# Declarative Taxonomic Constraint Enforcement in Conceptual Schemas (Extended Version)

Dolors Costal, Cristina Gómez and Ernest Teniente

Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Jordi Girona 1-3 E08034 Barcelona (Catalonia)
{dolors|cristina|teniente}@lsi.upc.edu
Fax number: 93 413 7833

**Abstract.** We propose to declaratively specify policies for the enforcement of taxonomic integrity constraints directly in the structural conceptual schema. These policies depend on the kind of constraint to be enforced (disjointness, covering or specialization) and on the particular event that may cause its violation. We provide a formal definition of these policies and of the repair actions that must be considered to ensure constraint enforcement. Those repairs may then be generated automatically by a repair process that is guaranteed to terminate. Our work eases conceptual modelling since defining taxonomic constraint enforcement declaratively allows omitting its specification from the external events in the behavioural conceptual schema.

## 1. Introduction

In recent years, proposals that give a key role to conceptual modeling in information systems development are emerging. We may remark among them conceptual schema-centric development (CSCD) [Oli05]. One of the main problems to solve to achieve the CSCD goal is facilitating the designer the task of conceptual modeling. Our work represents a step forward in this direction for the specification of taxonomic constraint enforcement. In this sense, we propose to declaratively specify enforcement policies in a single place in the structural conceptual schema (instead of spreading its behavior among the specification of all the external events) and we provide a formalization that serves as a basis for its implementation.

An information system maintains a representation of the state of a domain in its information base (IB). The conceptual schema of an information system must include all relevant knowledge about the domain. Hence, the structural conceptual schema defines the structure of the IB and its constraints while the behavioural conceptual schema defines how the IB changes when events occur.

Taxonomies are fundamental constructs of structural conceptual schemas [OT02]. In its most basic form, a taxonomy consists of a set of entity types and a set of specialization relationships among them. A specialization relationship implies a constraint between the populations of two types. Usually, taxonomies also include other constraints like disjointness and covering.

Figure 1.1 illustrates a simple taxonomy that specializes entities of type *Person* as entities of types *Single*, *Married*, *Divorced* or *Widower*. We assume a disjointness constraint ensuring that a person has no more than one marital status and also a covering constraint that guarantees each person has at least one.

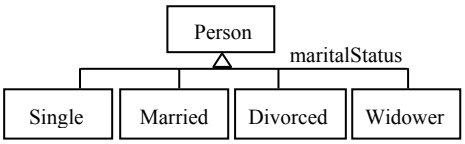



**Figure 1.1 Example of a taxonomy**

Integrity enforcement ensures that constraints are satisfied after each update of the IB. This may be achieved either by integrity checking, which rejects any update leading to an inconsistent state of the IB; or integrity maintenance, aimed at repairing constraint violations by performing additional updates that maintain the IB consistent.

The classical approach to specify how to perform integrity enforcement in conceptual modelling has been to spread the enforcement of those constraints among the various events (possibly a large number) of the behavioural schema that may violate the constraint. Apart from increasing the difficulty of the conceptual modelling task (since the designer must be able to determine all events that may violate each constraint), this approach has a negative impact on the understandability and modifiability of the resulting conceptual schemas (since integrity enforcement is not localised in a single place but distributed among the behavioural schema).

In our example, an event of *Marriage*, aimed at adding two entities in the *Married* subtype, will need to include also the removal of those entities from the type of their previous marital status to maintain the IB consistent. Such additional removals are also required into an event of *Divorce* or into the one of becoming a widower.

The approach we propose in this paper, which is independent of the particular language used, overcomes the previous limitations by specifying taxonomic constraint enforcement declaratively in the structural conceptual schema. We define a set of predetermined policies for each taxonomic constraint, which depend on the type of change that causes the constraint violation, and that can be easily specified by the designer.

Another significant contribution of our work is the formalization of the proposed constraint enforcement policies by a precise definition of the repair actions or rejecting conditions for events they imply. This formalization serves as a basis for the automatic generation of events that preserve consistency and for ensuring that this generation process always terminates. An implementation of our taxonomic constraint enforcement policies by means of SQL:1999 standard triggers is also presented.

In this way, our approach allows specifying integrity constraint enforcement declaratively (without the need to know which events may violate a given constraint) and in a localised way in the structural conceptual schema. As a consequence, the definition of external events becomes simpler because only its intended effect must be considered and taxonomic integrity enforcement actions or conditions may be omitted.

In the previous example, our approach allows specifying that violations of the disjointness constraint due to a subtype insertion must always be repaired by a policy of removal of the entity from the previous subtype. Thus, the designer does not need

to care about including additional changes into the events of *Marriage*, *Divorce* nor becoming a widower to specify integrity constraint enforcement since they will be automatically determined from the specified policy.

This paper is structured as follows. Next section presents basic concepts. Sections 3 and 4 describe the intuition and formalization of the proposed policies, respectively. Section 5 shows termination for any procedure based on our formalization. Section 6 presents a procedure to repair events by means of SQL triggers. Section 7 reviews related work. Finally, section 8 gives the conclusions and outlines future work.

## 2. Basic Concepts

In this section we present basic concepts of structural and behavioral conceptual schemas in terms of first-order logic in order to be independent of particular conceptual modeling languages.

### 2.1 Structural conceptual schemas

Structural conceptual schemas define the structure of the IB and its constraints. We assume that entities are instances of their types at particular time points [Bub77], which are expressed in a common base time unit. We represent by *E(e,t)* the fact entity *e* is instance of entity type *E* at time *t*. For example, *Employee(Marta,D1)* means that *Marta* is an instance of *Employee* at time *D1*. The population of *E* at *t* is the set of entities that are instances of *E* at *t*.

Taxonomies of entity types consist of a set of entity types and a set of specialization relationships among them. A specialization is a relationship between two entity types *E'* and *E*, that we denote by *E' IsA E*. For instance, *JuniorEmp IsA Employee*. *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. A specialization implies a taxonomic specialization constraint between the populations of the subtype and the supertype. In terms of the population of *E'* and *E*, this implies that all instances of *E'* at a certain time point are also instances of *E* at that time point.

Generalization and specialization are two different viewpoints of the same IsA relationship, viewed from the supertype or from the subtype. We denote by *E Gens* $E_1,...,E_n$ the generalization of entity types $E_1,...,E_n$ to *E*. For instance, *Employee Gens JuniorEmp, SeniorEmp*.

Given a generalization *E Gens* $E_1,...,E_n$, we denote by *Disjoint*($E_1,...,E_n$) the taxonomic disjointness constraint which indicates that an entity can be instance of at most one $E_i$ at *t* [Len87]. For example, *Disjoint(JuniorEmp, SeniorEmp)* would indicate that an employee cannot be both junior and senior.

We also denote by *Covering*($E,E_1,...,E_n$) the taxonomic covering constraint of a given generalization which indicates that every instance of *E* at *t* is instance of at least one $E_i$ at *t* [Len87]. For example, *Covering(Employee, JuniorEmp, SeniorEmp)* states that an employee must be either junior or senior.

Note that generalizations labelled as overlapping (incomplete) in some specification languages correspond to the absence of disjointess (covering) constraint.

Satisfaction of taxonomic constraints can be ensured by the schema or by enforcement. A constraint is satisfied by the schema when the schema entails the constraint. For example, in ORM [Hal01] formal subtype definitions are required which may imply the satisfaction of some taxonomic constraints. UML [OMG03] admits derived types which can also ensure taxonomic constraints. A constraint must be enforced when it is not satisfied by the schema, but it must be satisfied by the information base [OT02]. Our work deals with the latter case.

### 2.2 Behavioural conceptual schemas

Behavioural conceptual schemas define how the IB changes when events occur. Our description of behavioural conceptual schemas is mainly based on that of [Oli00].

The state of a domain at some time point is the set of instances of the entity and relationship types that exist in the domain at that time. When the state of a domain changes, the IB must change accordingly. We say that there is a change in the state of a domain at time *t* if the entity or relationship instances at time *t* are different from those existing at previous time *t-1*.

External events cause changes in the state of the domain and, consequently, in the IB. External events can be defined precisely in terms of a more basic concept, a structural event. A structural event causes an elementary change in the IB. These event types are called structural because they are completely determined from the structural part of a conceptual schema. We use two structural event types [COS97], to update the contents of IB taxonomies: entity insertion and entity deletion.

