TOWARDS AUTOMATIC FEATURE TYPE PUBLICATION

Vânia Vidal, Marcel Teixeira, Fábio Feitosa
Departamento de Computação - Universidade Federal do Ceará (UFC)

Abstract: The purpose of the WFS specification, proposed by the the OpenGIS Consortium (OGC), is to describe the manipulation operations over geospatial data using GML. Web servers providing WFS service are called WFS Servers. Their objective is to provide queries, updates and exchange of geospatial data as geographic features instances encoded in GML.

This work proposes an approach for automatic feature type publication by WFS servers, where a feature type is specified by the feature type schema and a set of correspondence assertions. The feature type correspondence assertions formally specify relationships between the feature type schema and the base source schema. The adoption of correspondence assertions allows the automation of feature type publishing and maintenance. This paper presents an algorithm that automatically generates, based on the feature type correspondence assertion, the configuration file required to publish the feature type by a Deegree WFS.

Key words: Automatic Publication, Feature Type, WFS, GML, Correspondence Assertions, Deegree.

1. INTRODUCTION

The mission of the OpenGIS Consortium (OGC) is to promote the development and use of advanced open system standards and techniques in the area of geoprocessing and related information technologies. OGC manages a global consensus process that results in approved interfaces and encoding specifications that enable interoperability among diverse geospatial data stores, services, and applications[1]. Two important OGC's initiatives are:

the Geography Markup Language (GML)[2] and the Web Feature Service (WFS)[3] specifications. The purpose of the WFS specification is to describe the manipulation operations over geospatial data using GML.

According to OGC, a geographic feature is an "abstraction of a real world phenomenon associated with a location relative to the Earth". It is possible to describe feature form and localization through its geometric attributes, remaining the other attributes to represent its non-geographic properties. Given that, WFS servers publish GML views of geographic features stored in data sources, the user can query and update data sources through the feature view (feature type).

Today, the publication of a feature type is largely manual. It requires the generation of the feature type configuration file, which, usually, is a tedious and error-prone task. In case of modifications of the database schema, many feature types can be affected, which requires the re-generation of their configuration file.

We proposed an approach for automatic geographic feature publication which consists of four steps:

1. The User defines the XML schema of the feature type;
2. The base source schema is converted to an XML schema. Hence, the feature type schema and the base source schema are expressed in a "common" data model[4];
3. The correspondence assertions of the feature type are generated by matching the *feature type schema* and the *base source XML schema*[4]. The feature correspondence assertions formally specify relationships between the feature type schema and the base source schema;
4. The configuration file required to publish the feature type is automatically generated based on the feature type schema and its set of correspondence assertions.

In this paper we present an algorithm that generates the configuration file required to publish a feature type by a Deegree WFS[5]. We chose the Deegree WFS because it is a free implementation, and it enables one to define complex feature types, including feature types obtained by joining of tables in an Oracle database.

This paper is organized as follows. In Section 2, we describe how to convert a relational schema into an XML schema. In Section 3, we discuss how to generate feature correspondence assertions. In Section 4, we present the **GenerateDeegreeDataStoreConfigurationFile** algorithm that automatically generates, based on the feature type correspondence assertion, the configuration file required to publish a feature by a Deegree WFS. Finally, Section 5 contains the conclusions.

2. MAPPING RELATIONAL SCHEMA TO XMLS$^+$ SCHEMA

We assume that each relational schema **R** is first mapped to an XML schema **S**, generated according to the mapping rules described in[4]. Briefly, S has a root complex type, denoted **Troot**[**S**], which contains an element **R_i** of type **TR_i** for each relation schemes **R_i** in **R**. The complex type **TR_i** contains a sequence of elements **tuple_R_i** of type **Ttuple_R_i**. The complex type **Ttuple_R_i** contains an element **e** of a built-in XML data type for each attribute **e** of **R_i**. The referential integrity constraints in **R** are represented by keyref constraints[6].

The mapping rules guarantee that the relational schema **R** and the XML schema **S** are semantically equivalent, in the sense that each database state of **R** will correspond to a XML document that has **S** as XML schema, and vice-versa.

