Dipartimento di Informatica
Università degli Studi di Verona

Rapporto di ricerca
Research report
October 2009

RR 78/2009

Validating Spatial Integrity Constraints through SQL Queries

Alberto Belussi
Sara Migliorini
Mauro Negri
Giuseppe Pelagatti

Questo rapporto è disponibile su Web all'indirizzo:
This report is available on the web at the address:
http://www.di.univr.it/report

Abstract

The validation of spatial integrity constraints specified at conceptual level is an important activity, both for checking the quality of the provided datasets resulting from a production process and for monitoring the consistency of information stored into a spatial database, in particular when updates have to be handled.

This document presents a methodology for validating spatial integrity constraints defined at conceptual level through the GeoUML modeling language, by translating them into SQL spatial queries. The GeoUML language allows designers not only to represent spatial and non-spatial properties of a dataset, but also to specify spatial integrity constraints, using some predefined OCL templates referring to topological and part-whole relations. The aim of this work is to define a set of mapping rules for translating these constraints into spatial SQL queries that return the violating objects. Namely, for each kind of GeoUML spatial constraints an SQL query template is defined, that can be automatically customized with respect to the particular considered constraint.

1 Introduction

The validation of spatial integrity constraints specified at conceptual level is an important activity, both for checking the quality of the provided datasets resulting from a production process and for monitoring the consistency of information stored into a spatial database, in particular when updates have to be handled.

The GeoUML modelling language provides some predefined templates in OCL (Object Constrained Language) [3] for expressing spatial integrity constraints at conceptual level. This choice is motivated by the fact that designers of spatial databases for geographical applications very often are not familiar with such formalism; moreover the high expressive power of OCL admits the representation of the same spatial property in different but equivalent expressions, and this makes the optimization of constraints checking procedures a severe task. The GeoUML approach overcomes these limitations, but reduces the expressive power of the constraints expressions. However, the obtained language for constraints has been applied with success in many real projects, in particular in the definition of the Italian National Core, which is the reference spatial database schema for the Italian Spatial Data Infrastructure.

This report presents the translation of the OCL templates provided by GeoUML into SQL query templates that can be used for checking a dataset loaded into a geo-relational database. Each of these templates can be specialized according to the particular properties of the involved objects. In order to present the SQL expressions also the mapping rules for implementing a GeoUML schema in a geo-relation database are presented.

The report is organized as follows: Section 2 illustrates the main constructs of GeoUML, focusing on geometric types, segmented properties and spatial integrity constrains. The mapping to the geo-relational model is presented in Section 3, while in Section 4 the SQL templates for the validation of GeoUML spatial integrity constrains are defined.

2 The GeoUML Modeling Language

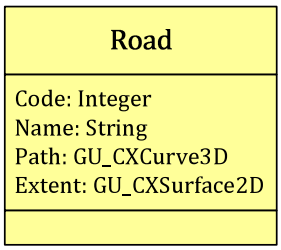
The GeoUML is an UML based language for the conceptual modeling and representation of spatial information. It inherits from UML some basic concepts such as the notion of *class*, *attribute*, *association*, *class inheritance* and *data type* and it defines some new constructs for the representation of spatial information. Beyond the definition of new constructs, the most evident difference between these two languages is about the syntax: GeoUML has both

a graphical and a textual representation, and the main syntax is considered the textual one, while the graphical representation is used only to enhance the readability of a specification.

Beside the basic concepts that GeoUML inherits form the UML language, it supplies some additional non-geometric constructs for supporting the schema designer in the specification of properties very often used in describing spatial data. These constructs are: *primary key*, *enumerated attribute* and *hierarchical enumerated attribute*, whose presentation is out of the scope of this report.

The main additional part provided by GeoUML concerns the definition of *spatial attributes* and their correlated properties (in particular *segmented properties* and *spatial integrity constraints*). Indeed, each class instance (called feature in spatial terminology) is characterized by one or more spatial attributes. A spatial attribute is an attribute whose values belong to a geometric domain. Each class can have one or more spatial attributes, which in the textual representation of a GeoUML schema are isolated from the thematic ones by the keyword `spatial attributes`, while in the graphical representation this distinction is loosed and the spatial attributes are represented as normal ones, as illustrated in Figure 1.

A detailed presentation of the GeoUML language is beyond the scope of this report, so we concentrate only on the main aspect of GeoUML as regard to the validation issue: *geometric data types*, *segmented properties* and *spatial integrity constraints*, which are presented into the following sections.




```
class Road
  attributes
    class attributes
    PK Code: String
    Name: String
    spatial attributes
    Path: GU_CXCurve3D
    Extent: GU_CXSurface2D
```

Figure 1: GeoUML graphical and textual representation of the class `Road` which is characterized by two spatial attributes: `Path` and `Extent`.

2.1 Geometric Data Types

The geometric data types provided by GeoUML for specifying spatial attributes of features, have been defined by using the point set model and in agreement with the Simple Feature Model (SFM) [4] for 2D geometries.

Moreover, for 2D types representing points and lines the corresponding version in 3D space has been defined. In the sequel, when the coordinate dimension is not important, we use the symbol * for denoting types that have both a 2D and a 3D version. The GeoUML data types can be divided into two major kinds: *geometric primitives* and *geometry collections*.

A *geometric primitive* is an atomic geometric value representing points, lines or surfaces, i.e. it is not separable into smaller parts. Geometric primitive types are: `GU_Point*D`, `GU_CPCurve*D`, `GU_CPSimpleCurve*D`, `GU_CPRing*D`, `GU_CPSurface2D`. An object of type `GU_Point*D` is a zero-dimensional geometric object (called point) which represents an individual position on the Earth surface. The two alternatives differ on the number of coordinates which are used to define the point. An object of type `GU_CPCurve*D` is a primitive curve in the 2D (3D) space. A curve is a one-dimensional geometric object usually represented by a sequence of points (vertices), where the type of interpolation can be different. The GeoUML model does not prescribe a particular interpolation method. The type `GU_CPSimpleCurve*D` specializes the previous type adding the condition of being simple (a curve is simple if it has no self-intersections), while the type `GU_CPRing*D` also includes the condition of being cycle (a curve is a cycle if its endpoints coincide). An object of type `GU_CPSurface2D` is the primitive surface in the 2D space, namely it is a bi-dimensional planar object which is connected and defined by a closed 2D curve, that represents the external boundary of the surface, and zero, one or more closed 2D curves that represent the holes (internal boundaries).

A *geometry collection* is a collection of various geometric primitives; it can be *homogeneous*, if it contains primitives of the same type (multi-point, multi-curve and multi-surface), or *non homogeneous*, if it contains primitives of different types (generic aggregate). *Non homogeneous collection geometries* are modeled using the type `GU_Aggregate*D`. This type models multi-dimensional objects composed of a collection of one or more objects of any primitive type. The 2D and 3D alternatives differs only in the coordinate dimensions of the objects involved; these collections are not subject to any other constraints.

Homogeneous geometry collection types are: `GU_CXPoint*D`, `GU_CXCurve*D`, `GU_CXRing*D`, `GU_CNCurve*D` and `GU_CXSurface2D`. In particular, an object of type `GU_CXPoint*D` is an aggregate of points. An object of type `GU_CXCurve*D` is an aggregate of curves in the 2D (3D) space with the constraint that its curve components may possibly intersect only in a finite number of points. An object of type `GU_CXRing*D` specializes the previous type with the constraint that all the components are closed and simple, while an object of type `GU_CNCurve*D` adds the constraint that the overall curve is also a connected point set. Finally, an object of type `GU_CXSurface2D` is an aggregate of sur-

faces with the constraint that each pair of surfaces is disjoint or intersects only in a finite number of points on their boundaries.

Figure 2 illustrates the UML class hierarchy diagram for the geometric types of GeoUML. Notice that, the root class of the hierarchy is `GU_Object`, that collects all common properties shared by all geometric types.

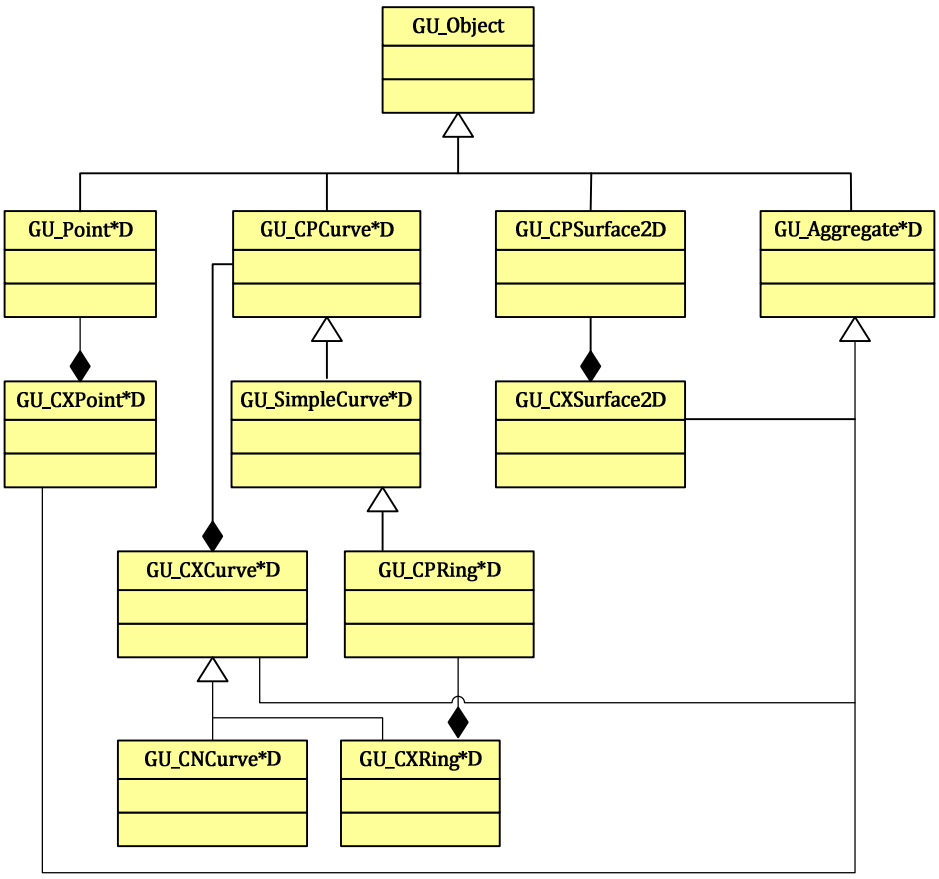


Figure 2: Class diagram of the GeoUML geometric data types. The symbol `*` stands for 2 or 3 in a coherent manner.