Let *E* be an entity type. We denote an entity insertion (resp. deletion) structural event in *E* by predicate *Insert_E(x,t)* (resp. *Delete_E(x,t)*). Its effect is the addition (removal) of the fact that entity *x* is instance of entity type *E* at time *t*. It is assumed that the event is applicable [VF85], that is, it ensures that the intended effect does not hold at previous state. For each one of them, we formally specify its effect (denoted by Eff) in the IB and also an applicability axiom (App) that guarantees that the event is applicable. Formally,

| *Insert_E(x,t)* | *Delete_E(x,t)* |
|---|---|
| App: $\forall x,t(Insert\_E(x,t) \rightarrow \neg E(x,t\text{-}1))$ | App: $\forall x,t(Delete\_E(x,t) \rightarrow E(x,t\text{-}1))$ |
| Eff: $\forall x,t(Insert\_E(x,t) \rightarrow E(x,t))$ | Eff: $\forall x,t(Delete\_E(x,t) \rightarrow \neg E(x,t))$ |

Entities existing (or not existing) at previous state of the IB that are not affected by the structural events remain unchanged [VF85].

An external event consists of a set of structural events which causes composite changes in the IB. The time at which the change occurs is the occurrence time of the event. The effect of an external event *Ev* is defined by an expression (written in some language) such that its evaluation gives the set *S* of structural event facts defined by *Ev*. We denote *S* by $\{SE_1,...,SE_i,...SE_n\}$ and our work is independent of the language used to specify the expression that permits to obtain the set *S*. For instance, an external event *Marriage* of *Pere* and *Maria* at time *25* in the example of figure 1.1 would be: *S={Insert_Married(Pere,25), Insert_Married(Maria,25)}*.

## 3. Specifying taxonomic constraint enforcement policies

Our proposal for taxonomic constraint enforcement specification consists of providing, for each type of constraint and type of change that may cause its violation, a set of predetermined policies (which may follow either integrity checking or maintenance) for integrity enforcement. Policies must consider the type of change that induces the violation because the enforcement strategies that make sense are different depending on it.

In order to facilitate their practical use to the designer, the provided policies must have a simple and clear meaning, there must be a reasonable number of them (not too large) and, at the same time, they must offer the necessary expressiveness to deal with the more usual cases of integrity enforcement. It is interesting to note that although the meaning of policies might be simple considered individually, their global implications to the enforcement of external events are usually not straightforward, since multiple policies may interact.

In the following, we give an intuitive explanation of the policies we propose and, finally, we provide an example of a taxonomy which combines several policies.

### 3.1 Disjointness Constraint

Given a generalization *E Gens* $E_1,...,E_n$, a disjointness constraint *Disjoint*($E_1,...,E_n$) indicates that every instance of *E* at time *t* is instance of at most one $E_i$ at *t*. Therefore, the only type of structural event that may induce a disjointness constraint violation is an entity insertion in one of the subtypes of the generalization. This subtype insertion may come directly from the external event definition or, indirectly, from the application of a previous integrity maintenance policy.

Policy *delete-when-subtype-insertion* is an integrity maintenance policy that repairs the external event by adding an action that deletes the involved entity from the subtype of the generalization to which it belonged in the previous time instant. For example, this policy is adequate to deal with the disjointness constraint of figure 1.1, and thus, it is able to repair the event of *Marriage* presented in section 2.2: *S={Insert_Married(Pere,25), Insert_Married(Maria,25)}*, by adding to it the deletion of *Pere* and *Maria* from the subtype of their respective previous marital status.

The alternative policy *restrict-when-subtype-insertion* is an integrity checking policy and, like all integrity checking policies, consists of rejecting the external event that violates the constraint.

These policies might be used for disjointness constraints among entity types which do not have a common direct supertype although we do not address this case.

### 3.2 Covering Constraint

The covering constraint for a generalization *E Gens* $E_1,...,E_n$ indicates that every instance of the supertype *E* at time *t* is instance of at least one subtype $E_i$ at *t*. Then, there are two types of structural events that may induce a covering constraint violation: an entity insertion in the supertype of the generalization or an entity deletion (a set of entity deletions) from one (some) of the subtypes.

Policy *insert-in-$E_i$-when-supertype-insertion* repairs a covering violation induced by an insertion in a supertype by performing an insertion of the involved entity in a particular subtype $E_i$. Observe that the subtype $E_i$ is selected by the designer among the subtypes of $E$ when he specifies the policy. Policy *restrict-when-supertype-insertion* deals with this case by rejecting the external event.

In case of a covering violation induced by deletions from subtypes, policy *insert-in-$E_i$-when-subtype-deletion* adds an insertion of the involved entity in a particular subtype $E_i$. Policy *delete-when-subtype-deletion* repairs this situation by performing a deletion of the involved entity from the supertype of the generalization. Finally, policy *restrict-when-subtype-deletion* rejects the external event.

### 3.3 Specialization Constraint

The specialization constraint for a relationship *E' IsA E* indicates that every instance of the subtype *E'* at time *t* is instance of the supertype *E* at *t*. This constraint may be violated only by two types of structural events: an entity insertion in the subtype of the specialization relationship or an entity deletion from the supertype.

Policy *insert-when-subtype-insertion* repairs a specialization violation induced by an insertion in the subtype by adding an insertion of the involved entity in the supertype and *restrict-when-subtype-insertion* rejects the external event in that case.

Additionally, policy *delete-when-supertype-deletion* repairs a specialization violation induced by a deletion from the supertype by performing a deletion of the involved entity from the subtype and *restrict-when-supertype-deletion* rejects the external event in the same situation.

### 3.4 Default policies

For practical purposes, we establish default policies. For disjointness constraint the default policy is *restrict-when-subtype-insertion*. Covering constraints have as default *restrict-when-supertype-insertion* and *restrict-when-subtype-deletion* depending on the type of change. Default policies for specialization constraints are *insert-when-subtype-insertion* and *delete-when-supertype-deletion*.

### 3.5 Application to an example

Consider the taxonomy of figure 3.1 that is part of a conceptual schema which represents information about people currently related with a company. People are first classified depending on whether they are employed by the company or not. Unemployed people include, among others, people who have applied for a job in the company. Employees may be temporary or permanent.

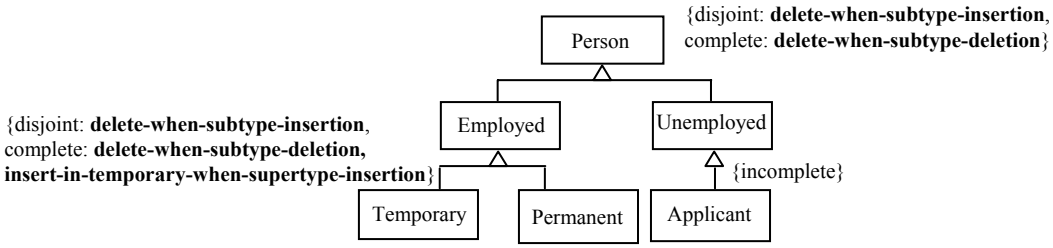



**Figure 3.1 Example of taxonomic constraint enforcement policies**

This taxonomy includes the specification of policies for all taxonomic constraints (default policies are omitted). Each policy is determined by the specific requirements of the company. The policy *insert-in-Temporary-when-supertype-insertion* for the complete constraint of generalization *Employed Gens Temporary, Permanent* is appropriate because employees usually are temporary when they are hired in the company. The policy *delete-when-subtype-insertion* for the disjoint constraint of generalization *Person Gens Employed, Unemployed* is adequate because usually when an unemployed person is inserted as employed we want to eliminate him as unemployed and vice versa. The rest of policies can be explained similarly.

These policies allow specifying taxonomic constraint enforcement in a declarative and localised way. Hence, as stated before, the definition of the external events becomes simpler because only its intended effect must be considered and taxonomic integrity enforcement actions or conditions to avoid them violating an integrity constraint may be omitted since they can be obtained from the declared taxonomic integrity enforcement policies.

In the following, we show this for a sample external event that violates several constraints of the given schema. We explain intuitively how to repair the external event according to the selected policies. The formal definition of this procedure is described in section 4.

Assume an IB at time *1* containing: *{Person(Pere,1), Person(Maria,1), Employed(Pere,1), Permanent(Pere,1), Unemployed(Maria,1), Applicant(Maria,1)}*. Consider now a *Substitution* external event occurring at time *2* with the intended effect of replacing an employee of the company (*Pere*) for another person (*Maria*) previously unemployed. Its definition consists of the following structural events: *{Insert_Employed(Maria,2), Delete_Employed(Pere,2)}*.

The *Substitution* external event induces the following violations:

1) The insertion of *Maria* as an employed person induces the violation of the covering constraint of the generalization *Employed Gens Temporary, Permanent* because *Maria* would be an employee neither temporary nor permanent. Then, according to the *insert-in-Temporary-when-supertype-insertion* declared policy, the insertion *Insert_Temporary(Maria,2)* must be included in the external event.

2) The insertion of *Maria* as an employed person also induces the violation of the disjointness constraint associated to *Person Gens Employed, Unemployed* because, after the insertion, *Maria* is both employed and unemployed. Due to the *delete-when-subtype-insertion* policy, the deletion *Delete_Unemployed(Maria,2)* must be added. In this case, the repair action of deleting Maria as unemployed induces itself a new violation:

2.1) The specialization constraint *Applicant IsA Unemployed* is violated since *Maria* is no more unemployed but is still an applicant. The policy to apply is *delete-when-supertype-deletion* (default policy), which establishes that the final external event must include *Delete_Applicant(Maria,2)*.