Consider in Figure 1(a) the relational schema **DB_School** for the base source. The corresponding XMLS schema, **XML_School**, is shown in Figure 2. The root type **Troot[XML_School]** contains an element for each relation in **DB_School**. For example, the element **Project**, of type **Tproject**, corresponds to the relation scheme **Project**. **Tproject** contains a sequence of zero or more

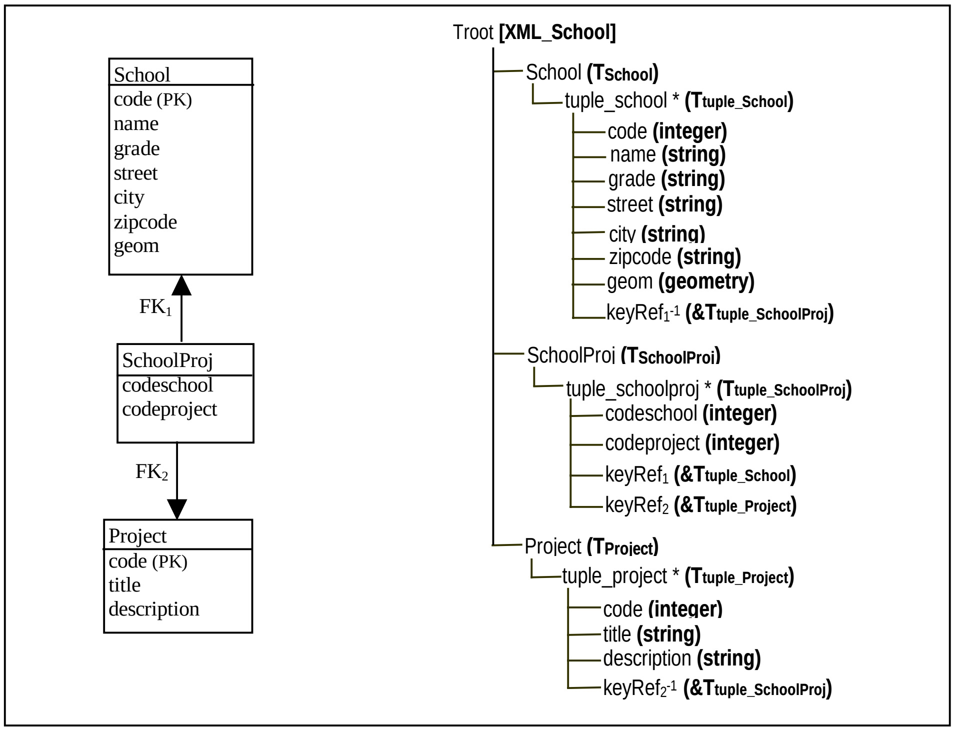


Figure 1. (a) Relational Schema DB_School; (b) Graphical Representation of XML_School Schema

tuple_project elements of type **Ttuple_project**, which in turn contains the elements **code**, **title** and **description**. The referential constraint **Fk$_1$:SchoolProj[codeschool]** $\subseteq$ **School[code]** in **R** is represented by the keyref constraint **keyref$_1$**.

We adopt a graphical notation, denoted XMLS$^+$ for Semantic XML Schema[4], to represent the types of a XML schema **S**. Briefly, the notation uses a tree-structured representation for the types of **S**, where bold fonts denote the name of the type, "&" denotes references, "@" denotes attributes and "*" denotes multiple occurrences of an element.

Figure 1(b) shows the graphical representation of the schema **XML_School**. In our graphical notation the keyref **keyRef$_1$** is represented by two reference elements: **Ttuple_SchoolProj** contains a reference element **keyRef$_1$** whose type is a reference to **Ttuple_School** (&**Ttuple_School**), and **Ttuple_School** contains a reference element **keyRef$_1^{-1}$** whose type is a reference to **Ttuple_SchoolProj** (&**Ttuple_SchoolProj**).

```
<schema>
  <element name="school"
     type="Troot[XML_School]">
    <key name="key1">
      <selector xpath=" School/tuple_school "/>
      <field xpath="code"/>
    </key>
    <key name="key2">
      <selector xpath="Project/tuple_project"/>
      <field xpath="code"/>
    </key>
    <key name="key3">
      <selector xpath="schoolproj/tuple_schoolproj"/>
      <field xpath="codeschool"/>
      <field xpath="codeproject"/>
    </key>
    <keyref name="keyRef1" refer="key1">
      <selector xpath="schoolproj/tuple_schoolproj "/>
      <field xpath=" codeschool"/>
    </keyref>
    <keyref name="keyRef2" refer="key2">
      <selector xpath="schoolproj/tuple_schoolproj"/>
      <field xpath="codeproject"/>
    </keyref>
  </element>
  <complexType name="Troot[XML_School]">
    <sequence>
      <element name="School" type="TSchool" />
      <element name="SchoolProj"
         type="TSchoolProj" />
     <element name="Project" type="TProject" />
    </sequence>
  </complexType>
  <complexType name="TSchool">
    <element name="tuple_school"
        type="Ttuple_School"
        minOccurs="0" maxOccurs="unbounded"/>
  </complexType>
```

```
<complexType name="TProject">
      <element name="tuple_project"
          type ="Ttuple_Project"
          minOccurs="0" maxOccurs="unbounded"/>
</complexType>
<complexType name="TSchoolProj">
     <element name="tuple_schoolproj"
         type="Ttuple_SchoolProj"
         minOccurs="0" maxOccurs="unbounded"/>
</complexType>
<complexType name= "Ttuple_School">
     <sequence>
        <element name="code" type="integer"/>
        <element name="geom" type="string"/>
        <element name="name" type="string"/>
        <element name="grade" type="string"/>
        <element name="street" type="string"/>
        <element name="city" type="string"/>
        <element name="zipcode" type="string"/>
     </sequence>
</complexType>
<complexType name= "Ttuple_Project">
     <sequence>
       <element name="code" type="integer"/>
       <element name="title" type="string"/>
       <element name="description" type="string"/>
     </sequence>
</complexType>
<complexType name="Ttuple_SchoolProj ">
     <sequence>
        <element name="codschool"
           type="integer"/>
        <element name="codproject"
           type="integer"/>
     </sequence>
 </complexType>
</schema>
```

