logic programming) clauses [25]. The classification is based on the

number of clauses thus obtained, and the structure of each of them, as we show in what follows.

Let $E(x,t) \leftrightarrow \phi(x,t)$ be a derivation rule. Assuming the semantics of the completion, the rule can be expressed by:

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

In general, $\phi(x,t)$ can be any valid first order formula, which makes its analysis difficult. However, using the well-known (at least in the logic programming and deductive databases fields) procedure proposed in [25], we can transform it (first step) into an equivalent set of n clauses:

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge L_{i,m} \quad \text{for } i = 1...n$$

where each of the $L_{i,j}$ in the body of the clause is a (positive or negative) literal. Other auxiliary clauses may be obtained as well, but they are irrelevant for our purposes. As usual, we require that the resulting clauses are safe and stratified [25, 13], thus guaranteeing that their computation is safe. We call *simple* a clause having exactly one literal of the form:

$$E(x,t) \leftarrow E_i(x,t)$$

In the example, transformation of DR1and DR4 yields the clauses (the transformation of the other rules is trivial):

DR1-1: Employee(e,t) ← Manager(e,t)
DR1-2: Employee(e,t) ← Engineer(e,t)
DR4-1: TopManager(m,t) ← Manager(m,t) ∧ ¬HasSuper(m,t)
DR4-2: TopManager(m,t) ← Owner(m,t)
Aux: HasSuper(m,t) ← HasSubordinate(super,m,t)

In the body of a clause $E(x,t) \leftarrow L_{i,1} \wedge ... \wedge L_{i,m}$, a literal $L_{i,j}$ may be:

- A positive entity fact $E_i(x,t_i)$ meaning the x is instance of E_i at time t_i.
- A positive relationship fact $R_i(x,y,t_i)$ including x as an argument. The position of x is irrelevant, and y is a vector of arguments. The meaning now is that x and y participate in a relationship of type R_i at time t_i.
- Other literals (positive facts not including x, negative, evaluable, *Time*). These are irrelevant for our current purposes. Note that in this paper we do not deal with aggregates.

There must be at least one positive $E_i(x,t_i)$ or $R_i(x,y,t_i)$ literal in the clause. This condition is implied by the safeness of clauses (recall that the head of the clauses is $E(x,t)$).

In conceptual modeling, participants p_i in relationships of type $R(p_1:E_1,...,p_n:E_n)$ at time t must be instance of their corresponding entity type E_i at time t (referential integrity constraint). Therefore, assuming that E_i is the type corresponding to x, whenever literal $R_i(x,y,t_i)$ is true, $E_i(x,t_i)$ must be true also. This justifies our second step: for each positive relationship fact $R_i(x,y,t_i)$ appearing in the body of a clause, we add the corresponding literal $E_i(x,t_i)$. These literals are redundant with respect to $R_i(x,y,t_i)$, but they will allow us to find the entity types specialized by E.

In the example, this step can be applied to DR2. Assuming the relationship type *HasContract*(*Employee,Contract*), we add *Employee*(*e,t*) to the body of DR2:

DR2': ContractEmployee(e,t) ← HasContract(e,contract,t) ∧ Employee(e,t)

In the third step we remove redundant positive entity fact literals. If we have a clause:

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge E_i(x,t_k) \wedge \ ... \wedge \ E_j(x,t_k) \wedge \ ... \wedge \ L_{i,m}$$

and we have also a direct or indirect E_i *IsA* E_j in the schema, then we remove the redundant literal $E_j(x,t_k)$.

In the fourth step, we deal with the literals in the body of a clause which are positive entity facts including argument x and with a time argument t_i distinct from t, where t is the time argument of the head of the clause, $E(x,t)$. In general, such t_i may range in the interval $[1,...,t]$. The case $t_i > t$ is not permitted, because then $E(x,t)$ would not be computable at time t. We want to distinguish the case when $t_i = t$ and the case when $t_i < t$. For this purpose, given that $t_i \leq t \leftrightarrow t_i = t \vee t_i < t$, we replace each clause having a $E_i(x,t_i)$ literal:

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge E_i(x,t_i) \wedge ... \wedge L_{i,m}$$

by the two equivalent clauses:

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge E_i(x,t) \wedge ... \wedge L_{i,m} \qquad (t_i \text{ is replaced by } t)$$

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge E_i(x,t_i) \wedge t_i < t \wedge ... \wedge L_{i,m}$$

If any of the clauses has a contradictory body, it is removed. After this step, all positive entity facts will have a time argument with t or with a t_i such that $t_i < t$. We call the former *current* facts, and the latter *past* facts.