3) The deletion of *Pere* as employee induces the violation of the covering constraint associated to *Person Gens Employed, Unemployed* since, after the deletion, *Pere* is a person who is neither employed nor unemployed. According to the *delete-when-subtype-deletion* policy, the deletion *Delete_Person(Pere,2)* must be included in the external event.

4) The deletion of *Pere* as employee also induces the violation of the specialization constraint *Permanent IsA Employed*. According to the *delete-when-supertype-deletion* policy, the final external event must include *Delete_Permanent(Pere,2)*.

Summing up, the resulting external event will include the original external event together with the five repair actions determined by the selected policies: *{Insert_Employed(Maria,2), Delete_Employed(Pere,2), Insert_Temporary(Maria,2), Delete_Unemployed(Maria,2)*, *Delete_Applicant(Maria,2)*, *Delete_Person(Pere,2), Delete_Permanent(Pere,2)}*.

## 4. Formalizing taxonomic constraint enforcement policies

We provide a formal definition of the proposed taxonomic constraint enforcement policies that gives them precise semantics. This formalization establishes a logical basis for the automatic repair of events to preserve consistency. Section 4.1 describes the framework used. Sections 4.2, 4.3 and 4.4 present definitions of disjointness, covering and specialization constraint policies, respectively.

### 4.1 Framework for policies definition

**Enforced external events.** The set of policies chosen for the taxonomic constraints of a given structural conceptual schema must determine how to obtain, for any external event to be applied to its IB, either an indication of rejection of the event or its corresponding *enforced external event*. Intuitively, an *enforced external event* is an event that preserves the intended effect of its original event but which maintains consistency. Therefore, it consists of a set of structural events which necessarily includes all structural events belonging to its original external event in order to preserve all its changes and which may include additional structural events with the purpose of repairing integrity. Figure 4.1 illustrates that the transitions of the IB are always performed by means of enforced external events in our proposal.

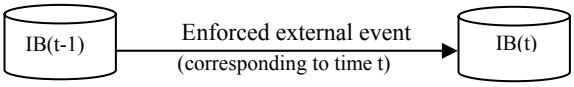



**Figure 4.1 Transition of the IB**

In some cases, an enforced external event cannot be obtained and the original event must be rejected. The reason may be that it violates a constraint for which an integrity checking policy has been selected or because it is irreparable. Irreparable situations

may arise when the (enforced) external event performs changes of the IB which are not jointly compatible. To indicate the rejection of the external event, we use an additional predicate *Abort*.

**States and transitions.** The impact of taxonomic constraint enforcement policies on enforced external events can be specified as declarative formulas by incorporating states into a logical framework as in Statelog [LLM98] or in Zaniolo's work [Zan93]. We take these approaches as a basis to define our policies.

We use states to describe the successive steps that, departing from an original external event and the IB to which it must be applied, obtain either an enforced external event or an indication of rejection.

Then, each transition from a state to its successor corresponds to the application of a single policy or to the detection of an irreparable situation. In case of integrity maintenance policies, the transition consists of the addition of a structural event to the initial set of events. In case of integrity checking policies or irreparable situations, the transition consists of the addition of the *Abort* fact. Intuitively, each intermediate state corresponds to the partial enforced external event obtained so far.

The state of a predicate is represented as a superscript; thus for example $Insert\_E^s(x,t)$ corresponds to *Insert_E(x,t)* at state *s*. The successor state of a state *s* is denoted by *s+1*. In general, state *s+1* contains facts existing at state *s* plus the fact added in the transition.

According to this framework, it is possible to define integrity maintenance policies by means of declarative formulas. Each formula describes a transition from a state *s* to its successor state *s+1* and represents either the application of a maintenance policy towards obtaining the enforced external event or the reaching of an indication of rejection. More specifically, the antecedent of the formula should describe the conditions at state *s* that must hold to apply the policy, and the consequent should describe its effect at state *s+1*. Figure 4.2 illustrates these transitions, where $Sef_i^{S_i}$ stands for a structural event fact added in a transition.

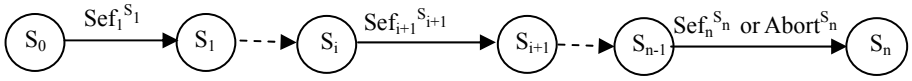



**Figure 4.2 Transitions to obtain the enforced external event or the rejection**

The initial state $s_0$ represents the starting point for the application of a selected set of policies to transform a given external event (which occurs at time *t* and that must be applied to the IB existing at time *t-1*) into its corresponding enforced external event. Therefore, the initial state consists of the set of structural events superscripted by $s_0$ from the original external event that occurs at time *t*, and the set of facts superscripted by $s_0$ that belong to the IB at *t-1*.

The final state $s_n$ is obtained when an indication of rejection is added or when no more policies can be applied, i.e. a fixpoint is reached for the set of formulas. In the last case, it is composed by a set of structural events superscripted by $s_n$ that includes all events from the original external event and the repair structural events added in the transitions; and the set of facts superscripted by $s_n$ that belong to the IB at time *t-1*.

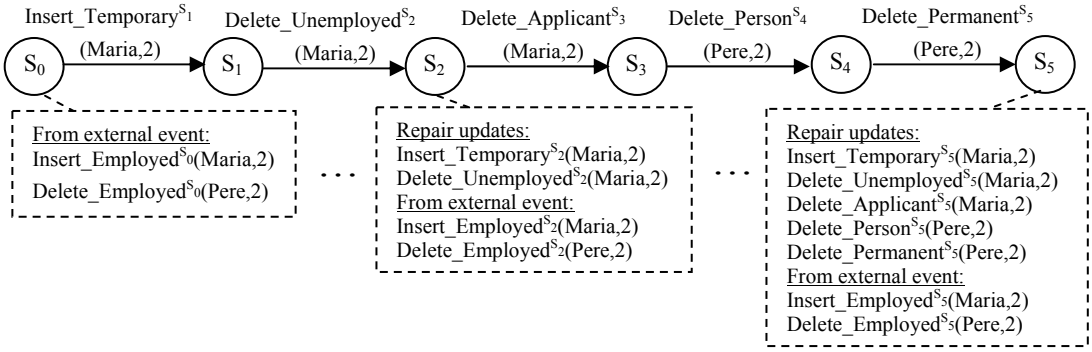



**Figure 4.3 Example of obtaining an enforced external event**

Then, the enforced external event is the set of structural events at the final state without the state-superscript. Figure 4.3 shows transitions and some states of the example presented in section 3.5. We have omitted facts from the IB at time *t-1* in the states of the figure. Appendix 1 gives part of the formulas of this example.

### 4.2 Policies for disjointness constraints

Given a disjointness constraint *d*, we use *Delete-when-subtype-insertion(d)* to denote the policy *delete-when-subtype-insertion* corresponding to *d*.

**Definition 1.** (*Delete-when-subtype-insertion* policy for disjointness). Let E Gens $E_1,...,E_i,...,E_j,...,E_n$ be a generalization and $D=Disjoint(E_1,...,E_i,...,E_j,...,E_n)$ a disjointness constraint such that *Delete-when-subtype-insertion(D)* holds. The application of this policy is defined by a set of formulas of the form:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge (\neg Insert\_E_1^s(x,t) \wedge ... \wedge \neg Insert\_E_{i-1}^s(x,t) \wedge \neg Insert\_E_{i+1}^s(x,t) \wedge ... \wedge \neg Insert\_E_n^s(x,t)) \wedge E_j^s(x,t\text{-}1) \wedge \neg Delete\_E_j^s(x,t) \rightarrow Delete\_E_j^{s+1}(x,t))$$

where $E_i$, $E_j$ are any pair of subtypes of *E* for the given generalization.

This set of formulas implies that disjointness constraint violation caused by a single event $Insert\_E_i(x,t)$ is repaired by the structural event $Delete\_E_j(x,t)$. For the example of section 3.5, one of the formulas obtained from this definition is: $\forall x,t\ (Insert\_Employed^s(x,t) \wedge \neg Insert\_Unemployed^s(x,t) \wedge Unemployed^s(x,t\text{-}1) \wedge \neg Delete\_Unemployed^s(x,t) \rightarrow Delete\_Unemployed^{s+1}(x,t))$, which repairs the disjointness violation induced by *Insert_Employed(Maria,2)* with the addition of *Delete_Unemployed(Maria,2)*.

Note that the previous set of formulas does not deal with external events that include the insertion of an entity in two or more subtypes that are disjoint. Obviously, those external events cannot be repaired without undoing their intended effect and must be rejected. This situation is handled by definition 2.