Figure 2. XML_School Schema

We adopt an extension of Xpath[7] that permits navigating through an reference element. Let **\$S** be an instance of **$T_{root[XML\_School]}$**. Let **\$t** in **\$S/SchoolProj/tuple_schoolproj**. The path expression **\$t/$keyRef_1$** returns the element **\$e** in **\$S/School/tuple_school** of type **T_{tuple_School}** where **\$t/codeschool=\$e/code**. Likewise, given **\$e** in **\$S/School/tuple_school**, the path expression **\$e/$keyRef_1^{-1}$** returns the elements **\$t** in **\$S/SchoolProj/tuple_schoolproj** where **\$e/code= \$t/codeschool**.

3. USING CORRESPONDENCE ASSERTIONS FOR SPECIFYING FEATURE TYPE

In general, we propose to define a feature type with the help of a *feature type schema*, as usual, and a set of *Path Correspondence Assertions (PCAs)*[4,8,9]. A feature type F is defined by a 4-tuple **$F=< T_F, R_M, A_F >$** where **T_F** is the feature type schema, **R_M** is the Master table, and **A_F** is the set of path correspondence assertions that matches the proprieties of **T_F** with attributes/path of **T_{tuple_RM}** (the Master table's type).

A WFS GetFeature request[3] delivers feature instances of a given feature type, where each feature instance matches a tuple of the Master table[10]. We say that a feature instance **f** matches a tuple **t**, denoted **$f \equiv t$** , if they represents the same real world object.