The `GU_Object` type exposes a set of methods (functions) that are specialized in each geometric type. In the following we use the usual notation `o.f()` to denote the application of the method `f()` on the object `o`.

- `o.boundary()`: it returns a geometric object which represents the boundary of `o`. The definition of boundary for an object depends on its specific type. In particular, the boundary is not defined for the type `GU_Aggregate*D`. For the types `GU_Point*D`, `GU_CXPoint*D`,

GU_CPRing*D and GU_ CXRing*D the boundary is empty; for the types GU_CPCurve*D and GU_CPSimpleCurve*D the boundary is constituted of the two curve endpoints, while the boundary of a GU_CXCurve*D or GU_CNCurve*D is defined by the *mod 2 union rule*, thus it combines the points that are boundary of an odd number of component curves, as defined in the SFM standard [4]. The boundary of a GU_CPSurface2D is composed of the curves that represent its internal and external boundaries, while the boundary of a GU_CXSurface2D is composed of the boundary of its component surfaces.

- `o.isSimple()`: it returns true if `o` is simple, i.e. if it does not contains point of self intersection or tangency.
- `o.isCycle()`: it returns true if the boundary of `o` is empty.
- `o.dimension()`: it returns 0 if `o` is a point, 1 if `o` is a line or 2 if `o` is a surface.
- `o.coordinateDimension()`: it returns the number of coordinates used for the definition of the object points (2 or 3).
- `o.spatialRS()`: it returns the spatial reference systems for the object coordinates.
- `o.planar()`: it returns the geometric object that represents the projection of `o` in the 2D space.
- `o.union(o1)`: it returns the geometric object representing the point set union of `o` and `o1`.

The topological relations used in GeoUML are: *disjoint* (DJ), *touch* (TC), *in* (IN), *contain* (CT), *equal* (EQ) and *overlap* (OV). They are a refinement of those defined by Clementini et al in [2]; moreover, the relations *intersect* (IT) and *cross* (CR) are added to this set. The semantics of these topological relations is given in Table 1 with the specification of the possible types for the involved objects.

The GeoUML geometric types allow one to represent surfaces only in the 2D space. In order to enrich this representation, the types GU_CPSurfaceB3D and GU_CXSurfaceB3D have been added. Using these types it is possible to represent a 2D surface together with its 3D boundary, called in short B3D surface. Each B3D surface can be intended as a GeoUML object with two spatial attributes: one of type GU_C*Surface2D and one of type GU_CXRing3D, which represent the surface extent and the surface boundary respectively. In

addition, for each B3D surface the planar projection of the GU_CXRing3D component has to be equal to the boundary of the 2D surface.

Relation	Types	Semantics
a DJ b	(A,A)	$a \cap b = \emptyset$
a TC b	(S,A),(C,A) (A,S),(A,C)	$(a^\circ \cap b^\circ = \emptyset) \wedge$ $(a \cap b \neq \emptyset)$
a IN b	(A,S),(C,C) (P,C),(S,A) (C,P)	$(a^\circ \cap b^\circ \neq \emptyset) \wedge$ $(a \cap b = a) \wedge$ $(a \cap b \neq b)$
a CT b	(S,A),(C,C) (C,P),(A,S) (P,C)	b IN a
a OV b	(S,S),(S,C) (C,C) (C,S)	$(a^\circ \cap b^\circ \neq \emptyset) \wedge$ $(a \cap b \neq a) \wedge$ $(a \cap b \neq b)$
a EQ b	(S,S),(C,C) (P,P)	$(a \cap b = a) \wedge$ $(a \cap b = b)$
a IT b	(A,A)	$(a \cap b \neq \emptyset) \wedge$ not a EQ b
a CR b	(C,C)	a OV b $\wedge$ $\dim(a \cap b) = 0 \wedge$ $\dim(a) = \dim(b) = 1$

Table 1: Semantics of the GeoUML topological relations (the keyword A stands for any geometric types excluding GU_Aggregate*D, P stands for all point types, C stands for all curve types and S stands for all surface types).

2.2 Segmented Properties

Modeling geographic information at conceptual level often requires specifying properties that change their value along the geometry of a feature. GeoUML allows one to represent this kind of properties by using a specific construct called *segmented property*: it is a feature attribute whose value is a function from a spatial attribute g of the same feature to a simple domain D_A. Therefore, its value is not constant for the whole geometric extent of g, but it changes on g in a discrete mode. According to the type of the spatial attribute on which it is defined, segmented properties can be distinguished into *linear* and *sub-regional segmented properties*. A linear segmented property is defined along a GU_C*Curve*D type: it divides the curve into segments with

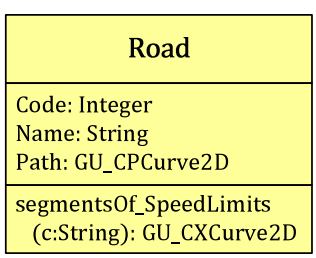
homogeneous value. Similarly, a sub-regional segmented property associates a particular value to different sub-regions of a GU_C*Surface2D.

In order to represent a segmented property in the textual representation, we insert after the definition of the interested spatial attribute the keyword segmented properties, followed by the list of segmented properties for this spatial component, as shown in the example of Figure 3.

Using the graphical representation each segmented property is defined by one of the following methods where A is the property name:

segmentsOf_*A*(condition on D_A): GU_CXCurve*D
subregionsOf_*A*(condition on D_A): GU_CXSurface2D

which return a complex curve (surface) composed of the set of curve segments (sub-regions) that satisfy a particular condition on the attribute domain D_A.




```
class Road
  attributes
    class attributes
    PK Code: String
    Name: String
    spatial attributes
    Path: GU_CPCurve2D
      segmented properties
      SpeedLimits: Integer
        segmented on Path
```

Figure 3: Graphical and textual representation of a linear segmented property defined on the Path spatial attribute of the class Road.

2.3 GeoUML Spatial Integrity Constraints

A spatial integrity constraint is a condition defined on spatial properties of features that must be satisfied by any database instance. In GeoUML these constraints are classified into two major kinds: *topological* and *part-whole constraints*. In the following sections we present the textual syntax that GeoUML provides for the definition of these constraints and we illustrate informally their semantics. Notice that, the GeoUML model defines for each constraint type an OCL template, that specifies formally its semantics; such OCL templates have been previously presented in [1] and are out of the scope of this report.

2.3.1 Topological Constraints

A topological constraint refers to a spatial property involving two features which is described by a topological relation between them. In order to define a

topological constraint we need to specify two aspects: (1) the spatial relation used in the constraint and (2) the logical structure of the constraint. The GeoUML reference set of topological relations has been defined in Table 1, while the logical structure of the available constraints types is presented in the following of this section.

Basic existential topological constraint The basic form of existential topological constraint requires that for each feature x of the constrained class X, there exists a feature y of the constraining class Y, such that between the spatial attribute g of x and the spatial attribute f of y a particular topological relation (or disjunction of topological relations) is satisfied. The textual syntax of this constraint is assigned as follows:

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ (\texttt{rel}_1\ |\ \ldots\ |\ \texttt{rel}_2)\ \underline{\texttt{exists}}\ \texttt{Y.f}$$

An existential topological constraint can also be defined on the geometry of a segmented property. In this case, in the constraint definition the spatial attribute is replaced by the function: `segmentsOf_`*A*`()` or `subregionsOf_`*A*`()` where *A* is the name of the segmented property. The replacement of a spatial attribute with a segmented property can be applied both to the constrained and the constraining class or to only one of them. This constraint is represented as follows:

$$\begin{array}{l}\underline{\texttt{constraint}}\ \texttt{X.segmentsOf\_}A\texttt{()}\ (\texttt{rel}_1\ |\ \ldots\ |\ \texttt{rel}_2)\\ \quad \underline{\texttt{exists}}\ \texttt{Y.segmentsOf\_}B\texttt{()}\end{array}$$

Many alternatives of the basic existential topological constraint can be defined by changing the form of both the constrained and the constraining class, or only one of them.

Existential topological constraint with selection An existential topological constraint is usually checked with respect to all the instances of the constrained and the constraining classes. Nevertheless, a selection can be applied in order to restrict the features involved in the constraint, in other words the constraint is checked only with respect to the features satisfying the selection. The selection condition can be applied to both the constrained and the constraining class or only one of them. Its textual representation is:

$$\begin{array}{l}\underline{\texttt{constraint}}\ (\sigma_1\texttt{(X)})\texttt{X.g}\ (\texttt{rel}_1\ |\ \ldots\ |\ \texttt{rel}_2)\\ \quad \underline{\texttt{exists}}\ (\sigma_2\texttt{(X,Y)})\texttt{Y.f}\end{array}$$

where σ_1 and σ_2 can be *basic selections* (i.e. logical expression like "`attribute comparator value`") or *join selections* (like "`attribute comparator constrainedClass.attribute`") which permits to bind constraining features to constrained features.