**Definition 2.** (Irreparable disjointness constraint). Let *E Gens* $E_1,...,E_i,...,E_j,...,E_n$ be a generalization and $D=Disjoint(E_1,...,E_i,...,E_j,...,E_n)$ a disjointness constraint. The following set of formulas defines the external event rejection for an irreparable disjointness constraint:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge Insert\_E_j^s(x,t) \rightarrow Abort^{s+1})$$

where $E_i$, $E_j$ are any pair of subtypes of *E* for the given generalization.

We use predicate *Restrict-when-subtype-insertion(d)* to denote the policy *restrict-when-subtype-insertion* corresponding to *d*.

**Definition 3.** (*Restrict-when-subtype-insertion* policy for disjointness). Let E Gens $E_1,...,E_i,...,E_j,...,E_n$ be a generalization and $D=Disjoint(E_1,...,E_i,...,E_j,...,E_n)$ a disjointness constraint such that *Restrict-when-subtype-insertion(D)* holds. The application of this policy is defined by a set of formulas of the form:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge (\neg Insert\_E_1^s(x,t) \wedge ... \wedge \neg Insert\_E_{i-1}^s(x,t) \wedge \neg Insert\_E_{i+1}^s(x,t) \wedge ... \wedge \neg Insert\_E_n^s(x,t)) \wedge E_j^s(x,t-1) \wedge \neg Delete\_E_j^s(x,t) \rightarrow Abort^{s+1})$$

where $E_i$, $E_j$ are any pair of subtypes of *E* for the given generalization.

Observe that this set of formulas implies that the situation of disjointness violation due to an event $Insert\_E_i(x,t)$ causes the rejection of the external event.

### 4.3 Policies for covering constraints

We use predicate *Insert-when-supertype-insertion(c,$e_i$)*, where *c* is a covering constraint $Covering(E,E_1,...,E_i,...,E_n)$ and $e_i$ is one of the subtypes $E_1,...,E_i,...,E_n$, to denote the policy *insert-in-$E_i$-when-supertype-insertion* corresponding to *c*.

**Definition 4.** (*Insert-in-$E_i$-when-supertype-insertion* policy for covering). Let *E Gens* $E_1,...,E_i,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_i,...,E_n)$ a covering constraint such that *Insert-when-supertype-insertion(C,$E_i$)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Insert\_E^s(x,t) \wedge (\neg Insert\_E_1^s(x,t) \wedge ... \wedge \neg Insert\_E_n^s(x,t)) \rightarrow Insert\_E_i^{s+1}(x,t))$$

It establishes that a covering violation caused by event *Insert_E(x,t)* is repaired by adding the structural event $Insert\_E_i(x,t)$ to the enforced external event. For the example of section 3.5, the application of this definition implies that the covering constraint violation induced by *Insert_Employed(Maria,2)* is repaired with the addition of *Insert_Temporary(Maria,2)*.

We use predicate *Restrict-when-supertype-insertion(c)* where *c* is a covering constraint to denote the policy *restrict-when-supertype-insertion* corresponding to *c*.

**Definition 5.** (*Restrict-when-supertype-insertion* policy for covering). Let *E Gens* $E_1,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_n)$ a covering constraint such that *Restrict-when-supertype-insertion(C)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Insert\_E^s(x,t) \wedge (\neg Insert\_E_1^s(x,t) \wedge ... \wedge \neg Insert\_E_n^s(x,t)) \rightarrow Abort^{s+1})$$

We use predicate *Insert-when-subtype-deletion(c,$e_i$)*, where *c* is a covering constraint $Covering(E,E_1,...,E_i,...,E_n)$ and $e_i$ is one of the subtypes $E_1,...,E_i,...,E_n$, to denote the policy *insert-in-$E_i$-when-subtype-deletion* corresponding to *c*.

**Definition 6.** (*Insert-in-$E_i$-when-subtype-deletion* policy for covering). Let *E Gens* $E_1,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_i,...,E_n)$ a covering constraint such that *Insert-when-subtype-deletion(C, $E_i$)* holds. The application of this policy is defined by a set of formulas of the form:

$$\forall x,t\ (Delete\_E_{i_1}^s(x,t) \wedge ... \wedge Delete\_E_{i_k}^s(x,t) \wedge \neg Delete\_E^s(x,t) \wedge (\neg E_{i_{k+1}}^s(x,t\text{-}1) \wedge ... \wedge \neg E_{i_n}^s(x,t\text{-}1)) \wedge (\neg Insert\_E_{i_{k+1}}^s(x,t) \wedge ... \wedge \neg Insert\_E_{i_n}^s(x,t)) \wedge \neg Delete\_E_i^s(x,t) \rightarrow Insert\_E_i^{s+1}(x,t))$$

where in each formula $1 \leq k \leq n$ and where $E_{i_1},...,E_{i_n}$ is any permutation of $E_1,...,E_n$.

The set of formulas implies that a covering violation caused by the deletion of an entity from all subtypes to which it belonged is repaired by adding an structural event *Insert_E*$_i$*(x,t)* to the enforced external event. Note that *k* corresponds to the number of subtypes from which the entity is deleted. If the involved generalization is disjoint previous formulas can be simplified as explained in [CGT05].

Observe that literal $\neg Delete\_E_i^s(x,t)$ guarantees that $E_j$ is not one of the subtypes from which the entity has been deleted because this leads to an irreparable situation since the repair action would undo the intended effect of the external event. Definition 7 corresponds to this case.

**Definition 7.** (Irreparable covering constraint). Let *E Gens* $E_1,...,E_i,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_i,...,E_n)$ a covering constraint such that *Insert-when-subtype-deletion(C, E*$_i$*)* holds. The following set of formulas defines the external event rejection for an irreparable covering constraint:

$$\forall x,t\ (Delete\_E_{i_1}^s(x,t) \wedge ... \wedge Delete\_E_{i_k}^s(x,t) \wedge \neg Delete\_E^s(x,t) \wedge (\neg E_{i_{k+1}}^s(x,t\text{-}1) \wedge ... \wedge \neg E_{i_n}^s(x,t\text{-}1)) \wedge (\neg Insert\_E_{i_{k+1}}^s(x,t) \wedge ... \wedge \neg Insert\_E_{i_n}^s(x,t)) \wedge Delete\_E_i^s(x,t) \rightarrow Abort^{s+1})$$

where in each formula $1 \leq k \leq n$ and where $E_{i_1},...,E_{i_n}$ is any permutation of $E_1,...,E_i,...,E_n$.

We use predicate *Delete-when-subtype-deletion(c)*, where *c* is a covering constraint *Covering(E,E*$_1$*,...,E*$_n$*)*, to denote the policy *Delete-when-subtype-deletion* corresponding to *c*.

**Definition 8.** (*Delete-when-subtype-deletion* policy for covering). Let *E Gens* $E_1,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_n)$ a covering constraint such that *Delete-when-subtype-deletion(C)* holds. The application of this policy is defined by a set of formulas of the form:

$$\forall x,t\ (Delete\_E_{i_1}^s(x,t) \wedge ... \wedge Delete\_E_{i_k}^s(x,t) \wedge \neg Delete\_E^s(x,t) \wedge (\neg E_{i_{k+1}}^s(x,t\text{-}1) \wedge ... \wedge \neg E_{i_n}^s(x,t\text{-}1)) \wedge (\neg Insert\_E_{i_{k+1}}^s(x,t) \wedge ... \wedge \neg Insert\_E_{i_n}^s(x,t)) \rightarrow Delete\_E^{s+1}(x,t))$$

where in each formula $1 \leq k \leq n$ and where $E_{i_1},...,E_{i_n}$ is any permutation of $E_1,...,E_n$.

These formulas imply that a covering violation caused by the deletion of an entity from all subtypes to which it belonged is repaired by adding a structural event *Delete_E(x,t)* to the enforced external event. Following them, in the example of section 3.5, the covering constraint violation induced by *Delete_Employed(Pere,2)* is repaired with the addition of *Delete_Person (Pere,2)*.

We use predicate *Restrict-when-subtype-deletion(c)* where *c* is a covering constraint to denote the policy *restrict-when-subtype-deletion* corresponding to *c*.

**Definition 9.** (*Restrict-when-subtype-deletion* policy for covering). Let *E Gens* $E_1,...,E_n$ be a generalization and $C=Covering(E,E_1,...,E_n)$ a covering constraint such that *Restrict-when-subtype-deletion(C)* holds. The application of this policy is defined by a set of formulas of the form:

$$\forall x,t\ (Delete\_E_{i_1}^s(x,t) \wedge ... \wedge Delete\_E_{i_k}^s(x,t) \wedge \neg Delete\_E^s(x,t) \wedge (\neg E_{i_{k+1}}^s(x,t\text{-}1) \wedge ... \wedge \neg E_{i_n}^s(x,t\text{-}1)) \wedge (\neg Insert\_E_{i_{k+1}}^s(x,t) \wedge ... \wedge \neg Insert\_E_{i_n}^s(x,t)) \rightarrow Abort^{s+1})$$

where in each formula $1 \leq k \leq n$ and where $E_{i_1},...,E_{i_n}$ is any permutation of $E_1,...,E_n$.