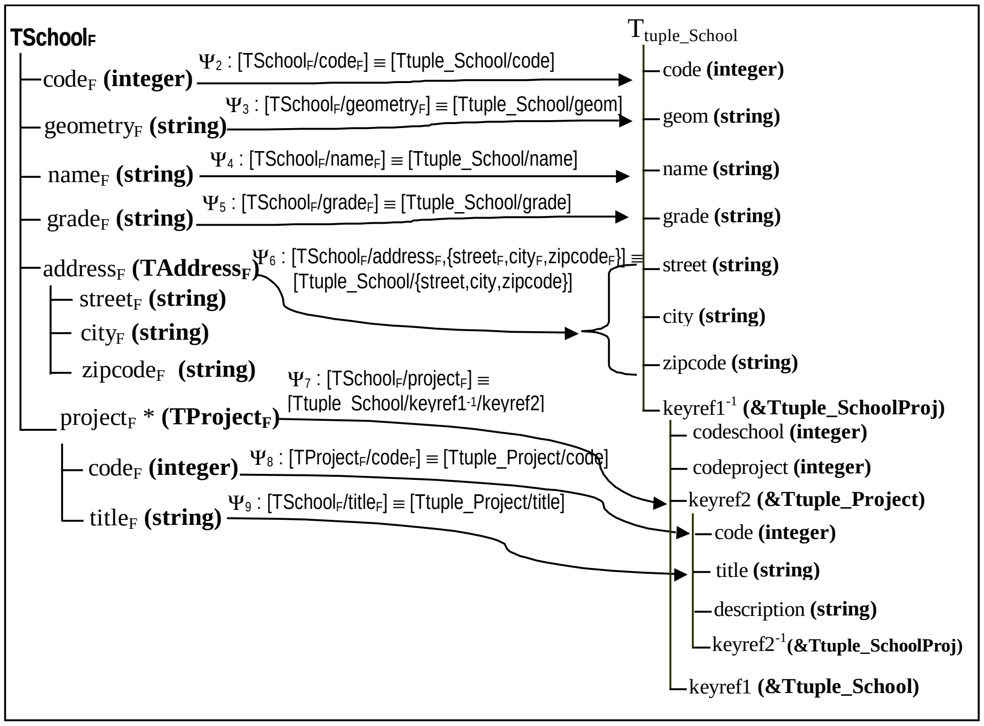

TSchool$_F$
code$_F$ (integer)
geometry$_F$ (string)
name$_F$ (string)
grade$_F$ (string)
address$_F$ (TAddress$_F$)
street$_F$ (string)
city$_F$ (string)
zipcode$_F$ (string)
project$_F$ * (TProject$_F$)
code$_F$ (integer)
title$_F$ (string)
Ψ_2 : [TSchool$_F$/code$_F$] ≡ [Ttuple_School/code]
Ψ_3 : [TSchool$_F$/geometry$_F$] ≡ [Ttuple_School/geom]
Ψ_4 : [TSchool$_F$/name$_F$] ≡ [Ttuple_School/name]
Ψ_5 : [TSchool$_F$/grade$_F$] ≡ [Ttuple_School/grade]
Ψ_6 : [TSchool$_F$/address$_F$,{street$_F$,city$_F$,zipcode$_F$}] ≡ [Ttuple_School/{street,city,zipcode}]
Ψ_7 : [TSchool$_F$/project$_F$] ≡ [Ttuple_School/keyref1^{-1}/keyref2]
Ψ_8 : [TProject$_F$/code$_F$] ≡ [Ttuple_Project/code]
Ψ_9 : [TSchool$_F$/title$_F$] ≡ [Ttuple_Project/title]
T$_{tuple\_School}$
code (integer)
geom (string)
name (string)
grade (string)
street (string)
city (string)
zipcode (string)
keyref1^{-1} (&Ttuple_SchoolProj)
codeschool (integer)
codeproject (integer)
keyref2 (&Ttuple_Project)
code (integer)
title (string)
description (string)
keyref2^{-1}(&Ttuple_SchoolProj)
keyref1 (&Ttuple_School)

Figure 3. Feature Type Schema and PCAs of School$_F$

Consider, for example, the feature type **$School_F$** whose schema is shown in Figure 3. Suppose that the table **School** of **DB_School** is the Master table for **$School_F$**. The PCAs of **$School_F$** are generated by matching the elements of **T_{School_F}** with the elements/paths of the base type **T_{tuple_School}**, and next recursively descend into sub-elements of **T_{School_F}** to define their correspondence. Figure 3 shows the set of PCAs for the feature type **T_{School_F}**.

The PCAs of **T_{School_F}** specify that:

(a) Given an instance **$\$s_f$** of **$T_{School_F}$** and an instance **$s** of **T_{tuple_School}**, if **$\$s_f$** *matches* **$s ($\$s_f \equiv \$s$)** then:

1. $\$s_f$/$code_f$ = \$s/code, from Ψ_2;
2. $\$s_f$/$geometry_f$ = \$s/geom, from Ψ_3;
3. $\$s_f$/$name_f$ = \$s/name, from Ψ_4;
4. $\$s_f$/$grade_f$ = \$s/grade, from Ψ_5;
5. $\$s_f$/$address_f$/$street_f$ = \$s/ street from Ψ_6;
6. $\$s_f$/$address_f$/ $city_f$ = \$s/city from Ψ_6;
7. $\$s_f$/$address_f$/ $zipcode_f$ = \$s/ zipcode from Ψ_6;
8. $\$p_f \in \$s_f/project_f$ iff exists $\$p \in \$s/keyref_1^{-1}/ keyref_2$ and $\$p_f \equiv \$p$, from Ψ_7;

(b) Given an instance **$\$p_f$** of **$T_{Project_F}$** and an instance **$p** of **$T_{tuple_Project}$**, if **$\$p_f \equiv \$p$** then:

1. $\$p_f$/$code_f$ = \$p/code, from Ψ_8;
2. $\$p_f$/$title_f$ = \$p/title, from Ψ_9;
3. $\$p_f$/$title_f$ = \$p/title, from Ψ_9;

4. AUTOMATIC FEATURE TYPE PUBLICATION BY DEEGREE WFS

In this section, let S be a relational schema, **S^+** be the corresponding $XMLS^+$ schema and **$F=< T_F, R_M, A_F >$** be a feature type over **S^+**, where **T_F** is the feature type schema, **R_M** is the Master table, and **A_F** is the set of PCAs that matches **T_F** with **T_{tuple_RM}** .