Existential topological constraint on boundary/planar projection The boundary function (`bnd()`) or the planar projection (`pln()`) can be applied on the spatial attributes involved in the constraint before checking it. These functions can be combined with the selection alternative presented above, but they cannot be applied on segmented properties. The textual representation of this constraint is:

$$\underline{\texttt{constraint}}\ \texttt{X.g.bnd()}\ \texttt{(rel}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{rel}_2\texttt{)}\ \underline{\texttt{exists}}\ \texttt{Y.f.pln()}$$

Existential topological constraint connected to an association This existential constraint considers only the features of the *constraining* class which are connected to the constrained one through a given association. The constraint is textually represented as:

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ \texttt{(rel}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{rel}_2\texttt{)}\ \underline{\texttt{exists}}\ \texttt{X.r.f}$$

where r is the role of Y in the association between X and Y (notice that $X.r$ identifies an instance of Y). This alternative can be combined with any of the previous one.

Union existential topological constraint Instead of requiring the existence of a Y instance that satisfies the constraint, the union existential topological constraint requires that the topological relation is satisfied between the spatial attribute g of the constrained instance and the union of the spatial attribute f of all the instances of the constraining class Y. The textual representation of this constraint is:

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ \texttt{(rel}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{rel}_2\texttt{)}\ \underline{\texttt{union}}\ \texttt{Y.f}$$

The three alternatives mentioned above can be defined also for the union existential topological constraint: the one that considers the selection of instances, the one referring to the boundary or planar projection and the one connected to an association.

Universal topological constraint By replacing the existential quantifier with the universal one, we obtain a new version of the topological constraint which requires that the topological relation exists between the spatial attribute g of the constrained feature and the spatial attribute f of *all* the features of the constraining class Y.

$$\underline{\texttt{constraint}}\ \texttt{X.g (rel}_1\ |\ \ldots\ |\ \texttt{rel}_2\texttt{)}\ \underline{\texttt{forAll}}\ \texttt{Y.f}$$

All the alternatives presented above for the existential constraint can also be applied to the universal one. In particular, we can define: a universal topological constraint with selection, a universal topological constraint on the boundary or planar projection and a universal topological constraint connected to an association. Finally, the universal topological constrain can also be applied on segmented properties.

2.3.2 Part-Whole Constraints

Part-whole constraints allow one to specify that two classes are in a (spatial) composition relationship. In GeoUML there are four main kinds of part-whole constraints: *strong composition*, *weak composition*, *membership* and *partition*. All these constraints have existential type, so they require that given a spatial attribute f of an instance of the composite class Y, there exists some instances of the component class X whose spatial attributes g together compose f.

The same alternatives presented for the topological constraints can also be defined for the part-whole constraints: part-whole constraint with selection, part-whole constraint on boundary or planar projection and part-whole constraint connected to an association. Finally, the part-whole constraint can also be defined on segmented properties.

Strong composition The strong composition constraint requires that for each instance of the constrained class X, its spatial attribute g must be *equal* to the union of the spatial attribute f of one or more instances of the constraining class Y.

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ \underline{\texttt{composedOf}}\ \texttt{Y.f}$$

Weak composition The weak composition constraint requires that for each instance of the constrained class X, its spatial attribute g must be *contained* into the union of the spatial attribute f of one or more instances of the constraining class Y.

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ \underline{\texttt{coveredBy}}\ \texttt{Y.f}$$

Membership The membership constraint of the spatial attribute g of the constrained class X to the spatial attribute f of the constraining class Y can be represented by an existential topological constraint between g and f, where the relation involved is *IN*. In GeoUML some alternatives of the pure membership constraints are defined:

- *Disjoint membership*: the membership constraint with disjunction requires that among the components of the same whole exists only the disjoint or touch relation.

`constraint X.g dj-IN Y.f`

- *Weak-Disjoint membership*: this constraint can be applied only to linear spatial attributes and admits that among components of the same whole is valid also the cross relation, further to disjoint or touch.

`constraint X.g wdj-IN Y.f`

Partition The partition constraint requires that the spatial attribute f of each instance of the partitioned class Y has to be composed by the spatial attribute g of one or more instances of the component class X with the following conditions:

- The union of the spatial attributes g of the X instances is equal to the spatial attribute f of the Y instance.
- The spatial attribute g of the X instances that compose f are not overlapping, at most they can touch.

The partition constraint is obtained combining the *strong composition* constraint with a *disjoint* or *weak-disjoint membership* constraint.

3 Basic Relational Mapping Rules

This section presents the mapping rules applied for translating a GeoUML conceptual schema into a geo-relational schema, with reference to PostGIS system for the mapping of data types. In particular, this section deals with the translation of the basic constructs, such as: classes, associations between classes and segmented properties; while the translation of spatial integrity constraints is treated in Section 4.

In the following we use the term S to denote a generic GeoUML conceptual schema, while during the table and query definitions we use strings in italics as placeholder for the name of a particular class, attribute, relation, and so on.

3.1 Class Mapping

For each class *className* in S we generate a table with the same name as the class and a primary key attribute `uuid`:

`Table name:` *`className`*
`Table columns:`
`uuid: varchar(64)`

Moreover, for each class attribute of type: *String*, *NumericString, Integer, Real, Boolean, Date, Time, DateTime* (i.e. basic types of GeoUML) we add to the class table an attribute as follows:

`attributeName``: Domain(`*`attributeDomain`*`)`

where the function `Domain(`*`attributeDomain`*`)` returns for each GeoUML basic type the corresponding SQL type. For example, if the attribute domain is *String(x)* or *NumericString(x)* then the function returns `varchar(x)`, while if the attribute domain is *DateTime* it returns `datetime`, and so on.

For each class attribute of type *dataTypeName* we add a column in the table *`className`* with the following characteristics:

`id_`*`attributeName`*`: varchar(64) references `*`dataTypeName`*`(id)`

where *`dataTypeName`* is the table implementing the data type. For each (hierarchical) enumerated class attribute *attributeName* with domain *domainName* we add a column in the table *`className`* as follows:

`code_`*`attributeName`*`: varchar(80) references `*`domainNameE`*`(code)`

where *`domainNameE`* is the table that implements the (hierarchical) enumerated domain. In particular, this table contains, for each possible value of the domain, its code and its description.

For each spatial attribute *geoAttributeName* of the class we add a column in the table *`className`* as follows:

`geoAttributeName``: corresponding PostGIS domain`

Table 2 defines the mapping between GeoUML geometric types and PostGIS types, the term *constraints* is used to denote the need for additional constraints during the column definition. For example, for ring objects it is nec-

essary to check that the curve is a simple cycle (i.e. `CHECK ST_IsClosed(`*`geoAttributeName`*`) AND ST_IsSimple(`*`geoAttributeName`*`)`).

GeoUML type	**PostGIS type**
GU_Point*D	POINT
GU_CPCurve*D	LINESTRING
GU_SimpleCurve*D	LINESTRING + constraints
GU_CPRing*2D	LINESTRING + constraints
GU_CPSurface2D	POLYGON
GU_Aggregate*D	GEOMETRYCOLLECTION
GU_CXPoint*2D	MULTIPOINT
GU_CXCurve*D GU_CNCurve*D GU_CXRing*D	MULTILINESTRING + constraints
GU_CXSurface2D	MULTIPOLYGON + constraints

Table 2: GeoUML and PostGIS geometric types mapping.

3.2 Class Hierarchy Mapping

For each class hierarchy with root C and children $C_1, \ldots, C_n$ in S we proceed as follows: for the root class we generate a table `C` with the usual mapping rules and then we add the column:

`type``:  varchar(64)`

which indicates the subtype $(C_1, \ldots, C_n)$ of each C instance. For each child class we create a table C_i as usual, that contains only the specific attributes of this child class; then we add the following integrity constraints on the `uuid` attribute of each C_i table to the table `C`:

`uuid:  varchar(64) references C(id)`

3.3 Association Mapping

Associations are mapped to the relational model by applying the well-known approach. In particular, for each binary association *associationName* in S between two classes $className_1$ and $className_2$ with cardinality one-to-many (or one-to-one), we add into the table implementing the class with cardinality one (suppose it is $className_1$) an attribute which represents the foreign

key of the table corresponding to $className_2$. The name of this attribute will be `uuid_class2Rule`, where *class2Rule* is the name of the rule assigned to $className_2$ into the association. If the rule does not have a name, we use `uuid_className2` as attribute name.

```
ALTER TABLE className1
ADD COLUMN uuid_class2Rule varchar(64)
REFERENCES className2(uuid)
```

If the binary association between $className_1$ and $className_2$ has cardinality many-to-many, we create a new table for the association with the following characteristics:

```
Table Name (if the association has a name):  associationName
Table Name (otherwise):  className1_className2
Table columns:
uuid_class1Rule:  varchar(64) references className1(uuid)
uuid_class2Rule:  varchar(64) references className2(uuid)
```

For each attribute *attributeName* of the association *associationName* we add a column into the table `associationName` as usual.

3.4 Linear Segmented Properties Mapping

For each group of linear segmented properties $AT_1, \ldots, AT_n$ defined on the same geometric linear attribute *geoAttributeName* of a class *className* in S, we create a table as follows:

```
Table name:  className_geoAttributeName
Table columns:
uuid:  varchar(64)
uuid_className:  varchar(64) references className (id)
geometry:  LINESTRING 2D/3D
```

where the `uuid` attribute is the identifier of a particular line segment, while `uuid_className` is the identifier of the class *className*. Hence, we add to this table the attributes $AT_1, \ldots, AT_n$ following the same mapping rules established for a class attribute.

3.5 Sub-Regional Segmented Properties Mapping

For each group of sub-regional segmented properties AS_1, ..., AS_n defined on the same geometric attribute *geoAttributeName* of a class *className* in S we create a table as follows:

```
Table name:  className_geoAttributeName
Table columns:
uuid:  varchar(64)
uuid_className:  varchar(64) references className (id)
geometry:  POLYGON 2D
```

where `uuid` attribute is the identifier of a particular sub-region, while `uuid_className` is the identifier of the class *className*. Hence, we add to the table the attributes AS_1, ..., AS_n with the same mapping rules established for a class attribute.