### 4.3 Policies for specialization constraints

We use predicate *Insert-when-subtype-insertion(s)* where *s* is a specialization relationship to denote the policy *insert-when-subtype-insertion* corresponding to *s*.

**Definition 10.** (*Insert-when-subtype-insertion* policy for specialization). Let $S=E_i$ *IsA E* be a specialization relationship such that *Insert-when-subtype-insertion(S)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge (\neg E^s(x,t\text{-}1) \wedge \neg Insert\_E^s(x,t)) \rightarrow Insert\_E^{s+1}(x,t))$$

It implies that the specialization constraint violation caused by event $Insert\_E_i(x,t)$ is repaired by adding the structural event *Insert_E(x,t)* to the enforced external event.

We use predicate *Restrict-when-subtype-insertion(s)* where *s* is a specialization relationship to denote the policy *restrict-when-subtype-insertion* corresponding to *s*.

**Definition 11.** (*Restrict-when-subtype-insertion* policy for specialization). Let $S=E_i$ *IsA E* be a specialization relationship such that *Restrict-when-subtype-insertion(S)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge (\neg E^s(x,t\text{-}1) \wedge \neg Insert\_E^s(x,t)) \rightarrow Abort^{s+1})$$

We use predicate *Delete-when-supertype-deletion(s)* where *s* is a specialization relationship to denote the policy *delete-when-supertype-deletion* corresponding to *s*.

**Definition 12.** (*Delete-when-supertype-deletion* policy for specialization). Let $S=E_i$ *IsA E* be a specialization relationship such that *Delete-when-supertype-deletion(S)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Delete\_E^s(x,t) \wedge (E_i^s(x,t\text{-}1) \wedge \neg Delete\_E_i^s(x,t)) \rightarrow Delete\_E_i^{s+1}(x,t))$$

It implies that the specialization constraint violation caused by event *Delete_E(x,t)* is repaired by adding event $Delete\_E_i(x,t)$ to the enforced external event.

We use predicate *Restrict-when-supertype-deletion(s)* where *s* is a specialization relationship to denote the policy *restrict-when-supertype-deletion* corresponding to *s*.

**Definition 13.** (*Restrict-when-supertype-deletion* policy for specialization). Let $S=E_i$ *IsA E* be a specialization relationship such that *Restrict-when-supertype-deletion(S)* holds. The application of this policy is defined by the following formula:

$$\forall x,t\ (Delete\_E^s(x,t) \wedge (E_i^s(x,t\text{-}1) \wedge \neg Delete\_E_i^s(x,t)) \rightarrow Abort^{s+1})$$

Specialization constraints are irreparable for external events which imply both an insertion of an entity in a subtype and a deletion of the same entity in its supertype. Obviously, this external event cannot be repaired without undoing its intended effect.

**Definition 14.** (Irreparable specialization constraint). Let $S=E_i$ IsA E be a specialization relationship. The following formula defines the external event rejection for an irreparable specialization constraint:

$$\forall x,t\ (Insert\_E_i^s(x,t) \wedge Delete\_E^s(x,t) \rightarrow Abort^{s+1})$$

## 5. Termination

As stated before, we use states to describe the successive steps for obtaining either an enforced external event or an indication of external event rejection (see figure 4.2). The transition from a state to its successor is defined by formulas presented in section

4. The final state is obtained when an indication of rejection is added or when no more policies can be applied, i.e. a fixpoint is reached for the set of formulas.

When formulas with a consequent part including a repair action to add to the enforced external event are applied, it may happen that the repair action induces a new constraint violation, and consequently, a new application of a formula. This situation might be cyclic and then a fixpoint (and the final state) would not be reached in a finite number of steps.

In the following, we prove that the above situation does not occur to guarantee that the application of our proposed set of formulas to obtain an enforced external event always terminates. Intuitively, the main idea of our proof is to show that given an entity that violates some taxonomic constraint, the repair actions we perform do not involve other entities of the same taxonomy (lemma1). Then, for each entity, we have to perform at most as much repairs as entity types exist. Clearly, termination is guaranteed if as usual the number of entity types in the conceptual schema is finite (theorem 1).

**Lemma 1.** (Termination for single instance external events). Let *S* be an external event composed by *s* structural events all of them corresponding to a particular instance *x* and to a time instant *t*. The application of the set of formulas that define taxonomic integrity constraint enforcement policies and that permit to obtain the enforced external event corresponding to *S*, that is, *EEE(S),* terminates if the number of entity types of the structural conceptual schema to which S must be applied is finite.

**Proof:**

Step 1) All structural event facts added to *EEE(S)* when a transition from a state to its successor is performed due to a formula application, correspond to instance *x* because all formulas are universally quantified over *x* and *t* and they do not include existential quantifiers.

Step 2) Two or more structural event facts that perform the insertion of *x* in the same entity type cannot be added to *EEE(S)* due to the application of the set of formulas. This situation is either explicitly prevented by our formulas which ensure that the same event fact does not hold in the previous state or implicitly because of the applicability conditions of the event (see section 2.2).

Step 3) For the same reasons as in step 2, two or more structural event facts that perform the deletion of *x* from the same entity type cannot be added to *EEE(S)* due to the application of the set of formulas.

Step 4) The number of insertion structural event facts of *x* that are added to *EEE(S)* corresponding to different entity types is a finite number *i* if the number of entity types of the conceptual schema is finite.

Step 5) Step 5 is similar to step 4 for deletion structural event facts. The number of deletion structural event facts of *x* that are added to *EEE(S)* corresponding to different entity types is a finite number *d* if the number of entity types of the conceptual schema is finite.

Finally, from steps 1, 4 and 5 we can conclude that the number of facts that are added to *EEE(S)* is the finite number *i*+*d*. From steps 2 and 3 we can conclude that each fact is added to *EEE(S)* due to a single application of a formula. Therefore, the application of the set of formulas that obtains *EEE(S)* terminates.

**Theorem 1.** (Termination for general external events). Let *S* be any external event composed by *s* structural events corresponding to a time instant *t*. The application of the set of formulas that define taxonomic integrity constraint enforcement policies and that permit to obtain the enforced external event corresponding to *S*, that is, *EEE(S),* terminates if the number of entity types of the structural conceptual schema to which *S* must be applied is finite.

**Proof:**

Step 1) If the number of structural event facts that compose *S* is a finite number *s* then the number of different instances that appear in *S* is a finite number *n*.

Step2) A structural event fact corresponding to instance *x* cannot induce the application of a formula which adds a structural event fact corresponding to an instance *y* such that $y<>x$ because all formulas are universally quantified over *x* and *t* and they do not include existential quantifiers.

Step 3) Let $S_1,...,S_j,...,S_k,...,S_n$ be disjoint subsets of *S* such that each one includes structural event facts corresponding to a single instance and additionally the instance corresponding to $S_j$ is different from the instance corresponding to $S_k$ for all *j*, *k* such that $j<>k$. Then, from step 2, *EEE(S)* is the union $EEE(S_1)\ U...U\ EEE(S_j)\ U\ ...\ U\ EEE(S_k)\ U\ ...U\ EEE(S_n)$.

Finally, from lemma 1 the application of the set of formulas that permit to obtain $EEE(S_1),...,EEE(S_j),\ ...\ ,EEE(S_k),\ ...\ ,EEE(S_n)$ terminates and from step 1 *n* is finite. Then from step 3 we can conclude that the application of the set of formulas that permits to obtain *EEE(S)* terminates.

## 6. Implementing taxonomic constraint enforcement policies

The implementation of the taxonomic constraint enforcement policies of a given conceptual schema performs the generation of enforced external events for any input external event and IB to which the external event must be applied. Figure 6.1 shows the input data and the output generated by the implementation.

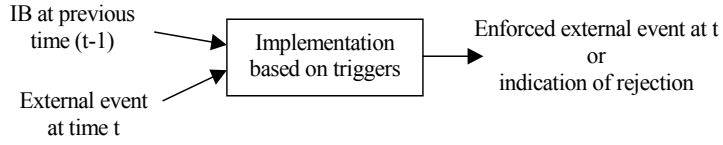



**Figure 6.1 Implementation**

We propose an implementation based on SQL:1999 standard triggers [MS02]. Triggers specify certain types of active rules and, as such, they allow specifying actions to be executed upon occurrence of a triggering event provided that a condition holds. The main components of a trigger definition are: the *triggering event*, the *condition* that must hold and the *action* to execute. Triggers are adequate to implement taxonomic constraint enforcement policies because policy definitions can be translated into them in a direct way.