In order to publish a feature type by a Deegree WFS, we need to generate an XML file, named **DataStoreConfiguration** file, which defines the correspondences between the feature type properties and the attributes of the base tables. Figure 4 shows the **DataStoreConfiguration** file for the feature type **$School_F$** (Figure 3).

The configuration file contains a <MappingField> element for each property of the feature type schema. Each <MappingField> element indicates the data type of its associated property and database attribute that matches that property. For example, in Figure 4, the second <MappingField> element (lines 31 to 35) defines that the property, **$name_F$**, matches the attribute, **name**,

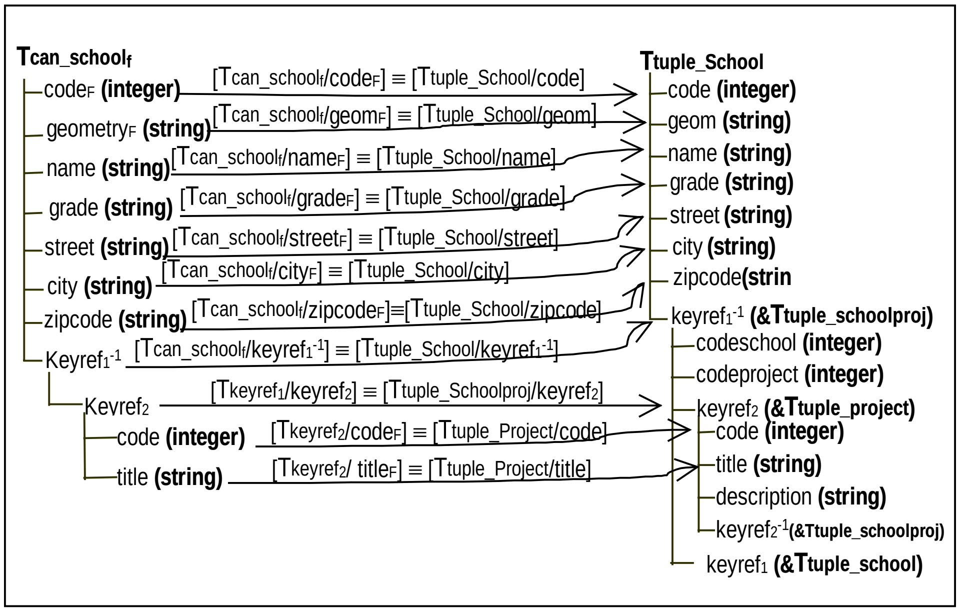
```
<?xml version="1.0" encoding="iso-8859-1"
  standalone="no"?>
<DatastoreConfiguration name="SCHOOL"
   type="ORACLESPATIAL">
  <Connection name="SCHOOL_con">
    <driver>oracle.jdbc.driver.OracleDriver</driver>
    <logon>
          jdbc:oracle:thin:@127.0.0.1:1521:dbGEOM
    </logon>
    <user>geoUser</user>
    <password> geoUser </password>
    <spatialversion>8.1.6</spatialversion>
  </Connection>
  <FeatureType name="SCHOOL">
   <OutputFormat>
      <GML2 responsibleClass="org.deegree_impl.
          services.wfs.oracle.DataStoreOutputGML">
         <Param name="FILTER"
          value="file:///.../SchoolTransf.xsl"/>
          <SchemaLocation>
            file:///…/SCHOOLSchema.xsd
         </SchemaLocation>
      </GML2>
   </OutputFormat>
   <MappingField>
         <Property name="KEYREF1^-1"
                type="xsd:integer"/>
         <DatastoreField name="SCHOOL.CODE"
                 type="NUMBER"/>
   </MappingField>
   <MappingField>
         <Property name="NAME" type="xsd:string"/>
         <DatastoreField name="SCHOOLL.NAME"
                type="VARCHAR2"/>
   </MappingField>
   <MappingField>
         <Property name="GRADE" type="xsd:string"/>
         <DatastoreField name="SCHOOL.GRADE"
                type="VARCHAR2"/>
   </MappingField>
   <MappingField>
         <Property name="STREET" type="xsd:string"/>
         <DatastoreField name="SCHOOL.STREET"
                type="VARCHAR2"/>
   </MappingField>
   <MappingField>
         <Property name="CITY" type="xsd:string"/>
         <DatastoreField name="SCHOOL.CITY"
                type="VARCHAR2"/>
   </MappingField>
   <MappingField>
         <Property name="ZIPCODE" type="xsd:string"/>
         <DatastoreField name="SCHOOL.ZIPCODE"
                type="VARCHAR2"/>
   </MappingField>
   <MappingField>
         <Property name="GEOM" type="xsd:string"/>
         <DatastoreField name="SCHOOL.GEOM"
                type="GEOMETRY"/>
   </MappingField>
   <MappingField>
         <Property name="KEYREF2" type="xsd:integer"/>
         <DatastoreField
   name="SCHOOLPROJ.CODEPROJECT"
                type="NUMBER"/>
   </MappingField>
   <MappingField>
         <Property name="CODE"
                type="xsd:integer"/>
         <DatastoreField name="PROJECT.CODE"
                type="NUMBER"/>
   </MappingField>
   <MappingField>
         <Property name="TITLE"
                type="xsd:string"/>
         <DatastoreField name="PROJECT.TITLE"
                type="VARCHAR2"/>
   </MappingField>
   <MasterTable name="SCHOOL"
         targetName="SCHOOL">
         <IdField number="true" auto="false">
           CODE
         </IdField>
         <Reference tableField="CODE"
                replaceable="true"
                targetTable="SCHOOLPROJ"
                targetField="CODESCHOOL"/>
         <GeoFieldIdentifier>
            GEOM
         </GeoFieldIdentifier>
   </MasterTable>
   <RelatedTable name="SCHOOLPROJ"
                targetName="SCHOOLPROJ"
                jointable="false">
         <IdField number="true" auto="false">
           CODESCHOOL
         </IdField>
         <Reference tableField="CODEPROJECT"
                replaceable="true"
                targetTable="PROJECT"
                targetField="CODE"/>
   </RelatedTable>
   <RelatedTable name="PROJECT"
                targetName="PROJECT"
                jointable="false">
         <IdField number="true" auto="false">
           CODE
         </IdField>
   </RelatedTable>
   <CRS>EPSG:4326</CRS>
  </FeatureType>
</DatastoreConfiguration>
```