4 Spatial Integrity Constraints Validation

Each spatial integrity constraint specified in a GeoUML schema has to be checked w.r.t the database content. This section presents the mapping rules that allow one to convert a GeoUML spatial integrity constraint into the corresponding SQL query. Each query returns the set of constrained class instances that violate the constraint. In particular, for each type of spatial constraint presented in Section 2.3 a SQL query template is defined, whose instantiation changes according to the particular alternative used in the constraint specification, for defining the constraining and the constraint class.

4.1 Topological Constraints Validation

This section deals with the translation of the GeoUML topological constraints presented in Section 2.3.1 into SQL query templates. In the following, we use strings in bold to denote the variable parts of each query template, namely the parts that have to be instantiated according to the particular constraint.

4.1.1 Basic existential topological constraint

The basic existential topological constraint:

$$\underline{\texttt{constraint}}\ \texttt{X.g}\ (\texttt{rel}_1 \mid \ldots \mid \texttt{rel}_n)\ \underline{\texttt{exists}}\ \texttt{Y.f}$$

PostGIS Function	Semantics
ST_Disjoint(a,b)	$a \cap b = \emptyset$
ST_Touches(a,b)	$(a^\circ \cap b^\circ = \emptyset) \wedge (a \cap b \neq \emptyset)$
ST_Within(a,b)	$(a^\circ \cap b^\circ \neq \emptyset) \wedge (a \cap b = a)$
ST_Contains(a,b)	ST_Within(b,a)
ST_Overlap(a,b)	$(a^\circ \cap b^\circ \neq \emptyset) \wedge$ $(a \cap b \neq a) \wedge (a \cap b \neq b) \wedge$ $(\dim(a^\circ \cap b^\circ) = \max(\dim(a), \dim(b))$
ST_Intersects(a,b)	not ST_Disjoint(a,b)
ST_Crosses(a,b)	$(a^\circ \cap b^\circ \neq \emptyset) \wedge$ $(a \cap b \neq a) \wedge (a \cap b \neq b) \wedge$ $(\dim(a^\circ \cap b^\circ) < \max(\dim(a), \dim(b))$
ST_Equal(a,b)	ST_Within(a,b) $\wedge$ ST_Within(b,a)

Table 3: Semantics of the PostGIS functions that evaluates the topological relation that exists between two objects.

can be translated into a query which returns the set of X instances x for which does not exist any Y instance, whose spatial attribute f is in one of the requested topological relations with the spatial attribute g of x, as in the following query template:

```
SELECT x.uuid as failedObjects, x.g as failedGeometries
FROM X as x
WHERE NOT EXISTS
   (SELECT * FROM Y as y
    WHERE Topo_Condition(x.g, y.f, rel_1 | ... | rel_n))
```

The function `Topo_Condition()` is defined as follows:

```
Topo_Condition(x.g, f.y, rel_1 | ... | rel_n) =
   ST_XXX_1(x.g, y.f) OR ... OR ST_XXX_n(x.g, y.f)
```

where $\texttt{ST\_XXX}_i$ is the PostGIS function corresponding to the relation $\texttt{rel}_i$. The mapping between GeoUML topological relations and PostGIS functions is illustrated in Table 4, while Table 3 gives the semantics of the PostGIS functions that evaluate topological relations between geometries. Notice that, because the PostGIS functions are implemented with respect to the 2D space, a particular treatment is required for relations between 3D objects. Indeed, if a topological relation exists between the planar projections of two objects,

it is not certain that the same relation between those objects exists in the 3D space. In this report we focus only on the validation procedure for relations in 2D space.

GeoUML Relation	PostGIS function
a DJ b	ST_Disjoint(a,b)
a TC b	ST_Touches(a,b)
a IN b	ST_Within(a,b) $\wedge$ not ST_Equal(a,b)
a CT b	ST_Contains(a,b) $\wedge$ not ST_Equal(a,b)
a OV b	ST_Overlap(a,b) $\vee$ ST_Crosses(a,b)
a EQ b	ST_Equal(a,b)
a IT b	ST_Intersects(a,b) $\wedge$ not ST_Equals(a,b)
a CR b	ST_Crosses(a,b)

Table 4: GeoUML topological relations and PostGIS functions.

Example 1 Suppose to consider two classes *Road* and *RoadArea* which are translated into the following tables:

```
Table Road                  Table RoadArea
uuid:  varchar(64)          uuid:  varchar(64)
<Road attributes>           <RoadArea attributes>
path:  multilinestring      extent:  multipolygon
```

The constraints saying that for each road there exists a road area whose extent contains the road path:

```
constraint Road.path (IN) exists RoadArea.extent
```

is translating in the following spatial SQL query:

```
SELECT x.uuid as failedObjects, x.g as failedGeometries
FROM Road as x
WHERE NOT EXISTS
   (SELECT * FROM CL_RoadArea as y
    WHERE ST_Within(x.path, y.extent))
```

4.1.2 Basic existential topological constraint alternatives

Each alternative of the basic existential topological constraint presented in Section 2.3.1 has the general form:

constraint <**contrained_class_expr**>(**rel**$_1$ | ... | **rel**$_n$)
exists <**constraining_class_expr**>

Therefore, by refining the SQL query template specified for the basic case, a generic template for all the existential topological constraint can be defined, which has to be specialized according to the particular form of the constrained and constraining class expressions. For simplifying the definition of a common generic template, two temporary tables, named `Constrained` and `Constraining`, are created. These tables contain the set of objects obtained by evaluating the constrained and constraining expression respectively.

```
CREATE TABLE Constrained AS(
   SELECT x.uuid, x.pk_1, ..., x.pk_n,
         <ed_add_idattrib>, <ed_add_attrib>,
         <ed_geom> as g
   FROM <ed_from_expr>
   WHERE <ed_geom> IS NOT NULL
         <ed_where_expr>
   <ed_groupby_expr>)

CREATE INDEX Constrained_G_Index ON Constrained USING GIST (g)

CREATE TABLE Constraining AS(
   SELECT y.uuid <ing_add_attrib>, <ing_geom> as f
   FROM <ing_from_expr>
   WHERE <ing_geom> IS NOT NULL
          <ing_where_expr>
   <ing_groupby_expr>

CREATE INDEX Constraining_G_Index ON Constraining USING GIST (f)
```

The SQL query template that retrieves the set of objects violating a generic existential topological constraint becomes:

```
SELECT x.uuid as failedObjects,
   x.pk_1, ..., x.pk_n,
   <ed_add_idattrib>
   x.g as failedGeometries
FROM Constrained as x
WHERE NOT EXISTS(
   SELECT *
```

```
FROM Constraining <const_from_expr>
WHERE <const_where_expr> AND
   Topo_Condition(g, f, rel_1 | ... | rel_n))
```

These template queries for the construction of the temporary tables and the constraint verification have to be instantiated according to the particular alternative chosen for the constrained and constraining class, as explained in the following sections.

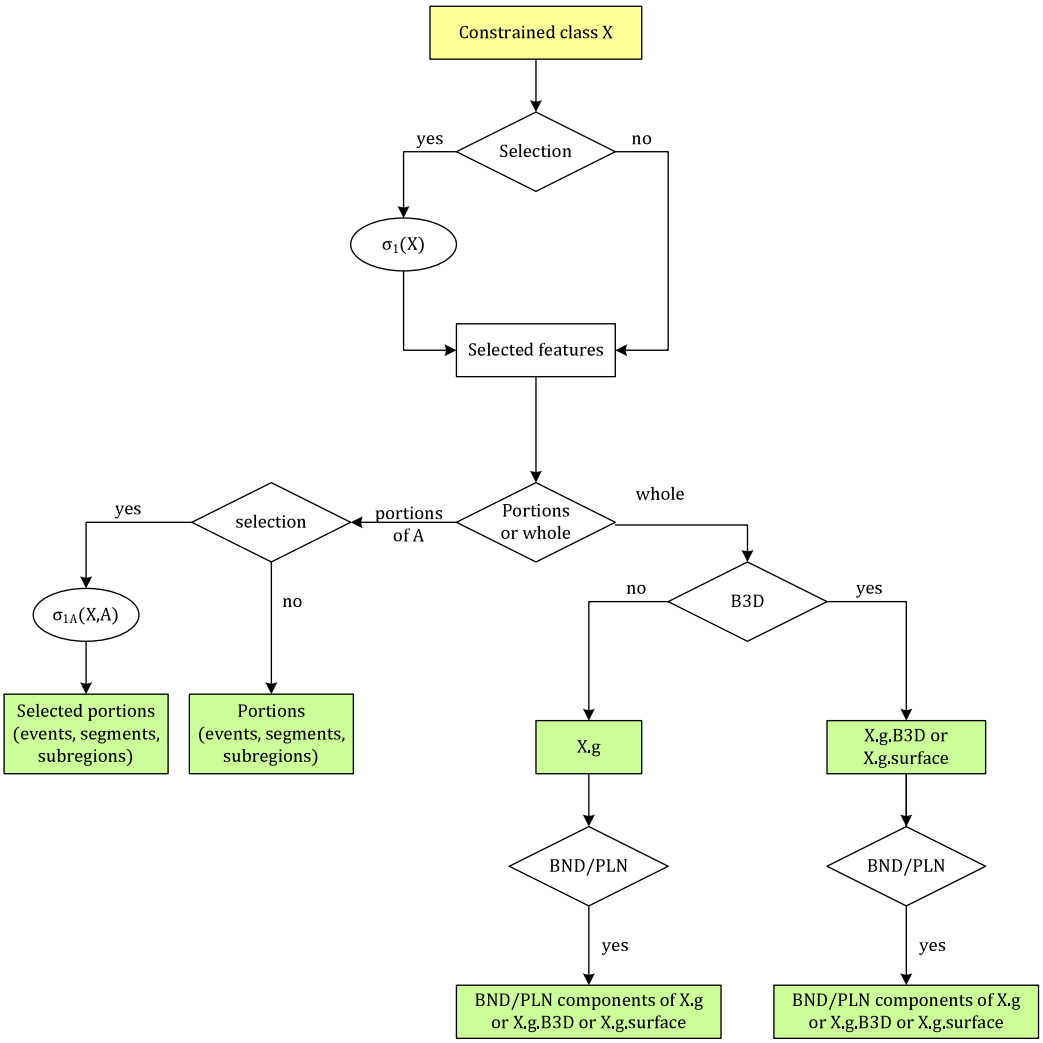


Figure 4: Schema about the selection process of the constrained geometries.