Each formula defined in section 4 is implemented by a set of triggers which can be automatically obtained from the formula. Roughly, for each structural event that

appears as a positive literal in the antecedent of the formula, there is a trigger with the following components: 1) a triggering event that corresponds to the mentioned structural event, 2) a condition that corresponds to the rest of the antecedent of the formula expressed as an SQL search condition, and 3) an action that corresponds to the consequent of the formula. We call taxonomic constraint enforcement triggers (TCE triggers), the triggers obtained.

Section 6.1 describes relational tables used to store data managed by TCE triggers and the transaction that activates them and section 6.2 describes TCE triggers themselves.

### 6.1 Relational tables

The following kinds of tables store data used by TCE triggers: IB tables, structural event tables, triggering tables and an abort table.

IB tables and structural event tables store IB facts at time *t-1* and structural event facts at time *t*, respectively. There is an IB table for each predicate of the IB and a structural event table for each structural event predicate. All of them have two columns named 'entity' and 'time', to store the two terms of the predicates. For example, for the IB predicate *Person*, we have table: `person(entity, time)` and for the structural event predicate *Delete_Person* we have table: `delete_person(entity, time).`

The content of IB tables and structural event tables is used to evaluate the condition part of the TCE triggers. IB facts at *t-1* are not altered by the TCE triggers actions and, thus, IB tables remain unchanged during the whole trigger processing. On the other hand, structural event tables contain, initially, the structural event facts of the external event. During TCE trigger processing, as TCE trigger actions may add new structural event facts to repair consistency, they contain the structural event facts of the enforced external event generated so far. At the end of the process, they contain the whole enforced external event provided that an indication of rejection has not been generated.

Triggering tables are similar to structural event tables but their content and use is different as will be seen in next paragraphs. For each structural event table there is a triggering table with the same name prefixed by 'triggering' and with the same columns. For example, for table `delete_person(entity, time)` we have `triggering_delete_person(entity, time).`

Triggering tables are used to fire TCE triggers i.e. TCE triggers are activated by insertions in triggering tables. They are initially empty. During TCE trigger processing they contain the structural event facts of the enforced external event that have already been processed or that are being processed by the trigger mechanism. Therefore, facts stored by triggering tables are always a subset of facts stored by structural event tables. At the end of the process, the content of triggering tables and structural event tables is the same provided an indication of rejection has not been generated. Note that structural event tables are used to evaluate conditions of TCE triggers, thus, they must keep track of all known structural event facts, even those that have not already been processed and do not belong to triggering tables. Consequently, neither structural event tables nor triggering tables can be omitted.

Finally, a one-column table `abort(indication)` is needed to store the possible indication of rejection that may be generated by the constraint enforcement process. Therefore, the `abort` table is initially empty and, after the TCE trigger processing, it may contain a value `'yes'` which indicates the rejection of the external event.

As TCE triggers are activated by insertions in triggering tables, we need a transaction that performs the adequate insertions to activate them. This transaction has an insertion in a triggering table for each structural event fact appearing in the external event. For example, for the following external event: *{Insert_Employed(Maria,2), Delete_Employed(Pere,2)}*, a transaction with the following statements performs the necessary insertions:

```
INSERT INTO triggering_insert_employed
            VALUES ('Maria',2);
INSERT INTO triggering_delete_employed
            VALUES ('Pere',2);
```

Before the execution of the transaction, structural event tables `insert_employed` and `delete_employed` contain `('Maria',2)` and `('Pere',2)`, respectively. When the transaction performs the first statement which inserts `('Maria',2)` in `triggering_insert_employed`, the adequate triggers are activated. To evaluate the condition part of those triggers, it is necessary to query structural event tables which have all known structural events. Note that triggering tables could not be used for this purpose because they still do not have `('Pere',2)` in `triggering_delete_employed`.

Observe that triggering events for TCE triggers are only insertions (not deletions or updates).

The processing of previous transaction fires the adequate TCE triggers which, in turn, generate an enforced external event or an indication of rejection.

### 6.2 TCE triggers

In the following paragraphs, we explain the details of the translation of our formulas into TCE triggers with the help of an example.

Consider the example of figure 3.1. For the disjointness constraint of *Person Gens Employed, Unemployed*, we have a *delete-when-supertype-insertion* policy. By definition 1 from section 4.2, one of the formulas that defines the application of this policy is:

$$\forall x,t\ (Insert\_Employed^s(x,t) \wedge \neg Insert\_Unemployed^s(x,t) \wedge Unemployed^s(x,t\text{-}1) \wedge \neg Delete\_Unemployed^s(x,t) \rightarrow Delete\_Unemployed^{s+1}(x,t))$$

As said before, there is a TCE trigger for each structural event that appears as a positive literal in the antecedent of the formula. The reason is that only positive structural event literals may make the antecedent hold because both external events and consequents of formulas consist only of positive structural event literals. In our example, there is a single literal of this type: $Insert\_Employed^s(x,t)$. Therefore, the formula is translated into a single trigger. The definition of this trigger is shown in figure 6.2. The rest of the triggers for a subset of the figure 3.1 example can be found in the appendix 2.

```
CREATE TRIGGER t1
AFTER INSERT ON triggering_insert_employed
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM insert_unemployed iu
                   WHERE iu.entity=nr.entity AND iu.time=nr.time)
    AND EXISTS (SELECT * FROM unemployed u
             WHERE u.entity=nr.entity AND u.time=nr.time – 1)
    AND NOT EXISTS ( SELECT * FROM delete_unemployed du
                   WHERE du.entity=nr.entity AND du.time=nr.time)
BEGIN ATOMIC
INSERT INTO delete_unemployed VALUES (nr.entity, nr.time);
INSERT INTO triggering_delete_unemployed VALUES (nr.entity, nr.time);
END;
```

**Figure 6.2 Example of a TCE trigger**

The triggering event is specified following the `AFTER` keyword. It is an insertion in the triggering table `triggering_pred` being *pred* the predicate of the positive literal that has originated the trigger. In our example, this predicate is *Insert_Employed*, thus, the translation obtains a triggering event consisting of the insertion in table `triggering_insert_employed`.

The condition of the trigger is specified following the `WHEN` keyword and is obtained from the antecedent of the formula not including the literal that has originated the trigger. It consists of an SQL search condition that evaluates this part of the antecedent in terms of the tables that implement its predicates.

More concretely, each positive/negative literal is translated into a subquery preceded by `EXISTS/NOT EXISTS`. The subquery has a `FROM` clause with the table that implements the predicate of the literal. Its `WHERE` clause has join conditions which relates that table with the row inserted by the triggering event. Translations of various literals are connected by the `AND` keyword.

In our example, the antecedent part of the formula to consider is: $\neg Insert\_Unemployed^{s}(x,t) \wedge Unemployed^{s}(x,t\text{-}1) \wedge \neg Delete\_Unemployed^{s}(x,t)$. For the first literal $\neg Insert\_Unemployed^{s}(x,t)$, we have a subquery preceded by `NOT EXISTS` with table `insert_unemployed` in the `FROM` clause and with join conditions which relates `insert_unemployed` and the inserted row. The rest of the literals of the example have been translated similarly.

The action is specified between `BEGIN ATOMIC` and `END` keywords. It corresponds to the consequent of the formula. When the consequent is $Abort^{s+1}$ the action is an insertion of value `'yes'` in the `abort` table. Otherwise, if the consequent has the form $Pred^{s+1}(x,t)$, the action consists of two insertions: an insertion in the structural event table `pred` and an insertion in the triggering table `triggering_pred`. In both cases, the values inserted coincide with the values inserted by the triggering event. The insertion in the structural event table is needed to take the new fact into account in the evaluation of the condition of subsequent triggers while the insertion in the triggering table is needed to fire TCE triggers with that insertion as triggering event.

In the example, the predicate of the consequent of the formula is *Delete_Unemployed*. Then, the action consists of insertions in table `delete_unemployed` and table `triggering_delete_unemployed`.

Observe that, although an action of a TCE trigger may fire new triggers, this process terminates as shown in section 5.

## 7. Related Work

Declarative constraint enforcement has received a lot of attention in the database field. For relational databases, the SQL standard supports declarative enforcement specification for referential integrity constraints [MS02]. Moreover, [BCP94,Ger94] offer several repair strategies, based on the SQL standard, for a broad spectrum of constraints expressed in a logic based language. In the context of object oriented databases, [BG98] proposes to extend the ODMG Object Model to specify declaratively referential and composite objects constraint enforcement. [ED98] describes high-level abstractions called stabilizer types that define how a database can be made consistent when an update exception occurs.

Other works as [CW90,UD90,CFPT94] in the context of databases propose alternative approaches which emphasize more the automatic generation of reactions than the declarative specification of reactions associated to constraints.