The application of this step to DR5 gives:

DR5-1: $\text{FormerEmployee}(e,t) \leftrightarrow \text{Employee}(e,t) \wedge t = t \wedge \text{Time}(t) \wedge t < t \wedge \neg \text{Employee}(e,t)$

DR5-2: $\text{FormerEmployee}(e,t) \leftrightarrow \text{Employee}(e,t1) \wedge \text{Time}(t) \wedge t1 < t \wedge \neg \text{Employee}(e,t)$

The first clause, DR5-1, is removed because of the contradiction $t = t$ and $t < t$. Such contradictions can be easily detected with the algorithms given in [18].

We now have sufficient machinery in place to classify derived entity types. We start by distinguishing two main cases: the derivation rule is transformed into a single clause (DR2',DR3,DR5-2) or into multiple clauses (DR1,DR4).

Single Clause. An entity type E is derived by *specialization* of $E_1,...,E_n$ if the above transformation of its derivation rule yields the single clause:

$$E(x,t) \leftarrow E_1(x,t) \wedge ... \wedge E_n(x,t) \wedge Res$$

with $n \geq 1$, and *Res* is a (possibly empty) residual set of literals, none of which is a positive entity fact $E_j(x,t)$. The above clause cannot be simple (that is, $n = 1$ and *Res* empty) because then E and E_1 would be redundant entity types.

If entity type E is derived by specialization of E_1, ..., E_n then the derivation rule of E entails the n specialization constraints: E *IsA* E_1, ..., E *IsA* E_n. If any of the literals in *Res* has the form $\neg E_p(x,t)$, then the derivation rule entails also the constraint: E *disjoint* E_p.

According to this definition, in our example *ContractEmployee* and *TemporaryEmployee* are derived by specialization of *Employee* (DR2' and DR3).

In conceptual modeling, many derived entity types are derived by specialization, and for some conceptual models, this is the only allowed type (for instance, NIAM [28] and Chimera [13])

A particularly interesting subcase of entity type derived by specialization is when the clause has exactly the form:

$$E(x,t) \leftarrow E'(x,t) \wedge \neg E_1(x,t) \wedge ... \wedge \neg E_m(x,t)$$

with $m \geq 1$. In this case, we say that E is derived by *exclusion*, because its instances are those of E' but excluding those of E_1 and ... and E_m. Now the derivation rule of E entails also:

- the disjointness constraints: E *disjoint* E_i, for $i = 1,..,m$.
- the covering constraint E' covered E, E_1, ..., E_m.

In the example, *TemporaryEmployee* is derived by exclusion (DR3). The set of constraints entailed by DR3 is:

TemporaryEmployee IsA Employee,
TemporaryEmployee disjoint ContractEmployee, and
Employee covered TemporaryEmployee, ContractEmployee.

Finally, we say that an entity type E is derived by *past specialization* of $E_1,...,E_n$ if the transformation of its derivation rule yields the clause:

$$E(x,t) \leftarrow E_1(x,t_1) \wedge ... \wedge E_n(x,t_n) \wedge Res$$

with $n \geq 1$, all $E_i(x,t_i)$ past, and where *Res* is a (possibly empty) residual set of literals, none of which is a positive entity fact $E_j(x,t)$. In this case the specializations E *IsA* E_i do not hold. If any of the literals in *Res* has the form $\neg E_p(x,t)$, then the derivation rule entails the constraint: E *disjoint* E_p.

The relationship between E_i and E that exists in this case is:

$$E(x,t) \rightarrow \exists t_k \, E_i(x,t_k) \wedge t_k < t$$

which can be seen as a past specialization. This specialization can be captured by the expression E *WasA* E_i, and could be considered a new kind of taxonomic constraint.

In the example, *FormerEmployee* is derived by past specialization of *Employee* (DR5-2). The constraint entailed by DR5-2 is *FormerEmployee disjoint Employee*.