Figure 4. DatastoreConfiguration File for the Feature Type School$_F$

of the table, **School**, as defined by the PCA **Ψ_4: [T_{School_F}/name$_F$] ≡ [T_{tuple_School}/name]**. It is important to notice that a <MappingField> element can only represent a PCA of the form **[T/p]≡[T_{tuple_R}/e]** where **e** is an element(attribute) of **T_{tuple_R}**. So, we cannot define a <MappingField> element for the PCA **Ψ_7: [T_{School_F}/project$_F$] ≡ [T_{tuple_School}/keyRef$_1^{-1}$/keyRef$_2$]**. However, we can solve the problem by making some adjustments to the feature type schema, by adding a new element **keyRef$_1^{-1}$** which matches the element **keyRef$_1^{-1}$** of **T_{tuple_School}**, and contains **project$_F$** as a sub element. The adjusted schema, named *canonical feature type schema* and its set of PCAs are shown in Figure 5. Note that all the PCAs of the canonical schema have the form **[T/p]≡[T_{tuple_R}/e]**. Therefore, we can define a <MappingField> element for each of them (see lines 25 to 78 of Figure 4).

In cases where the canonical feature type schema differs from the feature type schema defined by the user, we can define a filter for the GML output (see line 18 to 19 of Figure 4) transforming the result of a GetFeature request with "OutputFormat=GML2" to the schema defined in the referenced XSLT stylesheet. For example, we can define stylesheet that transforms instances of the canonical schema in Figure 5 to instances of **T_{School_F}**.

In our approach, we propose a three-step process to publish a feature type by a Deegree WFS: (i) The canonical feature type schema and its set of PCAs are automatically generated based on the feature type schema and PCAs. (ii) The **DataStoreConfiguration** file is generated for the canonical feature type schema. (iii) Whenever the canonical feature type schema is different from the feature type schema, generate the style file that specifies the rules to transform instances of the canonical schema into instances of the



Tcan_schoolf
codeF (integer)
geometryF (string)
name (string)
grade (string)
street (string)
city (string)
zipcode (string)
Keyref1-1
Kevref2
code (integer)
title (string)
[Tcan_schoolf/codeF] ≡ [Ttuple_School/code]
[Tcan_schoolf/geomF] ≡ [Ttuple_School/geom]
[Tcan_schoolf/nameF] ≡ [Ttuple_School/name]
[Tcan_schoolf/gradeF] ≡ [Ttuple_School/grade]
[Tcan_schoolf/streetF] ≡ [Ttuple_School/street]
[Tcan_schoolf/cityF] ≡ [Ttuple_School/city]
[Tcan_schoolf/zipcodeF]≡[Ttuple_School/zipcode]
[Tcan_schoolf/keyref1-1] ≡ [Ttuple_School/keyref1-1]
[Tkeyref1/keyref2] ≡ [Ttuple_Schoolproj/keyref2]
[Tkeyref2/codeF] ≡ [Ttuple_Project/code]
[Tkeyref2/ titleF] ≡ [Ttuple_Project/title]
Ttuple_School
code (integer)
geom (string)
name (string)
grade (string)
street (string)
city (string)
zipcode(strin
keyref1-1 (&Ttuple_schoolproj)
codeschool (integer)
codeproject (integer)
keyref2 (&Ttuple_project)
code (integer)
title (string)
description (string)
keyref2-1(&Ttuple_schoolproj)
keyref1 (&Ttuple_school)