Let us notice that the constrained and the constraining class expressions that appear in a general constraint have the form `feature_selector.geo-`

metry_selector, where feature_selector determines the features involved in the constraint, while the geometry_selector determines the geometric value involved in the constraint.

4.1.3 Constrained class expression

The constrained class expression, which appears in the generic existential topological constraint presented above, can be instantiated in different way, according to one of the alternatives presented in Section 2.3.1. The form of the constrained class expression determines a different instantiation of the queries for creating the constrained temporary table and for performing the validation.

In the following the possible alternatives for the feature_selector and the geometry_selector expression of the constrained class are deeply analyzed and the corresponding query templates instantiated. Figure 4 summarizes how the constrained class expression can be built.

Feature selector The feature selector expression for the constrained class can take one of the following forms:

1. *Basic instance selector* X
   If the feature_selector expression is of the basic form X, then the template for the Constrained temporary table is instantiated as:
   - <ed_from_expr> :== **X** as x
   - <ed_where_expr> :== true
2. *Selection condition on class instances* (σ1(X))X
   In case a selection is performer on the constrained class X, that is if the feature_selector expression has the form (σ1(X))X, then the where clause has to be augmented with the corresponding SQL selection condition, in order to restrict the considered X instances.
   - <ed_from_expr> :== **X** as x
     If σ1(X) refers to attributes of a structured attribute or to multi-valued attributes of the X class, then the tables representing these attributes have to be joined as follows:
     (a) For each simple (i.e. of basic type) multi-valued attribute **a** in σ_1(X):
     <ed_from_expr> :== <ed_from_expr> +
     LEFT JOIN **X_a** AS m1 ON x.uuid = m1.uuid

Moreover during the translation of the selection condition $\sigma 1$(X) the attribute **a** must be referred to as: `m1.a`.

(b) For each enumerated (or hierarchical enumerated) multi-valued attribute **a** in $\sigma 1$(X):
<ed_from_expr> :== <ed_from_expr> +
LEFT JOIN **X_a** AS m1 ON x.uuid = m1.uuid
Moreover during the translation of the selection condition σ_1(X) the attribute **a** must be referred to as: `m1.code_a` (`m1.codeH_a`).

(c) For each simple (i.e. of basic type) attribute **b** of a multi-valued attribute **s** of type *DataType* in σ_1(X):
<ed_from_expr> :== <ed_from_expr> +
LEFT JOIN **X_s_T** as s1 ON x.uuid = s1.uuid
Moreover during the translation of the selection condition σ_1(X) the attribute **b** must be referred to as: `s1.code_b` (`s1.codeH_b`).

(d) For each enumerated (or hierarchical enumerated) attribute **b** of a multi-valued attribute **s** of type *DataType* in σ_1(X):
<ed_from_expr> :== <ed_from_expr> +
LEFT JOIN **X_s_T** AS s1 ON x.uuid = m1.uuid
Moreover during the translation of the selection condition σ_1(X) the attribute **b** must be referred to as: `s1.code_b` (`s1.codeH_b`).

- <ed_where_expr> :== SelCondition(σ_1(x))
where SelCondition(σ_1(x)) translates the selection condition σ_1(x) = [not](α_1 logicOperator ... logicOperator α_n) into the corresponding SQL statement through the following rules:

(a) `or`, `and` and `not` remain unchanged;

(b) each `(X.a op Y.b` is translated into `(x.a trad(op) x.b)`, where `trad(op)` translates the comparison operator (=, <, >, ≤, ≥, ≠) into the corresponding SQL comparison operator;

(c) each `(X.a op const)` is translated into `(x.a trad(op) trad (const))`, where `trad(const)` is the translation of the constant value into SQL, in particular a numeric value is live unchanged, while a string value is putted within single mark and so on.

(d) each `(X.a = null)` is translated into `(x.a IS NULL)` and each `(X.a = not null)` is translated in `(x.a IS NOT NULL)`.

Geometry selector The geometry selector expression for the constrained class can take one of the following forms:

1. *Basic geometry selector* `g`
   If the `geometry_selector` expression is of the basic form `g`, then the template for the `Constrained` temporary table is instantiated as follows:

   - `<ed_geom> :== x.g`

2. *Boundary/planar projection* `g.bnd()/g.pln()`
   If the existential topological constraint refers to the boundary or planar projection of the spatial attribute of the constrained class X, then the function `ST_Boundary()` or `ST_Force2D()` has to be applied in the constrained geometry expression. Therefore, the query template is instantiated as bellow:

   - `<ed_geom> :== ST_Boundary(x.g)`
   - `<ed_geom> :== ST_Union(ST_Force2d(x.g))`

3. *Segmented properities* `g.segmentsOf_A()`
   If the existential topological constraint refers to the segmented property A of the spatial attribute g of the constrained class X, that is if the geometry selector has the form `g.segmentsOf_A()`, then the query template is instantiated as:

   - `<ed_add_idattrib> :== , t1.A`
   - `<ed_from_expr> :== <espr_from_ed> +`
     `JOIN X_g_SG as t1 ON x.uuid = t1.uuid_X`
   - `<ed_geom> :== ST_Union(t1.geometry)`
   - `<ed_groupby_expr> :== GROUP BY t1.A, x.uuid`

   Notice that the function `ST_Union()` and the `GROUP BY` condition are added in order to reconstruct the *homogeneous segments* of A, that is the set of segments characterized by the same value of the segmented property. Indeed, there can exists many disconnected segments characterized by the same value of A.

4. *Selection on segmented properties* `g.segmentsOf_A(`σ_{1_a}`(X,A))`
   An existential topological constraint can also refers to a selection on the segmented property A of the spatial attribute g of X. In that case the geometry selector has the form `g.segmentsOf_A(`σ_{1_a}`(X,A))`, where

σ_{1_a}(X,*A*) contains a selection on the segmented property A, possibly a comparison between A and an attribute of X. Note that if the attribute of X is a multi-valued attribute, then the from clause has to be completed as illustrated previously.

In presence of a selection on segmented properties, the query template is instantiated as follows:

- <ed_add_idattrib> :== , t1.uuid
- <ed_add_attrib> :== , t1.**A**
- <ed_from_expr> :== <ed_from_expr>
  + ‘‘ JOIN **X_g_SG** as t1’’
  + ‘‘ ON x.uuid = t1.uuid_**X**’’
- <ed_geom> :== ST_Union(t1.geometry)
- <ed_where_expr> :== SelCondition(σ_{1_a}(x,t1.**A**))
  where SelCondition(σ_{1_a}(x,t1.**A**)) translates the selection condition σ_{1_a}(x,t1.**A**) = [not](α_1 logicOperator ...logicOperator α_n) into the corresponding SQL statement through the rules illustrated above.
- <ed_groupby_expr> :== GROUP BY t1.**A**, x.uuid

As in the previous case, the function ST_Union() and the GROUP BY condition are added in order to reconstruct the *homogeneous segments*.

4.1.4 Constraining class expression

As done for the constrained class expression, in this section the possible alternatives for the feature_selector and the geometry_selector expression of the constraining class are deeply analyzed and the corresponding query templates instantiated. Figure 5 summarizes how the constraining class expression can be built.

Feature selector The feature selector expression for the constraining class can take one of the following forms:

1. *Basic instance selector* Y
   If the feature_selector expression is of the basic form Y, then the template for the Constraining temporary table is instantiated as:
   - <ing_from_expr> :== **Y** as y
   - <ing_where_expr> :== true