Derivation by specialization or by past specialization are the two only possible cases when the original derivation rule is transformed into a single clause. The reason is that, as we have said before, after the first step the body of the clause must contain at least one $E_i(x,t_i)$ or $R_i(x,y,t_i)$ literal. If it is a $R_i(x,y,t_i)$ literal, then after the second step the body will include at least one $E_i(x,t_i)$ literal.

Multiple Clauses. Now we deal with the case when the original derivation rule is equivalent to a set of n clauses ($n > 1$):

$$E(x,t) \leftarrow L_{i,1} \wedge ... \wedge L_{i,m} \qquad \text{for i = 1...n}$$

An entity type E is derived by *union* of E_1, ..., E_n if the n clauses are simple; that is, they have the form:

$$E(x,t) \leftarrow E_1(x,t)$$
$$...$$
$$E(x,t) \leftarrow E_n(x,t)$$

If entity type E is derived by union of E_1, ..., E_n then the derivation rule of E entails the n specialization constraints: E_1 *IsA* E, ..., E_n *IsA* E and the covering constraint: E *covered* E_1, ..., E_n.

In the example, *Employee* is derived by union of *Manager* and *Engineer* (DR1-1 and DR1-2). The constraints entailed are: *Manager IsA Employee*, *Engineer IsA Employee* and *Employee covered Manager, Employee*.

Some conceptual models allow defining entity types derived by specialization and union. Among them, we mention PSM [19].

If any of the *n* clauses is not simple, we say that *E* is derived by *union with implicit types*. The non-simple clauses will be either a specialization or a past specialization, which can be seen as defining an implicit type. If some clause is simple, then the derivation rule entails a corresponding specialization constraint.

In the example, *TopManager* is derived by union of implicit types (DR4-1,DR4-2). The simple clause DR4-2 entails *Owner IsA TopManager*. Note that we can always transform a type derived by union with implicit types into one derived by union, with the introduction of new types. Thus, *TopManager* could be derived by union of *Owner* and *TopEmployeeManager*, defined as:

DR4-1': TopManager(m,t) ← TopEmployeeManager(m,t)
DR4-2': TopManager(m,t) ← Owner(m,t)
New: TopEmployeeManager(m,t) ← Manager(m,t) ∧ ¬HasSuper(m,t)

where *TopEmployeeManager* is the implicit type (derived by specialization).

In some special cases, the transformation yields *n* clauses with one or more common $E_i(x,t)$ literals. If this occurs, then *E* is classified as derived by specialization of E_i. For example, in the derivation rule:

PromotionCandidate(e,t) ↔ Engineer(e,t) ∧
(TemporaryEmployee(e,t) ∨ (HasSalary(e,sal,t) ∧ sal < Min))

where *PromotionCandidate* becomes derived by specialization of *Engineer*.

3.3 Taxonomy and Constraints Entailed by Derivation Rules

As we have just seen, a derivation rule may entail taxonomic constraints. Now, the question is: must these constraints appear in the taxonomy? Our answer is positive, because for verification, validation, implementation, and evolution purposes it is important that a taxonomy be as complete as possible. It is also important in order to ensure the consistency between the taxonomy and the derivation rules. In our example, this means to check that the taxonomic constraints depicted graphically in Figure 2 are included in the taxonomy.

A constraint entailed by a derivation rule will be very often a base one. In these cases, derivation rules and taxonomy match perfectly.

In other cases, however, a constraint entailed by a derivation rule is not a base one. In these cases, the constraint must be derivable from the base ones, using a set of inference rules. [3, 24] give the complete and sound set of inference rules for taxonomic constraints. If a constraint is not derivable using that set of inference rules, then either the taxonomy or the derivation rules must be modified.

4 Satisfaction of Taxonomic Constraints

In this Section, we analyze how the constraints defined in a taxonomy can be satisfied. We apply the results to the particular case of partitions.

4.1 Satisfaction of Base Constraints

An information base comprises all temporal base and derived facts that are true in the domain of the information system. An information base *IB* satisfies a constraint *IC* if *IC* is true in *IB*. There are several different meanings of constraint satisfiability in logic [15, 16], but the differences will not be important in this paper.