Figure 5. Canonical Feature Type Schema and PCAs of School$_F$

feature type schema. The style file is automatically generated based on the Canonical schema, on the feature type schema and on their set of PCAs.

In Figure 6(a), we show the **GenerateDeegreeDataStoreConfigurationFile** Algorithm. The algorithm receives as input: $\mathbf{R_M}$ - the Master table, $\mathbf{T_C}$ - the canonical feature type and $\mathbf{A_C}$ - the PCAs of $\mathbf{T_C}$. In the following, we present a sequence of definitions used in the Algorithm. In those definitions, let **R** be a table of **S**.

Definition 4.1: Let $\mathbf{T_{tuple\_R}}$ be a type of **S** and **e** be a reference element of $\mathbf{T_{tuple\_R}}$ that represents the foreign key $R_t[F_t] \subseteq R[F_R]$ or the inverse of the foreign key $R[F_R] \subseteq R_t[F_t]$. Then, we say that: (i) R_t is the **targetTable** of **e**, (ii) F_t is the **targetField** of **e** and (iii) F_R is the **tableField** of **e**.

Definition 4.2: The set **Ref(R)** contains all the reference elements **e** of $\mathbf{T_{tuple\_R}}$ such that there exists a PCA of the form $[T/f] \equiv [T_{tuple\_R}/e]$ in $\mathbf{A_F}$.

Definition 4.3: The set **Tab(R)** contains all tables **R'** referenced by a reference element in **Ref(R)**. More formally, **Tab(R)** = { **R'** | exists **e** $\in$ **Ref(R)** where **targetTable(e)** = **R'**}.

Definition 4.4: The set **RelTab(R)** contains all tables **R'** related with **R**. More formally, **RelTab(R) = Tab(R)** $\cup$ { **RelTab(R')** | **R'**$\in$ **Tab(R)**}

The procedure **GenerateMappingFields** in Figure 6(b) generates a <MappingField> element for each property of $\mathbf{T_C}$ and, recursively, for the sub properties of $\mathbf{T_C}$. As we can see, the <MappingField> elements are directly defined based on the PCAs of the properties.

The procedure **GenerateRelatedTables** in Figure 6(c) generates the declaration of the <MasterTable> element and also the declaration of a <RelatedTable> element for each table in $\mathbf{RelTab(R_M)}$.

Given a table **R** (master table or related table), the procedure **GenerateReferences(R)** in Figure 6(d) generates the declaration of a <Reference> element for each reference element in **Ref(R)**.

```
Algorithm GenerateDeegreeDataStoreConfigurationFile ($T_C$, $R_M$, $A_C$)
    Let Map be a string, initially empty.
    Map + = GenerateFileHeading();
    Map + = GenerateMappingFields($T_C$, Ttuple_$R_M$);
    Map + = GenerateMappingTables($R_M$);
    Map + = GenerateFileFooting();
```

Figure 6. (a) – GenerateDeegreeConfigurationFile Algorithm

```
Procedure GenerateMappingFields (T, Ttuple_R)
  Let Map be a string, initially empty and visitedTables[ ] be a set.
  visitedTables[ ] = visitedTables[ ] ∪ R
  For each property p of T Do
    Case 1: If $\psi_p$ is of the form [T/p]≡[$T_{tuple\_R}$/e], where $T_e$ (the type of e) is a simple type.
      Map + = "<MappinField>"
              "<Property name=' "+ p + " ' type=' " + getGMLType(p) + " '/>"
              "<DatastoreField name=' " + R + "." + e + " ' type=' " + getType(e) + " '/>"
            "</MappingField>"
    Case 2: If $\psi_p$ is of the form [T/p]≡[$T_{tuple\_R}$/e], where e is a reference element and
            targetTable(e) = $R_t$ .
      Map + = "<MappinField>"
              "<Property name=' " + p + " ' type=' " + getGMLType(tableField(e)) + " '/>"
              "<DatastoreField name=' " + R + "."+ tableField(e) + " ' type= ' "+
                getType(tableField(e)) + " '/> "
            "</MappingField>"
      If $R_t \notin$ visitedTables[ ] Then
        GenerateMappingFields($T_p$ , Ttuple_$R_t$)
      End If
  End For
  Return Map
```