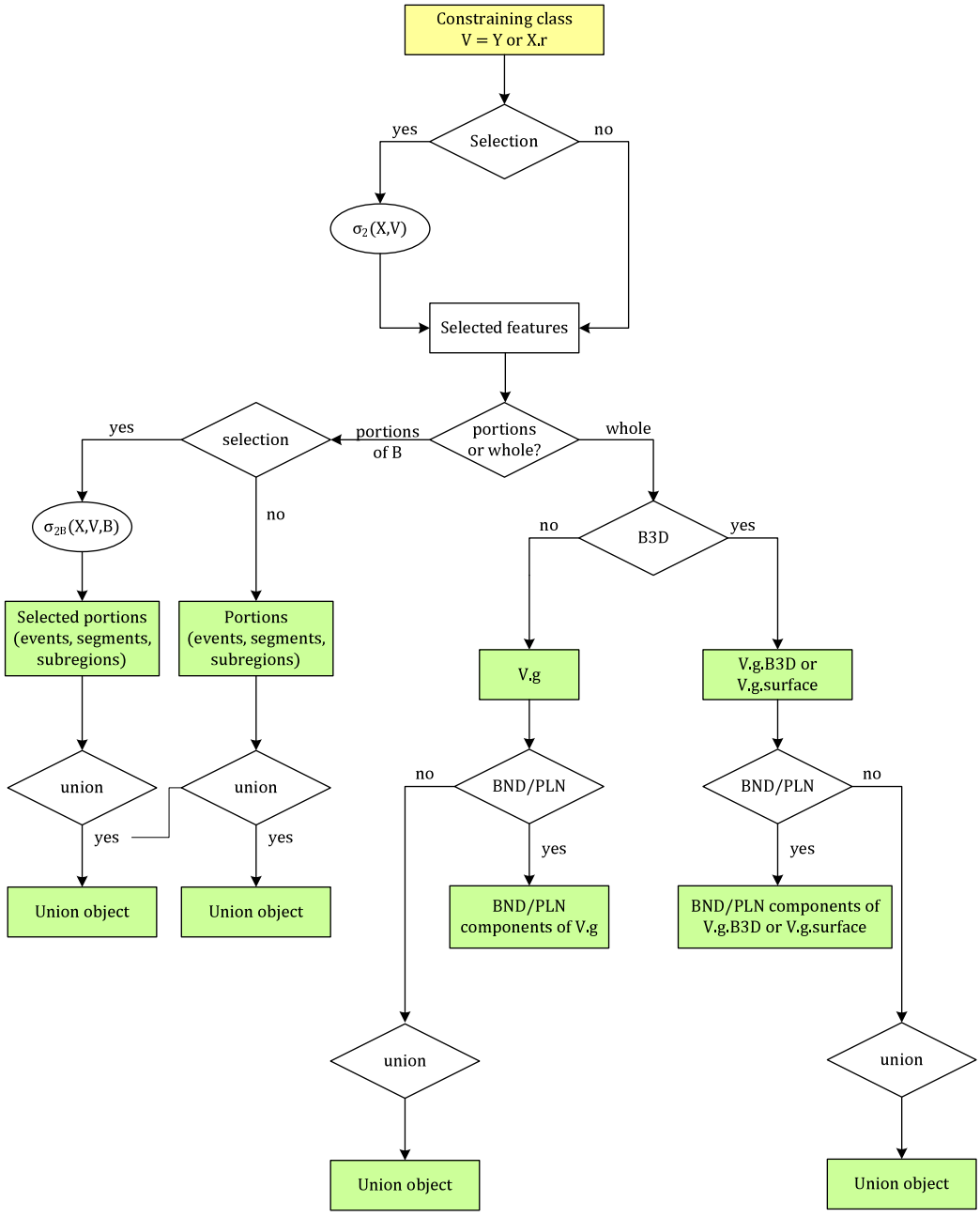


Figure 5: Schema about the selection process of the constraining geometries.

2. *Selection condition on class instances* $(\sigma_2$(X,Y))Y = $(\sigma_2$(Y))Y
For a condition $(\sigma_2$(X,Y))Y that does not involve attributes of X, simply denoted as $(\sigma_2$(Y))Y, the template is instantiated as:

   - <ing_from_expr> :== **Y** as y
     If the condition σ_2(Y)contains attributes of structured attributes (possibly multi-valued) or multi-valued attributes of the Y class, then the tables defining these attributes have to be joined in the from clause, as follows:

     (a) For each simple (i.e. of basic type) multi-valued attribute **a** in σ_2(Y):
     <ing_from_expr> :== <ing_from_expr> +
     LEFT JOIN **Y_a** AS m1 ON y.uuid = m1.uuid
     Moreover, during the translation of the selection condition σ_2(Y), the attribute **a** has to be referred to as m1.**a**.

     (b) For each enumerated (or hierarchical enumerated) multi-valued attribute **a** in σ_2(Y):
     <ing_from_expr> :== <ing_from_expr> +
     LEFT JOIN **Y_a** AS m1 ON y.uuid = m1.uuid
     Moreover, during the translation of the selection condition σ_2(Y), the attribute **a** has to be referred to as m1.code_**a** (m1.codeH_**a**).

     (c) For each simple (i.e. of basic type) attribute **b** of an attribute **s** of type ***DataType*** in σ_2(Y):
     <ing_from_expr> :== <ing_from_expr> +
     LEFT JOIN **Y_s_T** AS s1 ON x.uuid = s1.uuid
     Moreover, during the translation of the selection condition σ_2(Y), the attribute **b** has to be referred to as s1.**b**.

     (d) For each enumerated (or hierarchical enumerated) attribute **b** of an attribute **s** of type ***DataType*** in σ2(Y):
     <ing_from_expr> :== <ing_from_expr> +
     LEFT JOIN **Y_s_T** AS s1 ON x.uuid = s1.uuid
     Moreover, during the translation of the selection condition σ_2(Y), the attribute **b** has to be referred to as s1.code_**b** (s1.codeH_**b**).

   - <ing_where_expr> :== SelCondition(σ_2(y))
     where SelCondition(σ_2(y)) translates the condition σ_2(y)=[not] (α_1 logicOperator ... logicOperator α_n) into the corresponding SQL condition, using the rules presented above for σ_1(X).

3. *Selection condition on class instances* $(\sigma_2$(X,Y))Y
   For a condition of type $(\sigma_2$(X,Y))Y that involves the attributes $a_1, \ldots, a_n$ of X and the attributes $b_1, \ldots, b_m$ of Y, the template is instantiated as:

   - <ed_add_attrib> :== , x.$\mathbf{a}_1$, ..., x.$\mathbf{a}_n$
     Moreover, if the geometry selector of X refers to a segmented property, then the following condition has to be added:
     <ed_groupby_expr> :==
     <ed_groupby_expr>, x.$\mathbf{a}_1$, ..., x.$\mathbf{a}_n$
   - <ing_add_attrib> :==
     <ing_add_attrib>, y.$\mathbf{b}_1$, ..., y.$\mathbf{b}_m$
     Moreover, if the geometry selector of Y refers to a segmented property, then the following condition has to be added:
     <ing_groupby_expr> :==
     <ing_groupby_expr>, y.$\mathbf{b}_1$, ..., y.$\mathbf{b}_m$
   - <ing_from_expr> :== **Y** as y
     If σ_2(X,Y) refers to attributes of structured multi-valued attributes or to multi-valued attributes of the Y class, then the table containing the corresponding values have to be joined in the from clause as illustrated above.
     If σ_2(X,Y) refers to attributes of structured multi-valued attributes or to multi-valued attributes of the X class, then the table containing the corresponding values have to be joined into the query for the construction of the Constrained table as illustrated above.
   - <constr_where_expr> :== SelCondition(σ_2(x,y))
     where SelCondition(σ_2(x,y)) translate the condition σ_2(x,y) = [not](α_1 logicOperator ... logicOperator α_n) into the corresponding SQL expression, using the rules presended above.

4. *Constraining class connected to an association* X.r.f
   If the instances of the constraining class which are involved in the constraint are only those that are reachable from an association, then the <expr_where_exists> expression has to be enhanced in order to integrate the translation of $X.r.f$ and thus restrict the constraining class instances. In doing so, it is necessary to distinguish three cases:

   - r is a role with maximum cardinality 1 and it is represented into the table **X** by the attribute **uuid_r**:
     - <ed_add_attrib> :== <ed_add_attrib>, x.**uuid_r**
     - <constr_where_expr> :== <constr_where_expr>
       AND x.**uuid_r** = y.uuid

- r is a rule with maximum cardinality $n > 1$ and its inverse role r_i is represented into the table **Y** by the attribute uuid_**r**$_i$:
  - <ing_add_attrib> :== <ing_add_attrib>, y.uuid_**r**$_i$
  - <constr_where_expr> :== <constr_where_expr>
    AND y.uuid_**r** = x.uuid
- r is a rule with maximum cardinality $n > 1$ and its inverse role r_i is represented into an external table **A_X_Y** with attributes uuid_**r** and uuid_**r**$_i$:
  - <espr_from_vincolo> :==
    JOIN **A_X_Y** r ON r.uuid_**r** = y.uuid
  - <espr_where_vincolo> :== r.uuid_**ri** = x.uuid

Geometry selector The geometry selector expression for the constraining class can take one of the following forms:

1. *Basic geometry selector* f
   If the geometry_selector expression is of the basic form f, then the template for the Constraining temporary table is instantiated as:
   - <ing_geom> :== y.**f**
2. *Boundary/planar projection* f.bnd()/f.pln()
   If the existential topological constraint refers to the boundary or planar projection of the spatial attribute of the constrained class Y, then the function ST_Boundary() or ST_Force2D() has to be applied in the constrained geometry expression. Therefore, the query template is instantiated in the follow manner:
   - <ing_geom> :== ST_Boundary(y.**f**)
   - <ing_geom> :== ST_Union(ST_Force2d(y.**f**)
3. *Segmented properities* f.segmentsOf_*B()*
   If the existential topological constraint refers to the segmented property B of the spatial attribute f of the constraining class Y, that is the geometry selector has the form f.segmentsOf_*B*(), then the query template is instantiated as:
   - <ing_add_attrib > :== , t2.**B**
   - <ing_from_expr> :== <ing_from_expr> +
     JOIN **Y_f**_SG as t2 ON y.uuid = t2.uuid_**Y**

- <ing_geom> :== ST_Union(t2.geometry) AS f
- <ing_groupby_expr> :== GROUP BY t2.**B**, y.uuid

4. *Selection on segmented properties* f.segmentsOf_B(σ_{2_b}(y,B)
   An existential topological constraint can also refer to a selection σ_{2_b} on the segmented property B of the spatial attribute f of the constraining class Y. If the selection condition contains only selections on the attribute B and possibly comparisons between B and one or more attribute of Y, the templates are instantiated as:

   - <ing_add_attrib> :== , t2.**B**
   - <ing_from_expr> :== <ing_from_expr>
     + '' JOIN **Y_f_SG** as t2''
     + '' ON y.uuid = t1.uuid_**X**''
   - <ing_geom> :== ST_Union(t2.geometry)
   - <ing_where_expr> :== SelCondition(σ_{2_b}(y,t2.b))
     where SelCondition(σ_{2_b}(y, t2.b)) translate the condition σ_{2_b}(y, t2.b) = [not](α_1 logicOperator ... logicOperator α_n) into the corresponding SQL condition, using the rules explained previously.
   - <ing_groupby_expr> :== GROUP BY t2.**B**, y.uuid

   As explained for the constrained class expression, the function ST_Union() and the GROUP BY condition are added in order to reconstruct the *homogeneous segments*.

   Note that if the attribute of Y is a multi-valued attribute or an attribute of a structured multi-valued attribute, then the from clause of the query for constructing the Constraining temporal table has to be completed as explained previously.