Satisfaction of integrity constraints can be ensured by the schema or by enforcement. We say that a constraint *IC* is satisfied by the *schema* (or intensionally) when the schema entails *IC*. That is, the derivation rules and the (other) constraints defined in the schema imply *IC* or, in other terms, *IC* is a logical consequence of the schema. In this case no particular action must be taken at runtime to ensure the satisfaction of *IC*. In a way, *IC* is redundant, but it may be important to keep it in the schema for verification, validation, implementation or evolution purposes. For example, if the schema includes:

S = LongPaper IsA Paper

DR: LongPaper(p,t) ← Paper(p,t) ∧ Length(p,pages,t) ∧ pages > 50

then *DR* entails *S*, and therefore *S* is satisfied by the schema.

We say that a constraint *IC* is satisfied by *enforcement* (or extensionally) when it is not satisfied by the schema, but it is entailed by the information base. That is, *IC* is a condition true in the information base. In this case, the system has to enforce *IC* by means of checking and corrective actions, to be executed whenever the information base is updated. For example, if *Manager* and *Engineer* are base entity types and we have the constraint *D = Manager disjoint Engineer*, then *D* must be enforced. This means to check at runtime that no entity is classified in both types at the same time and, if it is, to reject the cause of the constraint violation or to perform some corrective action.

The enforcement of constraints is expensive, but it may be the only option available for some constraints. However, methods developed so far assume that all taxonomic constraints must be enforced [21]. We are going to show that, in some cases, taxonomic constraints may be satisfied by the schema and, thus, their enforcement is not necessary. We discuss each kind of taxonomic constraint in turn.

Let *S = E′ IsA E* be a base specialization constraint. We distinguish four cases:

(1) if both *E′* and *E* are base, then *S* must be enforced.

(2) if *E′* is derived and *E* is base, then the derivation rule of *E′* must entail *S*. The rationale is that if the derivation rule of *E′*: $E'(e,t) \leftrightarrow \varphi(e,t)$ does not entail *S*, then we can transform it into the equivalent one $E'(e,t) \leftrightarrow (\varphi(e,t) \wedge E(e,t))$, which now entails *S*. Note that this transformation does not change the stratification of *E′*.

In the particular case that *E′* is derived by specialization of *E*, the derivation rule of *E′* entails *S*.

(3) when *E′* is base and *E* is derived, *S* may be entailed by the schema; if it is not, then *S* must be enforced.

In the general case, it may be difficult to prove that *S* is entailed by the schema, see comment below. In many practical cases, however, it suffices to prove that *S* is entailed by the derivation rule of *E*. This happens, for instance, when *E* is derived by union of a set of types that includes *E'*.

(4) when both *E'* and *E* are derived, *S* may be entailed by the derivation rules of *E'* or *E,* or by the schema; if it is not, then *S* must be enforced.

As an example, consider the five specialization constraints of Figure 1. Assume that *Man* and *Woman* are base, and that the other entity types are derived. The derivation rules are:

DR1-1: Person(p,t) ← Man(p,t)
DR1-2: Person(p,t) ← Woman(p,t)
DR2: Child(p,t) ← Person(p,t) ∧ Age(p,a,t) ∧ a < 5
DR3: Young(p,t) ← Person(p,t) ∧ Age(p,a,t) ∧ a > 5 ∧ a < 18.
DR4: Adult(p,t) ← Person(p,t) ∧ Age(p,a,t) ∧ a ≥ 18.

S_1 = *Man IsA Person* is entailed by DR1-1 (case 3). Similarly, S_2 = *Woman IsA Person* is entailed by DR1-2 (case 3). S_3 = *Child IsA Person*; S_4 = *Young IsA Person* and S_5 = *Adult IsA Person* are entailed by DR2, DR3 and DR4, respectively (case 4).

In cases (3) and (4) it may be necessary to prove that *S* is entailed by the schema. If it cannot be proved, then a safe approach is to enforce *S*. Automation of this proof requires the use of some reasoner [18, 15]. When derivation rules are written in a language based on Description Logics, such a reasoner is usually available [7, 9].

Let $D = E_1$ *disjoint* E_2 be a base disjointness constraint. We distinguish three cases:

(1) if both E_1 and E_2 are base, then *D* must be enforced.