Figure 6. (b) – GenerateMappingFields Procedure

```
Procedure GenerateMappingTables($R_M$)
Note: Deegree assumes the Primary Keys (PKs) of the relations has only one attribute.
In our notation we use $k_R$ for the name of key attribute in Key(R).
  Let Map be a string, initially empty.
  Let $R_M$ be the Master table where Key($R_M$) = {$k_M$}. Let $A_g$ the geometric atribute of $R_M$.
  Map = "<MasterTable name=' "+ $R_M$ + " ' targetName= ' "+ $R_M$ + " '> "
           "<IdField number= ' " + IsNumber(getType($k_M$)) + " ' auto ='false'> " +
             $k_M$ + " </IdField> "
  Map +=     GenerateReferenceFields($R_M$)
  Map +=     "<GeomFieldIdentifier>" + $A_g$ + "</GeomFieldIdentifier>"
  Map += "</MasterTable>"
  For each R in RelTab($R_M$) Do
    Map += "<RelatedTable name=' "+ R + " ' targetName= ' "+ R + " '>"
             "<IdField number=' " + IsNumber(getType($k_R$)) + " ' auto ='false'> " +
               $K_R$ + "</IdField>"
    Map +=     GenerateReferenceFields (R)
    Map += "</RelatedTable>"
  End For
  Return Map
```

Figure 6. (c) – GenerateMappingTables Procedure

```
Procedure GenerateReferenceFields(R)
  Let Reference be a string, initially empty.
  For each e in Ref(R) Do
    Reference += <Reference tableField=" + tableField(e) + " replaceable="true"
                      targetTable=" + targetTable(e) + " targetField=" +
                      targetField(e) + ">
  End For
  Return Reference
```

Figure 6. (d) – GenerateReferenceFields Procedure

5. CONCLUSION

In this paper, we proposed an approach for automatic feature type publication by WFS servers. We first described how to generate the XMLS$^+$ schema for a relational schema. Then, we discussed how to specify feature type correspondence assertions, which formally specify the relationships between the feature type schema and the XMLS$^+$ base source schema. Our formalism handles schematic heterogeneity[11], and allows complex mappings to be specified quite simply. We also presented an algorithm that automatically generates, based on the feature type schema and its set of correspondence assertions, the **DataStoreConfiguration** file required to publish a feature type by a Deegree WFS. It is important to notice that the algorithm can be easily adapted to publish a feature type by other types of WFS implementation.

We have developed **DFP (Deegree Feature Publisher)**[12], a tool to support the publication of features by Deegree WFS.

6. REFERENCES

1. OpenGis Consortium. http://www.opengis.org/
2. S. Cox, P. Daisey, R. Lake, C. Portele, and A. Whiteside, OpenGIS® Geography Markup Language(GML) Implementation Specification. Version 3.00, 2003. http://www.opengis.org/ specs/?page=specs
3. P.A. Vretanos, Web Feature Service Implementation Specification. Version 1.0.0, 2002. http://www.opengis.org/specs/?page=specs
4. V.M.P. Vidal, and R. Vilas Boas, A Top-Down Approach for XML Schema Matching. In Proceedings of the 17th Brazilian Symposium on Databases. Gramado, Brazil, 2002.

5. Deegree. http://deegree.sourceforge.net/
6. World-Wide Web Consortium: Extensible Markup Language (XML). http://www.w3c.org/XML.
7. World-Wide Web Consortium: XML Path Language (XPath): Version 1.0 (*N*ovember 1999). http://www.w3.org/TR/xpath.
8. L. Popa, Y. Velegrakis, R.J. Miller, M.A. Hernandez, and R. Fagin, Translating Web Data. In VLDB, pages 598–609, August 2002.
9. E. Rahm, and P.A. Bernstein, A Survey of Approaches to Automatic Schema Matching. VLDB Journal, 10(4):334–350, 2001.
10. P. Rigaux, M. School, and A. Voisard, Spatial Database With Application To GIS. (Morgan Kaufmann Publishers, 2002)
11. F. Fonseca, and M. Egenhofer, Sistemas de Informações Geográficos Baseados em Ontologias. Informática Pública 1 (2):47-65, 2001.
12. M. Teixeira, Deegree Feature Publisher: Manual do Usuário, 2004. http://www.lia.ufc.br/~teixeira/ dfp