5. *Selection on segmented properties* f.segmentsOf_B(σ_{2_b}(x,y,B))
   If the selection condition contains selections on the attribute B and possibly comparisons among B and the attributes a_1, ..., a_n of X and the attributes b_1, ..., b_m of Y , the templates are instantiated as as:

   - <ed_add_attrib> :== , x.$\mathbf{a}_1$, ..., x.$\mathbf{a}_n$
     Moreover, if the geometry selector of X refers to a segmented property, then the following condition has to be added:
     <ed_groupby_expr> :== <ed_groupby_expr>, x.$\mathbf{a}_1$,..., x.$\mathbf{a}_n$
   - <ing_add_attrib> :== , t2.**B**, y.$\mathbf{b}_1$, ..., y.$\mathbf{b}_m$

- <ing_from_expr> :== <ing_from_expr> +
  JOIN **Y_f_SG** as t2 ON y.uuid = t1.uuid_**X**
- <ing_geom> :== ST_Union(t2.geometry)
- <ing_groupby_expr> :==
  GROUP BY t2.**B**, y.uuid, y.$\mathbf{b}_1$, ..., y.$\mathbf{b}_m$
- <join_where_expr> :== SelCondition(σ_{2_b}(x,y,t2.**b**))
  where SelCondition(σ_{2_b}(y,t2.b)) translate the condition σ_{2_b}(y, t2.b) = [not](α_1 logicOper ... logicOper α_n) into the corresponding SQL condition, using the rules explained previously.

As previously, the function ST_Union() and the GROUP BY condition are added in order to reconstruct the *homogeneous segments*.

Note that if the attribute of the Y or X class is a multi-valued attribute or an attribute of a structured multi-valued attribute, then the from clause of the query for the construction of the Constraining or Constrained temporal table has to be completed as previously.

Example 2: Existential topological constraint with selection Let us consider the previous example about the *Road* and *RoadArea* classes and suppose that the constraint has to be verified only by roads with identifier value greater than 10. The constraint to check becomes:

```
constraint (Road.id > 10)Road.path (IN)
   exists RoadArea.extent
```

and the query templates are instantiated as follows:

```
CREATE TABLE Constrained(
   SELECT x.uuid, x.path as g
   FROM Road as x
   WHERE x.g IS NOT NULL AND x.uuid > 10)

CREATE TABLE Constraining(
   SELECT y.uuid, y.extent as f
   FROM RoadArea
   WHERE y.f IS NOT NULL)

SELECT x.uuid as failedObjects, x.g as failedGeometries
FROM Constrained as x
WHERE NOT EXISTS(SELECT *
```

```
   FROM Constraining as y )
   WHERE true AND ST_Within (x.path,y.extent))
```

Example 3: Existential topological constraint on segmented property Let us consider the *Road* and *RoadArea* classes of Example 1 and suppose that on the *Road* spatial attribute *path* is defined a segmented property *speedLimit*. In order to check that for each segment introduced by this segmented property there exist a road area that contains it:

```
   constraint Road.segmentsOf_Path (IN)
      exists RoadArea.extent
```

the query templates are instantiated in the following manner:

```
CREATE TABLE Constrained(
   SELECT x.uuid, t1.speedLimit, ST_Union(t1.geometry) as g
   FROM Road as x JOIN Road_Path_SG as t1
   WHERE ST_Union(t1.geometry) IS NOT NULL
   GROUP BY t1.speedLimit, x.uuid)

CREATE TABLE Constraining(
   SELECT y.uuid, y.extent as f
   FROM RoadArea
   WHERE y.f IS NOT NULL)

SELECT x.uuid as failedObjects, x.g as failedGeometries
   FROM Constrained as x
   WHERE NOT EXISTS(SELECT *
      FROM Constraining as y )
      WHERE true AND ST_Within (x.path,y.extent))
```

Example 4: Existential topological constraint with selection on segmented property Let us consider the example illustrated above and suppose that the constraint has to be satisfied only by the segments were the speed limit is above 50 km/h:

```
   constraint (speed_limit > 50)Road.segmentsOf_Path (IN)
      exists RoadArea.extent
```

the query templates are instantiated in the following manner:

```
CREATE TABLE Constrained(
   SELECT x.uuid, t1.speedLimit,
      t1.uuid, ST_Union(t1.geometry) as g
   FROM Road as x JOIN Road_Path_SG as t1
   WHERE ST_Union(t1.geometry) IS NOT NULL AND t1.speedLimit > 50
   GROUP BY t1.A, x.uuid)

CREATE TABLE Constraining(
   SELECT y.uuid, y.extent as f
   FROM RoadArea
   WHERE y.f IS NOT NULL)

SELECT x.uuid as failedObjects, x.g as failedGeometries
   FROM Constrained as x
   WHERE NOT EXISTS(SELECT *
      FROM Constraining as y
      WHERE true AND ST_Within (x.path,y.extent))
```

4.1.5 Union existential topological constraint

In a union existential topological constraint we refer to the union of the spatial attributes of one or more instances of the constraining class Y.

constraint <**contrained_class_expr**>(**rel**$_1$ | ... | **rel**$_n$)
union <**constraining_class_expr**>

Performing the union of a large number of geometries is an inefficient operation: some preliminary experiments using an increasing number of geometries with the `ST_Union()` function of PostGIS have been performed, showing that over the threshold of 5000 geometries the execution time is greater than an hour. In order to overcome this limitation we can observe that, to check the violation of a union existential topological constraint between the spatial attribute g of the constrained class instances and the union of spatial attribute f of the constraining class instances, it is enough to check the violation with respect to the union of all the spatial attributes f that have a nonempty intersection with g. Therefore, the geometric union has to be performed only on the necessary geometries, i.e. those that intersect the constrained geometry.

For applying this optimization, we build another a temporary table, named `Constraining_Union`, starting from the `Constrained` and `Constraining` ones, which contains for each instance x of the constrained class, the spa-

tial union of the instances of the constraining class Y whose spatial attribute intersects the spatial attribute of x.

Number of elements	Non optimized query	Optimized query
10	796 ms	531 ms
100	14,23 sec	4,98 sec
200	35,09 sec	10,05 sec
300	57,58 sec	15,14 sec
400	1,31 min	20,06 sec
500	1,67 min	24,69 sec
600	2,17 min	29,37 sec
700	2,56 min	32,47 sec
800	2,98 min	39,09 sec
900	3,56 min	44,00 sec
1000	4,06 min	47,55 sec
1500	6,76 min	1,23 min
2000	12,25 min	1,64 min
2500	17,03 min	1,95 min
3000	22,89 min	2,55 min
3500	31,52 min	2,94 min
4000	42,03 min	3,34 min
4500	54,78 min	3,73 min
5000	1,13 hour	4,16 min
5500	1,39 hour	4,64 min
6000	1,58 hour	5,00 min

Table 5: Comparison between the duration of the optimized and not optimized query for the union existential topological constraint.

```
CREATE TABLE Constraining_Union AS(
   SELECT x.uuid, x.g, x.pk_1, ..., x.pk_n,
      <ed_add_idattrib>,
      coalesce(ST_union(y.f), empty_geometry) as f_union
   FROM Constrained x LEFT JOIN Constraining y ON
      ST_Intersects(x.g, y.f)
      <const_from_expr>
   WHERE <const_where_expr>
   GROUP BY x.uuid, x.g, x.pk_1, ..., x.pk_n, <ed_add_idattrib>)

CREATE INDEX Constraining_union_Index ON
   Constraining_union USING GIST (f)
```

The keyword *empty_geometry* stands for an empty geometry spatial attribute, in PostGIS it can be obtained by considering the boundary of a ring, for example: `ST_Boundary(ST_GeomFromText('LINESTRING(0 0, 1 1, 2 3, 0 0)',-1))`.

The SQL query that retrieves the violating objects becomes:

```
SELECT u.uuid as failedObjects,
   u.pk_1, ..., u.pk_n, <ed_add_idattrib>,
   u.g as failedGeometries
FROM Constraining_Union u
WHERE Neg_Topo_Condition(u.g, u.f_union, rel1|...|reln)
```

where: `Neg_Topo_Condition(g,f,`$\texttt{rel}_1$ | ... | $\texttt{rel}_n$`) = `$\texttt{ST\_XXX}_{n+1}$`(g,f) OR ... OR `$\texttt{ST\_XXX}_m$`(g,f)` and $\texttt{ST\_XXX}_{n+1}$, ..., $\texttt{ST\_XXX}_m$ is the set of PostGIS topological relations corresponding to the GeoUML topological relations obtained as: $\texttt{ST\_XXX}_{n+1}$, ..., $\texttt{ST\_XXX}_m$ = RelTopo $\setminus$ $\{\texttt{rel}_1, \ldots, \texttt{rel}_n\}$

Example 4 Let us consider again the *Road* and *RoadArea* classes of the first example. Instead of requiring that for each road there exists a road area that contains it, suppose now to require that each road has to be contained into the union of the road area instances:

```
   constraint Road.path (IN) union RoadArea.extent
```

In this case the queries for creating the temporary table and checking the constraint become:

```
CREATE TABLE Constraining_Union AS(
   SELECT x.uuid, x.g,
      coalesce(ST_union(y.f),empty_geometry) as f_union
   FROM Constrained x LEFT JOIN Constraining y ON
      ST_Intersects(x.g,y.f)
   WHERE true
   GROUP BY x.uuid, x.g)

SELECT u.uuid as failedObjects, u.g as failedGeometries
FROM Constraining_Union u
WHERE (ST_Disjoint(x.g,y.f) OR
```

```
ST_Touch(x.g,y.f) OR
ST_Contains(x.g,y.f) OR
ST_Overlap(x.g,y.f) OR
ST_Crosses(x.g,y.f) OR
ST_Intersect(x.g,y.f))
```

4.1.6 Universal topological constraint

The universal topological constraint requires that the relation is valid between the constrained object and *all* the instances of the constraining class.

constraint <**constrained_class_expr**> **(rel**$_1$ | ... | **rel**$_2$**)**
forAll <**constraining_class_expr**>

An instance x of the constrained class X violates a universal topological constraint if there exists at least one instance y of the constraining class Y whose spatial attribute f has a relation with the spatial attribute g of x not included in the set $(rel_1 \mid \ldots \mid rel_n)$. Therefore, the universal topological constraint can be checked through the following SQL query template:

```
SELECT x.uuid as failedObjects,
   x.pk_1, ..., x.pk_n, <add_idattrib_ed>
   x.g as failedGeometries
FROM Constrained x, Constraining y <espr_from_vincolo>
WHERE <const_where_expr>
   Neg_Topo_Condition(x.g, y.f, rel1|...|reln)
```

where Neg_Topo_Condition(g,f,rel$_1$ | ... | rel$_n$) = ST_XXX$_{n+1}$(g,f) OR ... OR ST_XXX$_m$(g,f) and ST_XXX$_{n+1}$, ..., ST_XXX$_m$ is the set of PostGIS topological relations corresponding to the GeoUML topological relations obtained as: ST_XXX$_{n+1}$, ..., ST_XXX$_m$ = RelTopo \ {rel$_1$, ..., rel$_n$}.