Following CSCD, we are convinced that declarative constraint enforcement must be necessarily provided for conceptual modelling and, thus, at the early stages of information systems development and in a technological independent way. However, as far as we know, declarative constraint enforcement has mainly been addressed at a database level. An exception is [ST99] where constraint enforcement is studied for formal state oriented specifications. The basic idea of this work is to replace inconsistent operations by new consistent ones preserving at the same time its intention. Nevertheless, it does not address the problem of providing policies for integrity enforcement.

This paper differs from most of that work in several aspects: (1) we specify taxonomic constraint enforcement at a conceptual level; (2) we define declarative integrity enforcement specific for taxonomic constraints; (3) we generate events that maintain consistency from inconsistent events, preserving at the same time its intended effect; and (4) the automatic generation of events that preserves consistency terminates.

## 8. Conclusions and future work

We have proposed a set of predetermined policies for the enforcement of taxonomic integrity constraints. These policies are declaratively specified in the structural part of the conceptual schema by stating the kind of enforcement to apply when a certain type of structural event violates a taxonomic constraint.

Our approach facilitates conceptual modelling since taxonomic constraint enforcement policies must not be spread along all the events that may violate an integrity constraint. Moreover, it has a positive impact on the understandability and modifiability of the conceptual schemas. In this way, we overcome the limitations of previous proposals for the specification of taxonomic constraint enforcement.

Another important contribution of our work has been to provide a formal definition of each policy in terms of the repair actions they entail or the conditions that need to be checked to guarantee that the constraint is not violated. We have shown how to generate enforced external events that preserve consistency according to our formalization and we have proved that the application of our taxonomic constraint enforcement policies always terminates.

Finally, we have also explained how our taxonomic constraint enforcement policies may be implemented by means of SQL:1999 standard triggers.

Further work may include extending our approach to the enforcement of other types of integrity constraints. We could also define a UML Profile for taxonomic constraint policies with the aim of allowing the specification of our policies in a structural conceptual schema in UML. In the same direction, we may extend a CASE tool to incorporate the mentioned profile.

## References


[BCP94] Baralis, E.; Ceri, S.; Paraboschi, S. “Declarative Specification of Constraint Maintenance”, In Proc. ER 1994, LNCS 881, pp. 205-222.

[BG98] Bertino, E.; Guerrini, G. “Extending the ODMG Object Model with Composite Objects”, In Proc. of ACM SIGPLAN, 1998, pp. 259-270.

[Bub77] Bubenko, J.A. “The Temporal Dimension in Information Modelling” In Architecture and Models in Data Base Management Systems, North-Holland, 1977, pp. 93-113.

[CFPT94] Ceri, S.; Fraternali, P.; Paraboschi, S.; Tanca, L. “Automatic Generation of Production Rules for Integrity Maintenance”, In ACM Transactions on Database Systems, vol. 19, no. 3, 1994, pp. 367-422.

[COS97] Costal, D.;Olivé, A.; Sancho, M.R ”Temporal Features of Class Populations and Attributes in Conceptual Models”, In Proc. ER 1997, LNCS 1331, pp. 57-70.

[CW90] Ceri, S.;Widom, J. ”Deriving Production Rules for Constraint Maintenance”, In Proc. VLDB 1990, pp. 566-577.

[ED98] Etzion, O.; Dahav, B. ”Patterns of self-stabilization in database consistency maintenance”, In DKE 1998, pp. 299-319.

[Ger94] Gertz, M. “Specifying reactive integrity control for active databases”, In Proc. of the IEEE RIDE-ADS, 1994, pp.62-70.

[Hal01] Halpin, T. Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design, Morgan Kaufmann, 2001.

[Len87] Lenzerini, M. "Covering and Disjointness Constraints in Type Networks” In Proc. ICDE’87, IEEE Computer Society Press, 1987, pp. 386-393.

[LLM98] Lausen, G.; Ludäscher, B.; May, W. "On Logical Foundations of Active Databases” In Logics for Databases and Information Systems, Kluwer Academic Publishers, 1998, pp. 389-422.

[MS02] Melton, J.; Simon, A.R. "SQL:1999. Understanding Relational Language Components”, Morgan Kaufmann, 2002.

[Oli00] Olivé, A. "An Introduction to Conceptual Modeling of Information Systems". In Advanced Database Technology and Design, (Piattini, M.; Diaz, O. Eds.), Artech House, 2000, pp. 25-57

[Oli05] Olivé, A. “Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research”, In Proc. CAiSE’05, LNCS 3520, pp.1-15.

[OMG03] OMG. “UML2.0 Superstructure Specification”, OMG Adopted Specification, 2003.

[OT02] Olivé, A.; Teniente, E. “Derived types and taxonomic constraints in conceptual modeling”, In Information Systems, vol. 27, no. 6, Sept. 2002, pp.365-389.

[ST99] Schewe, K.D.; Thalheim, B. “Towards a theory of consistency enforcement”, In Acta Informatica, vol. 36, 1999, pp. 97-141.

[UD90] Urban, S.D.; Delcambre, L.M.L. “Constraint Analysis: A Design Process for Specifying Operations on Objects”, In IEEE Transactions on Knowledge and Data Engineering, vol. 2, no. 4, December 1990, pp. 391- 400.

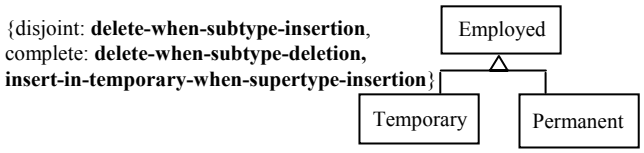
[VF85] Veloso, P.A.S.; Furtado, A.L. “Towards simpler and yet complete formal specifications”, In Information Systems: Theoretical and Formal Aspects, North Holland, pp. 175-190.

[WC96] Widom, J.; Ceri, S. Active Database Systems: Triggers and Rules for Advanced Database Processing, Morgan Kaufmann, 1996

[Zan93] Zaniolo, C. “A Unified Semantics for Active and Deductive Databases”, In Proc. RIDS’93, Springer, pp.271-287.

## Appendix 1

This appendix includes the set of formulas that define policies and irreparable situations of the part of the example of figure 3.1 illustrated by the following figure. All these formulas have been obtained according to the definitions presented in section 4.




The set of formulas that defines the *delete-when-subtype-insertion* policy declared for the disjointness constraint of *Employed Gens Temporary, Permanent* is (by definition 1):

(1) $\forall x,t\ (Insert\_Temporary^{s}(x,t) \wedge \neg Insert\_Permanent^{s}(x,t) \wedge Permanent^{s}(x,t\text{-}1) \wedge \neg Delete\_Permanent^{s}(x,t) \rightarrow Delete\_Permanent^{s+1}(x,t))$

and:

(2) $\forall x,t\ (Insert\_Permanent^{s}(x,t) \wedge \neg Insert\_Temporary^{s}(x,t) \wedge Temporary^{s}(x,t\text{-}1) \wedge \neg Delete\_Temporary^{s}(x,t) \rightarrow Delete\_Temporary^{s+1}(x,t))$

The formula that defines irreparable situations for previous disjointness constraint is (by definition 2):

(3) $\forall x,t\ (Insert\_Temporary^{s}(x,t) \wedge Insert\_Permanent^{s}(x,t) \rightarrow Abort^{s+1})$

The set of formulas corresponding to the *delete-when-subtype-deletion* policy declared for the covering constraint of *Employed Gens Temporary, Permanent*, is (according to definition 8):

(4) $\forall x,t\ (Delete\_Temporary^{s}(x,t) \wedge \neg Delete\_Employed^{s}(x,t) \wedge \neg Insert\_Permanent^{s}(x,t) \rightarrow Delete\_Employed^{s+1}(x,t))$

and:

(5) $\forall x,t\ (Delete\_Permanent^{s}(x,t) \wedge \neg Delete\_Employed^{s}(x,t) \wedge \neg Insert\_Temporary^{s}(x,t) \rightarrow Delete\_Employed^{s+1}(x,t))$

The formula corresponding to the *insert-in-temporary-when-supertype-insertion* policy declared for the covering constraint of *Employed Gens Temporary, Permanent*, is (according to definition 4):

(6) $\forall x,t\ (Insert\_Employed^{s}(x,t) \wedge \neg Insert\_Temporary^{s}(x,t) \wedge \neg Insert\_Permanent^{s}(x,t) \rightarrow Insert\_Temporary^{s+1}(x,t))$

For the default *insert-when-subtype-insertion* policy corresponding to the specialization relationship *Temporary IsA Employed*, we have the following formula (by definition 10):

(7) $\forall x,t\ (Insert\_Temporary^{s}(x,t) \wedge \neg Employed^{s}(x,t\text{-}1) \wedge \neg Insert\_Employed^{s}(x,t) \rightarrow Insert\_Employed^{s+1}(x,t))$