(2) if E_1 is derived and E_2 is base, then the derivation rule of E_1 must entail *D*. The rationale is that if the derivation rule of E_1: $E_1(e,t) \leftrightarrow \varphi(e,t)$ does not entail *D*, then we can transform it into the equivalent one $E_1(e,t) \leftrightarrow (\varphi(e,t) \wedge \neg E_2(e,t))$, which now entails *D*. Note that this transformation does not change the stratification of E_1.

In the particular case that E_1 is derived by specialization of some type *E* and exclusion of some types including E_2, *D* is entailed by the derivation rule of E_1.

(3) when E_1 and E_2 are derived, *D* may be entailed by the schema; if it is not, then *D* must be enforced.

Continuing our previous example, consider now the disjointness constraint D_1 = *Man disjoint Woman*. Given that both entity types are base, D_1 must be enforced (case 1). In D_2 = *Young disjoint Child*, D_3 = *Young disjoint Adult* and D_4 = *Child disjoint Adult* both entity types are derived (case 3). It is easy to see that their derivation rules entail them. In this particular example, the algorithms described in [18] determine the entailment efficiently.

Let $C = E$ *covered* $E_1,...,E_n$ be a base covering constraint. We distinguish only two cases:

(1) if all entity types are base, then *C* must be enforced.

(2) in any entity type is derived, *C* may be entailed by the schema; if it is not, then *C* must be enforced.

There are two particular subcases, frequently found in practice, for which C is entailed by a derivation rule. The first is when E is derived by union of $E_1,...,E_n$. The second is when some E_j is derived by specialization of E and exclusion of $\{E_1,...,E_n\} - \{E_j\}$.

Continuing again our previous example, consider now the covering constraint C_1 = *Person covered Man, Woman*. Given that *Person* is derived by union of *Man* and *Woman*, C_1 is satisfied by the schema. The constraint C_2 = *Person covered Child, Young, Adult* is entailed by the derivation rules of *Child*, *Young* and *Adult*. Again, in this particular example, the algorithms described in [18] determine the entailment efficiently.

Note, on the other hand, that if the derivation rule of *Adult* were:

$$\text{Adult}(p,t) \leftrightarrow \text{Person}(p,t) \wedge \neg \text{Child}(p,t) \wedge \neg \text{Young}(p,t)$$

then *Adult* would be derived by specialization of *Person* and exclusion of *Child* and *Young* and, therefore, C_2 would be entailed by this derivation rule.

4.2 Satisfaction of Partitions

Partitions can be classified according to the derivability of their supertype and that of their subtypes. All combinations are possible. The only exception, at least in conceptual modeling, is when all entity types are base, because in this case the supertype could be derived by union of its subtypes. The classification is useful in order to analyze how the partition constraints can be satisfied in each case. We discuss below the two most popular kinds of partitions.

Partition by Specialization. Let $P = E$ *Partd* $E_1,...,E_n$ be a partition. We say that P is a *partition by specialization* when all E_i, $i = 1,..,n$, are derived by specialization of E. We have an example in Figure 1: *Person Partd Child, Young, Adult*, with all subtypes derived by specialization of *Person*, as shown in the derivation rules DR2 - DR4 given above.

Let's see how the partition constraints may be satisfied in this kind of partition. First, the specialization constraints $S_i = E_i$ *IsA* E are satisfied by the schema. More specifically, they are entailed by the derivation rule of E_i.

Second, the disjointness constraints $D_{i,j} = E_i$ *disjoint* E_j should be satisfied by the derivation rules of E_i and E_j. In the general case, when both E_i and E_j are derived, the constraint *may* be entailed by the schema, but in this case it seems sensible to require it to be entailed and, in particular, to be entailed by the derivation rules. The reason is that it should not be difficult to write the derivation rules in a way that ensures disjointness.

In many cases, a partition by specialization has a subtype for each value of some single-valued attribute of E. In these cases, disjointness is naturally ensured. For example, the partition *Person Partd Single, Married, Divorced, Widower*, where *Person* has attribute *MaritalStatus* and the derivation rules are:

$$\text{Single}(p,t) \leftrightarrow \text{Person}(p,t) \wedge \text{MaritalStatus}(p,\text{Single},t)$$

and similarly for the other subtypes. The difference in the argument *ms* of the literal *MaritalStatus(p,ms,t)* ensures disjointness.