Note that if <constrained_class> = <constraining_class>, then the condition x.uuid <> y.uuid AND has to be added into the <const_where_expr> clause.

Example 5 Let us consider the *RoadArea* class presented in Example 1 and a *Building* class translated in the following table:

```
Table Building
uuid:  varchar(64)
<Building attributes>
extent:  polygon
```

In order to check the constraint that a building can only touch or be disjoint from every road area:

```
constraint Building.extent (DJ|TC) forall RoadArea.extent
```

the query template has to be instantiated as follows:

```
SELECT x.uuid as failedObjects, x.g as failedGeometries
FROM Constrained x, Constraining y
WHERE (ST_Within(x.g,y.f) OR
   ST_Contains(x.g,y.f) OR
   ST_Overlap(x.g,y.f) OR
   ST_Equal(x.g,y.f))
```

4.2 Part-Whole Constraints Validation

This section deals with the translation into SQL queries of the GeoUML part-whole constraints presented in Section 2.3.2.

4.2.1 Strong composition constraint

In order to check that the spatial attribute f of the constrained class X is equal to the union of the spatial attribute g of one or more instances of the constraining class Y:

```
constraint <constrained_class_expr>
   composedOf <constraining_class_expr>
```

we first create a temporary table that, for each instance x of the constrained class, contains the spatial union of the instances of the constraining class Y, whose spatial attribute f is within the spatial attribute of x. The query template for creating this table is:

```
CREATE TABLE Constraining_Compo AS(
   SELECT x.uuid, x.g,
      x.pk_1, ..., x.pk_n, <ed_add_idattrib>,
      coalesce(ST_union(y.f), empty_geometry) as f_union
   FROM CONSTRAINED x LEFT JOIN CONSTRAINING y ON
      ST_Contains(x.g, y.f) OR
      ST_Covers(x.g, y.f) OR
```

```
      ST_Equals(x.g, y.f)
      <const_from_expr>
   WHERE <const_where_expr>
   GROUP BY x.uuid, x.g, x.pk_1, ..., x.pk_n,
         <ed_add_idattrib>)

CREATE INDEX Constraining_Compo_Index ON
   Constraining_compo USING GIST (f)
```

This template can be instantiated according to the constrained and the constraining class expression, as done for the topological constraint templates.

After the creation of this temporary table, the following query can be performed in order to find the X instances whose spatial attribute is not equal to the computed union of the spatial attributes of Y instances:

```
SELECT u.uuid as failedObjects,
   u.pk_1, ..., u.pk_n,
   <add_idattrib_ed>
   u.g as failedGeometries
FROM Constraining_Compo u
WHERE NOT ST_Equals(u.g, u.f_union)
```

As usual, this template has to be instantiated according to the particular constrained and constraing class expressions.

Example 6 Let us consider two classes *Region* and *County* which are translated into the following two tables:

Table Region	Table County
uuid: varchar(64)	uuid: varchar(64)
<Region attributes>	<County attributes>
extent: polygon	extent: polygon

In order to check the constraint that the region extent has to be equal to the union of the extent of some countries:

```
   constraint Region.extent composedOf County.extent
```

the query template for creating the table and checking the constraint become:

```
CREATE TABLE Constraining_Compo AS (
   SELECT x.uuid, x.g,
```

```
      coalesce(ST_union(y.f), empty_geometry) as f_union
   FROM Constrained x LEFT JOIN Constraining y ON
      ST_Contains(x.g,y.f) OR
      ST_Covers(x.g,y.f) OR
      ST_Equals(x.g,y.f)
   WHERE true
   GROUP BY x.uuid, x.g)

SELECT u.uuid as failedObjects, u.g as failedGeometries
FROM Constraining_Compo u
WHERE NOT ST_Equals(u.g,u.f_union)
```

4.2.2 Weak composition constraint

As in the previous case we build a temporary table, named `Constraining_WCompo`, which contains for each instance x of the constrained class X, the spatial union of the instances of the constraining class Y whose spatial attribute is within the spatial attribute of x.

```
CREATE TABLE Constraining_WCompo AS (
   SELECT x.uuid, x.g,
      x.pk_1, ..., x.pk_n, <ed_add_idattrib>,
      coalesce(ST_union(y.f), empty_geometry) as f_union
   FROM CONSTRAINED x LEFT JOIN CONSTRAINING y ON
      ST_Intersect(x.g,y.f)
      <const_from_expr>
   WHERE <const_where_expr>
   GROUP BY x.uuid, x.g, x.pk_1, ..., x.pk_n, <ed_add_idattrib>)

CREATE INDEX Constraining_wcompo_Index ON
   Constraining_wcompo USING GIST (f)
```

The constraint template query becomes:

```
SELECT u.uuid as failedObjects,
   u.pk_1, ..., u.pk_n,
   <ed_add_idattrib>
   u.g as failedGeometries
FROM CONSTRAINING_WCOMPO u
WHERE NOT ST_Within(u.g, u.f_union)
```

4.2.3 Membership constraint

The basic membership constraint can be translated into the query template defined for the existential topological constraint in which the relation required is fixed to *IN*. Instead, a particular attention is required for the *disjoint* (*dj-IN*) and *weak-disjoint* (*wdj-IN*) membership constraints. The disjoint membership constraint:

```
constraint <constrained_class_expr>
   dj-IN <constraining_class_expr>
```

an instance x of the constrained class X violates the constraint, if either: (1) its spatial attribute g is not contained into the spatial attribute f of any constraining instance y of Y, or (2) between g and the spatial attribute any other instance x' which is contained in the same f there exist a relation different from "disjoint or touch".

Therefore given the `Constrained` and `Constraining` tables built as above, the check of this constraint is performed in two steps: first the *IN* topological constraint is checked:

```
constraint <constrained_class_expr> (IN)
   exists <constraining_class_expr>
```

If this constraint is not satisfied the validation can terminate, otherwise the second step is performed which checks the disjunction of the "brothers" of x and returns the pairs of brothers that violate the *DJ/IN* constraint. The query that implements this second step is:

```
SELECT x1.uuid as failedObjects1,
   x1.pk_1, ..., x1.pk_n,
   x1.g as failedGeometries1
   x2.uuid as failedObjects2,
   x2.pk_1, ..., x2.pk_n,
   x2.g as failedGeometries2
FROM Constrained x1, Constrained x2
WHERE x1.uuid <> x2.uuid AND
   EXISTS (SELECT * FROM Constraining y
      WHERE ST_Within(x1.g,y.f) AND
         ST_Within(x2.g,y.f)) AND
         NOT(ST_Disjoint(x1.g,x2.g) OR ST_Touches(x1.g,x2.g))
```

The query template for the *wdj-IN* constraint is very similar to the one presented above. Compared to that one, the `ST_Cross(x1.g, x2.g)` relation has to be added in the where clause, as well as a test on the dimension of the geometric objects involved, in order to verify that they are both of linear type.

Example 7 Let us consider two classes *Region* and *County*, both with a polygonal spatial extent attribute. In order to check that each county extent has to be contained into a region and has to touch or be disjoint from the extent of all the other counties in the same region:

```
constraint County.extent dj-IN Region.extent
```

the query for checking the constraint become:

```
SELECT x1.uuid as failedObjects1,
   x1.g as failedGeometries1
   x2.uuid as failedObjects2,
   x2.g as failedGeometries2
FROM Constrained x1, Constrained x2
WHERE x1.uuid <> x2.uuid AND
   EXISTS (SELECT * FROM Constraining y
      WHERE ST_Within(x1.g,y.f) AND
         ST_Within(x2.g, y.f)) AND
         NOT(ST_Disjoint(x1.g,x2.g) OR ST_Touches(x1.g,x2.g))
```

4.2.4 Partition Constraint

As mentioned above a partition constraint can be obtained combining the strong composition constraint with a disjoint or weak-disjoint membership constraint. Therefore, in order to find the constrained instances that violate a particular partition constraint, it is enough to merge the result produced by the query for the strong composition constrain with the result produced by the query for the (weak-)disjoint membership constraint.

References

[1] Alberto Belussi, Mauro Negri, and Giuseppe Pelagatti. An iso tc 211 conformant approach to model spatial integrity constraints in the con-

ceptual design of geographical databases. In *ER (Workshops)*, pages 100–109, 2006.

[2] Eliseo Clementini, Paolino Di Felice, and Peter van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *SSD '93: Proceedings of the Third International Symposium on Advances in Spatial Databases*, pages 277–295, London, UK, 1993. Springer-Verlag.

[3] Object Management Group. Object Constraint Language (OCL). Specification 06-05-01, OMG, 2006.

[4] Open GeoSpatial Consortium Inc. Specification for Geographic Information - Simple Feature Access - Part 1: Common Architecture, 2006.

University of Verona
Department of Computer Science
Strada Le Grazie, 15
I-37134 Verona
Italy

http://www.di.univr.it