For the default *delete-when-supertype-deletion* policy corresponding to the specialization relationship *Temporary IsA Employed*, we have the following formula (by definition 12):

(8) $\forall x,t\ (Delete\_Employed^{s}(x,t) \wedge Temporary^{s}(x,t\text{-}1) \wedge \neg Delete\_Temporary^{s}(x,t) \rightarrow Delete\_Temporary^{s+1}(x,t))$

The formula that defines irreparable situations for previous specialization constraint is (by definition 14):

(9) $\forall x,t\ (Insert\_Temporary^{s}(x,t) \wedge Delete\_Employed^{s}(x,t) \rightarrow Abort^{s+1})$

For the default *insert-when-subtype-insertion* policy corresponding to the specialization relationship *Permanent IsA Employed*, we have the following formula (by definition 10):

(10) $\forall x,t\ (Insert\_Permanent^{s}(x,t) \wedge \neg Employed^{s}(x,t\text{-}1) \wedge \neg Insert\_Employed^{s}(x,t) \rightarrow Insert\_Employed^{s+1}(x,t))$

For the default *delete-when-supertype-deletion* policy corresponding to the specialization relationship *Permanent IsA Employed*, we have the following formula (by definition 12):

(11) $\forall x,t\ (Delete\_Employed^{s}(x,t) \wedge Permanent^{s}(x,t\text{-}1) \wedge \neg Delete\_Permanent^{s}(x,t) \rightarrow Delete\_Permanent^{s+1}(x,t))$

The formula that defines irreparable situations for previous specialization constraint is (by definition 14):

(12) $\forall x,t\ (Insert\_Permanent^{s}(x,t) \wedge Delete\_Employed^{s}(x,t) \rightarrow Abort^{s+1})$

## Appendix 2

Appendix 2 presents TCE triggers for formulas of appendix 1 (which correspond to part of the example of figure 3.1).

First of all, we list relational tables used for the TCE triggers.

IB tables:

```
employed(entity, time)
temporary(entity, time)
permanent(entity, time)
```

Structural event tables:

```
insert_employed(entity, time)
delete_employed(entity, time)
insert_temporary(entity, time)
delete_temporary(entity, time)
insert_permanent(entity, time)
delete_permanent(entity, time)
```

Triggering tables:

```
triggering_insert_employed(entity, time)
triggering_delete_employed(entity, time)
triggering_insert_temporary(entity, time)
triggering_delete_temporary(entity, time)
```

```
      triggering_insert_permanent(entity, time)
      triggering_delete_permanent(entity, time)
```

Abort table:

```
      abort(indication)
```

Next, we list the TCE triggers corresponding to the formulas of the appendix 1.

Trigger for formula 1:

```
CREATE TRIGGER t1
AFTER INSERT ON triggering_insert_temporary
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM insert_permanent ip
                    WHERE ip.entity=nr.entity AND ip.time=nr.time)
     AND EXISTS (SELECT * FROM permanent p
                    WHERE p.entity=nr.entity AND p.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM delete_permanent dp
                    WHERE dp.entity=nr.entity AND dp.time=nr.time)
BEGIN ATOMIC
 INSERT INTO delete_permanent VALUES (nr.entity, nr.time);
 INSERT INTO triggering_delete_permanent VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 2:

```
CREATE TRIGGER t2
AFTER INSERT ON triggering_insert_permanent
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM insert_temporary it
                    WHERE it.entity=nr.entity AND it.time=nr.time)
     AND EXISTS (SELECT * FROM temporary t
                    WHERE t.entity=nr.entity AND t.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM delete_temporary dt
                    WHERE dt.entity=nr.entity AND dt.time=nr.time)
BEGIN ATOMIC
  INSERT INTO delete_temporary VALUES (nr.entity, nr.time);
  INSERT INTO triggering_delete_temporary VALUES (nr.entity, nr.time);
END;
```

Triggers for formula 3:

```
CREATE TRIGGER t3_1
AFTER INSERT ON triggering_insert_temporary
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM insert_permanent ip
               WHERE ip.entity=nr.entity AND ip.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES ('yes');
END;

CREATE TRIGGER t3_2
AFTER INSERT ON triggering_insert_permanent
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM insert_temporary it
               WHERE it.entity=nr.entity AND it.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES ('yes');
END;
```

Trigger for formula 4:

```
CREATE TRIGGER t4
AFTER INSERT ON triggering_delete_temporary
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM delete_employed de
                   WHERE de.entity=nr.entity AND de.time=nr.time)
     AND NOT EXISTS (SELECT * FROM insert_permanent ip
                   WHERE ip.entity=nr.entity AND ip.time=nr.time)
BEGIN ATOMIC
  INSERT INTO delete_employed VALUES (nr.entity, nr.time);
  INSERT INTO triggering_delete_employed VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 5:

```
CREATE TRIGGER t5
AFTER INSERT ON triggering_delete_permanent
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM delete_employed de
                   WHERE de.entity=nr.entity AND de.time=nr.time)
     AND NOT EXISTS (SELECT * FROM insert_temporary it
                   WHERE it.entity=nr.entity AND it.time=nr.time)
BEGIN ATOMIC
   INSERT INTO delete_employed VALUES (nr.entity, nr.time);
   INSERT INTO triggering_delete_employed VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 6:

```
CREATE TRIGGER t6
AFTER INSERT ON triggering_insert_employed
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM insert_temporary it
                   WHERE it.entity=nr.entity AND it.time=nr.time)
     AND NOT EXISTS (SELECT * FROM insert_permanent ip
                   WHERE ip.entity=nr.entity AND ip.time=nr.time)
BEGIN ATOMIC
  INSERT INTO insert_temporary VALUES (nr.entity, nr.time);
  INSERT INTO triggering_insert_temporary VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 7:

```
CREATE TRIGGER t7
AFTER INSERT ON triggering_insert_temporary
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM employed e
                   WHERE e.entity=nr.entity AND e.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM insert_employed ie
                   WHERE ie.entity=nr.entity AND ie.time=nr.time)
BEGIN ATOMIC
   INSERT INTO insert_employed VALUES (nr.entity, nr.time);
   INSERT INTO triggering_insert_employed VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 8:

```
CREATE TRIGGER t8
AFTER INSERT ON triggering_delete_employed
REFERENCING NEW ROW AS nr
```

```
WHEN EXISTS (SELECT * FROM temporary t
                  WHERE t.entity=nr.entity AND t.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM delete_temporary dt
                  WHERE dt.entity=nr.entity AND dt.time=nr.time)
BEGIN ATOMIC
  INSERT INTO delete_temporary VALUES (nr.entity, nr.time);
  INSERT INTO triggering_delete_temporary VALUES (nr.entity, nr.time);
END;
```

Triggers for formula 9:

```
CREATE TRIGGER t9_1
AFTER INSERT ON triggering_insert_temporary
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM delete_employed de
               WHERE de.entity=nr.entity AND de.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES ('yes');
END;

CREATE TRIGGER t9_2
AFTER INSERT ON triggering_delete_employed
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM insert_temporary it
               WHERE it.entity=nr.entity AND it.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES ('yes');
END;
```

Trigger for formula 10:

```
CREATE TRIGGER t10
AFTER INSERT ON triggering_insert_permanent
REFERENCING NEW ROW AS nr
WHEN NOT EXISTS (SELECT * FROM employed e
                  WHERE e.entity=nr.entity AND e.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM insert_employed ie
                  WHERE ie.entity=nr.entity AND ie.time=nr.time)
BEGIN ATOMIC
   INSERT INTO insert_employed VALUES (nr.entity, nr.time);
   INSERT INTO triggering_insert_employed VALUES (nr.entity, nr.time);
END;
```

Trigger for formula 11:

```
CREATE TRIGGER t11
AFTER INSERT ON triggering_delete_employed
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM permanent p
                  WHERE p.entity=nr.entity AND p.time=nr.time-1)
     AND NOT EXISTS (SELECT * FROM delete_permanent dp
                  WHERE dp.entity=nr.entity AND dp.time=nr.time)
BEGIN ATOMIC
  INSERT INTO delete_permanent VALUES (nr.entity, nr.time);
  INSERT INTO triggering_delete_permanent VALUES (nr.entity, nr.time);
END;
```

Triggers for formula 12:

```
CREATE TRIGGER t12_1
```

```
AFTER INSERT ON triggering_insert_permanent
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM delete_employed de
               WHERE de.entity=nr.entity AND de.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES (‘yes’);
END;

CREATE TRIGGER t12_2
AFTER INSERT ON triggering_delete_employed
REFERENCING NEW ROW AS nr
WHEN EXISTS (SELECT * FROM insert_permanent ip
               WHERE ip.entity=nr.entity AND ip.time=nr.time)
BEGIN ATOMIC
     INSERT INTO abort VALUES (‘yes’);
END;
```