Finally, the covering constraint $C = E$ *covered* $E_1,...,E_n$ should be satisfied by the derivation rules of $E_1,...,E_n$, for the same reason as above. In the example, this requires to check only that there is a subtype for each possible value of *MaritalStatus*.

When one of the subtypes, E_j, is derived by specialization of E and exclusion of $\{E_1,...,E_n\}$-$\{E_j\}$, then C is satisfied by the derivation rule of E_j.

In summary, partitions by specialization does not require enforcement of any partition constraint. They only require writing carefully the derivation rules, so that the disjointness and covering constraints are satisfied by the schema.

Partition by Union. Let $P = E$ *Partd* $E_1,...,E_n$ be a partition. We say that P is a *partition by union* when E is derived by union of $E_1,...,E_n$. We have also an example in Figure 1: *Person Partd Man, Woman*, with *Person* derived by union of *Man* and *Woman*, as shown in the derivation rule DR1 given above.

Let's see how the partition constraints may be satisfied in this kind of partition. First, the specialization constraints $S_i = E_i$ *IsA* E are satisfied by the schema. More specifically, they are entailed by the derivation rule of E.

Second, the disjointness constraints $D_{i,j} = E_i$ *disjoint* E_j can be satisfied, as indicated in subsection 4.1, either by the schema or by enforcement, depending on the derivability of E_i and E_j. In the example, *Man disjoint Woman* must be enforced.

Finally, the covering constraint $C = E$ *covered* $E_1,...,E_n$ is satisfied by the schema. More specifically, it is entailed by the derivation rule of E.

In summary, partitions by union may require the enforcement of only some disjointness constraints. All other partition constraints are enforced by the schema.

5 Conclusions

This paper has focused on the relationships between taxonomic constraints and derivation rules. The objectives were to see which taxonomic constraints are entailed by derivation rules, and must be included in a taxonomy, and to analyze how taxonomic constraints can be satisfied in presence of derived types.

Based on an analysis of their derivation rules, we have classified derived entity types into derived by specialization, by past specialization, by union or by union with implicit types. The analysis reveals the taxonomic constraints entailed in each case. These constraints must be base constraints (defined in the taxonomy) or be derivable from them. We have shown how the base taxonomic constraints can be satisfied, either by the derivation rules (or the whole schema), or by enforcement. Our analysis can be extended naturally to taxonomies of relationship types [1,2].

Our results are general and could be incorporated into many conceptual modeling environments and tools. The expected benefits are an improvement in the verification of the consistency between taxonomic constraints and derivation rules, and a guide (to the information system designer) for the determination of the taxonomic constraints that must be enforced in the final system.

We see our results as complementary to those of Description Logics [9, 12], mainly because we assume a different context: temporal conceptual schemas, taxonomies of entity and relationship types, and derivation rules written in the FOL

language. On the other hand, we deal with the design problem of determining which constraints must be enforced.

The work reported here can be extended in several directions. The analysis of derivation rules given in Subsections 3.2 and 5.1 can be completed by taking into account aggregate literals. The classifications given in these subsections may be refined, and further entailed constraints may be defined. We have focused on a single derivation rule to see the constraints entailed by it, but it should be possible to consider also sets of such rules. The analysis of partitions given in Subsection 4.2 can be extended to other frequent kinds. Finally, the analysis can be extended to taxonomies of relationship types.

References

1. Analyti, A.; Constantopoulos, P.; Spyratos, N. "Specialization by restriction and Schema Derivations", Information Systems, 23(1), 1998, pp. 1 - 38.
2. Analyti, A.; Spyratos, N.; Constantopoulos, P. "Property Covering: A Powerful Construct for Schema Derivations", Proc. ER'97, LNCS 1331, Springer-Verlag, pp. 271-284.
3. Atzeni, P.; Parker, D.S. "Formal Properties of Net-based Knowledge Representation Systems", Procs. ICDE'86, Los Angeles, 1986, pp. 700-706.
4. Batini, C.; Ceri, S.; Navathe, S. Conceptual Database Design. An Entity-Relationship Approach. The Benjamin/Cummings Pub. Co., 1992, 470 p.
5. 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.
6. Boman, M.; Bubenko, J.A.; Johannesson, P.; Wangler, B. Conceptual Modelling. Prentice Hall, 1997, 269 p.
7. Bergamaschi, S.; Sartori, C. "On Taxonomic Reasoning in Conceptual Design", ACM TODS 17(3), 1992, pp. 385-422.
8. Borgida, A.; Mylopoulos, J.; Wong, H.K.T. "Generalization/Specialization as a Basis for Software Specification". In Brodie, M.L.; Mylopoulos, J.; Schmidt, J.W. (Eds.) "On Conceptual Modelling", Springer-Verlag, 1984, pp. 87-117.
9. Borgida, A. "Description Logics in Data Management", IEEE Trans. on Knowledge and Data Eng., 7(5), 1995, pp. 671-682.
10. Borgida, A. "On the Relative Expressiveness of Description Logics and Predicate Logics". Artificial Intelligence 82(1-2), 1996, pp. 353-367.
11. Brachman, R.J.; Schmolze, J.G. "An Overview of the KL-ONE Knowledge Representation System". Cognitive Science 9, 1985, pp. 171-216.
12. 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 Academic Press, 1998, pp. 229-263.
13. Ceri, S.; Fraternali, P. Designing Database Applications with Objects and Rules. The IDEA Methodology. Addison-Wesley, 1997, 579 p.
14. Delcambre, L.M.L.; Davis, K.C. "Automatic Verification of Object-Oriented Database Structures". Procs. ICDE'89, Los Angeles, pp. 2-9.
15. Decker, H.; Teniente, E.; Urpí, T. "How to Tackle Schema Validation by View Updating". Proc. EDBT'96, Avignon, LNCS 1057, Springer, 1996, pp. 535-549.
16. Godfrey, P.; Grant, J.; Gryz, J.; Minker, J. "Integrity Constraints: Semantics and Applications", In Chomicki, J.; Saake, G. (Eds.) Logics for Databases and Information Systems, Kluwer Academic Press, 1998, pp. 265-306.

17. Gustaffsson, M.R.; Karlsson, T.; Bubenko, J.A. jr. "A Declarative Approach to Conceptual Information Modeling". 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.
18. Guo, S.; Sun, W.; Weiss, M.A. "Solving Satisfiability and Implication Problems in Database Systems". ACM TODS 21(2), 1996, pp. 270-293.
19. Halpin, T.A.; Proper, H.A. "Subtyping and polymorphism in object-role modelling", Data and Knowledge Eng., 15, 1995, pp. 251-281.
20. Hammer, M.; McLeod, D. "Database Description with SDM: A Semantic Database Model", ACM TODS, 6(3), 1981, pp. 351-386.
21. Hainaut, J-L.; Hick, J-M.; Englebert, V.; Henrard,J.; Roland, D. "Understanding the Implementation of IS-A Relations". Proc. ER'96, Cottbus, LNCS 1157, Springer, pp. 42-57.
22. Hull, R.; King, R. "Semantic Database Modeling: Survey, Applications, and Research Issues", ACM Computing Surveys, 19(3), 1987, pp. 201-260.
23. ISO/TC97/SC5/WG3. "Concepts and Terminology for the Conceptual Schema and Information Base", J.J. van Griethuysen (ed.), March, 1982.
24. Lenzerini, M. "Covering and Disjointness Constraints in Type Networks", Proc. ICDE'87, Los Angeles, IEEE, 1987, pp. 386-393.
25. Lloyd, J.W. Foundations of Logic Programming. 2nd Edition. Springer-Verlag, 1987, 212 p.
26. Martin, J.; Odell, J.J. Object-Oriented Methods: A Foundation. Prentice Hall, 1995, 412 p.
27. Markowitz, V.M.; Shoshani, A. "Representing Extended Entity-Relationship Structures in Relational Databases: A Modular Approach", ACM TODS 17(3), 1992, pp. 423-464.
28. Nijssen, G.M.; Halpin, T.A. Conceptual Schema and Relational Database Design. Prentice Hall, 1989, 342 p.
29. Olivé, A.; Costal, D.; Sancho, M-R. "Entity Evolution in IsA Hierarchies", Proc. ER'99, LNCS 1728, Springer, 1999, pp. 62-80.
30. Olivé, A. "Relationship Reification: A Temporal View", Proc. CAiSE'99, LNCS 1626, Springer, 1999, pp. 396-410.
31. Rumbaugh, J.; Jacobson, I.; Booch, G. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999, 550 p.